Python für Data Science

Release 24.1.0

Veit Schiele

11.04.2024

Inhaltsverzeichnis

Dies ist ein Tutorium über Data Science mit Python. Das wirft sofort die Frage auf: Was ist Data Science? Der Begriff ist mittlerweile allgegenwärtig, aber es gibt keine einheitliche Definition. Manche halten den Begriff sogar für überflüssig, denn welche Wissenschaft hat nicht mit Daten zu tun? Dennoch scheint mir, dass Data Science mehr als nur ein Hype ist: Wissenschaftliche Daten werden immer umfangreicher und lassen sich mit herkömmlichen mathematischen und statistischen Methoden allein oft nicht mehr adäquat erschließen – zusätzliche Hacking-Fähigkeiten sind gefragt. Es handelt sich jedoch nicht um ein neues Wissensgebiet, das ihr erlernen müsst, sondern um eine Reihe von Fähigkeiten, die ihr in eurem Bereich anwenden könnt. Ob ihr nun astronomische Objekte oder Maschinen analysiert, Börsenkurse prognostiziert oder in anderen Bereichen mit Daten arbeitet, das Ziel dieses Tutorials ist es, euch in die Lage zu versetzen, Aufgaben in eurem Bereich programmatisch zu lösen.

Dieses Tutorial ist nicht als Einführung in Python oder in die Programmierung im Allgemeinen gedacht; dafür gibt es das Python Basics-Tutorial. Stattdessen soll es den Python Data Science Stack – Bibliotheken wie *IPython*, *NumPy*, *pandas*, Matplotlib und verwandte Tools – vorstellen, damit ihr anschließend eure Daten effektiv analysieren und visualisieren könnt.

KAPITEL 1

Einführung

1.1 Zielgruppe

Die Zielgruppen sind vielfältig, von Data-Scientists über Data-Engineers und -Analysts bis hin zu Systems-Engineers. Ihre Fähigkeiten und Arbeitsabläufe sind sehr unterschiedlich. Eine der großen Stärken von Python für Data Science ist jedoch, dass es diesen verschiedenen Expert*innen ermöglicht, in funktionsübergreifenden Teams eng zusammenzuarbeiten.

Data-Scientists
: untersuchen Daten mit verschiedenen Parametern und fassen die Ergebnisse zusammen.

Data-Engineers
: überprüfen die Qualität des Codes und machen ihn robuster, effizienter und skalierbarer.

Data-Analysts
: nutzen den von Data-Engineers bereitgestellten Code, um die Daten systematisch zu analysieren.

Systems-Engineers
: stellen die Forschungsplattform auf Basis von JupyterHub bereit, auf der die anderen ihre Arbeit verrichten können.

In diesem Tutorial wenden wir uns an Systems-Engineers, die eine auf Jupyter-Notebooks basierende Plattform aufbauen und betreiben wollen. Wir erklären dann, wie diese Plattform von Data-Scientists, Data-Engineers und -Analysts effektiv genutzt werden kann.

1.2 Aufbau des Tutorials Python für Data Science

Ab Kapitel 2 folgt das Tutorial dem Prototyp eines Forschungsprojekts:

2. *Arbeitsbereich* mit der Installation und Konfiguration von *IPython*, Jupyter notebooks mit nbextensions und ipywidgets.
3. *Daten lesen, speichern und bereitstellen* entweder über eine *REST API* oder direkt über eine *HTML-Seite*.
4. *Daten bereinigen und validieren* ist eine wiederkehrende Aufgabe, bei der redundante, inkonsistente oder falsch formatierte Daten entfernt oder geändert werden.
5. *Daten visualisieren* wurde in ein separates Tutorial mit den vielen verschiedenen Möglichkeiten verschoben.
6. *Performance* stellt Möglichkeiten vor, wie ihr euren Code schneller laufen lassen könnt.
7. *Produkt erstellen* product zeigt, was notwendig ist, um reproduzierbare Ergebnisse zu erzielen: Es werden nicht nur *reproducible environments* benötigt, sondern auch die Versionierung des *Quellcodes* und der *Daten*. Der Quellcode sollte in *Programmbibliotheken verpackt werden* mit *Dokumentation*, *Lizenz(en)*, *Rests* und *Logging*. Schließlich enthält das Kapitel Ratschläge zur *Verbesserung der Codequalität* und des *sicheren Betriebs*.
8. *Web-Applikationen erstellen* kann entweder Dashboards aus Jupyter-Notebooks generieren oder eine umfassendere Anwendungslogik erfordern, wie in Bokeh-Plots in Flask einbinden, demonstriert, oder Daten über eine RESTful API bereitstellen.

:

1.3 Status

:

1.4 Folge uns

- GitHub
- Mastodon

1.5 Pull-Requests

Wenn ihr Vorschläge für Verbesserungen und Ergänzungen habt, empfehle ich euch, einen Fork meines GitHub-Repository zu erstellen und darin eure Änderungen vorzunehmen. Gerne dürft ihr auch einen *Pull Request* stellen. Sofern die darin enthaltenen Änderungen klein und atomar sind, schaue ich mir eure Vorschläge gerne an.

Da eine englischsprachige Übersetzung gepflegt wird, beachtet bitte folgende Richtlinien:

- Commit messages auf Englisch
- Commit messages mit einem Gitmoji am Anfang
- Namen von Ordnern und Dateien auf Englisch.

KAPITEL 2

Arbeitsbereich

Das Einrichten des Arbeitsbereichs umfasst das Installieren und Konfigurieren von *IPython* und Jupyter mit nbextensions und ipywidgets sowie *NumPy*.

2.1 IPython

IPython oder *Interactive Python* war zunächst ein erweiterter Python-Interpreter, der nun zu einem umfangreichen Projekt geworden ist, das Tools für den gesamten Lebenszyklus der Forschungsdatenverarbeitung bereitstellen soll. So ist IPython heute nicht nur eine interaktive Schnittstelle zu Python, sondern bietet auch eine Reihe nützlicher syntaktischer Ergänzungen für die Sprache. Darüberhinaus ist IPython eng mit dem Jupyter-Projekt verbunden.

Siehe auch:

- Miki Tebeka - IPython: The Productivity Booster

2.1.1 Starten der IPython-Shell

Ihr könnt IPython einfach in einer Konsole starten:

```
$ pipenv run ipython
Python 3.7.0 (default, Aug 22 2018, 15:22:29)
Type 'copyright', 'credits' or 'license' for more information
IPython 7.6.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Alternativ könnt ihr IPython auch in einem Jupyter-Notebook verwenden. Hierfür startet ihr zunächst den Notebook-Server:

```
$ pipenv run jupyter notebook
[I 17:35:02.419 NotebookApp] Serving notebooks from local directory: /Users/veit/cusy/
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪trn/Python4DataScience
[I 17:35:02.419 NotebookApp] The Jupyter Notebook is running at:
[I 17:35:02.427 NotebookApp] http://localhost:8888/?
↪token=72209334c2e325a68115902a63bd064db436c0c84aeced7f
[I 17:35:02.428 NotebookApp] Use Control-C to stop this server and shut down all kernels␣
↪(twice to skip confirmation).
[C 17:35:02.497 NotebookApp]
```

Anschließend sollte der Standardbrowser mit der angegebenen URL geöffnet werden. Häufig ist dies `http://localhost:8888`.

Nun könnt ihr im Browser einen Python-Prozess starten, indem ihr ein neues Notebook erstellt.

2.1.2 IPython-Beispiele

Ausführen von Python-Code

Python-Version anzeigen

[1]:
```
import sys


sys.version_info
```

[1]:
```
sys.version_info(major=3, minor=11, micro=4, releaselevel='final', serial=0)
```

Versionen von Python-Paketen anzeigen

Die meisten Python-Pakete bieten hierfür eine Methode `__version__`:

[2]:
```
import pandas as pd


pd.__version__
```

[2]:
```
'2.0.3'
```

Alternativ könnt ihr auch `version` aus `importlib_metadata` verwenden:

[3]:
```
from importlib_metadata import version


print(version("pandas"))
```

```
2.0.3
```

Informationen über das Host-Betriebssystem und die Versionen installierter Python-Pakete

[4]:
```
pd.show_versions()
```

```
/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/_
↪distutils_hack/__init__.py:33: UserWarning: Setuptools is replacing distutils.
  warnings.warn("Setuptools is replacing distutils.")


INSTALLED VERSIONS
------------------
commit           : 0f437949513225922d851e9581723d82120684a6
python           : 3.11.4.final.0
python-bits      : 64
OS               : Darwin
OS-release       : 22.5.0
Version          : Darwin Kernel Version 22.5.0: Thu Jun  8 22:22:23 PDT 2023; root:xnu-
↪8796.121.3~7/RELEASE_ARM64_T6020
machine          : arm64
processor        : arm
byteorder        : little
LC_ALL           : None
LANG             : de_DE.UTF-8
LOCALE           : de_DE.UTF-8

pandas           : 2.0.3
numpy            : 1.23.5
pytz             : 2023.3
dateutil         : 2.8.2
setuptools       : 68.0.0
pip              : 23.1.2
Cython           : 3.0.0
pytest           : 7.4.0
hypothesis       : 6.82.0
sphinx           : 7.1.2
blosc            : None
feather          : None
xlsxwriter       : None
lxml.etree       : 4.9.3
html5lib         : None
pymysql          : None
psycopg2         : None
jinja2           : 3.1.2
IPython          : 8.14.0
pandas_datareader: None
bs4              : 4.12.2
bottleneck       : None
brotli           : None
fastparquet      : 2023.7.0
fsspec           : 2023.6.0
gcsfs            : None
matplotlib       : 3.7.2
numba            : 0.57.1
numexpr          : None
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
odfpy           : None
openpyxl        : 3.1.2
pandas_gbq      : None
pyarrow         : 12.0.1
pyreadstat      : None
pyxlsb          : None
s3fs            : 2023.6.0
scipy           : 1.11.1
snappy          : None
sqlalchemy      : None
tables          : None
tabulate        : None
xarray          : 2023.7.0
xlrd            : None
zstandard       : None
tzdata          : 2023.3
qtpy            : None
pyqt5           : None
```

Nur Python-Versionen 3.8 verwenden

[5]:
```
import sys


assert sys.version_info[:2] >= (3, 8)
```

Shell-Kommandos

[6]:
```
!python3 -V
```

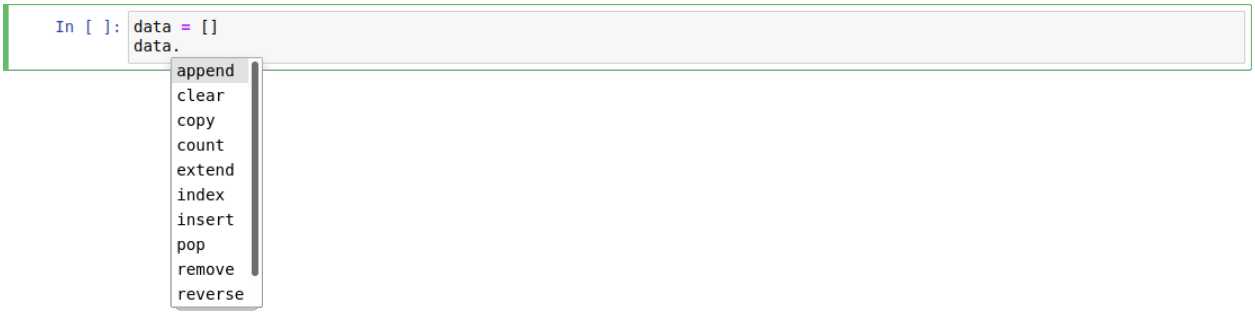
```
Python 3.11.4
```

[7]:
```
!python3 -m pip --version
```

```
pip 23.1.2 from /Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/
↪site-packages/pip (python 3.11)
```

Tab-Vervollständigung

… für Objekte mit Methoden und Attributen:

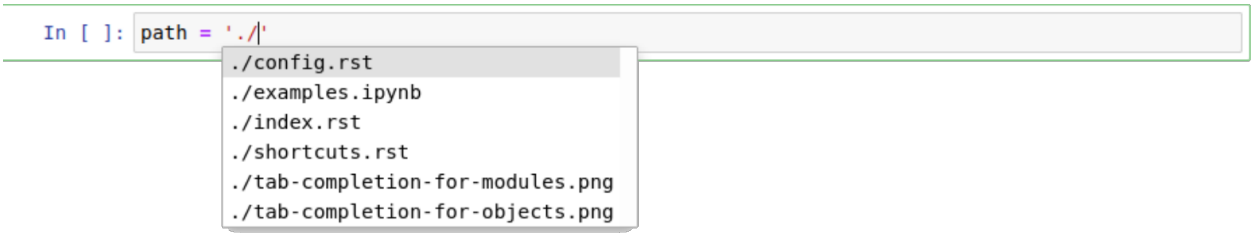


... und auch für Module:

Bemerkung

Wie ihr jetzt vielleicht verwundert festgestellt habt, wird die oben verwendete Methode `__version__` in der Auswahl nicht angeboten. IPython blendet diese *privaten* Methoden und Attribute, , die mit Unterstrichen beginnen, zunächst aus. Sie können jedoch auch mit einem Tabulator vervollständigt werden, wenn ihr zunächst einen Unterstrich eingebt. Alternativ könnt ihr diese Einstellung auch in der IPython-Konfiguration ändern.

... für fast alles:



Informationen über ein Objekt anzeigen

Mit einem Fragezeichen (?) könnt ihr euch Informationen über ein Objekt anzeigen lassen, wenn es z.B. eine Methode `multiply` mit folgendem Docstring gibt:

```
[8]: import numpy as np
```

```
[9]: np.mean?
```

```
Signature:
np.mean(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    a,
    axis=None,
    dtype=None,
    out=None,
    keepdims=<no value>,
    *,
    where=<no value>,
)
Docstring:
Compute the arithmetic mean along the specified axis.

Returns the average of the array elements.  The average is taken over
the flattened array by default, otherwise over the specified axis.
`float64` intermediate and return values are used for integer inputs.

Parameters
----------
a : array_like
    Array containing numbers whose mean is desired. If `a` is not an
    array, a conversion is attempted.
axis : None or int or tuple of ints, optional
    Axis or axes along which the means are computed. The default is to
    compute the mean of the flattened array.

    .. versionadded:: 1.7.0

    If this is a tuple of ints, a mean is performed over multiple axes,
    instead of a single axis or all the axes as before.
dtype : data-type, optional
    Type to use in computing the mean.  For integer inputs, the default
    is `float64`; for floating point inputs, it is the same as the
    input dtype.
out : ndarray, optional
    Alternate output array in which to place the result.  The default
    is ``None``; if provided, it must have the same shape as the
    expected output, but the type will be cast if necessary.
    See :ref:`ufuncs-output-type` for more details.

keepdims : bool, optional
    If this is set to True, the axes which are reduced are left
    in the result as dimensions with size one. With this option,
    the result will broadcast correctly against the input array.

    If the default value is passed, then `keepdims` will not be
    passed through to the `mean` method of sub-classes of
    `ndarray`, however any non-default value will be.  If the
    sub-class' method does not implement `keepdims` any
    exceptions will be raised.

where : array_like of bool, optional
    Elements to include in the mean. See `~numpy.ufunc.reduce` for details.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    .. versionadded:: 1.20.0

Returns
-------
m : ndarray, see dtype parameter above
    If `out=None`, returns a new array containing the mean values,
    otherwise a reference to the output array is returned.

See Also
--------
average : Weighted average
std, var, nanmean, nanstd, nanvar

Notes
-----
The arithmetic mean is the sum of the elements along the axis divided
by the number of elements.

Note that for floating-point input, the mean is computed using the
same precision the input has.  Depending on the input data, this can
cause the results to be inaccurate, especially for `float32` (see
example below).  Specifying a higher-precision accumulator using the
`dtype` keyword can alleviate this issue.

By default, `float16` results are computed using `float32` intermediates
for extra precision.

Examples
--------
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([2., 3.])
>>> np.mean(a, axis=1)
array([1.5, 3.5])

In single precision, `mean` can be inaccurate:

>>> a = np.zeros((2, 512*512), dtype=np.float32)
>>> a[0, :] = 1.0
>>> a[1, :] = 0.1
>>> np.mean(a)
0.54999924

Computing the mean in float64 is more accurate:

>>> np.mean(a, dtype=np.float64)
0.55000000074505806 # may vary

Specifying a where argument:
>>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
>>> np.mean(a)
12.0
>>> np.mean(a, where=[[True], [False], [False]])
9.0
File:      ~/spack/var/spack/environments/python-38/.spack-env/view/lib/python3.8/site-
↪packages/numpy/core/fromnumeric.py
Type:      function
```

Durch die Verwendung von ?? wird auch der Quellcode der Funktion angezeigt, sofern dies möglich ist:

[10]:
```
np.mean??
```

```
Signature:
np.mean(
    a,
    axis=None,
    dtype=None,
    out=None,
    keepdims=<no value>,
    *,
    where=<no value>,
)
Source:
@array_function_dispatch(_mean_dispatcher)
def mean(a, axis=None, dtype=None, out=None, keepdims=np._NoValue, *,
         where=np._NoValue):
    """
    Compute the arithmetic mean along the specified axis.

    Returns the average of the array elements.  The average is taken over
    the flattened array by default, otherwise over the specified axis.
    `float64` intermediate and return values are used for integer inputs.

    Parameters
    ----------
    a : array_like
        Array containing numbers whose mean is desired. If `a` is not an
        array, a conversion is attempted.
    axis : None or int or tuple of ints, optional
        Axis or axes along which the means are computed. The default is to
        compute the mean of the flattened array.

        .. versionadded:: 1.7.0

        If this is a tuple of ints, a mean is performed over multiple axes,
        instead of a single axis or all the axes as before.
    dtype : data-type, optional
        Type to use in computing the mean.  For integer inputs, the default
        is `float64`; for floating point inputs, it is the same as the
        input dtype.
    out : ndarray, optional
        Alternate output array in which to place the result.  The default
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        is ``None``; if provided, it must have the same shape as the
        expected output, but the type will be cast if necessary.
        See :ref:`ufuncs-output-type` for more details.

    keepdims : bool, optional
        If this is set to True, the axes which are reduced are left
        in the result as dimensions with size one. With this option,
        the result will broadcast correctly against the input array.

        If the default value is passed, then `keepdims` will not be
        passed through to the `mean` method of sub-classes of
        `ndarray`, however any non-default value will be.  If the
        sub-class' method does not implement `keepdims` any
        exceptions will be raised.

    where : array_like of bool, optional
        Elements to include in the mean. See `~numpy.ufunc.reduce` for details.

        .. versionadded:: 1.20.0

    Returns
    -------
    m : ndarray, see dtype parameter above
        If `out=None`, returns a new array containing the mean values,
        otherwise a reference to the output array is returned.

    See Also
    --------
    average : Weighted average
    std, var, nanmean, nanstd, nanvar

    Notes
    -----
    The arithmetic mean is the sum of the elements along the axis divided
    by the number of elements.

    Note that for floating-point input, the mean is computed using the
    same precision the input has.  Depending on the input data, this can
    cause the results to be inaccurate, especially for `float32` (see
    example below).  Specifying a higher-precision accumulator using the
    `dtype` keyword can alleviate this issue.

    By default, `float16` results are computed using `float32` intermediates
    for extra precision.

    Examples
    --------
    >>> a = np.array([[1, 2], [3, 4]])
    >>> np.mean(a)
    2.5
    >>> np.mean(a, axis=0)
    array([2., 3.])
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    >>> np.mean(a, axis=1)
    array([1.5, 3.5])

    In single precision, `mean` can be inaccurate:

    >>> a = np.zeros((2, 512*512), dtype=np.float32)
    >>> a[0, :] = 1.0
    >>> a[1, :] = 0.1
    >>> np.mean(a)
    0.54999924

    Computing the mean in float64 is more accurate:

    >>> np.mean(a, dtype=np.float64)
    0.55000000074505806 # may vary

    Specifying a where argument:
    >>> a = np.array([[5, 9, 13], [14, 10, 12], [11, 15, 19]])
    >>> np.mean(a)
    12.0
    >>> np.mean(a, where=[[True], [False], [False]])
    9.0

    """
    kwargs = {}
    if keepdims is not np._NoValue:
        kwargs['keepdims'] = keepdims
    if where is not np._NoValue:
        kwargs['where'] = where
    if type(a) is not mu.ndarray:
        try:
            mean = a.mean
        except AttributeError:
            pass
        else:
            return mean(axis=axis, dtype=dtype, out=out, **kwargs)

    return _methods._mean(a, axis=axis, dtype=dtype,
                          out=out, **kwargs)
File:      ~/spack/var/spack/environments/python-38/.spack-env/view/lib/python3.8/site-
↪packages/numpy/core/fromnumeric.py
Type:      function
```

? kann auch zur Suche im IPython-Namensraum verwendet werden. Dabei kann eine Reihe von Zeichen mit dem Platzhalter (*) dargestellt werden. Um z.B. eine Liste aller Funktionen im NumPy-Namensraum der obersten Ebene erhalten, die mean enthalten:

[11]:
```
np.*mean*?
```

```
np.mean
np.nanmean
```

2.1.3 IPython-Magie

IPython ermöglicht nicht nur Python interaktiv zu verwenden, sondern erweitert auch die Python-Syntax um sog. *Magic Commands*, die mit dem Präfix `%` versehen werden. Sie wurden entwickelt, um häufig auftretende Probleme bei der Datenanalyse schnell und einfach lösen zu können. Dabei wird zwischen zwei verschiedenen Arten von *Magic Commands* unterschieden:

- *line magics*, die durch einen einzelnen `%`-Präfix gekennzeichnet sind und auf einer einzelnen Eingabezeile ausgeführt werden
- *cell magics*, denen ein doppeltes Symbol `%%` vorangestellt wird und die innerhalb einer Notebook-Zelle ausgeführt werden.

Externen Code ausführen: `%run`

Wenn ihr anfangt, umfangreicheren Code zu entwickeln, arbeitet ihr vermutlich sowohl in IPython für interaktive Erkundungen als auch in einem Texteditor zum Speichern von Code, den ihr wiederverwenden möchtet. Mit der `%run`-Magie könnt ihr diesen Code direkt in eurer IPython-Sitzung ausführen.

Stellt euch vor, ihr hättet eine `myscript.py`-Datei mit folgendem Inhalt erstellt:

```
def square(x):
    return x ** 2

for N in range(1, 4):
    print(N, "squared is", square(N))
```

```
[1]: %run myscript.py
```

```
1 squared is 1
2 squared is 4
3 squared is 9
```

Beachtet, dass nach dem Ausführen dieses Skripts alle darin definierten Funktionen für die Verwendung in eurer IPython-Sitzung verfügbar sind:

```
[2]: square(4)
```

```
[2]: 16
```

Es gibt verschiedene Möglichkeiten, die Ausführung eures Codes zu verbessern. Wie üblich, könnt ihr euch die Dokumentation in IPython anzeigen lassen mit `%run?`.

Timing-Code ausführen: `%timeit`

Ein weiteres Beispiel für eine Magic-Funktion ist `%timeit`, die automatisch die Ausführungszeit der darauf folgenden einzeiligen Python-Anweisung ermittelt. So können wir uns z.B. die Performance einer *list comprehension* ausgeben lassen mit:

```
[3]: %timeit L = [n**2 for n in range(1000)]
```

```
27.4 µs ± 186 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Der Vorteil von `%timeit` ist, dass bei kurzen Befehlen automatisch mehrere Läufe ausgeführt werden, um robustere Ergebnisse zu erzielen. Bei mehrzeiligen Anweisungen wird durch Hinzufügen eines zweiten `%`-Zeichens eine Zellenmagie erzeugt, die mehrere Eingabezeilen verarbeiten kann. Hier ist zum Beispiel die äquivalente Konstruktion mit einer `for`-Schleife:

[4]:
```
%%timeit
L = []
for n in range(1000):
    L.append(n**2)
```

```
29.9 µs ± 170 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Wir können sofort erkennen, dass die *list comprehension* etwa 10% schneller ist als das entsprechende Äquivalent mit einer `for` Schleife. Ausführlicher beschreiben wir Performance-Messungen und -Optimierungen dann in *Profiling*.

Code anderer Interpreter ausführen

IPython verfügt über eine `%%script`-Magie, mit der ihr eine Zelle in einem Unterprozess eines Interpreters auf eurem System ausführen könnt, z.B. `bash`, `ruby`, `perl`, `zsh`, `R` usw. Dies kann auch ein eigenes Skript sein, das Eingaben in `stdin` erwartet. Hierzu wird einfach eine Pfadangabe oder ein Shell-Befehl an das Programm übergeben, das in der `%%script`-Zeile angegeben ist. Der Rest der Zelle wird von diesem Skript ausgeführt, `stdout` oder `err` aus dem Unterprozess erfasst und angezeigt.

[1]:
```
%%script python2
import sys


print 'Python %s' % sys.version
```

```
Python 2.7.15 (default, Oct 22 2018, 19:33:46)
[GCC 4.2.1 Compatible Apple LLVM 8.0.0 (clang-800.0.42.1)]
```

[2]:
```
%%script python3
import sys


print('Python: %s' % sys.version)
```

```
Python: 3.11.4 (main, Jun 20 2023, 17:23:00) [Clang 14.0.3 (clang-1403.0.22.14.1)]
```

[3]:
```
%%ruby
puts "Ruby #{RUBY_VERSION}"
```

```
Ruby 2.6.10
```

[4]:
```
%%bash
echo "$BASH"
```

```
/bin/bash
```

Ihr könnt `stdout` und `err` aus diesen Unterprozessen in Python-Variablen erfassen:

[5]:
```
%%bash --out output --err error
echo "stdout"
echo "stderr" >&2
```

[6]:
```
print(error)
print(output)
```

```
stderr

stdout
```

Standard-`script`-Magie konfigurieren

Die Liste der Aliase für die `script`-Magie ist konfigurierbar. Standardmäßig können ggf. einige gängige Interpreter verwendet werden. Ihr könnt jedoch in `ipython_config.py` auch eigene Interpreter angeben:

```
c.ScriptMagics.scripts = ['R', 'pypy', 'myprogram']
c.ScriptMagics.script_paths = {'myprogram': '/path/to/myprogram'}
```

Hilfe-Funktionen: `?`, `%magic` und `%lsmagic`

Wie normale Python-Funktionen verfügen auch die magischen IPython-Funktionen über *docstrings*, auf die einfach zugegriffen werden können. Um z.B. die Dokumentation der `%timeit`-Magie zu lesen, gebt einfach Folgendes ein:

[7]:
```
%timeit?
```

Auf die Dokumentation für andere Funktionen kann auf ähnliche Weise zugegriffen werden. Um auf eine allgemeine Beschreibung der verfügbaren `%magic`-Funktionen einschließlich einiger Beispiele zuzugreifen, könnt ihr Folgendes eingeben:

[8]:
```
%magic
```

Um schnell eine Liste aller verfügbaren `magic`-Funktionen zu erhalten, gebt Folgendes ein:

[9]:
```
%lsmagic
```

[9]:
```
Available line magics:
%alias  %alias_magic  %autoawait  %autocall  %automagic  %autosave  %bookmark  %cat  %cd␣
↪ %clear  %colors  %conda  %config  %connect_info  %cp  %debug  %dhist  %dirs  %doctest_
↪mode  %ed  %edit  %env  %gui  %hist  %history  %killbgscripts  %ldir  %less  %lf  %lk
↪%ll  %load  %load_ext  %loadpy  %logoff  %logon  %logstart  %logstate  %logstop  %ls
↪%lsmagic  %lx  %macro  %magic  %man  %matplotlib  %mkdir  %more  %mv  %notebook  %page␣
↪ %pastebin  %pdb  %pdef  %pdoc  %pfile  %pinfo  %pinfo2  %pip  %popd  %pprint
↪%precision  %prun  %psearch  %psource  %pushd  %pwd  %pycat  %pylab  %qtconsole
↪%quickref  %recall  %rehashx  %reload_ext  %rep  %rerun  %reset  %reset_selective  %rm␣
↪ %rmdir  %run  %save  %sc  %set_env  %store  %sx  %system  %tb  %time  %timeit
↪%unalias  %unload_ext  %who  %who_ls  %whos  %xdel  %xmode

Available cell magics:
%%!  %%HTML  %%SVG  %%bash  %%capture  %%debug  %%file  %%html  %%javascript  %%js  %
↪%latex  %%markdown  %%perl  %%prun  %%pypy  %%python  %%python2  %%python3  %%ruby  %
↪%script  %%sh  %%svg  %%sx  %%system  %%time  %%timeit  %%writefile

Automagic is ON, % prefix IS NOT needed for line magics.
```

Ihr könnt auch einfach eure eigenen `magic`-Funktionen definieren. Weitere Informationen hierzu erhaltet ihr unter Defining custom magics.

2.1.4 Shell-Befehle in IPython

IPython Notebooks ermöglichen die Ausführung einfacher UNIX/Linux-Befehle in einer Eingabezelle. Es gibt keine Einschränkungen, aber denkt bitte daran, dass ihr im Gegensatz zu einer normalen UNIX/Linux-Shell jeden Shell-Befehl `!` voranstellen müsst, zum Beispiel `!ls`für den Befehl `ls` (weitere Erläuterungen zum Befehl folgen weiter unten). Außerdem wird jeder Shell-Befehl in einer eigenen Subshell ausgeführt. Aus diesem Grund stehen Euch die Ergebnisse früherer Shell-Befehle nicht zur Verfügung.

Zunächst listet der Befehl `ls` die Dateien im aktuellen Arbeitsverzeichnis auf. Die Ausgabe wird unter der Eingabezelle angezeigt und listet eine einzelne Datei`shell.ipynb` auf:

[1]:
```
!ls
```

```
debugging.ipynb                 myscript.py
display.ipynb                   shell.ipynb
examples.ipynb                  start.rst
extensions.rst                  tab-completion-for-anything.png
importing.ipynb                 tab-completion-for-modules.png
index.rst                       tab-completion-for-objects.png
magics.ipynb                    unix-shell
mypackage
```

Der Befehl `!pwd` zeigt den Pfad zum Arbeitsverzeichnis (engl: **p**ath to the **w**orking **d**irectory) an:

[2]:
```
!pwd
```

```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython
```

Der Befehl `!echo` gibt Text aus, der dem `echo`-Befehl als Parameter übergeben wurde. Das folgende Beispiel zeigt, wie ihr *Hello world!* ausgeben könnt:

[3]:
```
!echo "Hello world!"
```

```
Hello world!
```

Werte an und von der Shell übergeben

Es gibt eine clevere Möglichkeit, auf die Ausgabe eines UNIX/Linux-Befehls als Variable in Python zuzugreifen, z.B. mit:

[4]:
```
contents = !ls
```

Hier wurde der Python-Variable `contents` die Ausgabe des Befehls `ls` zugewiesen. Als Ergebnis von `contents` entsteht eine Liste, wobei jedes Listenelement einer Zeile in der Ausgabe entspricht. Mit dem `print`-Befehl gebt ihr den Listeninhalt aus:

[5]:
```
print(contents)
```

```
['debugging.ipynb', 'display.ipynb', 'examples.ipynb', 'extensions.rst', 'importing.ipynb
↪', 'index.rst', 'magics.ipynb', '\x1b[34mmypackage\x1b[m\x1b[m', 'myscript.py', 'shell.
↪ipynb', 'start.rst', '\x1b[31mtab-completion-for-anything.png\x1b[m\x1b[m', '\
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪x1b[31mtab-completion-for-modules.png\x1b[m\x1b[m', '\x1b[31mtab-completion-for-
↪objects.png\x1b[m\x1b[m', '\x1b[34munix-shell\x1b[m\x1b[m']
```

Das gleiche Ergebnis seht ihr unten, wenn ihr den Befehl `pwd` ausführt:

[6]:
```
directory = !pwd
```

[7]:
```
print(directory)
```
```
['/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython']
```

Shell-bezogene Magic-Befehle

[8]:
```
!pwd
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython
```

[9]:
```
!cd ..
```

[10]:
```
!pwd
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython
```

[11]:
```
%cd ..
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace
```

Neben `%cd` gibt es noch folgende andere Shell-bezogene Magic-Befehle: `%cat`, `%cp`, `%env`, `%ls`, `%man`, `%mkdir`, `%more`, `%mv`, `%pwd`, `%rm` und `%rmdir`.

`%automagic`-Funktion

Mit der `%automagic`-Funktion lassen sich diese auch ohne vorangestelltes `%`-Zeichen verwenden:

[14]:
```
%automagic
```
```

Automagic is ON, % prefix IS NOT needed for line magics.
```

[15]:
```
cd ..
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs
```

2.1.5 Unix-Shell

Jeder Befehl auf der Kommandozeile funktioniert auch in Jupyter-Notebooks, wenn `!` vorangestellt wird. Die Ergebnisse können dann mit dem Jupyter-Namensraum interagieren, siehe *Werte an und von der Shell übergeben*.

Navigieren durch Dateien und Verzeichnisse

Zuerst wollen wir herausfinden, wo wir uns befinden, indem wir den Befehl `pwd` ausführen:

[1]:
```
!pwd
```

```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython/unix-shell
```

Hier ist die Antwort das iPython-Kapitel des Jupyter-Tutorials in meinem Home-Verzeichnis `/Users/veit`.

Unter Windows sieht das Home-Verzeichnis aus wie `C:\Documents and Settings\veit` oder `C:\Users\veit` und unter Linux wie `/home/veit`.

Um den Inhalt dieses Verzeichnisses zu sehen, können wir `ls` verwenden:

To see the contents of our this directory, we can use `ls`:

[2]:
```
!ls
```

```
create-delete.ipynb   grep-find.ipynb       pipes-filters.ipynb
file-system.ipynb     index.rst             shell-variables.ipynb
```

- ein nachgestelltes `/` kennzeichnet ein Verzeichnis
- `@` zeigt einen Link an
- `*` steht für eine ausführbare Datei

Abhängig von euren Standardoptionen kann die Shell auch Farben verwenden, um anzuzeigen, ob ein Eintrag eine Datei oder ein Verzeichnis ist.

`ls`-Optionen und Argumente

[3]:
```
!ls -F ../
```

```
debugging.ipynb                 myscript.py
display.ipynb                   shell.ipynb
examples.ipynb                  start.rst
extensions.rst                  tab-completion-for-anything.png*
importing.ipynb                 tab-completion-for-modules.png*
index.rst                       tab-completion-for-objects.png*
magics.ipynb                    unix-shell/
mypackage/
```

`ls` ist der Befehl, mit der Option `-F` und dem Argument `../`.

- Optionen beginnen entweder mit einem einzelnen Bindestrich (`-`) oder zwei Bindestrichen (`--`) und ändern das Verhalten eines Befehls.
- Argumente sagen dem Befehl, womit er operieren soll.
- Optionen und Argumente werden manchmal auch als Parameter bezeichnet.
- Jeder Teil wird durch Leerzeichen getrennt.

- Auch die Großschreibung ist wichtig, z. B.
  - `ls -s` zeigt neben den Namen auch die Größe der Dateien und Verzeichnisse an, während
  - `ls -S` die Dateien und Verzeichnisse nach Größe sortiert.

[4]:
```
!ls -s
```

```
total 136
24 create-delete.ipynb   40 grep-find.ipynb       16 pipes-filters.ipynb
32 file-system.ipynb      8 index.rst             16 shell-variables.ipynb
```

[5]:
```
!ls -S
```

```
grep-find.ipynb       create-delete.ipynb   shell-variables.ipynb
file-system.ipynb     pipes-filters.ipynb   index.rst
```

Alle Optionen und Argumente anzeigen

`ls` hat viele weitere Optionen, und mit `man` könnt ihr euch diese anzeigen lassen:

[6]:
```
!man ls
```

```
LS(1)                        General Commands Manual                        LS(1)

NNAAMMEE
     llss - list directory contents

SSYYNNOOPPSSIISS
     llss [--
↪@@AABBCCFFGGHHIILLOOPPRRSSTTUUWWaabbccddeeffgghhiikkllmmnnooppqqrrssttuuvvwwxxyy11%%,,
↪] [----ccoolloorr=_w_h_e_n]
        [--DD _f_o_r_m_a_t] [_f_i_l_e _._._.]

DDEESSCCRRIIPPTTIIOONN
     For each operand that names a _f_i_l_e of a type other than directory, llss
     displays its name as well as any requested, associated information.  For
     each operand that names a _f_i_l_e of type directory, llss displays the names
     of files contained within that directory, as well as any requested,
     associated information.

     If no operands are given, the contents of the current directory are
     displayed.  If more than one operand is given, non-directory operands are
     displayed first; directory and non-directory operands are sorted
     separately and in lexicographical order.

     The following options are available:

     --@@      Display extended attribute keys and sizes in long (--ll) output.

     --AA      Include directory entries whose names begin with a dot (‘_.’)
             except for _. and _._..  Automatically set for the super-user unless
             --II is specified.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    ...

macOS 13.4                      August 31, 2020                 macOS 13.4
```

Unzulässige Optionen

Wenn ihr versucht, eine Option zu verwenden, die nicht unterstützt wird, geben die Befehle normalerweise eine Fehlermeldung aus, z. B. für:

[7]:
```
!ls -z
```

```
ls: invalid option -- z
usage: ls [-@ABCFGHILOPRSTUWabcdefghiklmnopqrstuvwxy1%,] [--color=when] [-D format]␣
↪[file ...]
```

Versteckte Dateien

Mit der Option `-a` könn ihr alle Dateien anzeigen:

[8]:
```
!ls -a
```

```
.                    create-delete.ipynb   index.rst
..                   file-system.ipynb     pipes-filters.ipynb
.ipynb_checkpoints   grep-find.ipynb       shell-variables.ipynb
```

Zusätzlich zu den versteckten Verzeichnissen `..` und `.` seht ihr möglicherweise auch ein Verzeichnis namens `.ipynb_checkpoints`. Diese Datei enthält normalerweise Schnappschüsse der Jupyter-Notebooks.

Verzeichnisbaum anzeigen

[9]:
```
!tree
```

```
.
├── create-delete.ipynb
├── file-system.ipynb
├── grep-find.ipynb
├── index.rst
├── pipes-filters.ipynb
└── shell-variables.ipynb

1 directory, 6 files
```

Verzeichnis wechseln

Auf den ersten Blick mag für manche irritierend wirken, dass sie mit `!cd` nicht in ein anderes Verzeichnis wechseln können.

[10]:
```
!pwd
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython/unix-shell
```

[11]:
```
!cd ..
```

[12]:
```
!pwd
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython/unix-shell
```

Der Grund dafür ist, dass Jupyter eine temporäre Subshell verwendet. Wenn ihr dauerhaft in ein anderes Verzeichnis wechseln wollt, müsst ihr den *magischen Befehl* `%cd` verwenden.

[13]:
```
%cd ..
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython
```

[14]:
```
!pwd
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython
```

Mit der Funktion `%automagic` können diese auch ohne das vorangestellte `%`-Zeichen verwendet werden:

[16]:
```
%automagic
```
```

Automagic is ON, % prefix IS NOT needed for line magics.
```

[17]:
```
cd ..
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace
```

Absolute und relative Pfade

[18]:
```
cd .
```
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace
```

[19]:
```
cd ../..
```
```
/Users/veit/cusy/trn/Python4DataScience-de
```

[20]:
```
cd ..
```
```
/Users/veit/cusy/trn
```

[21]:
```
cd /
```
```
/
```

[22]:
```
cd
```
```
/Users/veit
```

[23]:
```
cd ~
```
```
/Users/veit
```

[24]:
```
cd /Users/veit
```
```
/Users/veit
```

Dateien und Verzeichnisse erstellen, aktualisieren und löschen

Legt ein neues Verzeichnis `test` an und überprüft dieses anschließend mit `ls`:

[1]:
```
!mkdir tests
```

[2]:
```
!ls
```
```
create-delete.ipynb   index.rst             tests
file-system.ipynb     pipes-filters.ipynb
grep-find.ipynb       shell-variables.ipynb
```

Dann legen wir in diesem Verzeichnis die Datei `test_file.txt` an:

[3]:
```
!touch tests/test_file.txt
```

[4]:
```
!ls tests
```
```
test_file.txt
```

Nun ändern wir das Suffix der Datei:

[5]:
```
!mv tests/test_file.txt tests/test_file.py
```

[6]:
```
!ls tests
```
```
test_file.py
```

Nun erstellen wir eine Kopie dieser Datei:

[7]:
```
!cp tests/test_file.py tests/test_file2.py
```

[8]:
```
!ls tests
```
```
test_file.py  test_file2.py
```

Auch ein Verzeichnis mit allen darin enthaltenen Dateien kann rekursiv mit der Option `-r` kopiert werden:

[9]:
```
!cp -r tests tests.bak
```

[10]:
```
!ls tests.bak
```

```
test_file.py  test_file2.py
```

Zum Schluss löschen wir die Verzeichnisse `tests` und `tests.bak` wieder:

[11]:
```
!rm -r tests tests.bak
```

[12]:
```
!ls
```

```
create-delete.ipynb   grep-find.ipynb       pipes-filters.ipynb
file-system.ipynb     index.rst             shell-variables.ipynb
```

Dateien übertragen

wget

[13]:
```
!wget https://dvc.org/deb/dvc.list
```

```
--2023-08-03 16:56:03--  https://dvc.org/deb/dvc.list
Auflösen des Hostnamens dvc.org (dvc.org)... 104.21.81.205, 172.67.164.76
Verbindungsaufbau zu dvc.org (dvc.org)|104.21.81.205|:443 ... verbunden.
HTTP-Anforderung gesendet, auf Antwort wird gewartet ... 303 See Other
Platz: https://s3-us-east-2.amazonaws.com/dvc-s3-repo/deb/dvc.list [folgend]
--2023-08-03 16:56:07--  https://s3-us-east-2.amazonaws.com/dvc-s3-repo/deb/dvc.list
Auflösen des Hostnamens s3-us-east-2.amazonaws.com (s3-us-east-2.amazonaws.com)... 52.
↪219.103.65, 52.219.94.129, 52.219.94.153, ...
Verbindungsaufbau zu s3-us-east-2.amazonaws.com (s3-us-east-2.amazonaws.com)|52.219.103.
↪65|:443 ... verbunden.
HTTP-Anforderung gesendet, auf Antwort wird gewartet ... 200 OK
Länge: 51 [binary/octet-stream]
Wird in »dvc.list« gespeichert.

dvc.list            100%[===================>]      51  --.-KB/s    in 0s

2023-08-03 16:56:20 (874 KB/s) - »dvc.list« gespeichert [51/51]
```

- `-r` crawlt rekursiv andere Dateien und Verzeichnisse
- `-np` vermeidet das Crawlen in übergeordneten Verzeichnissen
- `-D` zielt nur auf den folgenden Domainnamen
- `-nH` vermeidet das Anlegen eines Unterverzeichnisses für den Inhalt der Website
- `-m` spiegelt mit Zeitstempel, unendlicher Rekursionstiefe und Erhaltung der FTP-Verzeichniseinstellungen
- `-q` unterdrückt die Ausgabe auf dem Bildschirm

cURL

Alternativ könnt ihr cURL verwenden, das eine viel größere Auswahl an Protokollen unterstützt.

[14]:
```
!curl -o dvc.list https://dvc.org/deb/dvc.list
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100    85  100    85    0     0    271      0 --:--:-- --:--:-- --:--:--   274
```

Pipes und Filter

`ls` zeigt alle Dateien und Verzeichnisse an dieser Stelle an.

[1]:
```
!ls
```

```
create-delete.ipynb   grep-find.ipynb       shell-variables.ipynb
dvc.list              index.rst
file-system.ipynb     pipes-filters.ipynb
```

Mit `*.rst` schränken wir die Ergebnisse auf alle Dateien mit der Endung `.rst` ein:

[2]:
```
!ls *.rst
```

```
index.rst
```

Wir können auch die Anzahl der Zeilen, Wörter und Zeichen in diesen Dokumenten ausgeben:

[3]:
```
!wc *.rst
```

```
      17      43     441 index.rst
```

Nun schreiben wir die Anzahl der Zeichen in die Datei `length.txt` und geben den Text anschließend mit `cat` aus:

[4]:
```
!wc -m *.rst > length.txt
```

[5]:
```
!cat length.txt
```

```
     439 index.rst
```

Wir können die Dateien auch nach der Anzahl der Zeichen sortieren lassen:

[6]:
```
!sort -n length.txt
```

```
     439 index.rst
```

[7]:
```
!sort -n length.txt > sorted-length.txt
```

Wir können die vorhandene Datei auch überschreiben:

[8]:
```
!sort -n length.txt > length.txt
```

Wenn wir nur die Gesamtzahl der Zeichen wissen wollen, also nur die letzte Zeile ausgeben wollen, können wir dies mit `tail` tun:

[9]:
```
!tail -n 1 length.txt
```

Mit > wird eine Datei überschrieben, während >> zum Anhängen an eine Datei verwendet wird.

[10]:
```
!echo Anzahl der Zeichen >> length.txt
```

[11]:
```
!cat length.txt
```

```
Anzahl der Zeichen
```

Pipe |

Ihr könnt Befehle mit einer Pipe (|) verbinden. Im folgenden Einzeiler wollen wir die Anzahl der Zeichen für die kürzeste Datei anzeigen:

[12]:
```
!wc -l *.rst | sort -n | head
```

```
      17 index.rst
```

Wenn wir uns die ersten Zeilen des Haupttextes (ohne die ersten drei Zeilen für den Titel) anzeigen lassen wollen:

[13]:
```
!cat index.rst | head -n 5 | tail -n 2
```

```
Jeder Befehl auf der Kommandozeile funktioniert auch in Jupyter-Notebooks, wenn
``!`` vorangestellt wird. Die Ergebnisse können dann mit dem Jupyter-Namensraum
```

grep und find

grep

grep findet Zeilen in Dateien, die einem regulären Ausdruck entsprechen. Im folgenden Beispiel wird nach der Zeichenkette Python gesucht:

[1]:
```
!grep Python ../index.rst
```

```
IPython
`IPython <https://ipython.org/>`_ oder *Interactive Python* war zunächst ein
erweiterter Python-Interpreter, der nun zu einem umfangreichen Projekt geworden
bereitstellen soll. So ist IPython heute nicht nur eine interaktive
Schnittstelle zu Python, sondern bietet auch eine Reihe nützlicher syntaktischer
Ergänzungen für die Sprache. Darüberhinaus ist IPython eng mit dem
    * `Miki Tebeka - IPython: The Productivity Booster
```

Die Option -w begrenzt die Treffer auf die Wortgrenzen, so dass IPython ignoriert wird:

[2]:
```
!grep  -w Python ../index.rst
```

```
`IPython <https://ipython.org/>`_ oder *Interactive Python* war zunächst ein
erweiterter Python-Interpreter, der nun zu einem umfangreichen Projekt geworden
Schnittstelle zu Python, sondern bietet auch eine Reihe nützlicher syntaktischer
```

-n zeigt die Zeilennummern an, die übereinstimmen:

[3]:
```
!grep  -n -w Python ../index.rst
```

```
4:`IPython <https://ipython.org/>`_ oder *Interactive Python* war zunächst ein
5:erweiterter Python-Interpreter, der nun zu einem umfangreichen Projekt geworden
8:Schnittstelle zu Python, sondern bietet auch eine Reihe nützlicher syntaktischer
```

`-v` invertiert unsere Suche

[4]:
```
!grep  -n -v "^ " ../index.rst
```

```
1:IPython
2:=======
3:
4:`IPython <https://ipython.org/>`_ oder *Interactive Python* war zunächst ein
5:erweiterter Python-Interpreter, der nun zu einem umfangreichen Projekt geworden
6:ist, das Tools für den gesamten Lebenszyklus der Forschungsdatenverarbeitung
7:bereitstellen soll. So ist IPython heute nicht nur eine interaktive
8:Schnittstelle zu Python, sondern bietet auch eine Reihe nützlicher syntaktischer
9:Ergänzungen für die Sprache. Darüberhinaus ist IPython eng mit dem
10:`Jupyter-Projekt <https://jupyter.org/>`_ verbunden.
11:
12:.. seealso::
15:
16:.. toctree::
20:
```

`grep` hat viele andere Optionen. Um herauszufinden, welche das sind, könnt ihr folgendes eingeben:

[5]:
```
!grep --help
```

```
usage: grep [-abcdDEFGHhIiJLlMmnOopqRSsUVvwXxZz] [-A num] [-B num] [-C[num]]
        [-e pattern] [-f file] [--binary-files=value] [--color=when]
        [--context[=num]] [--directories=action] [--label] [--line-buffered]
        [--null] [pattern] [file ...]
```

Im folgenden Beispiel verwenden wir die Option `-E` und setzen das Muster in Anführungszeichen, damit die Shell nicht versucht, es zu interpretieren. `^` im Muster verankert die Übereinstimmung am Anfang der Zeile und `.` entspricht einem einzelnen Zeichen.

[6]:
```
!grep -n -E "^.Python" ../index.rst
```

```
1:IPython
```

find

`find .` sucht in diesem Verzeichnis, wobei die Suche mit `-type d` auf Verzeichnisse beschränkt wird.

[7]:
```
!find .. -type d
```

```
..
../mypackage
../.hypothesis
../.hypothesis/unicode_data
../.hypothesis/unicode_data/14.0.0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
../unix-shell
../.ipynb_checkpoints
```

Mit `-type f` ist die Suche auf Dateien beschränkt.

[8]:
```
!find . -type f
```

```
./index.rst
./sorted-length.txt
./create-delete.ipynb
./length.txt
./dvc.list
./file-system.ipynb
./pipes-filters.ipynb
./shell-variables.ipynb
./grep-find.ipynb
```

Mit `-mtime` wird die Suche auf die letzten `X` Tage beschränkt, in unserem Beispiel auf den letzten Tag:

[9]:
```
!find . -mtime -1
```

```
.
./sorted-length.txt
./create-delete.ipynb
./length.txt
./dvc.list
./file-system.ipynb
./pipes-filters.ipynb
./grep-find.ipynb
```

Mit `-name` könnt ihr die Suche nach Dateinamen filtern.

[10]:
```
!find .. -name "*.rst"
```

```
../index.rst
../unix-shell/index.rst
../extensions.rst
../start.rst
```

Jetzt zählen wir die Zeichen in den Dateien mit der Endung `.rst`:

[11]:
```
!wc -c $(find .. -name "*.rst")
```

```
     891 ../index.rst
     441 ../unix-shell/index.rst
    2281 ../extensions.rst
    1216 ../start.rst
    4829 total
```

Es ist auch möglich, in diesen Dateien nach einem regulären Ausdruck zu suchen:

[12]:
```
!grep "ipython.org" $(find .. -name "*.rst")
```

```
../index.rst:`IPython <https://ipython.org/>`_ oder *Interactive Python* war zunächst ein
```

Schließlich filtern wir alle Ergebnisse heraus, deren Pfad `ipynb_checkpoints` enthält:

[13]:
```
!find . -name "*.ipynb" | grep -v ipynb_checkpoints
```

```
./create-delete.ipynb
./file-system.ipynb
./pipes-filters.ipynb
./shell-variables.ipynb
./grep-find.ipynb
```

Shell-Variablen

Anzeige aller Shell-Variablen

[1]:
```
!set
```

```
...
HOME=/Users/veit
...
PATH=/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/bin:/opt/homebrew/Cellar/
↪pipenv/2023.6.18/libexec/tools:/Users/veit/spack/bin:/opt/homebrew/bin:/opt/homebrew/
↪sbin:/usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/
↪Library/TeX/texbin:/usr/local/MacGPG2/bin:/Library/Apple/usr/bin:/var/run/com.apple.
↪security.cryptexd/codex.system/bootstrap/usr/local/bin:/var/run/com.apple.security.
↪cryptexd/codex.system/bootstrap/usr/bin:/var/run/com.apple.security.cryptexd/codex.
↪system/bootstrap/usr/appleinternal/bin
...
```

Anzeigen des Wertes einer Variablen

[2]:
```
!echo $HOME
```

```
/Users/veit
```

Die `path`-Variable

Sie definiert den Suchpfad der Shell, d.h. die Liste der Verzeichnisse, in denen die Shell nach ausführbaren Programmen sucht.

[1]:
```
!echo $PATH
```

```
/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/bin:/opt/homebrew/Cellar/pipenv/
↪2023.6.18/libexec/tools:/Users/veit/spack/bin:/opt/homebrew/bin:/opt/homebrew/sbin:/
↪usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/bin:/bin:/usr/sbin:/sbin:/Library/TeX/
↪texbin:/usr/local/MacGPG2/bin:/Library/Apple/usr/bin:/var/run/com.apple.security.
↪cryptexd/codex.system/bootstrap/usr/local/bin:/var/run/com.apple.security.cryptexd/
↪codex.system/bootstrap/usr/bin:/var/run/com.apple.security.cryptexd/codex.system/
↪bootstrap/usr/appleinternal/bin
```

Erstellen und Ändern von Variablen

Anlegen oder Überschreiben von Variablen

```
[4]: !export SPACK_ROOT=~/spack
```

Zusätzliche Spezifikationen anhängen

```
[5]: !export PATH=/usr/local/opt/python@3.7/bin:$PATH
```

2.1.6 Objekte anzeigen mit `display`

IPython kann Objekte anzeigen wie z.B. `HTML`, `JSON`, `PNG`, `JPEG`, `SVG` und `Latex`.

Bilder

Um Bilder (`JPEG`, `PNG`) in IPython und Notebooks anzuzeigen, könnt ihr die `Image`-Klasse verwenden:

```
[1]: from IPython.display import Image


     Image("https://www.python.org/images/python-logo.gif")
```

```
[1]: <IPython.core.display.Image object>
```

```
[2]: from IPython.display import SVG


     SVG(
         "https://upload.wikimedia.org/wikipedia/commons/c/c3/Python-logo-notext.svg"
     )
```

```
[2]:
```

Nicht-eingebettete Bilder

- Standardmäßig sind Bilddaten eingebettet:

  ```
  Image ('img_url')
  ```

- Wenn jedoch `url` als `kwarg` angegeben ist, wird dies als *Softlink* interpretiert:

  ```
  Image (url='img_url')
  ```

- `embed` kann jedoch auch explizit angegeben werden:

  ```
  Image (url='img_url', embed = True)
  ```

HTML

Python-Objekte können HTML-Repräsentationen deklarieren, die in einem Notebook angezeigt werden:

[3]:
```
from IPython.display import HTML
```

[4]:
```
%%html
<ul>
    <li>foo</li>
    <li>bar</li>
</ul>
```

```
<IPython.core.display.HTML object>
```

Javascript

Mit Notebooks können Objekte auch eine JavaScript-Darstellung deklarieren. Dies ermöglicht dann z.B. Datenvisualisierungen mit Javascript-Bibliotheken wie d3.js.

[5]:
```
from IPython.display import Javascript


welcome = Javascript(
    'alert("Dies ist ein Beispiel für eine durch IPython angezeigte Javascript-Warnung.")
↪'
)
display(welcome)
```

```
<IPython.core.display.Javascript object>
```

Für umfangreicheres Javascript könnt ihr auch die `%%javascript`-Syntax verwenden.

LaTeX

`IPython.display` verfügt außerdem über eine integrierte Unterstützung für die Anzeige von mathematischen Ausdrücken, die in LaTeX gesetzt sind und im Browser mit MathJax gerendert werden:

[6]:
```
from IPython.display import Math


Math(r"F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k} dx")
```

[6]:
$$F(k) = \int_{-\infty}^{\infty} f(x) e^{2\pi i k} dx$$

Bei der `Latex`-Klasse müsst ihr die Begrenzungen selbst angeben. Auf diese Weise könnt ihr jedoch auch andere LaTeX-Modi verwenden, wie z.B. `eqnarray`:

[7]:
```
from IPython.display import Latex


Latex(
    r"""\begin{eqnarray}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
\nabla \times \vec{\mathbf{B}} -\, \frac1c\, \frac{\partial\vec{\mathbf{E}}}{\partial t}␣
↪& = \frac{4\pi}{c}\vec{\mathbf{j}} \\
\end{eqnarray}"""
)
```

[7]:

$$\nabla \times \vec{\mathbf{B}} - \frac{1}{c}\frac{\partial \vec{\mathbf{E}}}{\partial t} = \tfrac{4\pi}{c}\vec{\mathbf{j}} \tag{2.1}$$

(2.2)

Audio

IPython ermöglicht auch das interaktive Arbeiten mit Sounds. Mit der `display.Audio`-Klasse könnt ihr ein Audio-Control erstellen, das in das Notebook eingebettet ist. Die Schnittstelle ist analog zu denjenigen der `Image`-Klasse. Alle vom Browser unterstützten Audioformate können verwendet werden.

[8]:
```
from IPython.display import Audio
```

Im folgenden werden wir die Sinusfunktion eines NumPy-Array als Audiosignal ausgeben. Dabei normalisiert und codiert die `Audio`-Klasse die Daten und bettet das resultierende Audio in das Notebook ein.

[9]:
```
import numpy as np


f = 500.0
rate = 8000
L = 3
times = np.linspace(0, L, rate * L)
signal = np.sin(f * times)

Audio(data=signal, rate=rate)
```

[9]:
```
<IPython.lib.display.Audio object>
```

Links zu lokalen Dateien

IPython bietet integrierte Klassen zum Generieren von Links zu lokalen Dateien. Erstellt hierzu eine Verknüpfung zu einer einzelnen Datei mit dem `FileLink`-Objekt:

[10]:
```
from IPython.display import FileLink, FileLinks


FileLink("magics.ipynb")
```

[10]:
```
/Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/ipython/magics.ipynb
```

Alternativ könnt ihr auch eine Liste mit Links zu allen Dateien in einem Verzeichnis generieren, z.B.:

[11]:
```
FileLinks(".")
```

[11]:
```
./
  index.rst
  tab-completion-for-modules.png
  tab-completion-for-objects.png
  tab-completion-for-anything.png
  debugging.ipynb
  magics.ipynb
  shell.ipynb
  display.ipynb
  examples.ipynb
  myscript.py
  importing.ipynb
  extensions.rst
  start.rst
./.hypothesis/unicode_data/14.0.0/
  charmap.json.gz
./.ipynb_checkpoints/
  examples-checkpoint.ipynb
  magics-checkpoint.ipynb
  display-checkpoint.ipynb
  shell-checkpoint.ipynb
./mypackage/
  __init__.py
  foo.ipynb
./unix-shell/
  index.rst
  sorted-length.txt
  create-delete.ipynb
  length.txt
  dvc.list
  file-system.ipynb
  pipes-filters.ipynb
  shell-variables.ipynb
  grep-find.ipynb
./unix-shell/.ipynb_checkpoints/
  grep-find-checkpoint.ipynb
  shell-variables-checkpoint.ipynb
```

Notebooks anzeigen

[12]:
```
import os
import sys
import types

import nbformat

from IPython.display import HTML, display
from pygments import highlight
from pygments.formatters import HtmlFormatter
from pygments.lexers import PythonLexer
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
formatter = HtmlFormatter()
lexer = PythonLexer()

# publish the CSS for pygments highlighting
display(
    HTML(
        """
<style type='text/css'>
%s
</style>
"""
        % formatter.get_style_defs()
    )
)
```

```
<IPython.core.display.HTML object>
```

[13]:
```
def show_notebook(fname):
    """display a short summary of the cells of a notebook"""
    nb = nbformat.read(fname, as_version=4)
    html = []
    for cell in nb.cells:
        html.append(f"<h4>{cell.cell_type} cell</h4>")
        if cell.cell_type == "code":
            html.append(highlight(cell.source, lexer, formatter))
        else:
            html.append(f"<pre>{cell.source}</pre>")
    display(HTML("\n".join(html)))


show_notebook(os.path.join("mypackage/foo.ipynb"))
```

```
<IPython.core.display.HTML object>
```

2.1.7 foo.ipynb

[1]:
```
def bar():
    return "bar"
```

[2]:
```
def has_ip_syntax():
    listing = !ls
    return listing
```

[3]:
```
def whatsmyname():
    return __name__
```

2.1.8 Notebooks importieren

Um modularer entwickeln zu können, ist der Import von Notebooks erforderlich. Da Notebooks jedoch keine Python-Dateien sind, lassen sie sich auch nicht so einfach importieren. Glücklicherweise stellt Python einige Hooks für den Import bereit, sodass IPython-Notebooks schließlich doch importiert werden können.

[1]:
```
import os, sys, types
```

[2]:
```
import nbformat

from IPython import get_ipython
from IPython.core.interactiveshell import InteractiveShell
```

Import-Hooks haben normalerweise zwei Objekte:

- **Module Loader**, der einen Modulnamen (z.B. `IPython.display`) annimmt und ein Modul zurückgibt
- **Module Finder**, der herausfindet, ob ein Modul vorhanden ist, und Python mitteilt, welcher *Loader* verwendet werden soll

Zunächst jedoch schreiben wir eine Methode, die ein Notebook anhand des vollständig qualifizierten Namen und des optionalen Pfades findet. So wird z.B. aus `mypackage.foo` `mypackage/foo.ipynb` und ersetzt `Foo_Bar` durch `Foo Bar`, wenn `Foo_Bar` nicht existiert.

[3]:
```
def find_notebook(fullname, path=None):
    name = fullname.rsplit(".", 1)[-1]
    if not path:
        path = [""]
    for d in path:
        nb_path = os.path.join(d, name + ".ipynb")
        if os.path.isfile(nb_path):
            return nb_path
        # let import Foo_Bar find "Foo Bar.ipynb"
        nb_path = nb_path.replace("_", " ")
        if os.path.isfile(nb_path):
            return nb_path
```

Notebook Loader

Der Notebook Loader führt die folgenden drei Schritte aus:

1. Laden des Notebook-Dokuments in den Speicher
2. Erstellen eines leeren Moduls
3. Ausführen jeder Zelle im Modul-Namensraum

   Da IPython-Zellen eine erweiterte Syntax haben können, wird mit `transform_cell` jede Zelle in reinen Python-Code umgewandelt, bevor er ausgeführt wird.

[4]:
```
class NotebookLoader(object):
    """Module Loader for IPython Notebooks"""

    def __init__(self, path=None):
        self.shell = InteractiveShell.instance()
        self.path = path
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    def load_module(self, fullname):
        """import a notebook as a module"""
        path = find_notebook(fullname, self.path)

        print("importing notebook from %s" % path)

        # load the notebook object
        nb = nbformat.read(path, as_version=4)

        # create the module and add it to sys.modules
        # if name in sys.modules:
        #    return sys.modules[name]
        mod = types.ModuleType(fullname)
        mod.__file__ = path
        mod.__loader__ = self
        mod.__dict__["get_ipython"] = get_ipython
        sys.modules[fullname] = mod

        # extra work to ensure that magics that would affect the user_ns
        # magics that would affect the user_ns actually affect the
        # notebook module's ns
        save_user_ns = self.shell.user_ns
        self.shell.user_ns = mod.__dict__

        try:
            for cell in nb.cells:
                if cell.cell_type == "code":
                    # transform the input to executable Python
                    code = self.shell.input_transformer_manager.transform_cell(
                        cell.source
                    )
                    # run the code in the module
                    exec(code, mod.__dict__)
        finally:
            self.shell.user_ns = save_user_ns
        return mod
```

Notebook Finder

Der Finder ist ein einfaches Objekt, das angibt, ob ein Notebook anhand seines Dateinamens importiert werden kann, und das den entsprechenden Loader zurückgibt.

[5]:
```
class NotebookFinder(object):
    """Module Finder finds the transformed IPython Notebook"""

    def __init__(self):
        self.loaders = {}

    def find_module(self, fullname, path=None):
        nb_path = find_notebook(fullname, path)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        if not nb_path:
            return

        key = path
        if path:
            # lists aren't hashable
            key = os.path.sep.join(path)

        if key not in self.loaders:
            self.loaders[key] = NotebookLoader(path)
        return self.loaders[key]
```

Hook registrieren

Jetzt registrieren wir `NotebookFinder` mit `sys.meta_path`:

[6]:
```
sys.meta_path.append(NotebookFinder())
```

Überprüfen

Nun sollte unser Notebook *mypackage/foo.ipynb* importierbar sein mit

[7]:
```
from mypackage import foo
```

```
importing notebook from /Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/
↪ipython/mypackage/foo.ipynb
```

Wird die Python-Methode `bar` ausgeführt?

[8]:
```
foo.bar()
```

[8]:
```
'bar'
```

… und die IPython-Syntax?

[9]:
```
foo.has_ip_syntax()
```

[9]:
```
['debugging.ipynb',
 'display.ipynb',
 'examples.ipynb',
 'extensions.rst',
 'importing.ipynb',
 'index.rst',
 'magics.ipynb',
 '\x1b[34mmypackage\x1b[m\x1b[m',
 'myscript.py',
 'shell.ipynb',
 'start.rst',
 '\x1b[31mtab-completion-for-anything.png\x1b[m\x1b[m',
 '\x1b[31mtab-completion-for-modules.png\x1b[m\x1b[m',
 '\x1b[31mtab-completion-for-objects.png\x1b[m\x1b[m',
 '\x1b[34munix-shell\x1b[m\x1b[m']
```

Wiederverwendbarer Import-Hook

Der Import-Hook kann auch einfach in anderen Notebooks ausgeführt werden mit

```
[10]: %run importing.ipynb
```

```
importing notebook from /Users/veit/cusy/trn/Python4DataScience-de/docs/workspace/
↪ipython/mypackage/foo.ipynb
```

2.1.9 IPython-Erweiterungen

IPython-Erweterungen sind Python-Module, die das Verhalten der Shell ändern. Sie werden mit einem importierbaren Modulnamen bezeichnet und befinden sich üblicherweise in `.ipython/extensions/`.

Einige wichtige Erweiterungen sind bereits in IPython enthalten: `autoreload` und `storemagic`. Andere Erweiterungen findet ihr im Extensions Index oder auf PyPI unter dem IPython tag.

Siehe auch:

- IPython extensions docs

Erweiterungen verwenden

Die `%load_ext`-Magie kann verwendet werden um Erweiterungen zu laden während IPython ausgeführt wird, z.B. (zum Beispiel) :samp:` %load_ext {MYEXTENSION}`.

Alternativ kann eine Erweiterung auch bei jedem Start von IPython geladen werden, indem sie in der IPython-Konfigurationsdate aufgelistet wird, z.B. `c.InteractiveShellApp.extensions = ["MYEXTENSION"]`.

Falls ihr noch keine IPython-Konfigurationsdatei erstellt habt, könnt ihr dies mit `$ ipython profile create [PROFILENAME]`,

Falls kein Profilname angegeben wird, wird `default` verwendet. Üblicherweise wird die Datei dann in `~/.ipython/profile_default/` erstellt und je nach Verwendungszweck benannt: `ipython_config.py` wird für alle IPython-Befehle verwendet, während `ipython_notebook_config.py` nur für Befehle in IPython-Notebooks Verwendung findet.

IPython-Erweiterungen schreiben

Eine IPython-Erweiterung ist ein importierbares Python-Modul, das über spezielle Funktionen zum Laden und Entladen verfügt:

```
def load_ipython_extension(ipython):
    # The `ipython` argument is the currently active `InteractiveShell`
    # instance, which can be used in any way. This allows you to register
    # new magics or aliases, for example.

def unload_ipython_extension(ipython):
    # If you want your extension to be unloadable, put that logic here.
```

Siehe auch:

- `Defining custom magics`

2.1.10 Fehleranalyse

IPython enthält verschiedene Werkzeuge, um fehlerhaften Code zu analysieren, im Wesentlichen das *Exception Reporting* und den Debugger.

Exceptions kontrollieren mit `%xmode`

Wenn die Ausführung eines Python-Skripts fehlschlägt, wird meist eine sog. *Exception* ausgelöst und relevante Informationen zur Fehlerursache in einen *Traceback* geschrieben. Mit der `%xmode`-Magic-Funktion könnt ihr in IPython die Menge der Informationen steuern, die euch angezeigt werden. Betrachten wir hierfür den folgenden Code:

[1]:
```
def func1(a, b):
    return a / b


def func2(x):
    a = x
    b = x - 1
    return func1(a, b)
```

[2]:
```
func2(1)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[2], line 1
----> 1 func2(1)

Cell In[1], line 8, in func2(x)
      6 a = x
      7 b = x - 1
----> 8 return func1(a, b)

Cell In[1], line 2, in func1(a, b)
      1 def func1(a, b):
----> 2     return a / b

ZeroDivisionError: division by zero
```

Der Aufruf von `func2`führt zu einem Fehler, und der `Traceback` zeigt genau, was passiert ist: in jeder Zeile wird der Kontext jedes Schritts angezeigt, der schließlich zum Fehler geführt hat. Mit der `%xmode`-Magic-Funktion (kurz für *Exception-Modus*) können wir steuern, welche Informationen uns angezeigt werden sollen.

`%xmode` nimmt ein einziges Argument, den Modus, und es gibt drei Möglichkeiten: * `Plain` * `Context` * `Verbose`

Die Standardeinstellung ist `Context` und gibt eine Ausgabe wie die obige aus. `Plain` ist kompakter und liefert weniger Informationen:

[3]:
```
%xmode Plain
func2(1)
```

```
Exception reporting mode: Plain
```

```
Traceback (most recent call last):

  Cell In[3], line 2
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    func2(1)

  Cell In[1], line 8 in func2
    return func1(a, b)
  Cell In[1], line 2 in func1
    return a / b

ZeroDivisionError: division by zero
```

Der `Verbose`-Modus zeigt einige zusätzliche Informationen an, einschließlich der Argumente für alle aufgerufenen Funktionen:

```
[4]: %xmode Verbose
     func2(1)
```

```
Exception reporting mode: Verbose
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
Cell In[4], line 2
      1 get_ipython().run_line_magic('xmode', 'Verbose')
----> 2 func2(1)

Cell In[1], line 8, in func2(x=1)
      6 a = x
      7 b = x - 1
        a = 1
----> 8 return func1(a, b)
        b = 0

Cell In[1], line 2, in func1(a=1, b=0)
      1 def func1(a, b):
        a = 1
----> 2     return a / b
        b = 0

ZeroDivisionError: division by zero
```

Diese zusätzlichen Informationen können helfen, den Grund für die *Exception* einzugrenzen. Umgekehrt kann der `Verbose`-Modus jedoch bei komplexem Code zu extrem langen Tracebacks führen, bei denen kaum noch die wesentlichen Stellen erkannt werden können.

Debugging

Wenn durch das Lesen eines `Traceback` ein Fehler nicht gefunden werden kann, hilft Debugging weiter. Der Python-Standard für interaktives Debugging ist der Python-Debugger `pdb`. Mit ihm könnt ihr euch zeilenweise durch den Code navigieren, um festzustellen, was möglicherweise einen Fehler verursacht. Die erweiterte Version für IPython ist `ipdb`.

In IPython ist der `%debug`-Magic-Befehl möglicherweise die bequemste Art zum Debugging. Wenn ihr ihn aufruft, nachdem eine Exception ausgegeben wurde, wird automatisch ein interaktiver Debug-Prompt während der Exception geöffnet. Mit der `ipdb`-Eingabeaufforderung könnt ih den aktuellen Status des Stacks untersuchen, die verfügbaren Variablen untersuchen und sogar Python-Befehle ausführen.

Schauen wir uns die letzte Exception an und führen dann einige grundlegende Aufgaben aus:

[5]:
```
%debug
```

```
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_80098/3792871231.
↪py(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
      4
      5 def func2(x):

ipdb> s
```

Der interaktive Debugger bietet jedoch viel mehr – wir können im Stack auch auf und ab gehen und die Werte von Variablen untersuchen:

[6]:
```
%debug
```

```
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_80098/3792871231.
↪py(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
      4
      5 def func2(x):

ipdb> up
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_80098/3792871231.
↪py(8)func2()
      4
      5 def func2(x):
      6     a = x
      7     b = x - 1
----> 8     return func1(a, b)

ipdb> print(x)
1
ipdb> up
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_80098/1541833627.py(2)
↪<module>()
      1 get_ipython().run_line_magic('xmode', 'Verbose')
----> 2 func2(1)

ipdb> down
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_80098/3792871231.
↪py(8)func2()
      4
      5 def func2(x):
      6     a = x
      7     b = x - 1
----> 8     return func1(a, b)

ipdb> quit
```

Dies vereinfacht die Suche nach den Funktionsaufrufen, die zum Fehler geführt haben, enorm.

Wenn der Debugger automatisch gestartet werden soll, wenn eine Ausnahme ausgelöst wird, könnt ihr die %pdb-Magic-Funktion verwenden, um dieses Verhalten zu aktivieren:

[7]:
```
%xmode Plain
%pdb on
func2(1)
```

```
Exception reporting mode: Plain
Automatic pdb calling has been turned ON
```

```
Traceback (most recent call last):

  Cell In[7], line 3
    func2(1)

  Cell In[1], line 8 in func2
    return func1(a, b)
  Cell In[1], line 2 in func1
    return a / b

ZeroDivisionError: division by zero
```

```
> /var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_80098/3792871231.
↪py(2)func1()
      1 def func1(a, b):
----> 2     return a / b
      3
      4
      5 def func2(x):

ipdb> print(b)
0
ipdb> quit
```

Wenn ihr ein Skript habt, das ihr von Anfang an im interaktiven Modus ausführen möchtet, so könnt ihr dies mit dem Befehl `%run -d`.

Wesentliche Befehle des `ipdb`

Befehl	Beschreibung
`list`	Zeige den aktuellen Ort in der Datei
`h(elp)`	Liste der Befehle anzeigen oder Hilfe zu einem bestimmten Befehl suchen
`q(uit)`	Beendet den Debugger und das Programm
`c(ontinue)`	Den Debugger beenden, im Programm fortfahren
`n(ext)`	Gehe zum nächsten Schritt des Programms
`<enter>`	Wiederhole den vorherigen Befehl
`p(rint)`	Druckvariablen
`s(tep)`	Schritt in eine Unterroutine
`r(eturn)`	Rückkehr aus einem Unterprogramm

Weitere Informationen zum IPython-Debugger erhaltet ihr unter ipdb.

2.2 Jupyter

wir haben das Jupyter-Kapitel verschoben in ein eigenes Tutorial: Jupyter-Tutorial.

2.3 NumPy

NumPy ist die Abkürzung für numerisches Python. Viele Python-Pakete, die wissenschaftliche Funktionen bereitstellen, verwenden die Array-Objekte von NumPy als eine der Standardschnittstellen für den Datenaustausch. Im folgenden möchte ich einen kurzen Überblick über den wesentlichen Funktionsumfang von NumPy geben:

- *ndarray*, ein effizientes mehrdimensionales Array, das schnelle Array-basierte Operationen bietet, wie das Mischen und Bereinigen von Daten, Untergruppenbildung und Filterung, Transformation und alle anderen Arten von Berechnungen. Zudem gibt es flexible Funktionen für das Broadcasting, also von Auswertungen unterschiedlich großer Arrays.
- Mathematische Funktionen für schnelle Operationen auf ganzen Arrays von Daten, wie Sortieren, Eindeutigkeit und Mengenoperationen. Dabei werden die Ausdrücke, anstelle von Schleifen mit `if-elif-else`-Verzweigungen, in bedingter Logik geschrieben.
- Werkzeuge zum Lesen und Schreiben von Array-Daten auf die Festplatte und zur Arbeit mit Memory-Mapped-Dateien.
- Funktionen für Lineare Algebra, Zufallszahlengenerierung und Fourier-Transformation.
- Eine C-API für die Verbindung von NumPy mit Bibliotheken, die in C, C++ oder FORTRAN geschrieben sind.

Bemerkung: Dieser Abschnitt führt euch in die Grundlagen der Verwendung von NumPy-Arrays ein und sollte ausreichen, um dem Rest des Tutorials zu folgen. Für viele datenanalytische Anwendungen ist es zwar nicht notwendig, ein tiefes Verständnis von NumPy zu haben, aber die Beherrschung der array-orientierten Programmierung und Denkweise ist ein wichtiger Schritt auf dem Weg zum Data-Scientist.

Siehe auch:

- Home
- Docs
- GitHub
- Tutorials

2.3.1 Einführung in NumPy

NumPy-Operationen führen komplexe Berechnungen auf ganzen Arrays durch, ohne dass Python `for`-Schleifen erforderlich sind, die bei großen Sequenzen langsam sein können. Die Schnelligkeit von NumPy erklärt sich aus den C-basierten Algorithmen, die den Overhead des Python-Codes vermeidet. Um euch einen Eindruck zu vermitteln vom Performance-Unterschied, messen wir den Unterschied zwischen einem NumPy-Array un einer Python-Liste mit hunderttausend Ganzzahlen:

[1]:
```
import numpy as np
```

[2]:
```
myarray = np.arange(100000)
mylist = list(range(100000))
```

[3]:
```
%time for _ in range(10): myarray2 = myarray ** 2
```

```
CPU times: user 0 ns, sys: 2.55 ms, total: 2.55 ms
Wall time: 2.28 ms
```

[4]:
```
%time for _ in range(10): mylist2 = [x ** 2 for x in mylist]
```

```
CPU times: user 390 ms, sys: 25.5 ms, total: 416 ms
Wall time: 423 ms
```

2.3.2 ndarray – ein N-dimensionales Array-Objekt

ndarray erlaubt mathematische Operationen auf ganzen Datenblöcken und verwendet dabei eine ähnliche Syntax wie bei ähnlichen Operationen zwischen skalaren Elementen. In NumPy gibt viele verschiedene Typen zur Beschreibung von Skalaren, die größtenteils auf Typen aus der Sprache C basieren und denjenigen, die mit Python kompatibel sind.

Siehe auch

- Array Scalars

Bemerkung

Wann immer in diesem Tutorial von *Array*, *NumPy-Array* oder `ndarray` geredet wird, bezieht sich dies in den meisten Fällen auf das `ndarray`-Objekt.

[1]:
```
import numpy as np
```

[2]:
```
py_list = [2020, 2021, 20222]
array_1d = np.array(py_list)
```

[3]:
```
array_1d
```

[3]:
```
array([ 2020,  2021, 20222])
```

Verschachtelte Sequenzen, wie eine Liste von Listen gleicher Länge, können in ein mehrdimensionales Array umgewandelt werden:

[4]:
```
list_of_lists = [[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]
array_2d = np.array(list_of_lists)
```

[5]:
```
array_2d
```

[5]:
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

Da `list_of_lists` eine Liste mit drei Listen war, hat das NumPy-Array `array_2d` zwei Dimensionen, deren Form aus den Daten abgeleitet wird. Mit den Attributen ndim und shape können wir uns die Anzahl der Dimensionen und den Umriss von `array_2d` ausgeben lassen:

[6]:
```
array_2d.ndim
```

[6]:
```
2
```

```
[7]: array_2d.shape
```

```
[7]: (3, 4)
```

Um euch einen Eindruck von der Syntax zu vermitteln, erstelle ich zunächst ein Array aus Zufallszahlen mit fünf Spalten und sieben `Slices` (engl.: Scheiben):

```
[8]: data = np.random.randn(7, 3)
```

`ndarray` ist ein generischer mehrdimensionaler Container. Jedes Array hat eine Form, ein Tupel, das die Größe der einzelnen Dimensionen angibt. Mit `shape` kann ich mir die Anzahl der Zeilen und Spalten eines Arrays ausgeben lassen:

Zusätzlich zu `np.array` gibt es eine Reihe weiterer Funktionen zur Erstellung neuer Arrays. zeros und ones erzeugen beispielsweise Arrays aus Nullen bzw. Einsen mit einer bestimmten Länge oder Form. empty erzeugt ein Array, ohne dessen Werte auf einen bestimmten Wert zu initialisieren. Um ein höherdimensionales Array mit diesen Methoden zu erstellen, übergebt ein Tupel für die Form:

```
[9]: np.zeros(4)
```

```
[9]: array([0., 0., 0., 0.])
```

```
[10]: np.ones((3,4))
```

```
[10]: array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

```
[11]: np.empty((2,3,4))
```

```
[11]: array([[[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]],

       [[0., 0., 0., 0.],
        [0., 0., 0., 0.],
        [0., 0., 0., 0.]]])
```

Bemerkung

Ihr dürft nich sicher annehmen, dass die die Funktion `np.empty` ein Array mit lauter Nullen zurückgibt, da sie uninitialisierten Speicher zurückgibt und kann daher auch *garbage*-Werte enthalten kann.

arange ist eine array-bewertete Version der Built-in Python range-Funktion:

```
[12]: np.arange(4)
```

```
[12]: array([0, 1, 2, 3])
```

Weitere NumPy-Standardfunktionen zur Erstellung von Arrays sind:

Funktion	Beschreibung
`array`	konvertiert Eingabedaten (Liste, Tupel, Array oder andere Sequenztypen) in ein `ndarray`, indem entweder ein `dtype` abgeleitet oder explizit ein `dtype` angegeben wird; kopiert standardmäßig die Eingabedaten in das Array
`asarray`	konvertiert die Eingabe in ein `ndarray`, kopiert aber nicht, wenn die Eingabe bereits ein `ndarray` ist
`arange`	wie Python built-in `range`, gibt aber ein `ndarray` statt einer Liste zurück
`ones`, `ones_lik`	`ones` erzeugt ein Array mit 1en in der gegebenen Form und dem gegebenen `dtype`; `ones_like` nimmt ein anderes Array und erzeugt ein `ones`-Array in der gleichen Form und dem gleichen `dtype`
`zeros`, `zeros_li`	wie `ones` und `ones_like`, erzeugt aber stattdessen Arrays mit 0en
`empty`, `empty_li`	erzeugt neue Arrays durch Zuweisung neuen Speichers, füllt sie aber nicht mit Werten wie `ones` und `zeros`
`full`, `full_lik`	erzeugt ein Array der angegebenen `shape` und des angegebenen `dtype`, wobei alle Werte auf den angegebenen *Füllwert* gesetzt werden; `full_like` nimmt ein anderes Array und erzeugt ein gefülltes Array mit denselben `shape` und `dtype`
`eye`, `identity`	erzeugt eine quadratische N × N-Identitätsmatrix (1en auf der Diagonale und 0en anderswo)

2.3.3 dtype

Der `ndarray` ist ein Container für homogene Daten, d.h. alle Elemente müssen vom gleichen Typ sein. Jedes Array hat einen `dtype`, ein Objekt, das den Datentyp des Arrays beschreibt:

[8]:
```
dt = data.dtype
dt
```

[8]:
```
dtype('float64')
```

NumPy-Datentypen:

Typ	Typ-Code	Beschreibung
int8, uint8	i1, u1	Vorzeichenbehaftete und vorzeichenlose 8-Bit (1 Byte) Ganzzahltypen
int16, uint16	i2, u2	Vorzeichenbehaftete und vorzeichenlose 16-Bit (2 Byte) Ganzzahltypen
int32, uint32	i4, u4	Vorzeichenbehaftete und vorzeichenlose 32-Bit (4 Byte) Ganzzahltypen
int64, uint64	i8, u8	Vorzeichenbehaftete und vorzeichenlose 64-Bit (8 Byte) Ganzzahltypen
float16	f2	Standard-Gleitkomma mit halber Genauigkeit
float32	f4 oder f	Standard-Gleitkomma mit einfacher Genauigkeit; kompatibel mit C float
float64	f8 oder d	Standard-Gleitkomma mit doppelter Genauigkeit; kompatibel mit C double und Python float-Objekt
complex64, complex128, complex256	c8, c16, c32	Komplexe Zahlen, die durch zwei 32-, 64- bzw. 128-Gleitkommazahlen dargestellt werden
bool	?	Boolescher Typ, der die Werte True und False speichert
object	O	Python-Objekttyp; ein Wert kann ein beliebiges Python-Objekt sein
string_	S	ASCII-Stringtyp mit fester Länge (1 Byte pro Zeichen); um z.B. einen Stringtyp mit der Länge 7 zu erstellen, verwendet S7; längere Eingaben werden ohne Warnung abgeschnitten
unicode_	U	Unicode-Typ mit fester Länge wobei die Anzahl der Bytes plattformspezifisch ist; verwendet dieselbe Spezifikationssemantik wie string_, z.B. U7

Anzahl der Elemente mit itemsize ermitteln:

```
[9]: dt.itemsize
```

```
[9]: 8
```

Name des Datentypes ermitteln:

```
[10]: dt.name
```

```
[10]: 'float64'
```

Datentyp überprüfen:

```
[11]: dt.type is np.float64
```

```
[11]: True
```

Datentyp ändern mit astype:

```
[12]: data_float32 = data.astype(np.float32)
      data_float32
```

```
[12]: array([[ 0.33215025,  0.9821482 ,  0.14965022],
             [-0.5039629 ,  0.79987854, -0.55183125],
             [-0.9200971 , -0.746871  ,  0.37547055],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
       [-1.1687789 ,  1.6087633 , -1.6145438 ],
       [ 0.14729111,  0.4293929 , -0.11391696],
       [-0.9159697 , -0.6969758 , -0.36380735],
       [-0.34818023, -0.9103621 , -0.47645596]], dtype=float32)
```

2.3.4 Arithmetik

Arrays ermöglichen euch, Stapeloperationen auf Daten durchzuführen ohne `for`-Schleifen verwenden zu müssen. Dies wird bei NumPy *Vektorisierung* genannt. Bei allen arithmetischen Operationen zwischen Arrays gleicher Größe wird die Operation elementweise angewendet:

[1]:
```
import numpy as np


data = np.random.randn(7, 3)
data
```

[1]:
```
array([[ 0.07282305, -0.5454217 ,  0.07199181],
       [ 2.16112354,  0.30496674,  0.73963605],
       [-1.61359344, -0.19812259, -0.76049151],
       [-0.86957267, -0.5841333 , -0.15678665],
       [-0.69817046,  1.86730036, -1.15614376],
       [ 0.85112655, -0.61805251, -0.63671618],
       [-1.78018583,  1.17915059, -0.6853148 ]])
```

[2]:
```
1 / data
```

[2]:
```
array([[13.73191506, -1.83344373, 13.89046966],
       [ 0.46272228,  3.27904615,  1.3520163 ],
       [-0.61973479, -5.04737999, -1.31493907],
       [-1.14999015, -1.71193802, -6.37809388],
       [-1.43231497,  0.53553248, -0.86494434],
       [ 1.17491341, -1.61798549, -1.57055848],
       [-0.56173911,  0.8480681 , -1.45918342]])
```

[3]:
```
data**2
```

[3]:
```
array([[0.0053032 , 0.29748483, 0.00518282],
       [4.67045494, 0.09300471, 0.54706149],
       [2.6036838 , 0.03925256, 0.57834734],
       [0.75615662, 0.34121171, 0.02458205],
       [0.48744199, 3.48681063, 1.33666839],
       [0.7244164 , 0.38198891, 0.4054075 ],
       [3.16906158, 1.39039611, 0.46965638]])
```

Vergleich zweier Arrays:

[4]:
```
data2 = np.random.randn(7, 3)
data > data2
```

[4]:
```
array([[ True,  True,  True],
       [ True,  True,  True],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
       [False, False, False],
       [False, False, False],
       [False,  True, False],
       [ True, False, False],
       [False, False,  True]])
```

2.3.5 Indizierung und Slicing

Indizierung ist die Auswahl einer Teilmenge eurer Daten oder einzelner Elemente. In eindimensionalen Arrays ist das sehr einfach; verhalten sie sich doch ähnlich wie Python-Listen:

[1]:
```
import numpy as np
```

[2]:
```
data = np.random.randn(7, 3)
data
```

[2]:
```
array([[ 0.33050115, -0.24519424,  0.66232884],
       [ 0.22530574,  0.41200361,  1.21487789],
       [ 0.20448321,  0.91945497,  1.51575382],
       [ 1.28360113, -0.14962714,  0.22448488],
       [ 0.04452226,  0.88346213,  1.75586787],
       [-0.48891048, -1.73687276, -2.47068694],
       [ 0.96051183,  2.09280516,  0.28880627]])
```

[3]:
```
data[4]
```

[3]:
```
array([0.04452226, 0.88346213, 1.75586787])
```

[4]:
```
data[2:4]
```

[4]:
```
array([[ 0.20448321,  0.91945497,  1.51575382],
       [ 1.28360113, -0.14962714,  0.22448488]])
```

[5]:
```
data[2:4] = np.random.randn(2, 3)
```

[6]:
```
data
```

[6]:
```
array([[ 0.33050115, -0.24519424,  0.66232884],
       [ 0.22530574,  0.41200361,  1.21487789],
       [ 1.46155705, -1.42462688, -0.64299548],
       [ 0.37948229,  1.29346366, -0.84481165],
       [ 0.04452226,  0.88346213,  1.75586787],
       [-0.48891048, -1.73687276, -2.47068694],
       [ 0.96051183,  2.09280516,  0.28880627]])
```

Bemerkung

Array-Slices unterscheiden sich von Python-Listen dadurch, dass sie Sichten (*views*) auf das ursprüngliche Array sind. Das bedeutet, dass die Daten nicht kopiert werden und dass alle Änderungen an der View im Ausgangsarray wiedergegeben werden.

Wenn ihr eine Kopie eines Teils eines `ndarray` erstellen wollt, könnt ihr das Array explizit kopieren – z.B. mit `data[2:5].copy()`.

Slicing in dieser Weise führt immer zu Array-Ansichten mit der gleichen Anzahl von Dimensionen. Wenn ihr jedoch ganzzahlige Indizes und Slices mischt, erhaltet ihr Slices mit niedrigeren Dimensionen. So können wir z.B. die zweite Zeile, aber nur die ersten beiden Spalten wie folgt auswählen:

[7]:
```
data[1, :2]
```

[7]:
```
array([0.22530574, 0.41200361])
```

Ein Doppelpunkt bedeutet, dass die gesamte Achse genommen wird, so dass ihr auch höherdimensionale Achsen auswählen könnt:

[8]:
```
data[:, :1]
```

[8]:
```
array([[ 0.33050115],
       [ 0.22530574],
       [ 1.46155705],
       [ 0.37948229],
       [ 0.04452226],
       [-0.48891048],
       [ 0.96051183]])
```

Boolesche Indizierung

Betrachten wir ein Beispiel, bei dem wir einige Daten in einem Array und ein Array von Namen mit Duplikaten haben. Ich werde hier die Funktion `randn` in `numpy.random` verwenden, um einige zufällige normalverteilte Daten zu erzeugen:

[9]:
```
names = np.array(
    [
        "Liam",
        "Olivia",
        "Noah",
        "Liam",
        "Noah",
        "Olivia",
        "Liam",
        "Emma",
        "Oliver",
        "Ava",
    ]
)
data = np.random.randn(10, 4)
```

[10]:
```
names
```

[10]:
```
array(['Liam', 'Olivia', 'Noah', 'Liam', 'Noah', 'Olivia', 'Liam', 'Emma',
       'Oliver', 'Ava'], dtype='<U6')
```

[11]:
```
data
```

[11]:
```
array([[-0.1353117 ,  0.81134636,  1.27451527,  1.05352755],
       [-1.13514458,  0.54537619,  1.82156905, -1.01267531],
       [ 1.09013813,  0.87177598, -0.62723958,  0.40875051],
       [ 0.2705055 , -1.95543273,  1.69239456, -1.63625339],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
       [ 0.51631929,  1.05609607,  0.83749348,  0.03833032],
       [-1.15009072, -0.24253528, -0.63441305,  0.05945932],
       [-0.02354212,  0.97168249,  0.81879396, -1.45633432],
       [-0.0938235 , -0.76430898,  1.58021321,  0.84109544],
       [-1.06054123, -0.29494841, -1.71638201,  1.25419797],
       [ 1.04045002,  1.04053515, -1.26912014,  1.04517256]])
```

Angenommen, jeder Name entspricht einer Zeile im Datenarray und wir wollen alle Zeilen mit dem entsprechenden Namen *Liam* auswählen. Wie arithmetische Operationen werden auch Vergleiche wie == mit Arrays vektorisiert. Der Vergleich von Namen mit der Zeichenkette `Liam` ergibt also ein boolesches Array:

```
[12]: names == "Liam"
```

```
[12]: array([ True, False, False,  True, False, False,  True, False, False,
       False])
```

Dieses boolesche Array kann beim Indizieren des Arrays übergeben werden:

```
[13]: data[names == "Liam"]
```

```
[13]: array([[-0.1353117 ,  0.81134636,  1.27451527,  1.05352755],
       [ 0.2705055 , -1.95543273,  1.69239456, -1.63625339],
       [-0.02354212,  0.97168249,  0.81879396, -1.45633432]])
```

Dabei muss das boolesche Array die gleiche Länge haben wie die Arrayachse, die es indiziert.

Bemerkung

Die Auswahl von Daten aus einem Array durch boolesche Indizierung und die Zuweisung des Ergebnisses an eine neue Variable erzeugt immer eine Kopie der Daten, selbst wenn das zurückgegebene Array unverändert ist.

Im folgenden Beispiel wähle ich die Zeilen aus, in denen `Namen == "Liam"` ist und indiziere auch die Spalten:

```
[14]: data[names == "Liam", 2:]
```

```
[14]: array([[ 1.27451527,  1.05352755],
       [ 1.69239456, -1.63625339],
       [ 0.81879396, -1.45633432]])
```

Um alles außer *Liam* auszuwählen, könnt ihr entweder `!=` verwenden oder die Bedingung mit `~` negieren:

```
[15]: names != "Liam"
```

```
[15]: array([False,  True,  True, False,  True,  True, False,  True,  True,
        True])
```

```
[16]: cond = names == "Liam"
data[~cond]
```

```
[16]: array([[-1.13514458,  0.54537619,  1.82156905, -1.01267531],
       [ 1.09013813,  0.87177598, -0.62723958,  0.40875051],
       [ 0.51631929,  1.05609607,  0.83749348,  0.03833032],
       [-1.15009072, -0.24253528, -0.63441305,  0.05945932],
       [-0.0938235 , -0.76430898,  1.58021321,  0.84109544],
       [-1.06054123, -0.29494841, -1.71638201,  1.25419797],
       [ 1.04045002,  1.04053515, -1.26912014,  1.04517256]])
```

Wenn ihr zwei der drei Namen auswählt, um mehrere boolesche Bedingungen zu kombinieren, könnt ihr die booleschen arithmetischen Operatoren & (und) und | (oder) verwenden.

Warnung

Die Python-Schlüsselwörter `and` und `or` funktionieren nicht mit booleschen Arrays.

[17]:
```
mask = (names == "Liam") | (names == "Olivia")
```

[18]:
```
mask
```

[18]:
```
array([ True,  True, False,  True, False,  True,  True, False, False,
       False])
```

[19]:
```
data[mask]
```

[19]:
```
array([[-0.1353117 ,  0.81134636,  1.27451527,  1.05352755],
       [-1.13514458,  0.54537619,  1.82156905, -1.01267531],
       [ 0.2705055 , -1.95543273,  1.69239456, -1.63625339],
       [-1.15009072, -0.24253528, -0.63441305,  0.05945932],
       [-0.02354212,  0.97168249,  0.81879396, -1.45633432]])
```

Ganzzahlige Array-Indizierung

Ganzzahlige Array-Indizierung ermöglicht die Auswahl beliebiger Elemente im Array auf der Grundlage eures N-dimensionalen Index. Jedes Integer-Array repräsentiert eine Anzahl von Indizes in dieser Dimension.

Siehe auch

- Integer array indexing

2.3.6 Transponieren von Arrays und Vertauschen von Achsen

Transponieren ist eine spezielle Form der Umformung, die ebenfalls eine Sicht auf die zugrunde liegenden Daten liefert, ohne etwas zu kopieren. Arrays haben die Transpose-Methode und auch das spezielle T-Attribut:

[1]:
```
import numpy as np
```

[2]:
```
data = np.arange(16)
```

[3]:
```
data
```

[3]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15])
```

[4]:
```
reshaped_data = data.reshape((4, 4))
```

[5]:
```
reshaped_data
```

[5]:
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15]])
```

```
[6]: reshaped_data.T
```

```
[6]: array([[ 0,  4,  8, 12],
       [ 1,  5,  9, 13],
       [ 2,  6, 10, 14],
       [ 3,  7, 11, 15]])
```

numpy.dot gibt das Skalarprodukt zweier Arrays zurück, z.B.:

```
[7]: np.dot(reshaped_data.T, reshaped_data)
```

```
[7]: array([[224, 248, 272, 296],
       [248, 276, 304, 332],
       [272, 304, 336, 368],
       [296, 332, 368, 404]])
```

Der `@` Infix-Operator ist eine weitere Möglichkeit, eine Matrixmultiplikation durchzuführen. Er implementiert die Semantik des `@`-Operators, der mit PEP 465 in Python 3.5 eingeführt wurde und ist eine Abkürzung von np.matmul.

```
[8]: data.T @ data
```

```
[8]: 1240
```

Bei höherdimensionalen Arrays akzeptiert transpose ein Tupel von Achsennummern, um die Achsen zu vertauschen:

```
[9]: array_3d = np.arange(16).reshape((2, 2, 4))
```

```
[10]: array_3d
```

```
[10]: array([[[ 0,  1,  2,  3],
        [ 4,  5,  6,  7]],

       [[ 8,  9, 10, 11],
        [12, 13, 14, 15]]])
```

```
[11]: array_3d.transpose((1, 0, 2))
```

```
[11]: array([[[ 0,  1,  2,  3],
        [ 8,  9, 10, 11]],

       [[ 4,  5,  6,  7],
        [12, 13, 14, 15]]])
```

Hier wurden die Achsen neu geordnet, wobei die zweite Achse an erster Stelle steht, die erste Achse an zweiter Stelle und die letzte Achse unverändert bleibt.

`ndarray` verfügt auch über eine Methode swapaxes, die ein Paar von Achsennummern nimmt und die angegebenen Achsen vertauscht, um die Daten neu anzuordnen:

```
[12]: array_3d.swapaxes(1, 2)
```

```
[12]: array([[[ 0,  4],
        [ 1,  5],
        [ 2,  6],
        [ 3,  7]],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
       [[ 8, 12],
        [ 9, 13],
        [10, 14],
        [11, 15]]])
```

2.3.7 Universelle Funktionen (ufunc)

Eine universelle Funktion, oder `ufunc`, ist eine Funktion, die elementweise Operationen auf Daten in `ndarrays` durchführt. Man kann sie sich als schnelle vektorisierte Wrapper für einfache Funktionen vorstellen, die einen oder mehrere skalare Werte annehmen und ein oder mehrere skalare Ergebnisse erzeugen.

Viele `ufuncs` sind einfache elementweise Transformationen, wie sqrt oder exp:

[1]:
```
import numpy as np


data = np.arange(10)
```

[2]:
```
data
```

[2]:
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

[3]:
```
np.sqrt(data)
```

[3]:
```
array([0.        , 1.        , 1.41421356, 1.73205081, 2.        ,
       2.23606798, 2.44948974, 2.64575131, 2.82842712, 3.        ])
```

[4]:
```
np.exp(data)
```

[4]:
```
array([1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01,
       5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03,
       2.98095799e+03, 8.10308393e+03])
```

Diese werden als einstellige ufuncs bezeichnet. Andere, wie add oder maximum, nehmen zwei Arrays (also binäre ufuncs) und geben ein einziges Array als Ergebnis zurück:

[5]:
```
x = np.random.randn(8)
```

[6]:
```
y = np.random.randn(8)
```

[7]:
```
x
```

[7]:
```
array([ 0.33223956,  0.41802595,  0.67933838,  2.49505346, -0.99815963,
       -0.26031943,  0.66244814,  0.74554742])
```

[8]:
```
y
```

[8]:
```
array([ 1.45323312, -0.09441452,  0.87310632,  0.8724353 , -0.20123837,
       -0.3807043 ,  2.70549913, -2.02891263])
```

[9]:
```
np.maximum(x, y)
```

```
[9]: array([ 1.45323312,  0.41802595,  0.87310632,  2.49505346, -0.20123837,
        -0.26031943,  2.70549913,  0.74554742])
```

Hier berechnete `numpy.maximum` das elementweise Maximum der Elemente in `x` und `y`.

Manche `ufunc`, wie z.B. modf , eine vektorisierte Version des eingebauten Python divmod, geben mehrere Arrays zurückgeben: die Bruch- und Integralteile eines Gleitkomma-Arrays:

```
[10]: data = x * 5
```

```
[11]: data
```

```
[11]: array([ 1.66119778,  2.09012976,  3.39669191, 12.47526728, -4.99079815,
        -1.30159713,  3.31224069,  3.7277371 ])
```

```
[12]: remainder, whole_part = np.modf(x)
```

```
[13]: remainder
```

```
[13]: array([ 0.33223956,  0.41802595,  0.67933838,  0.49505346, -0.99815963,
        -0.26031943,  0.66244814,  0.74554742])
```

```
[14]: whole_part
```

```
[14]: array([ 0.,  0.,  0.,  2., -0., -0.,  0.,  0.])
```

Ufuncs akzeptieren ein optionales `out`-Argument, das es euch erlaubt, eure Ergebnisse an ein bestehendes Array zu übertragen, anstatt ein neues zu erstellen:

```
[15]: out = np.zeros_like(data)
```

```
[16]: np.add(data, 1)
```

```
[16]: array([ 2.66119778,  3.09012976,  4.39669191, 13.47526728, -3.99079815,
        -0.30159713,  4.31224069,  4.7277371 ])
```

```
[17]: np.add(data, 1, out=out)
```

```
[17]: array([ 2.66119778,  3.09012976,  4.39669191, 13.47526728, -3.99079815,
        -0.30159713,  4.31224069,  4.7277371 ])
```

```
[18]: out
```

```
[18]: array([ 2.66119778,  3.09012976,  4.39669191, 13.47526728, -3.99079815,
        -0.30159713,  4.31224069,  4.7277371 ])
```

Einige einstellige ufuncs:

Funktion	Beschreibung
`abs`, `fabs`	berechnet den absoluten Wert elementweise für Ganzzahl-, Gleitkomma- oder komplexe Werte
`sqrt`	berechnet die Quadratwurzel aus jedem Element (entspricht `data ** 0,5`)
`square`	berechnet das Quadrat eines jeden Elements (entspricht `data ** 2`)
`exp`	berechnet den Exponenten ex eines jeden Elements
`log`, `log10`, `log2`, `log1p`	Natürlicher Logarithmus (Basis e), log Basis 10, log Basis 2 bzw. log(1 + x)
`sign`	berechnet das Vorzeichen jedes Elements: `1` (positiv), `0` (Null), oder `-1` (negativ)
`ceil`	berechnet die Obergrenze jedes Elements (d.h. die kleinste ganze Zahl, die größer oder gleich dieser Zahl ist)
`floor`	berechnet die Untergrenze jedes Elements (d.h. die größte ganze Zahl, die kleiner oder gleich jedem Element ist)
`rint`	rundet Elemente auf die nächste Ganzzahl, wobei der `dtype` erhalten bleibt
`modf`	gibt den gebrochenen und ganzzahligen Teile des Arrays als separate Arrays zurück
`isnan`	gibt ein boolesches Array zurück, das angibt, ob jeder Wert `NaN` (Not a Number) ist
`isfinite`, `isinf`	gibt ein boolesches Array zurück, das angibt, ob jedes Element endlich (non-`inf`, not-`NaN`) bzw. unendlich ist
`cos`, `cosh`, `sin`, `sinh`, `tan`, `tanh`	reguläre und hyperbolische trigonometrische Funktionen
`arccos`, `arccosh`, `arcsin`, `arcsinh`, `arctan`, `arctanh`	Inverse trigonometrische Funktionen
`logical_not`	berechnet den Wahrheitswert von `not x` elementweise (entspricht `~data`)

Einige binäre universelle Funktionen:

Funktion	Beschreibung
`add`	hinzufügen entsprechender Elemente in Arrays
`subtract`	subtrahiert Elemente im zweiten Array vom ersten Array
`multiply`	Array-Elemente multiplizieren
`divide`, `floor_divide`	Dividieren oder Abschneiden des Rests
`power`	erhöht Elemente im ersten Array auf die im zweiten Array angegebenen Potenzen
`maximum`, `fmax`	elementweises Maximum; `fmax` ignoriert `NaN`
`minimum`, `fmin`	elementweises Minimum; `fmin` ignoriert `NaN`
`mod`	Elementweiser Modulus (Rest der Division)
`copysign`	kopiert das Vorzeichen der Werte im zweiten Argument auf die Werte im ersten Argument
`greater`, `greater_equal`, `less`, `less_equal`, `equal`, `not_equal`	Elementweise Vergleiche durchführen, die ein boolesches Array ergeben (entspricht den Infix-Operatoren >, >=, <, <=, ==, !=)
`logical_and`	berechnet den elementweisen Wahrheitswert der logischen Operation AND (&).
`logical_or`	berechnet den elementweisen Wahrheitswert der logischen Operation OR (|).
`logical_xor`	berechnet den elementweisen Wahrheitswert der logischen Operation XOR (^).

Hinweis

Eine vollständige Übersicht über binäre universelle Funktionen findet ihr in Universal functions (ufunc).

2.3.8 Array-orientierte Programmierung – Vektorisierung

Die Verwendung von NumPy-Arrays ermöglicht euch, viele Arten von Datenverarbeitungsaufgaben als prägnante Array-Ausdrücke auszudrücken, die andernfalls das Schreiben von `for`-Schleifen erfordern würden. Diese Praxis, Schleifen durch Array-Ausdrücke zu ersetzen, wird auch *Vektorisierung* genannt. Im Allgemeinen sind vektorisierte Array-Operationen deutlich schneller als ihre reinen Python-Entsprechungen.

[1]:
```
import numpy as np
```

Zunächst erstellen wir ein NumPy-Array mit hunderttausend Ganzzahlen:

[2]:
```
myarray = np.arange(100000)
```

Anschließend quadrieren wir alle Elemente in diesem Array mit numpy.square:

[3]:
```
%time np.square(myarray)
```

```
CPU times: user 513 µs, sys: 2.92 ms, total: 3.44 ms
Wall time: 342 µs
```

[3]:
```
array([         0,          1,          4, ..., 9999400009, 9999600004,
       9999800001])
```

Zum Vergleich messen wir nun die Zeit der quadratischen Funktion von Python:

[4]:
```
%time for _ in range(10): myarray2 = myarray**2
```

```
CPU times: user 1.77 ms, sys: 9.21 ms, total: 11 ms
Wall time: 1.03 ms
```

Und schließlich vergleichen wir die Zeit noch mit der Berechnung der quadratischen Funktion aller Werte einer Python-Liste:

[5]:
```
mylist = list(range(100000))
%time for _ in range(10): mylist2 = [x**2 for x in mylist]
```

```
CPU times: user 111 ms, sys: 351 ms, total: 461 ms
Wall time: 45.6 ms
```

2.3.9 Bedingte Logik als Array-Operationen – `where`

Die Funktion numpy.where ist eine vektorisierte Version von `if` und `else`.

Im folgenden Beispiel erzeugen wir zunächst ein boolesches Array und zwei Arrays mit Werten:

[1]:
```
import numpy as np
```

[2]:
```
cond = ([False,  True, False,  True, False, False, False])
data1 = np.random.randn(1, 7)
data2 = np.random.randn(1, 7)
```

Nun wollen wir Nehmen wir die Werte aus `data1` übernehmen, wenn der entsprechende Wert in `cond` `True` ist und ansonsten den Wert aus `data2` übernommen wird. Mit Pythons `if-else` könnte das wie folgt aussehen:

[3]:
```
result = [(x if c else y) for x, y, c in zip(data1, data2, cond)]

result
```

[3]:
```
[array([-0.37431791,  0.11490952, -0.24917534,  0.35700256, -0.3293716 ,
        -1.51677151,  0.351892  ])]
```

Dies hat jedoch die folgenden beiden Probleme:

- bei großen Arrays wird die Funktion nicht sehr schnell sein
- dies funktioniert nicht mit mehrdimensionalen Arrays

Mit `np.where` könnt ihr diese Probleme in einem einzigen Funktionsaufruf umgehen:

[4]:
```
result = np.where(cond, data1, data2)

result
```

[4]:
```
array([[-0.37431791,  0.90681988, -0.24917534,  0.0425698 , -0.3293716 ,
        -1.51677151,  0.351892  ]])
```

Das zweite und dritte Argument von `np.where` müssen keine Arrays sein; eines oder beide können auch Skalare sein. Eine typische Anwendung von `where` in der Datenanalyse besteht darin, ein neues Array von Werten auf der Grundlage eines anderen Arrays zu erzeugen. Angenommen, ihr habt eine Matrix mit zufällig generierten Daten und möchtet alle negativen Werte zu positiven Werten machen:

[5]:
```
data = np.random.randn(4, 4)

data
```

[5]:
```
array([[-1.52714845, -0.17217264,  0.48149727,  0.18465047],
       [ 0.02691677, -0.39642089, -1.54266224,  1.40343846],
       [-0.1541781 , -1.94429536, -1.55113023, -1.27231227],
       [ 0.44520634,  1.17590632,  1.30634966, -1.8479735 ]])
```

[6]:
```
data < 0
```

[6]:
```
array([[ True,  True, False, False],
       [False,  True,  True, False],
       [ True,  True,  True,  True],
       [False, False, False,  True]])
```

[7]:
```
np.where(data < 0, data * -1, data)
```

[7]:
```
array([[1.52714845, 0.17217264, 0.48149727, 0.18465047],
       [0.02691677, 0.39642089, 1.54266224, 1.40343846],
       [0.1541781 , 1.94429536, 1.55113023, 1.27231227],
       [0.44520634, 1.17590632, 1.30634966, 1.8479735 ]])
```

2.3.10 Mathematische und statistische Methoden

Eine Reihe von mathematischen Funktionen, die Statistiken über ein ganzes Array oder über die Daten entlang einer Achse berechnen, sind als Methoden der Array-Klasse zugänglich. So könnt ihr Aggregationen verwenden wie Summe, Mittelwert und Standardabweichung, indem ihr entweder die Array-Instanzmethode aufruft oder die NumPy-Funktion der obersten Ebene verwendet.

Im folgenden generiere ich einige normalverteilte Zufallsdaten und berechne einige aggregierte Statistiken:

[1]:
```
import numpy as np


data = np.random.randn(7, 3)

data
```

[1]:
```
array([[ 0.76703659, -0.83933196,  0.4645299 ],
       [-1.78256805,  0.1250087 ,  1.34358408],
       [-0.69649245,  0.38336052,  0.68022167],
       [-0.14760562, -1.481958  , -0.4465985 ],
       [ 0.26499335,  1.21256197,  0.5816795 ],
       [-0.04407397,  0.87961945, -0.58604436],
       [-1.15613167, -0.16129107,  0.53232522]])
```

[2]:
```
data.mean()
```

[2]:
```
-0.00510355699323813
```

[3]:
```
np.mean(data)
```

[3]:
```
-0.00510355699323813
```

[4]:
```
data.sum()
```

[4]:
```
-0.10717469685800074
```

Funktionen wie `mean` und `sum` benötigen ein optionales Achsenargument, das die Statistik über die angegebene Achse berechnet, was zu einem Array mit einer Dimension weniger führt:

[5]:
```
data.mean(axis=0)
```

[5]:
```
array([-0.39926312,  0.0168528 ,  0.36709964])
```

[6]:
```
data.sum(axis=0)
```

[6]:
```
array([-2.79484182,  0.11796961,  2.56969751])
```

Mit `data.mean(0)`, was dasselbe ist wie `data.mean(axis=0)`, wird der Mittelwert über die Zeilen berechnet, während `data.sum(0)` die Summe über die Zeilen berechnet.

Andere Methoden wie `cumsum` und `cumprod` aggregieren hingegen nicht, sondern erzeugen ein neues Array mit den Zwischenergebnissen.

In mehrdimensionalen Arrays geben Akkumulationsfunktionen wie `cumsum` und `cumprodm` ein Array derselben Größe zurück, aber mit den entlang der angegebenen Achse berechneten Teilaggregaten:

[7]:
```
data.cumsum()
```

[7]:
```
array([ 0.76703659, -0.07229536,  0.39223454, -1.3903335 , -1.2653248 ,
        0.07825928, -0.61823317, -0.23487265,  0.44534901,  0.2977434 ,
       -1.1842146 , -1.6308131 , -1.36581975, -0.15325778,  0.42842172,
        0.38434775,  1.26396719,  0.67792283, -0.47820884, -0.63949992,
       -0.1071747 ])
```

[8]:
```
data.cumprod()
```

[8]:
```
array([ 7.67036595e-01, -6.43798325e-01, -2.99063574e-01,  5.33101170e-01,
        6.66422856e-02,  8.95395139e-02, -6.23635955e-02, -2.39077405e-02,
       -1.62625630e-02,  2.40044565e-03, -3.55735963e-03,  1.58871146e-03,
        4.20997972e-04,  5.10486129e-04,  2.96939317e-04, -1.30872955e-05,
       -1.15118396e-05,  6.74644874e-06, -7.79978307e-06,  1.25803538e-06,
        6.69683962e-07])
```

Grundlegende statistische Methoden für Arrays:

Methode	Beschreibung
`sum`	Summe aller Elemente im Array oder entlang einer Achse
`mean`	Arithmetisches Mittel; bei Arrays mit der Länge Null wird `NaN` zurückgegeben
`std`, `var`	Standardabweichung bzw. Varianz
`min`, `max`	Minimum und Maximum
`argmin`, `argmax`	Indizes der minimalen bzw. maximalen Elemente
`cumsum`	Kumulative Summe der Elemente, beginnend mit `0`
`cumprod`	Kumulatives Produkt der Elemente, beginnend mit `1`

2.3.11 Methoden für boolesche Arrays

Boolesche Werte wurden in den vorangegangenen Methoden in 1 (`True`) und 0 (`False`) umgewandelt. Daher wird `sum` oft zum Zählen der `True`-Werte in einem booleschen Array verwendet:

[1]:
```
import numpy as np
```

[2]:
```
data = np.random.randn(7, 3)
```

Anzahl der positiven Werte:

[3]:
```
(data > 0).sum()
```

[3]:
```
10
```

Anzahl der negativen Werte:

[4]:
```
(data < 0).sum()
```

[4]:
```
11
```

Es gibt zwei zusätzliche Methoden, `any` und `all`, die besonders für boolesche Arrays nützlich sind:

- `any` prüft, ob ein oder mehrere Werte in einem Array wahr sind
- `all` prüft, ob jeder Wert wahr ist

[5]:
```
data2 = np.random.randn(7, 3)
bools = data > data2

bools
```

[5]:
```
array([[ True, False, False],
       [ True,  True, False],
       [False,  True, False],
       [False,  True, False],
       [ True,  True, False],
       [ True, False,  True],
       [False,  True,  True]])
```

[6]:
```
bools.any()
```

[6]:
```
True
```

[7]:
```
bools.all()
```

[7]:
```
False
```

2.3.12 Sortieren

Wie in Pythons `list` können NumPy-Arrays mit der Sortiermethode numpy.sort in-place sortiert werden. Dabei könnt ihr jeden eindimensionalen Abschnitt von Werten in einem mehrdimensionalen Array an Ort und Stelle entlang einer Achse sortieren, indem ihr die Achsennummer zum Sortieren übergebt:

[1]:
```
import numpy as np

data = np.random.randn(7, 3)

data
```

[1]:
```
array([[-0.56284094, -0.33779884,  0.01886388],
       [-1.61021704,  0.52704153,  0.96303615],
       [ 0.81173664, -4.40632737,  0.58281267],
       [ 0.24646619,  0.73165879,  0.64359417],
       [ 2.06398772, -0.74374727, -0.24729191],
       [-0.19820382,  1.28179855, -0.47229339],
       [-0.40045568,  0.55265251,  0.49467883]])
```

[2]:
```
data.sort(0)

data
```

[2]:
```
array([[-1.61021704, -4.40632737, -0.47229339],
       [-0.56284094, -0.74374727, -0.24729191],
       [-0.40045568, -0.33779884,  0.01886388],
       [-0.19820382,  0.52704153,  0.49467883],
       [ 0.24646619,  0.55265251,  0.58281267],
       [ 0.81173664,  0.73165879,  0.64359417],
       [ 2.06398772,  1.28179855,  0.96303615]])
```

`np.sort` gibt hingegen eine sortierte Kopie eines Arrays zurück, anstatt das Array an Ort und Stelle zu verändern:

[6]:
```
np.sort(data, axis=1)
```

[6]:
```
array([[-4.40632737, -1.61021704, -0.47229339],
       [-0.74374727, -0.56284094, -0.24729191],
       [-0.40045568, -0.33779884,  0.01886388],
       [-0.19820382,  0.49467883,  0.52704153],
       [ 0.24646619,  0.55265251,  0.58281267],
       [ 0.64359417,  0.73165879,  0.81173664],
       [ 0.96303615,  1.28179855,  2.06398772]])
```

2.3.13 `unique` und andere Mengenlogik

NumPy hat einige grundlegende Mengenoperationen für eindimensionale `ndarray`. Eine häufig verwendete ist numpy.unique, die die sortierten eindeutigen Werte in einem Array zurückgibt:

[1]:
```
import numpy as np


names = np.array(
    [
        "Liam",
        "Olivia",
        "Noah",
        "Liam",
        "Noah",
        "Olivia",
        "Liam",
        "Emma",
        "Oliver",
        "Ava",
    ]
)
```

[2]:
```
np.unique(names)
```

[2]:
```
array(['Ava', 'Emma', 'Liam', 'Noah', 'Oliver', 'Olivia'], dtype='<U6')
```

Mit numpy.in1d lässt sich die Zugehörigkeit der Werte in einem eindimensionalen Array zu einem anderen Array überprüfen wobei ein boolesches Array zurückgegeben wird:

[3]:
```
np.in1d(names, ["Emma", "Ava", "Charlotte"])
```

[3]:
```
array([False, False, False, False, False, False, False,  True, False,
        True])
```

Array-Mengenoperationen:

Methode	Beschreibung
`unique(x)`	berechnet die sortierten, eindeutigen Elemente in `x`
`intersect1d(x, y)`	berechnet die sortierten, gemeinsamen Elemente in `x` und `y`
`union1d(x, y)`	berechnet die sortierte Vereinigung von Elementen
`in1d(x, y)`	berechnet ein boolesches Array, das angibt, ob jedes Element von `x` in `y` enthalten ist
`setdiff1d(x, y)`	setzt die Differenz der Elemente in `x`, die nicht in `y` enthalten sind
`setxor1d(x, y)`	setzt symmetrische Differenzen; Elemente, die in einem der Arrays enthalten sind, aber nicht in beiden

2.3.14 Dateieingabe und -ausgabe mit Arrays

NumPy ist in der Lage, Daten in einigen Text- oder Binärformaten auf der Festplatte zu speichern und von dort zu laden. In diesem Abschnitt gehe ich jedoch nur auf das NumPy-eigene Binärformat ein, da meist pandas oder andere Werkzeuge zum Laden von Text- oder Tabellendaten verwendet werden (siehe *Daten lesen, speichern und bereitstellen*).

`np.save` und `np.load` sind die beiden wichtigsten Funktionen zum effizienten Speichern und Laden von Array-Daten auf der Festplatte. Arrays werden standardmäßig in einem unkomprimierten Rohbinärformat mit der Dateierweiterung `.npy` gespeichert:

[1]:
```
import numpy as np


data = np.random.randn(7, 3)

np.save("my_data", data)
```

Wenn der Dateipfad nicht bereits auf `.npy` endet, wird die Erweiterung angehängt. Das Array auf der Festplatte kann dann mit `np.load` geladen werden:

[2]:
```
np.load("my_data.npy")
```

[2]:
```
array([[-1.84346172, -0.53302864,  1.37131024],
       [ 0.23457529,  1.27414327,  1.77885434],
       [ 0.52351246, -1.79912077, -0.55748198],
       [ 0.24762959, -0.90100598,  0.18478303],
       [-0.73525977, -1.63940599, -1.60407133],
       [-0.62536449, -0.03524507, -0.82175049],
       [-0.46539262, -0.26317502,  0.74846956]])
```

Ihr könnt mehrere Arrays in einem unkomprimierten Archiv speichern indem ihr `np.savez` verwendet und die Arrays als Schlüsselwortargumente übergebt:

[3]:
```
np.savez("data_archive.npz", a=data, b=np.square(data))
```

[4]:
```
archive = np.load("data_archive.npz")

archive["b"]
```

[4]:
```
array([[3.39835111e+00, 2.84119531e-01, 1.88049177e+00],
       [5.50255688e-02, 1.62344108e+00, 3.16432278e+00],
       [2.74065298e-01, 3.23683556e+00, 3.10786156e-01],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
       [6.13204151e-02, 8.11811769e-01, 3.41447698e-02],
       [5.40606923e-01, 2.68765198e+00, 2.57304483e+00],
       [3.91080744e-01, 1.24221481e-03, 6.75273860e-01],
       [2.16590291e-01, 6.92610896e-02, 5.60206683e-01]])
```

2.4 pandas

pandas ist eine Python-Bibliothek zur Datenanalyse, die in den letzten Jahren sehr populär geworden ist. Auf der Website wird pandas so beschrieben:

> »pandas ist ein schnelles, leistungsfähiges, flexibles und einfach zu bedienendes Open-Source-Tool zur Datenanalyse und -manipulation, das auf der Programmiersprache Python aufbaut.«

Genauer ist pandas ein In-Memory-Analysewerkzeug, das SQL-ähnliche Konstrukte, sowie statistische und analytische Werkzeuge bietet. Dabei baut pandas auf Cython und *NumPy* auf, wodurch es weniger speicherintensiv und schneller ist als reiner Python-Code. Meist wird pandas genutzt, um

- *Excel* und Power BI zu ersetzen
- einen ETL-Prozess zu realisieren
- *CSV*- oder *JSON*-Daten zu verarbeiten
- maschinelles Lernen vorzubereiten

Siehe auch:

- Home
- User guide
- API reference
- GitHub

2.4.1 Einführung in die Datenstrukturen von pandas

Um mit pandas zu beginnen, solltet ihr euch zunächst mit den beiden wichtigsten Datenstrukturen vertraut machen: *Series* und *DataFrame*.

Series

Eine Serie ist ein eindimensionales Array-ähnliches Objekt, das eine Folge von Werten (von ähnlichen Typen wie die NumPy-Typen) und ein zugehöriges Array von Datenbeschriftungen, genannt Index, enthält. Die einfachste Serie wird nur aus einem Array von Daten gebildet:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
s = pd.Series(np.random.randn(7))
s
```

[2]:
```
0   -0.734713
1    0.003809
2    0.291996
3    0.024081
4    1.115924
5    0.551363
6    1.029384
dtype: float64
```

Die Zeichenkettendarstellung einer interaktiv angezeigten Reihe zeigt den Index auf der linken Seite und die Werte auf der rechten Seite. Da wir keinen Index für die Daten angegeben haben, wird ein Standardindex erstellt, der aus den ganzen Zahlen `0` bis `N - 1` besteht (wobei `N` die Anzahl (*Length*) der Daten ist). Ihr könnt die Array-Darstellung und das Index-Objekt der Reihe über ihre pandas.Series.array- bzw. pandas.Series.index-Attribute erhalten:

[3]:
```
s.array
```

[3]:
```
<PandasArray>
[  -0.7347134268241727, 0.0038087194642710165,   0.29199556053746856,
  0.024081493463603963,    1.1159238685642536,    0.5513634668178448,
     1.029384005342762]
Length: 7, dtype: float64
```

[4]:
```
s.index
```

[4]:
```
RangeIndex(start=0, stop=7, step=1)
```

Oft werdet ihr einen Index erstellen wollen, der jeden Datenpunkt mit einer Bezeichnung kennzeichnet:

[5]:
```
idx = pd.date_range("2022-01-31", periods=7)

s2 = pd.Series(np.random.randn(7), index=idx)
```

[6]:
```
s2
```

[6]:
```
2022-01-31   -1.168981
2022-02-01    0.127966
2022-02-02   -1.915208
2022-02-03    0.935589
2022-02-04   -1.806423
2022-02-05    1.076115
2022-02-06    0.738466
Freq: D, dtype: float64
```

Siehe auch

- Time series / date functionality

Im Vergleich zu NumPy-Arrays könnt ihr Label im Index verwenden, wenn ihr einzelne Werte oder eine Gruppe von Werten auswählen wollt:

[7]:
```
s2["2022-02-02"]
```

[7]:
```
-1.9152075916006905
```

[8]:
```
s2[["2022-02-02", "2022-02-03", "2022-02-04"]]
```

[8]:
```
2022-02-02   -1.915208
2022-02-03    0.935589
2022-02-04   -1.806423
dtype: float64
```

Hier wird `['2022-02-02', '2022-02-03', '2022-02-04']` als eine Liste von Indizes interpretiert, auch wenn sie Strings anstelle von ganzen Zahlen enthält.

Bei der Verwendung von NumPy-Funktionen oder NumPy-ähnlichen Operationen, wie z. B. dem Filtern mit einem booleschen Array, der skalaren Multiplikation oder der Anwendung mathematischer Funktionen, bleibt die Verknüpfung zwischen Index und Wert erhalten:

[9]:
```
s2[s2 > 0]
```

[9]:
```
2022-02-01    0.127966
2022-02-03    0.935589
2022-02-05    1.076115
2022-02-06    0.738466
dtype: float64
```

[10]:
```
s2**2
```

[10]:
```
2022-01-31    1.366515
2022-02-01    0.016375
2022-02-02    3.668020
2022-02-03    0.875327
2022-02-04    3.263166
2022-02-05    1.158024
2022-02-06    0.545331
Freq: D, dtype: float64
```

[11]:
```
np.exp(s2)
```

[11]:
```
2022-01-31    0.310684
2022-02-01    1.136514
2022-02-02    0.147311
2022-02-03    2.548715
2022-02-04    0.164241
2022-02-05    2.933262
2022-02-06    2.092722
Freq: D, dtype: float64
```

Ihr könnt euch eine Serie auch als ein *ordered dict* mit fester Länge vorstellen, da sie eine Zuordnung von Indexwerten zu Datenwerten darstellt. Sie kann in vielen Kontexten verwendet werden, in denen man ein *dict* verwenden könnte:

[12]:
```
"2022-02-02" in s2
```

[12]:
```
True
```

[13]:
```
"2022-02-09" in s2
```

[13]:
```
False
```

Fehlende Daten

`NA` und `null` werde ich synonym verwenden, um auf fehlende Daten hinzuweisen. Die Funktionen `isna` und `notna` in pandas sollten verwendet werden, um fehlende Daten zu erkennen:

[14]:
```
pd.isna(s2)
```

[14]:
```
2022-01-31    False
2022-02-01    False
2022-02-02    False
2022-02-03    False
2022-02-04    False
2022-02-05    False
2022-02-06    False
Freq: D, dtype: bool
```

[15]:
```
pd.notna(s2)
```

[15]:
```
2022-01-31    True
2022-02-01    True
2022-02-02    True
2022-02-03    True
2022-02-04    True
2022-02-05    True
2022-02-06    True
Freq: D, dtype: bool
```

Series hat diese auch als Instanzmethoden:

[16]:
```
s2.isna()
```

[16]:
```
2022-01-31    False
2022-02-01    False
2022-02-02    False
2022-02-03    False
2022-02-04    False
2022-02-05    False
2022-02-06    False
Freq: D, dtype: bool
```

Der Umgang mit fehlenden Daten wird im Abschnitt *Verwalten fehlender Daten mit pandas* ausführlicher behandelt.

Eine für viele Anwendungen nützliche Funktion von Series ist die automatische Ausrichtung nach Indexbezeichnungen bei arithmetischen Operationen:

[17]:
```
idx = pd.date_range("2022-02-07", periods=7)

s3 = pd.Series(np.random.randn(7), index=idx)
```

[18]:
```
s2, s3
```

[18]:
```
(2022-01-31   -1.168981
 2022-02-01    0.127966
 2022-02-02   -1.915208
 2022-02-03    0.935589
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2022-02-04   -1.806423
2022-02-05    1.076115
2022-02-06    0.738466
Freq: D, dtype: float64,
2022-02-07   -0.871571
2022-02-08   -0.599261
2022-02-09    0.803387
2022-02-10    0.448825
2022-02-11   -0.548544
2022-02-12    0.996237
2022-02-13   -0.414533
Freq: D, dtype: float64)
```

[19]:
```
s2 + s3
```

[19]:
```
2022-01-31   NaN
2022-02-01   NaN
2022-02-02   NaN
2022-02-03   NaN
2022-02-04   NaN
2022-02-05   NaN
2022-02-06   NaN
2022-02-07   NaN
2022-02-08   NaN
2022-02-09   NaN
2022-02-10   NaN
2022-02-11   NaN
2022-02-12   NaN
2022-02-13   NaN
Freq: D, dtype: float64
```

Wenn ihr Erfahrung mit SQL habt, ähnelt dies einem JOIN-Vorgang.

Sowohl das Series-Objekt selbst als auch sein Index haben ein `name`-Attribut, das sich in andere Bereiche der pandas-Funktionalität integrieren lässt:

[20]:
```
s3.name = "floats"
s3.index.name = "date"

s3
```

[20]:
```
date
2022-02-07   -0.871571
2022-02-08   -0.599261
2022-02-09    0.803387
2022-02-10    0.448825
2022-02-11   -0.548544
2022-02-12    0.996237
2022-02-13   -0.414533
Freq: D, Name: floats, dtype: float64
```

DataFrame

Ein DataFrame stellt eine rechteckige Datentabelle dar und enthält eine geordnete, benannte Sammlung von Spalten, von denen jede einen anderen Werttyp haben kann. Der DataFrame hat sowohl einen Zeilen- als auch einen Spaltenindex.

Bemerkung

Ein DataFrame ist zwar zweidimensional, ihr könnt ihn aber auch verwenden, um mit join, combine und Reshaping höherdimensionale Daten in einem Tabellenformat mit hierarchischer Indizierung darzustellen.

[21]:
```
data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
    "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
}
df = pd.DataFrame(data)

df
```

[21]:

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

Bei großen DataFrames wählt die `head`-Methode nur die ersten fünf Zeilen aus:

[22]:
```
df.head()
```

[22]:

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D

Ihr könnt auch die Reihenfolge der Spalten angeben:

[23]:
```
pd.DataFrame(data, columns=["Code", "Key"])
```

[23]:

	Code	Key
0	U+0000	NUL
1	U+0001	Ctrl-A
2	U+0002	Ctrl-B
3	U+0003	Ctrl-C
4	U+0004	Ctrl-D
5	U+0005	Ctrl-E

Wenn ihr eine Spalte übergeben wollt, die nicht im Dict enthalten ist, wird sie ohne Werte im Ergebnis erscheinen:

[24]:
```
df2 = pd.DataFrame(
    data, columns=["Code", "Decimal", "Octal", "Description", "Key"]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
)

df2
```

[24]:
```
     Code  Decimal Octal Description     Key
0  U+0000        0   001         NaN     NUL
1  U+0001        1   002         NaN  Ctrl-A
2  U+0002        2   003         NaN  Ctrl-B
3  U+0003        3   004         NaN  Ctrl-C
4  U+0004        4   004         NaN  Ctrl-D
5  U+0005        5   005         NaN  Ctrl-E
```

Ihr könnt eine Spalte in einem DataFrame mit einer Dict-ähnlichen Notation abrufen:

[25]:
```
df["Code"]
```

[25]:
```
0    U+0000
1    U+0001
2    U+0002
3    U+0003
4    U+0004
5    U+0005
Name: Code, dtype: object
```

So könnt ihr auch eine Spalte zum Index machen:

[26]:
```
df2 = pd.DataFrame(
    data, columns=["Decimal", "Octal", "Description", "Key"], index=df["Code"]
)

df2
```

[26]:
```
        Decimal Octal Description     Key
Code
U+0000        0   001         NaN     NUL
U+0001        1   002         NaN  Ctrl-A
U+0002        2   003         NaN  Ctrl-B
U+0003        3   004         NaN  Ctrl-C
U+0004        4   004         NaN  Ctrl-D
U+0005        5   005         NaN  Ctrl-E
```

Zeilen können nach Position oder Name mit dem pandas.DataFrame.loc-Attribut abgerufen werden:

[27]:
```
df2.loc["U+0001"]
```

[27]:
```
Decimal            1
Octal            002
Description      NaN
Key           Ctrl-A
Name: U+0001, dtype: object
```

Spaltenwerte können durch Zuweisung geändert werden. Zum Beispiel könnte der leeren Spalte *Description* ein Einzelwert oder ein Array von Werten zugewiesen werden:

[28]:
```
df2["Description"] = [
    "Null character",
    "Start of Heading",
    "Start of Text",
    "End-of-text character",
    "End-of-transmission character",
    "Enquiry character",
]

df2
```

[28]:

Code	Decimal	Octal	Description	Key
U+0000	0	001	Null character	NUL
U+0001	1	002	Start of Heading	Ctrl-A
U+0002	2	003	Start of Text	Ctrl-B
U+0003	3	004	End-of-text character	Ctrl-C
U+0004	4	004	End-of-transmission character	Ctrl-D
U+0005	5	005	Enquiry character	Ctrl-E

Das Zuweisen einer nicht existierenden Spalte erzeugt eine neue Spalte.

Mit pandas.DataFrame.drop können Spalten entfernt und mit `pandas.DataFrame.columns` angezeigt werden:

[29]:
```
df3 = df2.drop(columns=["Decimal", "Octal"])
```

[30]:
```
df2.columns
```

[30]:
```
Index(['Decimal', 'Octal', 'Description', 'Key'], dtype='object')
```

[31]:
```
df3.columns
```

[31]:
```
Index(['Description', 'Key'], dtype='object')
```

Eine weitere gängige Form von Daten sind verschachtelte Dict von Dicts:

[32]:
```
u = {
    "U+0006": {
        "Decimal": "6",
        "Octal": "006",
        "Description": "Acknowledge character",
        "Key": "Ctrl-F",
    },
    "U+0007": {
        "Decimal": "7",
        "Octal": "007",
        "Description": "Bell character",
        "Key": "Ctrl-G",
    },
}

df4 = pd.DataFrame(u)

df4
```

[32]:
```
                            U+0006          U+0007
Decimal                          6               7
Octal                          006             007
Description  Acknowledge character  Bell character
Key                          Ctrl-F          Ctrl-G
```

Ihr könnt den DataFrame transponieren, d.h. die Zeilen und Spalten vertauschen, mit einer ähnlichen Syntax wie bei einem NumPy-Array:

[33]:
```
df4.T
```

[33]:
```
       Decimal Octal            Description     Key
U+0006       6   006  Acknowledge character  Ctrl-F
U+0007       7   007          Bell character  Ctrl-G
```

Warnung

Beachtet, dass beim Transponieren die Datentypen der Spalten verworfen werden, wenn die Spalten nicht alle denselben Datentyp haben, so dass beim Transponieren und anschließenden Zurücktransponieren die vorherigen Typinformationen verloren gehen können. Die Spalten werden in diesem Fall zu Arrays aus reinen Python-Objekten.

Die Schlüssel in den inneren Dicts werden kombiniert, um den Index im Ergebnis zu bilden. Dies ist nicht der Fall, wenn ein expliziter Index angegeben wird:

[34]:
```
df5 = pd.DataFrame(u, index=["Decimal", "Octal", "Key"])
df5
```

[34]:
```
         U+0006  U+0007
Decimal       6       7
Octal       006     007
Key      Ctrl-F  Ctrl-G
```

2.4.2 Python-Datenstrukturen in pandas überführen

Python-Datenstrukuren wie Listen und Arrays lassen sich in pandas *Series* oder *DataFrames* überführen.

[1]:
```
import numpy as np
import pandas as pd
```

Series

Python Lists können einfach in pandas Series umgewandelt werden:

[2]:
```
list1 = [
    -0.751442,
    0.816935,
    -0.272546,
    -0.268295,
    -0.296728,
    0.176255,
    -0.322612,
]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
pd.Series(list1)
```

[2]:
```
0   -0.751442
1    0.816935
2   -0.272546
3   -0.268295
4   -0.296728
5    0.176255
6   -0.322612
dtype: float64
```

Auch mehrere Lists lassen sich einfach in eine pandas Series umwandeln:

[3]:
```
list2 = [
    -0.029608,
    -0.277982,
    2.693057,
    -0.850817,
    0.783868,
    -1.137835,
    -0.617132,
]

pd.Series(list1 + list2)
```

[3]:
```
0    -0.751442
1     0.816935
2    -0.272546
3    -0.268295
4    -0.296728
5     0.176255
6    -0.322612
7    -0.029608
8    -0.277982
9     2.693057
10   -0.850817
11    0.783868
12   -1.137835
13   -0.617132
dtype: float64
```

Es kann auch eine Liste als Index übergeben werden:

[4]:
```
date = [
    "2022-01-31",
    "2022-02-01",
    "2022-02-02",
    "2022-02-03",
    "2022-02-04",
    "2022-02-05",
    "2022-02-06",
]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
pd.Series(list1, index=date)
```

[4]:
```
2022-01-31   -0.751442
2022-02-01    0.816935
2022-02-02   -0.272546
2022-02-03   -0.268295
2022-02-04   -0.296728
2022-02-05    0.176255
2022-02-06   -0.322612
dtype: float64
```

Mit Python Dictionary könnt ihr nicht nur Werte sondern auch die zugehörigen Schlüssel an eine pandas Series übergeben:

[5]:
```
dict1 = {
    "2022-01-31": -0.751442,
    "2022-02-01": 0.816935,
    "2022-02-02": -0.272546,
    "2022-02-03": -0.268295,
    "2022-02-04": -0.296728,
    "2022-02-05": 0.176255,
    "2022-02-06": -0.322612,
}

pd.Series(dict1)
```

[5]:
```
2022-01-31   -0.751442
2022-02-01    0.816935
2022-02-02   -0.272546
2022-02-03   -0.268295
2022-02-04   -0.296728
2022-02-05    0.176255
2022-02-06   -0.322612
dtype: float64
```

Wenn ihr ein `dict` übergebt, berücksichtigt der Index in der resultierenden pandas Series die Reihenfolge der Schlüssel im Dict.

Mit collections.ChainMap könnt ihr auch mehrere Dicts in eine pandas.Series verwandeln.

Zunächst definieren wir hierfür ein zweites Dict:

[6]:
```
dict2 = {
    "2022-02-07": -0.029608,
    "2022-02-08": -0.277982,
    "2022-02-09": 2.693057,
    "2022-02-10": -0.850817,
    "2022-02-11": 0.783868,
    "2022-02-12": -1.137835,
    "2022-02-13": -0.617132,
}
```

[7]:
```
from collections import ChainMap


pd.Series(ChainMap(dict1, dict2))
```

[7]:
```
2022-02-07   -0.029608
2022-02-08   -0.277982
2022-02-09    2.693057
2022-02-10   -0.850817
2022-02-11    0.783868
2022-02-12   -1.137835
2022-02-13   -0.617132
2022-01-31   -0.751442
2022-02-01    0.816935
2022-02-02   -0.272546
2022-02-03   -0.268295
2022-02-04   -0.296728
2022-02-05    0.176255
2022-02-06   -0.322612
dtype: float64
```

DataFrame

Listen von Python list können in ein pandas DataFrame geladen werden mit:

[8]:
```
df = pd.DataFrame([list1, list2])
df
```

[8]:
```
          0         1         2         3         4         5         6
0 -0.751442  0.816935 -0.272546 -0.268295 -0.296728  0.176255 -0.322612
1 -0.029608 -0.277982  2.693057 -0.850817  0.783868 -1.137835 -0.617132
```

Ihr könnt auch eine Liste in einen DataFrame-Index überführen:

[9]:
```
pd.DataFrame([list1, list2], index=["2022-01-31", "2022-02-01"])
```

[9]:
```
                   0         1         2         3         4         5  \
2022-01-31 -0.751442  0.816935 -0.272546 -0.268295 -0.296728  0.176255
2022-02-01 -0.029608 -0.277982  2.693057 -0.850817  0.783868 -1.137835


                   6
2022-01-31 -0.322612
2022-02-01 -0.617132
```

Ein pandas DataFrame kann aus einem Dict mit Werten in Listen erstellt werden:

[10]:
```
data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
    "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
}
```

[11]:
```
pd.DataFrame(data)
```

[11]:
```
     Code  Decimal Octal     Key
0  U+0000        0   001     NUL
1  U+0001        1   002  Ctrl-A
2  U+0002        2   003  Ctrl-B
3  U+0003        3   004  Ctrl-C
4  U+0004        4   004  Ctrl-D
5  U+0005        5   005  Ctrl-E
```

Eine weitere gängige Form von Daten sind verschachtelte Dict von Dicts:

[12]:
```
data2 = {
    "U+0006": {"Decimal": "6", "Octal": "006", "Key": "Ctrl-F"},
    "U+0007": {"Decimal": "7", "Octal": "007", "Key": "Ctrl-G"},
}

df2 = pd.DataFrame(data2)

df2
```

[12]:
```
         U+0006  U+0007
Decimal       6       7
Octal       006     007
Key      Ctrl-F  Ctrl-G
```

Dicts von Series werden in ähnlicher Weise behandelt:

[13]:
```
data3 = {"U+0006": df2["U+0006"][2:], "U+0007": df2["U+0007"][2:]}

pd.DataFrame(data3)
```

[13]:
```
     U+0006  U+0007
Key  Ctrl-F  Ctrl-G
```

2.4.3 Indexierung

Index-Objekte

Die Index-Objekte von pandas sind für die Achsenbeschriftungen und andere Metadaten, wie den Achsennamen, verantwortlich. Jedes Array oder jede andere Sequenz von Beschriftungen, die ihr bei der Konstruktion einer Serie oder eines DataFrame verwendet, wird intern in einen Index umgewandelt:

[1]:
```
import pandas as pd


obj = pd.Series(range(7), index=pd.date_range("2022-02-02", periods=7))
```

[2]:
```
obj.index
```

[2]:
```
DatetimeIndex(['2022-02-02', '2022-02-03', '2022-02-04', '2022-02-05',
               '2022-02-06', '2022-02-07', '2022-02-08'],
              dtype='datetime64[ns]', freq='D')
```

[3]:
```
obj.index[3:]
```

[3]:
```
DatetimeIndex(['2022-02-05', '2022-02-06', '2022-02-07', '2022-02-08'], dtype=
↪'datetime64[ns]', freq='D')
```

Indexobjekte sind unveränderlich (*immutable*) und können daher nicht geändert werden:

[4]:
```
obj.index[1] = "2022-02-03"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[4], line 1
----> 1 obj.index[1] = "2022-02-03"

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexes/base.py:5157, in Index.__setitem__(self, key, value)
   5155 @final
   5156 def __setitem__(self, key, value):
-> 5157     raise TypeError("Index does not support mutable operations")

TypeError: Index does not support mutable operations
```

Die Unveränderlichkeit macht die gemeinsame Nutzung von Indexobjekten in Datenstrukturen sicherer:

[5]:
```
import numpy as np


labels = pd.Index(np.arange(3))

labels
```

[5]:
```
Index([0, 1, 2], dtype='int64')
```

[6]:
```
obj2 = pd.Series(np.random.randn(3),index=labels)
```

[7]:
```
obj2
```

[7]:
```
0   -0.426526
1    0.038709
2    0.316950
dtype: float64
```

[8]:
```
obj2.index is labels
```

[8]:
```
True
```

Um einem Array ähnlich zu sein verhält sich ein Index auch wie eine Menge mit fester Größe:

[9]:
```
data1 = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
}
df1 = pd.DataFrame(data1)
```

[10]:
```
df1
```

[10]:
```
     Code  Decimal Octal
0  U+0000        0   001
1  U+0001        1   002
2  U+0002        2   003
3  U+0003        3   004
4  U+0004        4   004
5  U+0005        5   005
```

[11]:
```
df1.columns
```

[11]:
```
Index(['Code', 'Decimal', 'Octal'], dtype='object')
```

[12]:
```
"Code" in df1.columns
```

[12]:
```
True
```

[13]:
```
"Key" in df1.columns
```

[13]:
```
False
```

Achsenindizes mit doppelten Labels

Anders als Python-Sets kann ein Pandas-Index doppelte Label enthalten:

[14]:
```
data2 = {
    "Code": ["U+0006", "U+0007"],
    "Decimal": [6, 7],
    "Octal": ["006", "007"],
}
df2 = pd.DataFrame(data2)
df12 = pd.concat([df1, df2])

df12
```

[14]:
```
     Code  Decimal Octal
0  U+0000        0   001
1  U+0001        1   002
2  U+0002        2   003
3  U+0003        3   004
4  U+0004        4   004
5  U+0005        5   005
0  U+0006        6   006
1  U+0007        7   007
```

Bei der *Auswahl* von doppelten Bezeichnungen werden alle Vorkommen der betreffenden Bezeichnung ausgewählt:

[15]:
```
df12.loc[1]
```

[15]:
```
     Code  Decimal Octal
1  U+0001        1   002
1  U+0007        7   007
```

Die Datenauswahl ist einer der Hauptpunkte, der sich bei Duplikaten anders verhält. Die Indizierung eines Labels mit mehreren Einträgen ergibt eine Serie, während einzelne Einträge einen Einzelwert ergeben. Dies kann euren Code komplizierter machen, da der Ausgabetyp der Indizierung je nachdem, ob ein Label wiederholt wird oder nicht, variieren kann. Zudem setzen viele Pandas-Funktionen, wie z.B. `reindex`, voraus, dass Label eindeutig sind. Anhand der Eigenschaft `is_unique` des Index könnt ihr feststellen, ob seine Label eindeutig sind oder nicht:

[16]:
```
df12.index.is_unique
```

[16]:
```
False
```

Um doppelte Label zu vermeiden, könnt ihr z.B. `ignore_index=True` verwenden:

[17]:
```
df12 = pd.concat([df1, df2], ignore_index=True)

df12
```

[17]:
```
     Code  Decimal Octal
0  U+0000        0   001
1  U+0001        1   002
2  U+0002        2   003
3  U+0003        3   004
4  U+0004        4   004
5  U+0005        5   005
6  U+0006        6   006
7  U+0007        7   007
```

Einige Indexmethoden und -eigenschaften

Jeder Index verfügt über eine Reihe von Methoden und Eigenschaften für die Mengenlogik, die andere allgemeine Fragen zu den darin enthaltenen Daten beantworten. Im folgenden einige nützliche Methoden und Eigenschaften:

Methode	Beschreibung
`append`	verkettet zusätzliche Indexobjekte, wodurch ein neuer Index entsteht
`difference`	berechnet die Differenz zweier Mengen als Index
`intersection`	berechnet die Schnittmenge
`union`	berechnet die Vereinigungsmenge
`isin`	berechnet ein boolesches Array, das angibt, ob jeder Wert in der übergebenen Sammlung enthalten ist
`delete`	berechnet einen neuen Index, wobei das Element in Index `i` gelöscht wird
`drop`	berechnet einen neuen Index durch Löschen der übergebenen Werte
`insert`	berechnet neuen Index durch Einfügen des Elements in den Index `i`
`is_monotonic`	gibt `True` zurück, wenn jedes Element größer oder gleich dem vorherigen Element ist
`is_unique`	gibt `True` zurück, wenn der Index keine doppelten Werte enthält.
`unique`	berechnet das Array der eindeutigen Werte im Index

Neuindizierung mit `reindex`

Eine wichtige Methode für Pandas-Objekte ist die Neuindizierung, d.h. die Erstellung eines neuen Objekts mit neu angeordneten Werten, die mit dem neuen Index übereinstimmen. Betrachtet z.B.:

[18]:
```
obj = pd.Series(range(7), index=pd.date_range("2022-02-02", periods=7))
```

[19]:
```
obj
```

[19]:
```
2022-02-02    0
2022-02-03    1
2022-02-04    2
2022-02-05    3
2022-02-06    4
2022-02-07    5
2022-02-08    6
Freq: D, dtype: int64
```

[20]:
```
new_index = pd.date_range("2022-02-03", periods=7)
```

[21]:
```
obj.reindex(new_index)
```

[21]:
```
2022-02-03    1.0
2022-02-04    2.0
2022-02-05    3.0
2022-02-06    4.0
2022-02-07    5.0
2022-02-08    6.0
2022-02-09    NaN
Freq: D, dtype: float64
```

`reindex` erstellt einen neuen Index und indiziert den DataFrame neu. Standardmäßig werden Werte im neuen Index, für die es keine entsprechenden Datensätze im DataFrame gibt, zu `NaN`.

Bei geordneten Daten wie Zeitreihen kann es wünschenswert sein, bei der Neuindizierung Werte zu interpolieren oder zu füllen. Die Option `method` ermöglicht dies mit einer Methode wie `ffill`, die die Werte vorwärts füllt:

[22]:
```
obj.reindex(new_index, method="ffill")
```

[22]:
```
2022-02-03    1
2022-02-04    2
2022-02-05    3
2022-02-06    4
2022-02-07    5
2022-02-08    6
2022-02-09    6
Freq: D, dtype: int64
```

Bei einem DataFrame kann `reindex` entweder den (Zeilen-)Index, die Spalten oder beides ändern. Wenn nur eine Sequenz übergeben wird, werden die Zeilen im Ergebnis neu indiziert:

[23]:
```
df1.reindex(range(7))
```

[23]:
```
     Code  Decimal Octal
0  U+0000      0.0   001
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
1  U+0001      1.0   002
2  U+0002      2.0   003
3  U+0003      3.0   004
4  U+0004      4.0   004
5  U+0005      5.0   005
6     NaN      NaN   NaN
```

Die Spalten können mit dem Schlüsselwort `columns` neu indiziert werden:

[24]:
```
encoding = ["Octal", "Code", "Description"]

df1.reindex(columns=encoding)
```

[24]:
```
  Octal    Code  Description
0   001  U+0000          NaN
1   002  U+0001          NaN
2   003  U+0002          NaN
3   004  U+0003          NaN
4   004  U+0004          NaN
5   005  U+0005          NaN
```

Argumente der Funktion `reindex`

Argument	Beschreibung
`labels`	Neue Sequenz, die als Index verwendet werden soll. Kann eine Index-Instanz oder eine andere sequenzähnliche Python-Datenstruktur sein. Ein Index wird genau so verwendet, wie er ist, ohne dass er kopiert wird.
`axis`	Die neu zu indizierende Achse, entweder `index` (Zeilen) oder `columns` (Spalten). Die Vorgabe ist `index`. Ihr könnt alternativ `reindex(index=new_labels)` oder `reindex(columns=new_labels)` verwenden.
`method`	Interpolationsmethode; `ffill` füllt vorwärts, während `bfill` rückwärts füllt.
`fill_v`	Ersatzwert, der zu verwenden ist, wenn fehlende Daten durch Neuindizierung eingefügt werden. Verwendet `fill_value='missing'` (das Standardverhalten), wenn die fehlenden Bezeichnungen im Ergebnis Nullwerte haben sollen.
`limit`	Beim Vorwärts- oder Rückwärtsfüllen die maximale Anzahl der zu füllenden Elemente.
`tolera`	Beim Vorwärts- oder Rückwärtsauffüllen die maximale Größe der Lücke, die bei ungenauen Übereinstimmungen gefüllt werden soll.
`level`	Einfachen Index auf Ebene von `MultiIndex` abgleichen; andernfalls Teilmenge auswählen.
`copy`	Wenn `True`, werden die zugrunde liegenden Daten immer kopiert, auch wenn der neue Index dem alten Index entspricht; wenn `False`, werden die Daten nicht kopiert, wenn die Indizes gleichwertig sind.

Achsenindizes umbenennen

Die Achsenbeschriftungen können durch eine Funktion oder ein Mapping umgewandelt werden, um neue, anders beschriftete Objekte zu erzeugen. Ihr könnt auch die Achsen an Ort und Stelle ändern, ohne eine neue Datenstruktur zu erstellen. Hier ist ein einfaches Beispiel:

[25]:
```
df3 = pd.DataFrame(
    np.arange(12).reshape((3, 4)),
    index=["Deutsch", "English", "Français"],
    columns=[1, 2, 3, 4],
)

df3
```

[25]:
```
          1  2   3   4
Deutsch   0  1   2   3
English   4  5   6   7
Français  8  9  10  11
```

Achsenindizes umbenennen mit `map`

Wie `Series` haben auch die Achsenindizes eine `map`-Methode:

[26]:
```
transform = lambda x: x[:2].upper()

df3.index.map(transform)
```

[26]:
```
Index(['DE', 'EN', 'FR'], dtype='object')
```

Ihr könnt den Index zuweisen und den DataFrame an Ort und Stelle ändern:

[27]:
```
df3.index = df3.index.map(transform)

df3
```

[27]:
```
    1  2   3   4
DE  0  1   2   3
EN  4  5   6   7
FR  8  9  10  11
```

Achsenindizes umbenennen mit `rename`

Wenn ihr eine umgewandelte Version eures Datensatzes erstellen möchtet ohne das Original zu verändern, könnt ihr `rename` verwenden:

[28]:
```
df3.rename(index=str.lower)
```

[28]:
```
    1  2   3   4
de  0  1   2   3
en  4  5   6   7
fr  8  9  10  11
```

Insbesondere kann `rename` in Verbindung mit einem `dict`-ähnlichen Objekt verwendet werden, das neue Werte für eine Teilmenge der Achsenbeschriftungen liefert:

[29]:
```
df3.rename(
    index={"DE": "BE", "EN": "DE", "FR": "EN"},
    columns={1: 0, 2: 1, 3: 2, 4: 3},
)
```

[29]:
```
    0  1   2   3
BE  0  1   2   3
DE  4  5   6   7
EN  8  9  10  11
```

`rename` erspart euch das manuelle Kopieren des DataFrame und die Zuweisung seiner Index- und Spaltenattribute. Wenn ihr einen Datensatz an Ort und Stelle ändern möchtet, übergebt zusätzlich noch `inplace=True`:

[30]:
```
df3.rename(
    index={"DE": "BE", "EN": "DE", "FR": "EN"},
    columns={1: 0, 2: 1, 3: 2, 4: 3},
    inplace=True,
)

df3
```

[30]:
```
    0  1   2   3
BE  0  1   2   3
DE  4  5   6   7
EN  8  9  10  11
```

Hierarchische Indizierung

Die hierarchische Indizierung ist eine wichtige Funktion von pandas, die euch ermöglicht, mehrere Indexebenen auf einer Achse zu haben. Dies bietet euch die Möglichkeit, mit höherdimensionalen Daten in einer niedrigdimensionalen Form zu arbeiten.

Beginnen wir mit einem einfachen Beispiel: Erstellen wir eine Reihe Liste von Listen als Index:

[31]:
```
hits = pd.Series(
    [83080, 20336, 11376, 1228, 468],
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "de", "en"],
    ],
)

hits
```

[31]:
```
Jupyter Tutorial  de    83080
                  en    20336
PyViz Tutorial    de    11376
Python Basics     de     1228
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                  en      468
dtype: int64
```

Was ihr seht, ist eine graphische Ansicht einer Serie mit einem pandas.MultiIndex. Die „Lücken“ in der Indexanzeige bedeuten, dass die Beschriftung darüber verwendet werden soll.

[32]:
```
hits.index
```

[32]:
```
MultiIndex([('Jupyter Tutorial', 'de'),
            ('Jupyter Tutorial', 'en'),
            (  'PyViz Tutorial', 'de'),
            (   'Python Basics', 'de'),
            (   'Python Basics', 'en')],
           )
```

Bei einem hierarchisch indizierten Objekt ist eine so genannte partielle Indizierung möglich, mit der ihr Teilmengen der Daten gezielt auswählen könnt:

[33]:
```
hits["Jupyter Tutorial"]
```

[33]:
```
de    83080
en    20336
dtype: int64
```

[34]:
```
hits["Jupyter Tutorial":"Python Basics"]
```

[34]:
```
Jupyter Tutorial  de    83080
                  en    20336
PyViz Tutorial    de    11376
Python Basics     de     1228
                  en      468
dtype: int64
```

[35]:
```
hits.loc[["Jupyter Tutorial", "Python Basics"]]
```

[35]:
```
Jupyter Tutorial  de    83080
                  en    20336
Python Basics     de     1228
                  en      468
dtype: int64
```

Die Auswahl ist sogar von einer „inneren“ Ebene aus möglich. Im folgenden wähle ich alle Werte mit dem Wert 1 aus der zweiten Indexebene aus:

[36]:
```
hits.loc[:, "de"]
```

[36]:
```
Jupyter Tutorial    83080
PyViz Tutorial      11376
Python Basics        1228
dtype: int64
```

Bei einem DataFrame kann jede Achse einen hierarchischen Index haben:

[37]:
```
version_hits = [
    [19651, 0, 30134, 0, 33295, 0],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    [4722, 1825, 3497, 2576, 4009, 3707],
    [2573, 0, 4873, 0, 3930, 0],
    [525, 0, 427, 0, 276, 0],
    [157, 0, 85, 0, 226, 0],
]

df = pd.DataFrame(
    version_hits,
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "de", "en"],
    ],
    columns=[
        ["12/2021", "12/2021", "01/2022", "01/2022", "02/2022", "02/2022"],
        ["latest", "stable", "latest", "stable", "latest", "stable"],
    ],
)

df
```

[37]:

		12/2021		01/2022		02/2022	
		latest	stable	latest	stable	latest	stable
Jupyter Tutorial	de	19651	0	30134	0	33295	0
	en	4722	1825	3497	2576	4009	3707
PyViz Tutorial	de	2573	0	4873	0	3930	0
Python Basics	de	525	0	427	0	276	0
	en	157	0	85	0	226	0

Die Hierarchieebenen können Namen haben (als Zeichenketten oder beliebige Python-Objekte). Wenn dies der Fall ist, werden diese in der Konsolenausgabe angezeigt:

[38]:
```
df.index.names = ["Title", "Language"]
df.columns.names = ["Month", "Version"]

df
```

[38]:

Month		12/2021		01/2022		02/2022	
Version		latest	stable	latest	stable	latest	stable
Title	Language						
Jupyter Tutorial	de	19651	0	30134	0	33295	0
	en	4722	1825	3497	2576	4009	3707
PyViz Tutorial	de	2573	0	4873	0	3930	0
Python Basics	de	525	0	427	0	276	0
	en	157	0	85	0	226	0

Warnung

Achtet darauf, dass die Indexnamen `Month` und `Version` nicht Teil der Zeilenbezeichnungen (der `df.index`-Werte)

sind.

Mit pandas.MultiIndex.from_arrays kann selbst ein `MultiIndex` erstellt und dann wiederverwendet werden; die Spalten im vorangehenden DataFrame mit Ebenennamen könnten so erstellt werden:

[39]:
```
pd.MultiIndex.from_arrays(
    [
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "de", "en"],
    ],
    names=["Title", "Language"],
)
```

[39]:
```
MultiIndex([('Jupyter Tutorial', 'de'),
            ('Jupyter Tutorial', 'en'),
            (  'PyViz Tutorial', 'de'),
            (   'Python Basics', 'de'),
            (   'Python Basics', 'en')],
           names=['Title', 'Language'])
```

Mit der Teilspaltenindizierung könnt ihr auf ähnliche Weise Spaltengruppen auswählen:

[40]:
```
df.loc["Jupyter Tutorial"]
```

[40]:
```
Month    12/2021        01/2022        02/2022
Version   latest stable  latest stable  latest stable
Language
de          19651      0   30134      0   33295      0
en           4722   1825    3497   2576    4009   3707
```

[41]:
```
df.loc["PyViz Tutorial":"Python Basics"]
```

[41]:
```
Month                   12/2021        01/2022        02/2022
Version                  latest stable  latest stable  latest stable
Title          Language
PyViz Tutorial de          2573      0    4873      0    3930      0
Python Basics  de           525      0     427      0     276      0
               en           157      0      85      0     226      0
```

`loc` mit Liste von Tupeln:

[42]:
```
df.loc[[("Jupyter Tutorial", "de"), ("PyViz Tutorial", "de")]]
```

[42]:
```
Month                     12/2021        01/2022        02/2022
Version                    latest stable  latest stable  latest stable
Title            Language
Jupyter Tutorial de          19651      0   30134      0   33295      0
PyViz Tutorial   de           2573      0    4873      0    3930      0
```

`loc` mit Liste des zweiten Spaltenindex:

[43]:
```
df.loc[:, ["de"], :]
```

[43]:
```
Month                    12/2021        01/2022        02/2022
Version                   latest stable  latest stable  latest stable
Title            Language
Jupyter Tutorial de         19651      0   30134      0   33295      0
PyViz Tutorial   de          2573      0    4873      0    3930      0
Python Basics    de           525      0     427      0     276      0
```

Verwenden von `slice`:

[44]:
```
df.loc[slice("Jupyter Tutorial"), :]
```

[44]:
```
Month                    12/2021        01/2022        02/2022
Version                   latest stable  latest stable  latest stable
Title            Language
Jupyter Tutorial de         19651      0   30134      0   33295      0
                 en          4722   1825    3497   2576    4009   3707
```

[45]:
```
df.loc[slice("Jupyter Tutorial", "PyViz Tutorial"), :]
```

[45]:
```
Month                    12/2021        01/2022        02/2022
Version                   latest stable  latest stable  latest stable
Title            Language
Jupyter Tutorial de         19651      0   30134      0   33295      0
                 en          4722   1825    3497   2576    4009   3707
PyViz Tutorial   de          2573      0    4873      0    3930      0
```

[46]:
```
df.loc[slice("Jupyter Tutorial", "PyViz Tutorial"), "12/2021"]
```

[46]:
```
Version                   latest  stable
Title            Language
Jupyter Tutorial de         19651       0
                 en          4722    1825
PyViz Tutorial   de          2573       0
```

Verwenden von `slice`, und Listen:

[47]:
```
df.loc[slice("Jupyter Tutorial", "PyViz Tutorial"), ["de"], :]
```

[47]:
```
Month                    12/2021        01/2022        02/2022
Version                   latest stable  latest stable  latest stable
Title            Language
Jupyter Tutorial de         19651      0   30134      0   33295      0
PyViz Tutorial   de          2573      0    4873      0    3930      0
```

Ansicht vs. Kopie

In Pandas hängt es von der Struktur und den Datentypen des ursprünglichen DataFrame ab, ob ihr einen View erhaltet oder nicht – und ob Änderungen, die an einem View vorgenommen werden, in den ursprünglichen DataFrame zurück übertragen werden.

`stack` und `unstack`

Die hierarchische Indizierung spielt eine wichtige Rolle bei der Umformung von Daten und gruppenbasierten Operationen wie der Bildung einer Pivot-Tabelle. Zum Beispiel könnt ihr diese Daten mit der pandas.Series.unstack-Methode in einen DataFrame umordnen:

```
[48]: hits.unstack()
```

```
[48]:                       de       en
      Jupyter Tutorial  83080.0  20336.0
      PyViz Tutorial    11376.0      NaN
      Python Basics      1228.0    468.0
```

Die umgekehrte Operation von unstack ist stack:

```
[49]: hits.unstack().stack()
```

```
[49]: Jupyter Tutorial  de    83080.0
                        en    20336.0
      PyViz Tutorial    de    11376.0
      Python Basics     de     1228.0
                        en      468.0
      dtype: float64
```

Umordnen und Sortieren von Ebenen

Es kann vorkommen, dass ihr die Reihenfolge der Ebenen auf einer Achse neu anordnen oder die Daten nach den Werten in einer bestimmten Ebene sortieren wollt. Die Funktion pandas.DataFrame.swaplevel nimmt zwei Ebenennummern oder -namen entgegen und gibt ein neues Objekt zurück, in dem die Ebenen vertauscht sind (die Daten bleiben jedoch unverändert):

```
[50]: df.swaplevel("Language", "Title")
```

```
[50]: Month                     12/2021        01/2022        02/2022
      Version                    latest stable  latest stable  latest stable
      Language Title
      de       Jupyter Tutorial   19651      0   30134      0   33295      0
      en       Jupyter Tutorial    4722   1825    3497   2576    4009   3707
      de       PyViz Tutorial      2573      0    4873      0    3930      0
               Python Basics        525      0     427      0     276      0
      en       Python Basics        157      0      85      0     226      0
```

pandas.DataFrame.sort_index hingegen sortiert die Daten nur nach den Werten in einer einzigen Ebene. Beim Vertauschen von Ebenen ist es nicht unüblich, auch `sort_index` zu verwenden, damit das Ergebnis lexikografisch nach der angegebenen Ebene sortiert wird:

```
[51]: df.sort_index(level=0)
```

[51]:
```
Month                     12/2021        01/2022        02/2022
Version                    latest stable  latest stable  latest stable
Title            Language
Jupyter Tutorial de          19651      0   30134      0   33295      0
                 en           4722   1825    3497   2576    4009   3707
PyViz Tutorial   de           2573      0    4873      0    3930      0
Python Basics    de            525      0     427      0     276      0
                 en            157      0      85      0     226      0
```

[52]:
```
df.swaplevel(0, 1).sort_index(level=0)
```

[52]:
```
Month                     12/2021        01/2022        02/2022
Version                    latest stable  latest stable  latest stable
Language Title
de       Jupyter Tutorial   19651      0   30134      0   33295      0
         PyViz Tutorial      2573      0    4873      0    3930      0
         Python Basics        525      0     427      0     276      0
en       Jupyter Tutorial    4722   1825    3497   2576    4009   3707
         Python Basics        157      0      85      0     226      0
```

Hinweis

Die Leistung der Datenauswahl ist bei hierarchisch indizierten Objekten wesentlich besser, wenn der Index lexikografisch sortiert ist, beginnend mit der äußersten Ebene, d.h. dem Ergebnis des Aufrufs von `sort_index(level=0)` oder `sort_index()`.

Zusammenfassende Statistiken nach Ebene

Viele deskriptive und zusammenfassende Statistiken für `DataFrame` und `Series` verfügen über eine Ebenenoption, mit der die Ebene angeben können, nach der ihr auf einer bestimmten Achse aggregieren könnt. Betrachtet den obigen `DataFrame`; wir können entweder die Zeilen oder die Spalten nach der Ebene aggregieren wie folgt:

[53]:
```
df.groupby(level="Language").sum()
```

[53]:
```
Month    12/2021        01/2022        02/2022
Version   latest stable  latest stable  latest stable
Language
de         22749      0   35434      0   37501      0
en          4879   1825    3582   2576    4235   3707
```

[54]:
```
df.groupby(level="Month", axis=1).sum()
```

[54]:
```
Month                     01/2022  02/2022  12/2021
Title            Language
Jupyter Tutorial de          30134    33295    19651
                 en           6073     7716     6547
PyViz Tutorial   de           4873     3930     2573
Python Basics    de            427      276      525
                 en             85      226      157
```

Intern wird dazu die pandas.DataFrame.groupby-Maschinerie von Pandas verwendet, die in *Gruppenoperationen* näher erläutert wird.

Indizierung mit den Spalten eines DataFrame

Es ist nicht ungewöhnlich, eine oder mehrere Spalten eines DataFrame als Zeilenindex zu verwenden; alternativ könnt ihr den Zeilenindex auch in die Spalten des DataFrame verschieben. Hier ist ein Beispiel-DataFrame:

[55]:
```
data = [
    ["Jupyter Tutorial", "de", 19651, 0, 30134, 0, 33295, 0],
    ["Jupyter Tutorial", "en", 4722, 1825, 3497, 2576, 4009, 3707],
    ["PyViz Tutorial", "de", 2573, 0, 4873, 0, 3930, 0],
    ["Python Basics", "de", 525, 0, 427, 0, 276, 0],
    ["Python Basics", "en", 157, 0, 85, 0, 226, 0],
]

df = pd.DataFrame(data)

df
```

[55]:

	0	1	2	3	4	5	6	7
0	Jupyter Tutorial	de	19651	0	30134	0	33295	0
1	Jupyter Tutorial	en	4722	1825	3497	2576	4009	3707
2	PyViz Tutorial	de	2573	0	4873	0	3930	0
3	Python Basics	de	525	0	427	0	276	0
4	Python Basics	en	157	0	85	0	226	0

Die Funktion pandas.DataFrame.set_index erstellt einen neuen DataFrame, der eine oder mehrere seiner Spalten als Index verwendet:

[56]:
```
df2 = df.set_index([0, 1])

df2
```

[56]:

		2	3	4	5	6	7
0	1						
Jupyter Tutorial	de	19651	0	30134	0	33295	0
	en	4722	1825	3497	2576	4009	3707
PyViz Tutorial	de	2573	0	4873	0	3930	0
Python Basics	de	525	0	427	0	276	0
	en	157	0	85	0	226	0

Standardmäßig werden die Spalten aus dem DataFrame entfernt, Ihr könnt sie aber auch drin lassen, indem ihr `drop=False` an `set_index` übergebt:

[57]:
```
df.set_index([0, 1], drop=False)
```

[57]:

		0	1	2	3	4	5	6	\
0	1								
Jupyter Tutorial	de	Jupyter Tutorial	de	19651	0	30134	0	33295	
	en	Jupyter Tutorial	en	4722	1825	3497	2576	4009	
PyViz Tutorial	de	PyViz Tutorial	de	2573	0	4873	0	3930	
Python Basics	de	Python Basics	de	525	0	427	0	276	
	en	Python Basics	en	157	0	85	0	226	

		7
0	1	
Jupyter Tutorial	de	0

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                en  3707
PyViz Tutorial  de     0
Python Basics   de     0
                en     0
```

pandas.DataFrame.reset_index hingegen bewirkt das Gegenteil von `set_index`; die hierarchischen Indexebenen werden in die Spalten verschoben:

[58]:
```
df2.reset_index()
```

[58]:

	0	1	2	3	4	5	6	7
0	Jupyter Tutorial	de	19651	0	30134	0	33295	0
1	Jupyter Tutorial	en	4722	1825	3497	2576	4009	3707
2	PyViz Tutorial	de	2573	0	4873	0	3930	0
3	Python Basics	de	525	0	427	0	276	0
4	Python Basics	en	157	0	85	0	226	0

2.4.4 Datum und Uhrzeit

Mit pandas könnt ihr `Series` mit Datums- und Zeitinformationen erstellen. Im folgenden werden wir häufige Operationen mit Datumsdaten zeigen.

Hinweis

pandas unterstützt Datumsangaben, die in UTC-Werten gespeichert sind und den Datentyp `datetime64[ns]` verwenden. Es werden auch lokale Zeiten aus einer einzigen Zeitzone unterstützt. Mehrere Zeitzonen werden durch ein pandas.Timestamp-Objekt unterstützt. Wenn ihr mit Zeiten aus mehreren Zeitzonen hantieren müsst, würde ich die Daten vermutlich nach Zeitzonen aufteilen und für jede Zeitzone einen eigenen `DataFrame` oder eine eigene `Series` verwenden.

Siehe auch

- Time series / date functionality

Laden von UTC-Zeitdaten

[1]:
```
import pandas as pd


dt = pd.date_range("2022-03-27", periods=6, freq="H")

dt
```

[1]:
```
DatetimeIndex(['2022-03-27 00:00:00', '2022-03-27 01:00:00',
               '2022-03-27 02:00:00', '2022-03-27 03:00:00',
               '2022-03-27 04:00:00', '2022-03-27 05:00:00'],
              dtype='datetime64[ns]', freq='H')
```

[2]:
```
utc = pd.to_datetime(dt, utc=True)

utc
```

```
[2]: DatetimeIndex(['2022-03-27 00:00:00+00:00', '2022-03-27 01:00:00+00:00',
                    '2022-03-27 02:00:00+00:00', '2022-03-27 03:00:00+00:00',
                    '2022-03-27 04:00:00+00:00', '2022-03-27 05:00:00+00:00'],
                   dtype='datetime64[ns, UTC]', freq='H')
```

Hinweis

Der Typ des Ergebnisses `dtype='datetime64[ns, UTC]'` zeigt an, dass die Daten als UTC gespeichert sind.

Konvertieren wir diese Reihe in die Zeitzone Europe/Berlin:

```
[3]: utc.tz_convert("Europe/Berlin")
```

```
[3]: DatetimeIndex(['2022-03-27 01:00:00+01:00', '2022-03-27 03:00:00+02:00',
                    '2022-03-27 04:00:00+02:00', '2022-03-27 05:00:00+02:00',
                    '2022-03-27 06:00:00+02:00', '2022-03-27 07:00:00+02:00'],
                   dtype='datetime64[ns, Europe/Berlin]', freq='H')
```

Umrechnung der Ortszeit in UTC

```
[4]: local = utc.tz_convert("Europe/Berlin")

     local.tz_convert("UTC")
```

```
[4]: DatetimeIndex(['2022-03-27 00:00:00+00:00', '2022-03-27 01:00:00+00:00',
                    '2022-03-27 02:00:00+00:00', '2022-03-27 03:00:00+00:00',
                    '2022-03-27 04:00:00+00:00', '2022-03-27 05:00:00+00:00'],
                   dtype='datetime64[ns, UTC]', freq='H')
```

Umrechnung in Unixzeit

Wenn ihr eine `Series` mit UTC- oder Ortszeitinformationen habt, könnt ihr mit diesem Code die Sekunden nach der Unixzeit ermitteln:

```
[5]: uts = pd.to_datetime(dt).view(int) / 10**9

     uts
```

```
[5]: array([1.6483392e+09, 1.6483428e+09, 1.6483464e+09, 1.6483500e+09,
            1.6483536e+09, 1.6483572e+09])
```

Um die Unixzeit in UTC zu laden, könnt ihr wie folgt vorgehen:

```
[6]: (pd.to_datetime(uts, unit='s').tz_localize("UTC"))
```

```
[6]: DatetimeIndex(['2022-03-27 00:00:00+00:00', '2022-03-27 01:00:00+00:00',
                    '2022-03-27 02:00:00+00:00', '2022-03-27 03:00:00+00:00',
                    '2022-03-27 04:00:00+00:00', '2022-03-27 05:00:00+00:00'],
                   dtype='datetime64[ns, UTC]', freq=None)
```

Manipulation von Terminen

Umwandeln in Strings

Mit pandas.DatetimeIndex habt ihr einige Möglichkeiten, Datum und Uhrzeit in Strings umzuwandeln, z.B. in den Namen des Wochentags:

[7]:
```
local.day_name(locale="de_DE.UTF-8")
```

[7]:
```
Index(['Sonntag', 'Sonntag', 'Sonntag', 'Sonntag', 'Sonntag', 'Sonntag'], dtype='object')
```

Welche `locale` bei euch zur Verfügung stehen, könnt ihr euch mit `locale -a` anzeigen lassen:

[8]:
```
!locale -a
```

```
en_NZ
nl_NL.UTF-8
pt_BR.UTF-8
fr_CH.ISO8859-15
eu_ES.ISO8859-15
en_US.US-ASCII
af_ZA
bg_BG
cs_CZ.UTF-8
fi_FI
zh_CN.UTF-8
eu_ES
sk_SK.ISO8859-2
nl_BE
fr_BE
sk_SK
en_US.UTF-8
en_NZ.ISO8859-1
de_CH
sk_SK.UTF-8
de_DE.UTF-8
am_ET.UTF-8
zh_HK
be_BY.UTF-8
uk_UA
pt_PT.ISO8859-1
en_AU.US-ASCII
kk_KZ.PT154
en_US
nl_BE.ISO8859-15
de_AT.ISO8859-1
hr_HR.ISO8859-2
fr_FR.ISO8859-1
af_ZA.UTF-8
am_ET
fi_FI.ISO8859-1
ro_RO.UTF-8
af_ZA.ISO8859-15
en_NZ.UTF-8
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
fi_FI.UTF-8
hr_HR.UTF-8
da_DK.UTF-8
ca_ES.ISO8859-1
en_AU.ISO8859-15
ro_RO.ISO8859-2
de_AT.UTF-8
pt_PT.ISO8859-15
sv_SE
fr_CA.ISO8859-1
fr_BE.ISO8859-1
en_US.ISO8859-15
it_CH.ISO8859-1
en_NZ.ISO8859-15
en_AU.UTF-8
de_AT.ISO8859-15
af_ZA.ISO8859-1
hu_HU.UTF-8
et_EE.UTF-8
he_IL.UTF-8
uk_UA.KOI8-U
be_BY
kk_KZ
hu_HU.ISO8859-2
it_CH
pt_BR
ko_KR
it_IT
fr_BE.UTF-8
ru_RU.ISO8859-5
zh_TW
zh_CN.GB2312
no_NO.ISO8859-15
de_DE.ISO8859-15
en_CA
fr_CH.UTF-8
sl_SI.UTF-8
uk_UA.ISO8859-5
pt_PT
hr_HR
cs_CZ
fr_CH
he_IL
zh_CN.GBK
zh_CN.GB18030
fr_CA
pl_PL.UTF-8
ja_JP.SJIS
sr_YU.ISO8859-5
be_BY.CP1251
sr_YU.ISO8859-2
sv_SE.UTF-8
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
sr_YU.UTF-8
de_CH.UTF-8
sl_SI
pt_PT.UTF-8
ro_RO
en_NZ.US-ASCII
ja_JP
zh_CN
fr_CH.ISO8859-1
ko_KR.eucKR
be_BY.ISO8859-5
nl_NL.ISO8859-15
en_GB.ISO8859-1
en_CA.US-ASCII
is_IS.ISO8859-1
ru_RU.CP866
nl_NL
fr_CA.ISO8859-15
sv_SE.ISO8859-15
hy_AM
en_CA.ISO8859-15
en_US.ISO8859-1
zh_TW.Big5
ca_ES.UTF-8
ru_RU.CP1251
en_GB.UTF-8
en_GB.US-ASCII
ru_RU.UTF-8
eu_ES.UTF-8
es_ES.ISO8859-1
hu_HU
el_GR.ISO8859-7
en_AU
it_CH.UTF-8
en_GB
sl_SI.ISO8859-2
ru_RU.KOI8-R
nl_BE.UTF-8
et_EE
fr_FR.ISO8859-15
cs_CZ.ISO8859-2
lt_LT.UTF-8
pl_PL.ISO8859-2
fr_BE.ISO8859-15
is_IS.UTF-8
tr_TR.ISO8859-9
da_DK.ISO8859-1
lt_LT.ISO8859-4
lt_LT.ISO8859-13
zh_TW.UTF-8
bg_BG.CP1251
el_GR.UTF-8
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
be_BY.CP1131
da_DK.ISO8859-15
is_IS.ISO8859-15
no_NO.ISO8859-1
nl_NL.ISO8859-1
nl_BE.ISO8859-1
sv_SE.ISO8859-1
pt_BR.ISO8859-1
zh_CN.eucCN
it_IT.UTF-8
en_CA.UTF-8
uk_UA.UTF-8
de_CH.ISO8859-15
de_DE.ISO8859-1
ca_ES
sr_YU
hy_AM.ARMSCII-8
ru_RU
zh_HK.UTF-8
eu_ES.ISO8859-1
is_IS
bg_BG.UTF-8
ja_JP.UTF-8
it_CH.ISO8859-15
fr_FR.UTF-8
ko_KR.UTF-8
et_EE.ISO8859-15
kk_KZ.UTF-8
ca_ES.ISO8859-15
en_IE.UTF-8
es_ES
de_CH.ISO8859-1
en_CA.ISO8859-1
es_ES.ISO8859-15
en_AU.ISO8859-1
el_GR
da_DK
no_NO
it_IT.ISO8859-1
en_IE
zh_HK.Big5HKSCS
hi_IN.ISCII-DEV
ja_JP.eucJP
it_IT.ISO8859-15
pl_PL
ko_KR.CP949
fr_CA.UTF-8
fi_FI.ISO8859-15
en_GB.ISO8859-15
fr_FR
hy_AM.UTF-8
no_NO.UTF-8
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
es_ES.UTF-8
de_AT
tr_TR.UTF-8
de_DE
lt_LT
tr_TR
C
POSIX
```

Weitere Attribute von `DatetimeIndex`, mit denen sich Datum und Zeit in Strings umwandeln lassen, sind:

Attribut	Beschreibung
`year`	das Jahr der `datetime`
`month`	der Monat als Januar 1 und Dezember 12
`day`	der Tag der `datetime`
`hour`	die Stunden der `datetime`
`minute`	die Minuten der `datetime`
`second`	die Sekunden der `datetime`
`microsecond`	die Mikrosekunden der `datetime`
`nanosecond`	die Nanosekunden von `datetime`
`date`	gibt ein Numpy-Array von Python `datetime.date`-Objekten zurück
`time`	gibt ein NumPy-Array von `datetime.time`-Objekten zurück
`timetz`	liefert ein numpy-Array von `datetime.time`-Objekten mit Zeitzoneninformationen
`dayofyear`, `day_of_year`	der ordinale Tag des Jahres
`dayofweek`	der Tag der Woche mit Montag (`0`) und Sonntag (`6`)
`day_of_week`	der Tag der Woche mit Montag (`0`) und Sonntag (`6`)
`weekday`	der Tag der Woche mit Montag (`0`) und Sonntag (`6`)
`quarter`	gibt das Jahresquartal zurück
`tz`	gibt die Zeitzone zurück
`freq`	gibt das Frequenzobjekt zurück, wenn es gesetzt ist, andernfalls `None`
`freqstr`	gibt das Frequenz-Objekt als String zurück, wenn es gesetzt ist, andernfalls `None`
`is_month_start`	zeigt an, ob das Datum der erste Tag des Monats ist
`is_month_end`	zeigt an, ob das Datum der letzte Tag des Monats ist
`is_quarter_start`	zeigt an, ob das Datum der erste Tag eines Quartals ist
`is_quarter_end`	zeigt an, ob das Datum der letzte Tag eines Quartals ist
`is_year_start`	zeigt an, ob das Datum der erste Tag eines Jahres ist
`is_year_end`	zeigt an, ob das Datum der letzte Tag eines Jahres ist
`is_leap_year`	Boolescher Indikator, ob das Datum in ein Schaltjahr fällt
`inferred_freq`	versucht, eine Zeichenkette zurückzugeben, die eine durch `infer_freq` ermittelte Frequenz darstellt

Es gibt jedoch auch einige Methoden, mit denen ihr den `DatetimeIndex` in Stringsumwandeln könnt, z.B. `strftime`:

```
[9]: local.strftime("%d.%m.%Y")
```

```
[9]: Index(['27.03.2022', '27.03.2022', '27.03.2022', '27.03.2022', '27.03.2022',
       '27.03.2022'],
      dtype='object')
```

Hinweis

In strftime() and strptime() Format Codes erhaltet ihr eine Übersicht über die verschiedenen Formattierungsmöglichkeiten von `strftime`.

Weitere Methoden sind:

Methode	Beschreibung
`normalize`	konvertiert Zeiten in Mitternacht
`strftime`	konvertiert in Index unter Verwendung des angegebenen Datumsformats
`snap`	rastet den Zeitstempel auf der nächsten auftretenden Frequenz ein
`tz_convert`	konvertiert ein `tz`-fähiges Datetime-Array/Index von einer Zeitzone in eine andere
`tz_localiz`	Lokalisiert `tz`-naives Datetime Array/Index in `tz`-kompatibles Datetime Array/Index
`round`	rundet die Daten zur nächsten angegebenen Frequenz
`floor`	rundet die Daten ab auf die angegebene Frequenz
`ceil`	rundet die Daten auf auf die angegebene Frequenz
`to_period`	wandelt die Daten um in einen PeriodArray/Index bei einer bestimmten Frequenz
`to_periodd`	Berechnet `TimedeltaArray` der Differenz zwischen den Indexwerten und dem in `PeriodArray` umgewandelten Index bei der angegebenen Frequenz
`to_pydatet`	gibt `Datetime` Array/Index als `ndarray`-Objekt von `datetime.datetime`-Objekten zurück
`to_series`	erzeugt eine `Series` mit Index und Werten, die den Indexschlüsseln entsprechen; nützlich mit `map` für die Rückgabe eines Indexers
`to_frame`	erzeugt einen `DataFrame` mit einer Spalte, die den Index enthält
`month_name`	gibt die Monatsnamen des `DateTimeIndex` mit dem angegebenen `locale` zurück
`day_name`	gibt die Tagesnamen des `DateTimeIndex` mit dem angegebenen `locale` zurück
`mean`	gibt den Mittelwert des Arrays zurück
`std`	gibt die Standardabweichung der Stichprobe über die angeforderte Achse zurück

2.4.5 Daten auswählen und filtern

Die Indizierung von Serien (`obj[...]`) funktioniert analog zur Indizierung von NumPy-Arrays, außer dass ihr Indexwerte der Serie statt nur Ganzzahlen verwenden könnt. Hier sind einige Beispiele dafür:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
idx = pd.date_range("2022-02-02", periods=7)
s = pd.Series(np.random.randn(7), index=idx)
```

[3]:
```
s
```

[3]:
```
2022-02-02    0.647976
2022-02-03   -0.456759
2022-02-04    1.645400
2022-02-05   -0.327128
2022-02-06    0.289596
2022-02-07   -0.443900
2022-02-08    0.688924
Freq: D, dtype: float64
```

[4]:
```
s["2022-02-03"]
```

[4]:
```
-0.456758843961271
```

```
[5]: s[1]
```

```
[5]: -0.456758843961271
```

```
[6]: s[2:4]
```

```
[6]: 2022-02-04    1.645400
     2022-02-05   -0.327128
     Freq: D, dtype: float64
```

```
[7]: s[["2022-02-04", "2022-02-03", "2022-02-02"]]
```

```
[7]: 2022-02-04    1.645400
     2022-02-03   -0.456759
     2022-02-02    0.647976
     dtype: float64
```

```
[8]: s[[1, 3]]
```

```
[8]: 2022-02-03   -0.456759
     2022-02-05   -0.327128
     Freq: 2D, dtype: float64
```

```
[9]: s[s > 0]
```

```
[9]: 2022-02-02    0.647976
     2022-02-04    1.645400
     2022-02-06    0.289596
     2022-02-08    0.688924
     dtype: float64
```

Zwar könnt ihr auf diese Weise Daten nach Label auswählen, doch die bevorzugte Methode zur Auswahl von Indexwerten ist der `loc`-Operator:

```
[10]: s.loc[["2022-02-04", "2022-02-03", "2022-02-02"]]
```

```
[10]: 2022-02-04    1.645400
      2022-02-03   -0.456759
      2022-02-02    0.647976
      dtype: float64
```

Der Grund für die Bevorzugung von `loc` liegt in der unterschiedlichen Behandlung von Ganzzahlen bei der Indexierung mit `[]`. Bei der regulären `[]`-basierten Indizierung werden Ganzzahlen als Label behandelt, wenn der Index Ganzzahlen enthält, so dass das Verhalten je nach Datentyp des Index unterschiedlich ist. In unserem Beispiel wird der Ausdruck `s.loc[[3, 2, 1]]` fehlschlagen, da der Index keine ganzen Zahlen enthält:

```
[11]: s.loc[[3, 2, 1]]
```

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[11], line 1
----> 1 s.loc[[3, 2, 1]]

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexing.py:1103, in _LocationIndexer.__getitem__(self, key)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
   1100 axis = self.axis or 0
   1102 maybe_callable = com.apply_if_callable(key, self.obj)
-> 1103 return self._getitem_axis(maybe_callable, axis=axis)

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexing.py:1332, in _LocIndexer._getitem_axis(self, key, axis)
   1329     if hasattr(key, "ndim") and key.ndim > 1:
   1330         raise ValueError("Cannot index with multidimensional key")
-> 1332     return self._getitem_iterable(key, axis=axis)
   1334 # nested tuple slicing
   1335 if is_nested_tuple(key, labels):

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexing.py:1272, in _LocIndexer._getitem_iterable(self, key, axis)
   1269 self._validate_key(key, axis)
   1271 # A collection of keys
-> 1272 keyarr, indexer = self._get_listlike_indexer(key, axis)
   1273 return self.obj._reindex_with_indexers(
   1274     {axis: [keyarr, indexer]}, copy=True, allow_dups=True
   1275 )

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexing.py:1462, in _LocIndexer._get_listlike_indexer(self, key, axis)
   1459 ax = self.obj._get_axis(axis)
   1460 axis_name = self.obj._get_axis_name(axis)
-> 1462 keyarr, indexer = ax._get_indexer_strict(key, axis_name)
   1464 return keyarr, indexer

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexes/base.py:5877, in Index._get_indexer_strict(self, key, axis_name)
   5874 else:
   5875     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 5877 self._raise_if_missing(keyarr, indexer, axis_name)
   5879 keyarr = self.take(indexer)
   5880 if isinstance(key, Index):
   5881     # GH 42790 - Preserve name from an Index

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/
↪core/indexes/base.py:5938, in Index._raise_if_missing(self, key, indexer, axis_name)
   5936     if use_interval_msg:
   5937         key = list(key)
-> 5938     raise KeyError(f"None of [{key}] are in the [{axis_name}]")
   5940 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
   5941 raise KeyError(f"{not_found} not in index")

KeyError: "None of [Index([3, 2, 1], dtype='int64')] are in the [index]"
```

Während der `loc`-Operator ausschließlich Label indiziert, indiziert der `iloc`-Operator ausschließlich mit ganzen Zahlen:

[12]:
```
s.iloc[[3, 2, 1]]
```

[12]:
```
2022-02-05   -0.327128
2022-02-04    1.645400
2022-02-03   -0.456759
Freq: -1D, dtype: float64
```

Ihr könnt auch mit Labels slicen, aber das funktioniert anders als das normale Python-Slicing, da der Endpunkt inklusive ist:

[13]:
```
s.loc["2022-02-03":"2022-02-04"]
```

[13]:
```
2022-02-03   -0.456759
2022-02-04    1.645400
Freq: D, dtype: float64
```

Durch die Einstellung mit diesen Methoden wird der entsprechende Abschnitt der Reihe geändert:

[14]:
```
s.loc["2022-02-03":"2022-02-04"] = 0

s
```

[14]:
```
2022-02-02    0.647976
2022-02-03    0.000000
2022-02-04    0.000000
2022-02-05   -0.327128
2022-02-06    0.289596
2022-02-07   -0.443900
2022-02-08    0.688924
Freq: D, dtype: float64
```

Die Indizierung in einem DataFrame dient dazu, eine oder mehrere Spalten entweder mit einem einzelnen Wert oder einer Folge abzurufen:

[15]:
```
data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
    "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
}

df = pd.DataFrame(data)
df = pd.DataFrame(data, columns=["Decimal", "Octal", "Key"], index=df["Code"])

df
```

[15]:

Code	Decimal	Octal	Key
U+0000	0	001	NUL
U+0001	1	002	Ctrl-A
U+0002	2	003	Ctrl-B
U+0003	3	004	Ctrl-C
U+0004	4	004	Ctrl-D
U+0005	5	005	Ctrl-E

[16]:
```
df["Key"]
```

[16]:
```
Code
U+0000       NUL
U+0001    Ctrl-A
U+0002    Ctrl-B
U+0003    Ctrl-C
U+0004    Ctrl-D
U+0005    Ctrl-E
Name: Key, dtype: object
```

[17]:
```
df[["Decimal", "Key"]]
```

[17]:
```
        Decimal     Key
Code
U+0000        0     NUL
U+0001        1  Ctrl-A
U+0002        2  Ctrl-B
U+0003        3  Ctrl-C
U+0004        4  Ctrl-D
U+0005        5  Ctrl-E
```

[18]:
```
df[:2]
```

[18]:
```
        Decimal Octal     Key
Code
U+0000        0   001     NUL
U+0001        1   002  Ctrl-A
```

[19]:
```
df[df["Decimal"] > 2]
```

[19]:
```
        Decimal Octal     Key
Code
U+0003        3   004  Ctrl-C
U+0004        4   004  Ctrl-D
U+0005        5   005  Ctrl-E
```

Die Zeilenauswahlsyntax `df[:2]` wird aus Gründen der Bequemlichkeit bereitgestellt. Durch die Übergabe eines einzelnen Elements oder einer Liste an den `[]`-Operator werden Spalten ausgewählt.

Ein weiterer Anwendungsfall ist die Indizierung mit einem booleschen DataFrame, der beispielsweise durch einen Skalarvergleich erzeugt wird:

[20]:
```
df["Decimal"] > 2
```

[20]:
```
Code
U+0000    False
U+0001    False
U+0002    False
U+0003     True
U+0004     True
U+0005     True
Name: Decimal, dtype: bool
```

[21]:
```
df[df["Decimal"] > 2] = "NA"

df
```

[21]:
```
       Decimal Octal     Key
Code
U+0000       0   001     NUL
U+0001       1   002  Ctrl-A
U+0002       2   003  Ctrl-B
U+0003      NA    NA      NA
U+0004      NA    NA      NA
U+0005      NA    NA      NA
```

Wie Series verfügt auch DataFrame über spezielle Operatoren `loc` und `iloc` für label-basierte bzw. ganzzahlige Indizierung. Da DataFrame zweidimensional ist, könnt ihr eine Teilmenge der Zeilen und Spalten mit NumPy-ähnlicher Notation auswählen, indem ihr entweder Achsenbeschriftungen (`loc`) oder Ganzzahlen (`iloc`) verwendet.

[22]:
```
df.loc["U+0002", ["Decimal", "Key"]]
```

[22]:
```
Decimal         2
Key        Ctrl-B
Name: U+0002, dtype: object
```

[23]:
```
df.iloc[[2], [1, 2]]
```

[23]:
```
       Octal     Key
Code
U+0002   003  Ctrl-B
```

[24]:
```
df.iloc[[0, 1], [1, 2]]
```

[24]:
```
       Octal     Key
Code
U+0000   001     NUL
U+0001   002  Ctrl-A
```

Beide Indizierungsfunktionen arbeiten mit Slices zusätzlich zu einzelnen Label oder Listen von Label:

[25]:
```
df.loc[:"U+0003", "Key"]
```

[25]:
```
Code
U+0000       NUL
U+0001    Ctrl-A
U+0002    Ctrl-B
U+0003        NA
Name: Key, dtype: object
```

[26]:
```
df.iloc[:3, :3]
```

[26]:
```
       Decimal Octal     Key
Code
U+0000       0   001     NUL
U+0001       1   002  Ctrl-A
U+0002       2   003  Ctrl-B
```

Es gibt also viele Möglichkeiten, die in einem pandas-Objekt enthaltenen Daten auszuwählen und neu anzuordnen. Im folgenden stelle ich für DataFrames eine kurze Zusammenfassung der meisten dieser Möglichkeiten zusammen:

Typ	Hinweis
`df[LABEL]`	wählt eine einzelne Spalte oder eine Folge von Spalten aus dem DataFrame aus
`df.loc[LABEL]`	wählt eine einzelne Zeile oder eine Teilmenge von Zeilen aus dem DataFrame nach Label aus
`df.loc[:, LABEL]`	wählt eine einzelne Spalte oder eine Teilmenge von Spalten nach dem Label aus
`df.loc[LABEL1, LABEL2]`	wählt sowohl Zeilen als auch Spalten nach dem Label aus
`df.iloc[INTEGER]`	wählt eine einzelne Zeile oder eine Teilmenge von Zeilen aus dem DataFrame anhand der Ganzzahlposition aus
`df.iloc[INTEGER1, INTEGER2]`	Wählt eine einzelne Spalte oder eine Teilmenge von Spalten anhand einer ganzzahligen Position aus
`df.at[LABEL1, LABEL2]`	wählt einen Einzelwert nach Zeilen- und Spaltenbezeichnung aus
`df.iat[INTEGER1, INTEGER2]`	wählt einen Einzelwert nach Zeilen- und Spaltenposition (Ganzzahlen) aus
`reindex` NEW_INDEX	wählt Zeilen oder Spalten nach Labels aus
`get_value, set_value`	veraltet seit Version 0.21.0: Verwendet stattdessen `.at[]` oder `.iat[]`.

2.4.6 Hinzufügen, Ändern und Löschen von Daten

Bei vielen Datensätzen möchtet ihr vielleicht eine Transformation basierend auf den Werten in einem Array, einer Serie oder einer Spalte in einem DataFrame durchführen. Hierfür betrachten wir die ersten Unicode-Zeichen:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.DataFrame(
    {
        "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
        "Decimal": [0, 1, 2, 3, 4, 5],
        "Octal": ["001", "002", "003", "004", "004", "005"],
        "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
    }
)

df
```

[2]:

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

Daten hinzufügen

Angenommen, ihr möchtet eine Spalte hinzufügen, in der die Zeichen dem C0- oder C1-Steuercode zugewiesen werden:

[3]:
```
control_code = {
    "u+0000": "C0",
    "u+0001": "C0",
    "u+0002": "C0",
    "u+0003": "C0",
    "u+0004": "C0",
    "u+0005": "C0",
}
```

Die map-Methode für eine Serie akzeptiert eine Funktion oder ein diktatähnliches Objekt, das eine Zuordnung enthält, aber hier haben wir ein kleines Problem, da einige die Codes in `control_code` kleingeschrieben sind, nicht jedoch in unserem DataFrame. Daher müssen wir jeden Wert mit der Methode `str.lower` in Kleinbuchstaben umwandeln:

[4]:
```
lowercased = df["Code"].str.lower()

lowercased
```

[4]:
```
0    u+0000
1    u+0001
2    u+0002
3    u+0003
4    u+0004
5    u+0005
Name: Code, dtype: object
```

[5]:
```
df["Control code"] = lowercased.map(control_code)

df
```

[5]:

	Code	Decimal	Octal	Key	Control code
0	U+0000	0	001	NUL	C0
1	U+0001	1	002	Ctrl-A	C0
2	U+0002	2	003	Ctrl-B	C0
3	U+0003	3	004	Ctrl-C	C0
4	U+0004	4	004	Ctrl-D	C0
5	U+0005	5	005	Ctrl-E	C0

Wir hätten auch eine Funktion übergeben können, die die ganze Arbeit erledigt:

[6]:
```
df["Code"].map(lambda x: control_code[x.lower()])
```

[6]:
```
0    C0
1    C0
2    C0
3    C0
4    C0
5    C0
Name: Code, dtype: object
```

Die Verwendung von map ist ein bequemer Weg, um elementweise Transformationen und andere Datenbereinigungsoperationen durchzuführen.

Daten ändern

Hinweis

Das Ersetzen fehlender Werte wird in *Verwalten fehlender Daten mit pandas* beschrieben.

```
[7]: pd.Series(["Manpower", "man-made"]).str.replace("Man", "Personal", regex=False)
```

```
[7]: 0    Personalpower
     1         man-made
     dtype: object
```

```
[8]: pd.Series(["Man-Power", "man-made"]).str.replace(
         "[Mm]an", "Personal", regex=True
     )
```

```
[8]: 0    Personal-Power
     1     Personal-made
     dtype: object
```

Hinweis

Die Methode replace unterscheidet sich von str.replace, dadurch, dass diese elementweise Zeichenketten ersetzt.

Daten löschen

Einen oder mehrere Einträge aus einer Achse zu löschen ist einfach, wenn ihr bereits ein Index-Array oder eine Liste ohne diese Einträge habt.

Zum Löschen von Duplikaten siehe *Daten deduplizieren*.

Da dies ein wenig Mengenlehre erfordern kann, geben wir die Drop-Methode als neues Objekt ohne den oder die gelöschten Werten zurück:

```
[9]: s = pd.Series(np.random.randn(7))

     s
```

```
[9]: 0   -1.200837
     1    2.444208
     2   -0.948290
     3   -1.409449
     4   -2.220925
     5    0.494626
     6    0.589888
     dtype: float64
```

```
[10]: new = s.drop(2)

      new
```

```
[10]: 0   -1.200837
      1    2.444208
      3   -1.409449
      4   -2.220925
      5    0.494626
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
6    0.589888
dtype: float64
```

[11]:
```
new = s.drop([2, 3])

new
```

[11]:
```
0   -1.200837
1    2.444208
4   -2.220925
5    0.494626
6    0.589888
dtype: float64
```

Bei DataFrames können Indexwerte auf beiden Achsen gelöscht werden. Um dies zu veranschaulichen, erstellen wir zunächst einen Beispiel-DataFrame:

[12]:
```
data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Decimal": [0, 1, 2, 3, 4, 5],
    "Octal": ["001", "002", "003", "004", "004", "005"],
    "Key": ["NUL", "Ctrl-A", "Ctrl-B", "Ctrl-C", "Ctrl-D", "Ctrl-E"],
}

df = pd.DataFrame(data)

df
```

[12]:

	Code	Decimal	Octal	Key
0	U+0000	0	001	NUL
1	U+0001	1	002	Ctrl-A
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

[13]:
```
df.drop([0, 1])
```

[13]:

	Code	Decimal	Octal	Key
2	U+0002	2	003	Ctrl-B
3	U+0003	3	004	Ctrl-C
4	U+0004	4	004	Ctrl-D
5	U+0005	5	005	Ctrl-E

Ihr könnt auch Werte aus den Spalten entfernen, indem ihr `axis=1` oder `axis='columns'` übergebt:

[14]:
```
df.drop("Decimal", axis=1)
```

[14]:

	Code	Octal	Key
0	U+0000	001	NUL
1	U+0001	002	Ctrl-A
2	U+0002	003	Ctrl-B
3	U+0003	004	Ctrl-C

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
4  U+0004   004  Ctrl-D
5  U+0005   005  Ctrl-E
```

Viele Funktionen wie `drop`, die die Größe oder Form einer Reihe oder eines DataFrame ändern, können ein Objekt an Ort und Stelle manipulieren, ohne ein neues Objekt zurückzugeben:

[15]:
```
df.drop(0, inplace=True)

df
```

[15]:
```
     Code  Decimal Octal     Key
1  U+0001        1   002  Ctrl-A
2  U+0002        2   003  Ctrl-B
3  U+0003        3   004  Ctrl-C
4  U+0004        4   004  Ctrl-D
5  U+0005        5   005  Ctrl-E
```

Warnung

Seid vorsichtig mit der `inplace`-Funktion, da die Daten unwiderbringlich gelöscht werden.

2.4.7 Manipulation von Zeichenketten

pandas bietet die Möglichkeit, String-Methoden und reguläre Ausdrücke von Python prägnant auf ganze Arrays von Daten anzuwenden.

See also

- string
- re

Vektorisierte String-Funktionen in pandas

Das Aufräumen eines unübersichtlichen Datensatzes für die Analyse erfordert oft eine Menge an String-Manipulationen. Erschwerend kommt hinzu, dass eine Spalte, die Strings enthält, manchmal fehlende Daten enthält:

[1]:
```
import numpy as np
import pandas as pd


addresses = {
    "Veit": np.nan,
    "Veit Schiele": "veit.schiele@cusy.io",
    "cusy GmbH": "info@cusy.io",
}
addresses = pd.Series(addresses)

addresses
```

[1]:
```
Veit                           NaN
Veit Schiele    veit.schiele@cusy.io
cusy GmbH              info@cusy.io
dtype: object
```

[2]:
```
addresses.isna()
```

[2]:
```
Veit             True
Veit Schiele    False
cusy GmbH       False
dtype: bool
```

Ihr könnt Methoden für Zeichenketten und reguläre Ausdrücke auf jeden Wert anwenden (durch Übergabe eines Lambdas oder einer anderen Funktion), indem ihr `data.map` verwendet, aber dies schlägt bei `NA`-Werten fehl. Um dies zu bewältigen, verfügt `Series` über array-orientierte Methoden für String-Operationen, die `NA`-Werte überspringen und weiterleiten. Auf diese wird über das `str`-Attribut von `Series` zugegriffen; zum Beispiel könnten wir mit `str.contains` prüfen, ob jede E-Mail-Adresse `veit` enthält:

[3]:
```
addresses.str.contains("veit")
```

[3]:
```
Veit              NaN
Veit Schiele     True
cusy GmbH       False
dtype: object
```

Reguläre Ausdrücke können ebenfalls verwendet werden, zusammen mit Optionen wie `IGNORECASE`:

[4]:
```
import re


pattern = r"([A-Z0-9._%+-]+)@([A-Z0-9.-]+)\.([A-Z]{2,4})"

addresses.str.findall(pattern, flags=re.IGNORECASE)
```

[4]:
```
Veit                                   NaN
Veit Schiele    [(veit.schiele, cusy, io)]
cusy GmbH               [(info, cusy, io)]
dtype: object
```

Es gibt mehrere Möglichkeiten, ein vektorisiertes Element abzurufen. Entweder verwendet ihr `str.get` oder den Index von `str`:

[5]:
```
matches = addresses.str.findall(pattern, flags=re.IGNORECASE).str[0]

matches
```

[5]:
```
Veit                                 NaN
Veit Schiele    (veit.schiele, cusy, io)
cusy GmbH               (info, cusy, io)
dtype: object
```

[6]:
```
matches.str.get(1)
```

[6]:
```
Veit             NaN
Veit Schiele    cusy
cusy GmbH       cusy
dtype: object
```

In ähnlicher Weise könnt ihr mit dieser Syntax auch Zeichenketten zerschneiden:

[7]:
```
addresses.str[:5]
```

[7]:
```
Veit              NaN
Veit Schiele    veit.
cusy GmbH       info@
dtype: object
```

Die pandas.Series.str.extract-Methode gibt die erfassten Gruppen eines regulären Ausdrucks als DataFrame zurück:

[8]:
```
addresses.str.extract(pattern, flags=re.IGNORECASE)
```

[8]:
```
                        0     1    2
Veit                   NaN   NaN  NaN
Veit Schiele  veit.schiele  cusy   io
cusy GmbH             info  cusy   io
```

Weitere vektorisierten Pandas-String-Methoden:

Methode	Beschreibung
`cat`	verknüpft Zeichenketten elementweise mit optionalem Trennzeichen
`contain`	gibt ein boolesches Array zurück, wenn jede Zeichenkette ein Muster/Regex enthält
`count`	zählt Vorkommen des Musters
`extract`	verwendet einen regulären Ausdruck mit Gruppen, um eine oder mehrere Zeichenketten aus einer Reihe von Zeichenketten zu extrahieren; das Ergebnis ist ein DataFrame mit einer Spalte pro Gruppe
`endswit`	Äquivalent zu `x.endswith(pattern)` für jedes Element
`startsw`	Äquivalent zu `x.startswith(pattern)` für jedes Element
`findall`	berechnet Liste aller Vorkommen von Muster/Regex für jede Zeichenkette
`get`	Index in jedem Element (`i`-tes Element abrufen)
`isalnum`	Äquivalent zu eingebautem `str.alnum`
`isalpha`	Entspricht dem eingebauten `str.isalpha`
`isdecim`	Äquivalent zu eingebautem `str.isdecimal`
`isdigit`	Gleichwertig zu eingebautem `str.isdigit`
`islower`	Gleichwertig zu eingebautem `str.islower`
`isnumer`	Gleichwertig zu eingebautem `str.isnumeric`
`isupper`	Äquivalent zur eingebauten `str.isupper`
`join`	verbindet Zeichenketten in jedem Element der Serie mit dem übergebenen Trennzeichen
`len`	berechnet die Länge jeder Zeichenkette
`lower, upper`	konvertiert Groß- und Kleinschreibung; entspricht `x.lower()` oder `x.upper()` für jedes Element
`match`	verwendet `re.match` mit dem übergebenen regulären Ausdruck für jedes Element, wobei `True` oder `False` zurückgegeben wird, wenn es übereinstimmt.
`extract`	erfasst Gruppenelemente (falls vorhanden) nach Index aus jeder Zeichenkette
`pad`	fügt Leerzeichen auf der linken, rechten oder beiden Seiten von Zeichenketten ein
`center`	Äquivalent zu `pad(side='both')`
`repeat`	Doppelte Werte (z.B. `s.str.repeat(3)` entspricht `x * 3` für jede Zeichenkette)
`replace`	ersetzt Muster/Regex durch eine andere Zeichenfolge
`slice`	schneidet jede Zeichenkette in der Serie auf
`split`	teilt Zeichenketten anhand von Begrenzungszeichen oder regulären Ausdrücken
`strip`	schneidet Leerzeichen auf beiden Seiten ab, einschließlich Zeilenumbrüchen
`rstrip`	schneidet Leerzeichen auf der rechten Seite ab
`lstrip`	schneidet Leerzeichen auf der linken Seite ab

2.4.8 Arithmetik

Eine wichtige Funktion von pandas ist das arithmetische Verhalten bei Objekten mit unterschiedlichen Indizes. Wenn beim Addieren von Objekten die Indexpaare nicht gleich sind, wird der entsprechende Index im Ergebnis die Vereinigung der Indexpaare sein. Für Benutzer mit Datenbankerfahrung ist dies vergleichbar mit einem automatischen OUTER JOIN auf den Indexbezeichnungen. Schauen wir uns ein Beispiel an:

[1]:
```
import numpy as np
import pandas as pd


s1 = pd.Series(np.random.randn(5))
s2 = pd.Series(np.random.randn(7))
```

Addiert man diese Werte, erhält man:

[2]:
```
s1 + s2
```

[2]:
```
0   -0.862485
1    0.474474
2   -0.357133
3   -0.625274
4    0.787645
5         NaN
6         NaN
dtype: float64
```

Der interne Datenabgleich führt zu fehlenden Werten an den Stellen der Labels, die sich nicht überschneiden. Fehlende Werte werden dann bei weiteren arithmetischen Berechnungen weitergegeben.

Bei DataFrames wird die Ausrichtung sowohl für die Zeilen als auch für die Spalten durchgeführt:

[3]:
```
df1 = pd.DataFrame(np.random.randn(5,3))
df2 = pd.DataFrame(np.random.randn(7,2))
```

Wenn die beiden DataFrames addiert werden, ergibt sich ein DataFrame, dessen Index und Spalten die Vereinigungen derjenigen in jedem der obigen DataFrames sind:

[4]:
```
df1 + df2
```

[4]:
```
          0         1   2
0 -0.228215 -0.922674 NaN
1 -1.515175  3.044930 NaN
2  0.501039  2.414039 NaN
3 -0.495267 -0.952028 NaN
4  0.630200  1.885596 NaN
5       NaN       NaN NaN
6       NaN       NaN NaN
```

Da die Spalte 2 nicht in beiden DataFrame-Objekten vorkommen, erscheinen sie im Ergebnis als fehlend. Das Gleiche gilt für die Zeilen, deren Bezeichnungen nicht in beiden Objekten vorkommen.

Arithmetische Methoden mit Füllwerten

Bei arithmetischen Operationen zwischen unterschiedlich indizierten Objekten kann es sinnvoll sein, einen speziellen Wert (z. B. `0`) zu verwenden, wenn eine Achsenbeschriftung in einem Objekt gefunden wird, im anderen aber nicht. Mit der `add`-Methode kann das Argument `fill_value` übergeben werden:

[5]:
```
df12 = df1.add(df2, fill_value=0)

df12
```

[5]:
```
          0         1         2
0 -0.228215 -0.922674 -1.091759
1 -1.515175  3.044930  0.018406
2  0.501039  2.414039 -0.252414
3 -0.495267 -0.952028  2.494162
4  0.630200  1.885596 -0.292126
5  1.651636 -0.611053       NaN
6 -0.320220  0.199775       NaN
```

Im folgenden Beispiel setzen wir die beiden verbleibenden NaN-Werte auf `0`:

[6]:
```
df12.iloc[[5,6], [2]] = 0
```

[7]:
```
df12
```

[7]:
```
          0         1         2
0 -0.228215 -0.922674 -1.091759
1 -1.515175  3.044930  0.018406
2  0.501039  2.414039 -0.252414
3 -0.495267 -0.952028  2.494162
4  0.630200  1.885596 -0.292126
5  1.651636 -0.611053  0.000000
6 -0.320220  0.199775  0.000000
```

Arithmetische Methoden

Methode	Beschreibung
`add`, `radd`	Methoden für Addition (+)
`sub`, `rsub`	Methoden für die Subtraktion (-)
`div`, `rdiv`	Methoden für die Division (/)
`floordiv`, `rfloordiv`	Methoden für die Abrundungsfunktion (engl.: floor divison) (//)
`mul`, `rmul`	Methoden für die Multiplikation (*)
`pow`, `rpow`	Methoden zur Potenzierung (**)

`r` (engl.: *reverse*) kehrt die Methode um.

Operationen zwischen DataFrame und Series

Wie bei NumPy-Arrays verschiedener Dimensionen ist auch die Arithmetik zwischen DataFrame und Series definiert.

[8]:
```
s1 + df12
```

[8]:
```
          0         1         2   3   4
0 -2.324582 -1.279211 -1.284821 NaN NaN
1 -3.611541  2.688394 -0.174656 NaN NaN
2 -1.595327  2.057503 -0.445476 NaN NaN
3 -2.591633 -1.308564  2.301100 NaN NaN
4 -1.466166  1.529060 -0.485188 NaN NaN
5 -0.444730 -0.967589 -0.193062 NaN NaN
6 -2.416587 -0.156761 -0.193062 NaN NaN
```

Wenn wir `s1` mit `df12` addieren, wird die Additon für jede Zeile einmal durchgeführt. Dies wird als *Broadcasting* bezeichnet. Standardmäßig entspricht die Arithmetik zwischen DataFrame und Serie dem Index der Serie in den Spalten des DataFrame, wobei die Zeilen nach unten übertragen werden.

Wenn ein Indexwert weder in den Spalten des DataFrame noch im Index der Serie gefunden wird, werden die Objekte neu indiziert, um die Vereinigung zu bilden:

Wenn ihr stattdessen die Spalten übertragen und die Zeilen abgleichen wollt, müsst ihr eine der arithmetischen Methoden verwenden, z.B.:

[9]:
```
df12.add(s2, axis="index")
```

[9]:
```
          0         1         2
0  1.005665  0.311206  0.142122
1 -0.684165  3.875940  0.849415
2  0.336968  2.249968 -0.416485
3 -0.307428 -0.764190  2.682000
4  1.990071  3.245467  1.067746
5  0.686948 -1.575741 -0.964688
6 -1.575925 -1.055929 -1.255704
```

Die Achsennummer, die ihr übergebt, ist die Achse, auf die abgeglichen werden soll. In diesem Fall soll der Zeilenindex des DataFrame (`axis='index'` oder `axis=0`) abgeglichen und übertragen werden.

Funktionsanwendung und Mapping

`numpy.ufunc` (elementweise Array-Methoden) funktionieren auch mit Pandas-Objekten:

[10]:
```
np.abs(df12)
```

[10]:
```
          0         1         2
0  0.228215  0.922674  1.091759
1  1.515175  3.044930  0.018406
2  0.501039  2.414039  0.252414
3  0.495267  0.952028  2.494162
4  0.630200  1.885596  0.292126
5  1.651636  0.611053  0.000000
6  0.320220  0.199775  0.000000
```

Eine weitere häufige Operation ist die Anwendung einer Funktion auf eindimensionale Arrays auf jede Spalte oder Zeile. Die pandas.DataFrame.apply-Methode tut genau dies:

[11]:
```
df12
```

[11]:
```
          0         1         2
0 -0.228215 -0.922674 -1.091759
1 -1.515175  3.044930  0.018406
2  0.501039  2.414039 -0.252414
3 -0.495267 -0.952028  2.494162
4  0.630200  1.885596 -0.292126
5  1.651636 -0.611053  0.000000
6 -0.320220  0.199775  0.000000
```

[12]:
```
f = lambda x: x.max() - x.min()

df12.apply(f)
```

[12]:
```
0    3.166811
1    3.996958
2    3.585921
dtype: float64
```

Hier wird die Funktion `f`, die die Differenz zwischen dem Maximum und dem Minimum einer Reihe berechnet, einmal für jede Spalte des Rahmens aufgerufen. Das Ergebnis ist eine Reihe mit den Spalten des Rahmens als Index.

Wenn ihr `axis='columns'` an `apply` übergebt, wird die Funktion stattdessen einmal pro Zeile aufgerufen:

[13]:
```
df12.apply(f, axis="columns")
```

[13]:
```
0    0.863544
1    4.560105
2    2.666453
3    3.446189
4    2.177721
5    2.262689
6    0.519995
dtype: float64
```

Viele der gebräuchlichsten Array-Statistiken (wie `sum` und `mean`) sind DataFrame-Methoden, so dass die Verwendung von `apply` nicht notwendig ist.

Die an apply übergebene Funktion muss keinen Einzelwert zurückgeben; sie kann auch eine Reihe mit mehreren Werten zurückgeben:

[14]:
```
def f(x):
    return pd.Series([x.min(), x.max()], index=["min", "max"])


df12.apply(f)
```

[14]:
```
            0         1         2
min -1.515175 -0.952028 -1.091759
max  1.651636  3.044930  2.494162
```

Es können auch elementweise Python-Funktionen verwendet werden. Angenommen, ihr möchtet aus jedem Fließkommawert in `df12` eine formatierte Zeichenkette berechnen. Dies könnt ihr mit pandas.DataFrame.applymap erreichen:

[15]:
```
f = lambda x: round(x, 2)

df12.applymap(f)
```

[15]:
```
      0     1     2
0 -0.23 -0.92 -1.09
1 -1.52  3.04  0.02
2  0.50  2.41 -0.25
3 -0.50 -0.95  2.49
4  0.63  1.89 -0.29
5  1.65 -0.61  0.00
6 -0.32  0.20  0.00
```

Der Grund für den Namen `applymap` ist, dass Series über eine `map`-Methode zur Anwendung einer elementweisen Funktion verfügt:

[16]:
```
df12[2].map(f)
```

[16]:
```
0   -1.09
1    0.02
2   -0.25
3    2.49
4   -0.29
5    0.00
6    0.00
Name: 2, dtype: float64
```

2.4.9 Deskriptive Statistik

pandas-Objekte sind mit einer Reihe von gängigen mathematischen und statistischen Methoden ausgestattet. Die meisten von ihnen fallen in die Kategorie der Reduktionen oder zusammenfassenden Statistiken, Methoden, die einen einzelnen Wert (wie die Summe oder den Mittelwert) aus einer Serie oder einer Reihe von Werten aus den Zeilen oder Spalten eines DataFrame extrahieren. Im Vergleich zu ähnlichen Methoden, die sich bei NumPy-Arrays finden, behandeln sie auch fehlende Daten.

[1]:
```
import numpy as np
import pandas as pd


df = pd.DataFrame(
    np.random.randn(7, 3), index=pd.date_range("2022-02-02", periods=7)
)
new_index = pd.date_range("2022-02-03", periods=7)
df2 = df.reindex(new_index)

df2
```

[1]:
```
                   0         1         2
2022-02-03 -0.620864  0.196322 -0.795490
2022-02-04  2.124922 -0.719332 -1.097980
2022-02-05 -1.836651 -1.733610  0.297066
2022-02-06  0.151839  0.011681 -1.157835
2022-02-07 -1.180802  0.263522 -0.895572
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2022-02-08 -1.310820  0.810454 -1.238823
2022-02-09       NaN       NaN       NaN
```

Der Aufruf der `pandas.DataFrame.sum`-Methode gibt eine Serie zurück, die Spaltensummen enthält:

[2]:
```
df2.sum()
```

[2]:
```
0   -2.672376
1   -1.170963
2   -4.888635
dtype: float64
```

Die Übergabe von `axis='columns'` oder `axis=1` summiert stattdessen über die Spalten:

[3]:
```
df2.sum(axis="columns")
```

[3]:
```
2022-02-03   -1.220032
2022-02-04    0.307609
2022-02-05   -3.273195
2022-02-06   -0.994315
2022-02-07   -1.812853
2022-02-08   -1.739189
2022-02-09    0.000000
Freq: D, dtype: float64
```

Wenn eine ganze Zeile oder Spalte alle NA-Werte enthält, ist die Summe `0`. Dies kann mit der Option `skipna` deaktiviert werden:

[4]:
```
df2.sum(axis="columns", skipna=False)
```

[4]:
```
2022-02-03   -1.220032
2022-02-04    0.307609
2022-02-05   -3.273195
2022-02-06   -0.994315
2022-02-07   -1.812853
2022-02-08   -1.739189
2022-02-09         NaN
Freq: D, dtype: float64
```

Einige Aggregationen, wie z.B. `mean`, erfordern mindestens einen Nicht-`NaN`-Wert, um ein wertvolles Ergebnis zu erhalten:

[5]:
```
df2.mean(axis="columns")
```

[5]:
```
2022-02-03   -0.406677
2022-02-04    0.102536
2022-02-05   -1.091065
2022-02-06   -0.331438
2022-02-07   -0.604284
2022-02-08   -0.579730
2022-02-09         NaN
Freq: D, dtype: float64
```

Optionen für Reduktionsmethoden

Methode	Beschreibung
`axis`	die Achse der zu reduzierenden Werte: `0` für die Zeilen des DataFrame und `1` für die Spalten
`skipna`	fehlende Werte ausschließen; standardmäßig `True`
`level`	nach Ebene gruppiert reduzieren, wenn die Achse hierarchisch indiziert ist (MultiIndex)

Einige Methoden, wie `idxmin` und `idxmax`, liefern indirekte Statistiken wie den Indexwert, bei dem der Mindest- oder Höchstwert erreicht wird:

[6]:
```
df2.idxmax()
```

[6]:
```
0   2022-02-04
1   2022-02-08
2   2022-02-05
dtype: datetime64[ns]
```

Andere Methoden sind Akkumulationen:

[7]:
```
df2.cumsum()
```

[7]:
```
                   0         1         2
2022-02-03 -0.620864  0.196322 -0.795490
2022-02-04  1.504058 -0.523010 -1.893470
2022-02-05 -0.332593 -2.256620 -1.596405
2022-02-06 -0.180754 -2.244939 -2.754240
2022-02-07 -1.361556 -1.981417 -3.649812
2022-02-08 -2.672376 -1.170963 -4.888635
2022-02-09       NaN       NaN       NaN
```

Eine andere Art von Methoden sind weder Reduktionen noch Akkumulationen. `describe` ist ein solches Beispiel, das mehrere zusammenfassende Statistiken auf einen Schlag erstellt:

[8]:
```
df2.describe()
```

[8]:
```
              0         1         2
count  6.000000  6.000000  6.000000
mean  -0.445396 -0.195161 -0.814773
std    1.429642  0.901030  0.569351
min   -1.836651 -1.733610 -1.238823
25%   -1.278316 -0.536579 -1.142872
50%   -0.900833  0.104002 -0.996776
75%   -0.041337  0.246722 -0.820510
max    2.124922  0.810454  0.297066
```

Bei nicht-numerischen Daten erzeugt `describe` alternative zusammenfassende Statistiken:

[9]:
```
data = {
    "Code": ["U+0000", "U+0001", "U+0002", "U+0003", "U+0004", "U+0005"],
    "Octal": ["001", "002", "003", "004", "004", "005"],
}
df3 = pd.DataFrame(data)

df3.describe()
```

[9]:
```
          Code Octal
count        6     6
unique       6     5
top     U+0000   004
freq         1     2
```

Deskriptive und zusammenfassende Statistiken:

Methode	Beschreibung
`count`	Anzahl der Nicht-NA-Werte
`describe`	Berechnung einer Reihe von zusammenfassenden Statistiken für Serien oder jede DataFrame-Spalte
`min`, `max`	Berechnung der Mindest- und Höchstwerte
`argmin`, `argmax`	Berechnung der Indexstellen (ganze Zahlen), an denen der Mindest- bzw. Höchstwert erreicht wurde
`idxmin`, `idxmax`	Berechnung der Indexbeschriftungen, an denen der Mindest- bzw. Höchstwert erreicht wurde
`quantile`	Berechnung des Stichprobenquantils im Bereich von 0 bis 1
`sum`	Summe der Werte
`mean`	Arithmetisches Mittel der Werte
`median`	Arithmetischer Median (50%-Quantil) der Werte
`mad`	Mittlere absolute Abweichung vom Mittelwert
`prod`	Produkt aller Werte
`var`	Stichprobenvarianz der Werte
`std`	Stichprobenstandardabweichung der Werte
`skew`	Stichprobenschiefe (drittes Moment) der Werte
`kurt`	Stichprobenwölbung (viertes Moment) der Werte
`cumsum`	Kumulierte Summe der Werte
`cummin`, `cummax`	Kumuliertes Minimum bzw. Maximum der Werte
`cumprod`	Kumuliertes Produkt der Werte
`diff`	Berechnung der ersten arithmetischen Differenz (nützlich für Zeitreihen)
`pct_change`	Berechnung der prozentualen Veränderungen

ydata_profiling

ydata-profiling erzeugt Profilberichte aus einem pandas DataFrame. Die Funktion `pandas df.describe()` ist praktisch, aber ein wenig einfach für die explorative Datenanalyse. ydata-profiling erweitert pandas DataFrame mit `df.profile_report()`, die automatisch einen standardisierten Bericht zum Verständnis der Daten erzeugt.

Installation

```
$ pipenv install ydata_profiling[notebook]
...
✓ Success!
Updated Pipfile.lock (cbc5f7)!
Installing dependencies from Pipfile.lock (cbc5f7)...
     ████████████████████████████████████████████████████████████████████████ 80/80 -␣
↪00:02:26
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
...
$ pipenv run jupyter nbextension enable --py widgetsnbextension
Enabling notebook extension jupyter-js-widgets/extension...
      - Validating: OK
```

Beispiel

[10]:
```
from ydata_profiling import ProfileReport


profile = ProfileReport(df2, title="Pandas Profiling Report")

profile.to_widgets()
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]
```

```
Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
Render widgets:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HTM
```

Konfiguration für große Datensätze

Standardmäßig fasst ydata-profiling den Datensatz so zusammen, dass er die meisten Erkenntnisse für die Datenanalyse liefert. Wenn die Berechnungszeit der Profilerstellung zu einem Engpass wird, bietet ydata-profiling mehrere Alternativen, um diesen zu überwinden. Für die folgenden Beispiele lesen wir zunächst einen größeren Datensatz in pandas ein:

[11]:
```
titanic = pd.read_csv(
    "https://raw.githubusercontent.com/datasciencedojo/datasets/master/titanic.csv"
)
```

1. Minimaler Modus

ydata-profiling enthält eine minimale Konfigurationsdatei config_minimal.yaml, in der die teuersten Berechnungen standardmäßig ausgeschaltet sind. Dies ist die empfohlene Ausgangsbasis für größere Datensätze.

[12]:
```
profile = ProfileReport(
    titanic, title="Minimal Pandas Profiling Report", minimal=True
)

profile.to_widgets()
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]
```

```
Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
Render widgets:   0%|          | 0/1 [00:00<?, ?it/s]
```

```
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HTM
```

Weitere Details zu Einstellungen und Konfiguration findet ihr unter Available settings.

2. Stichprobe

Eine alternative Möglichkeit bei sehr großen Datensätzen besteht darin, nur einen Teil davon für die Erstellung des Profiling-Berichts zu verwenden:

[13]:
```
sample = titanic.sample(frac=0.05)
profile = ProfileReport(sample, title="Sample Pandas Profiling Report")

profile.to_widgets()
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]
Render widgets:   0%|          | 0/1 [00:00<?, ?it/s]
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HT
```

3. Teure Berechnungen deaktivieren

Um den Rechenaufwand in großen Datensätzen zu verringern, aber dennoch einige interessante Informationen zu erhalten, können einige Berechnungen nur für bestimmte Spalten gefiltert werden:

[14]:
```
profile = ProfileReport()
profile.config.interactions.targets = ["Sex", "Age"]
profile.df = titanic

profile.to_widgets()
```

```
Summarize dataset:   0%|          | 0/5 [00:00<?, ?it/s]
Generate report structure:   0%|          | 0/1 [00:00<?, ?it/s]
Render widgets:   0%|          | 0/1 [00:00<?, ?it/s]
VBox(children=(Tab(children=(Tab(children=(GridBox(children=(VBox(children=(GridspecLayout(children=(HT
```

Die Einstellung `interactions.targets`, kann sowohl über Konfigurationsdateien wie auch über Umgebungsvariablen geändert werden; Einzelheiten hierzu findet ihr unter Changing settings.

4. Nebenläufigkeit

Aktuell wird an einem skalierbaren Spark-Backend für ydata-profiling gearbeitet, siehe Spark Profiling Development.

2.4.10 Sortieren und Ranking

Das Sortieren eines Datensatzes nach einem Kriterium ist eine weitere wichtige eingebaute Funktion. Um lexikografisch nach Zeilen- oder Spaltenindex zu sortieren, verwendet die Methoden pandas.Series.sort_index oder pandas.DataFrame.sort_index, die ein neues, sortiertes Objekt zurückgibt. Mit `ascending=False` wird die Sortierreihenfolge umgekehrt:

[1]:
```
import numpy as np
import pandas as pd
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
s = pd.Series(np.random.randn(7))

s.sort_index(ascending=False)
```

[1]:
```
6    0.681477
5    0.259093
4   -0.877553
3    0.539592
2    1.246258
1   -0.586841
0    0.392347
dtype: float64
```

Um eine Serie nach ihren Werten zu sortieren, könnt ihr die `sort_values`-Methode verwenden:

[2]:
```
s.sort_values()
```

[2]:
```
4   -0.877553
1   -0.586841
5    0.259093
0    0.392347
3    0.539592
6    0.681477
2    1.246258
dtype: float64
```

Alle fehlenden Werte werden standardmäßig an das Ende der Reihe sortiert:

[3]:
```
s = pd.Series(np.random.randn(7))
s[s < 0] = np.nan

s.sort_values()
```

[3]:
```
0    0.179928
3    0.565061
2    0.954890
6    1.017703
4    3.932341
1         NaN
5         NaN
dtype: float64
```

Mit einem DataFrame können ihr auf beiden Achsen nach Index sortieren:

[4]:
```
df = pd.DataFrame(np.random.randn(7, 3))

df
```

[4]:
```
          0         1         2
0  0.580062 -1.583831 -0.484063
1  0.413304 -1.005881 -0.580430
2  0.166762 -0.635044 -0.407102
3  0.206200  0.615862 -1.561290
4  0.863733  1.282648  0.087279
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
5  0.261991 -0.544270 -0.396647
6  1.194072 -0.703340 -1.313653
```

[5]:
```
df.sort_index(ascending=False)
```

[5]:
```
          0         1         2
6  1.194072 -0.703340 -1.313653
5  0.261991 -0.544270 -0.396647
4  0.863733  1.282648  0.087279
3  0.206200  0.615862 -1.561290
2  0.166762 -0.635044 -0.407102
1  0.413304 -1.005881 -0.580430
0  0.580062 -1.583831 -0.484063
```

[6]:
```
df.sort_index(axis=1, ascending=False)
```

[6]:
```
          2         1         0
0 -0.484063 -1.583831  0.580062
1 -0.580430 -1.005881  0.413304
2 -0.407102 -0.635044  0.166762
3 -1.561290  0.615862  0.206200
4  0.087279  1.282648  0.863733
5 -0.396647 -0.544270  0.261991
6 -1.313653 -0.703340  1.194072
```

Beim Sortieren eines DataFrame könnt ihr die Daten in einer oder mehreren Spalten als Sortierschlüssel verwenden. Dazu übergebt ihr eine oder mehrere Spaltennamen an die Option `by` von `sort_values`:

[7]:
```
df.sort_values(by=2)
```

[7]:
```
          0         1         2
3  0.206200  0.615862 -1.561290
6  1.194072 -0.703340 -1.313653
1  0.413304 -1.005881 -0.580430
0  0.580062 -1.583831 -0.484063
2  0.166762 -0.635044 -0.407102
5  0.261991 -0.544270 -0.396647
4  0.863733  1.282648  0.087279
```

Um nach mehreren Spalten zu sortieren, könnt ihr eine Liste von Namen übergeben.

Ranking weist Ränge von eins bis zur Anzahl der gültigen Datenpunkte in einem Array zu:

[8]:
```
df.rank()
```

[8]:
```
     0    1    2
0  5.0  1.0  4.0
1  4.0  2.0  3.0
2  1.0  4.0  5.0
3  2.0  6.0  1.0
4  6.0  7.0  7.0
5  3.0  5.0  6.0
6  7.0  3.0  2.0
```

Wenn beim Ranking Gleichstände auftauchen, weist `rank` jeder Gruppe den mittleren Rang zu.

[9]:
```
df.rank(method="max")
```

[9]:
```
     0    1    2
0  5.0  1.0  4.0
1  4.0  2.0  3.0
2  1.0  4.0  5.0
3  2.0  6.0  1.0
4  6.0  7.0  7.0
5  3.0  5.0  6.0
6  7.0  3.0  2.0
```

Weitere Verfahren mit `rank`

Methode	Beschreibung
`averag`	Standard: jedem Eintrag in der gleichen Gruppe den durchschnittlichen Rang zuweisen
`min`	verwendet den minimalen Rang für die gesamte Gruppe
`max`	verwendet den maximalen Rang für die gesamte Gruppe
`first`	weist die Ränge in der Reihenfolge zu, in der die Werte in den Daten erscheinen
`dense`	wie `method='min'`, aber die Ränge erhöhen sich zwischen den Gruppen immer um 1 und nicht nach der Anzahl der gleichen Elemente in einer Gruppe

2.4.11 Unterteilen und Kategorisieren von Daten

Kontinuierliche Daten werden häufig in Bereiche unterteilt oder auf andere Weise für die Analyse gruppiert.

Angenommen, ihr habt Daten über eine Gruppe von Personen in einer Studie, die ihr in diskrete Altersgruppen einteilen möchtet. Hierfür generieren wir uns einen Dataframe mit 250 Einträgen zwischen `0` und `99`:

[1]:
```
import numpy as np
import pandas as pd


ages = np.random.randint(0, 99, 250)
df = pd.DataFrame({"Age": ages})

df
```

[1]:
```
     Age
0     60
1     16
2     75
3     82
4     40
..   ...
245   16
246   35
247   21
248   26
249   87
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[250 rows x 1 columns]
```

Anschließend bietet uns pandas mit pandas.cut eine einfache Möglichkeit, die Ergebnisse in zehn Bereiche aufzuteilen. Um nur ganze Jahre zu erhalten, setzen wir zusätzlich `precision=0`:

[2]:
```
cats = pd.cut(ages, 10, precision=0)

cats
```

[2]:
```
[(59.0, 69.0], (10.0, 20.0], (69.0, 78.0], (78.0, 88.0], (39.0, 49.0], ..., (10.0, 20.0],
↪ (29.0, 39.0], (20.0, 29.0], (20.0, 29.0], (78.0, 88.0]]
Length: 250
Categories (10, interval[float64, right]): [(-0.1, 10.0] < (10.0, 20.0] < (20.0, 29.0] <␣
↪(29.0, 39.0] ... (59.0, 69.0] < (69.0, 78.0] < (78.0, 88.0] < (88.0, 98.0]]
```

Mit pandas.Categorical.categories könnt ihr euch die Kategorien anzeigen lassen:

[3]:
```
cats.categories
```

[3]:
```
IntervalIndex([(-0.1, 10.0], (10.0, 20.0], (20.0, 29.0], (29.0, 39.0], (39.0, 49.0], (49.
↪0, 59.0], (59.0, 69.0], (69.0, 78.0], (78.0, 88.0], (88.0, 98.0]], dtype=
↪'interval[float64, right]')
```

...oder auch nur eine einzelne Kategorie:

[4]:
```
cats.categories[0]
```

[4]:
```
Interval(-0.1, 10.0, closed='right')
```

Mit pandas.Categorical.codes könnt ihr euch ein Array anzeigen lassen, in dem für jeden Wert die zugehörige Kategorie angezeigt wird:

[5]:
```
cats.codes
```

[5]:
```
array([6, 1, 7, 8, 4, 7, 7, 5, 1, 6, 9, 7, 9, 1, 7, 7, 6, 8, 4, 4, 2, 5,
       5, 4, 9, 0, 5, 7, 9, 2, 7, 2, 4, 2, 8, 1, 2, 2, 0, 2, 1, 3, 1, 1,
       9, 0, 2, 6, 7, 5, 9, 2, 8, 0, 9, 7, 6, 6, 7, 6, 0, 2, 4, 8, 1, 1,
       9, 0, 7, 4, 8, 0, 0, 2, 6, 4, 2, 9, 0, 3, 3, 9, 3, 9, 3, 6, 5, 6,
       3, 5, 6, 3, 0, 3, 1, 2, 4, 3, 9, 5, 8, 0, 4, 2, 0, 9, 6, 9, 7, 0,
       8, 3, 1, 8, 8, 9, 3, 2, 7, 7, 0, 7, 5, 4, 7, 9, 1, 6, 0, 1, 0, 7,
       8, 5, 4, 1, 2, 8, 0, 3, 0, 4, 7, 4, 9, 2, 6, 4, 9, 9, 7, 0, 6, 1,
       9, 9, 9, 5, 1, 4, 8, 6, 9, 8, 5, 1, 8, 5, 4, 4, 8, 5, 1, 4, 7, 3,
       3, 7, 6, 7, 4, 0, 9, 7, 0, 7, 5, 6, 0, 6, 6, 2, 5, 4, 6, 2, 9, 0,
       1, 1, 3, 2, 2, 9, 0, 8, 7, 2, 7, 2, 7, 8, 2, 4, 7, 2, 2, 9, 2, 3,
       7, 2, 0, 6, 1, 4, 3, 0, 7, 8, 1, 2, 0, 3, 6, 8, 1, 9, 9, 1, 0, 2,
       3, 8, 8, 1, 3, 2, 2, 8], dtype=int8)
```

Mit `value_counts` können wir uns nun anschauen, wie sich die Anzahl auf die einzelnen Bereiche verteilt:

[6]:
```
pd.value_counts(cats)
```

[6]:
```
(20.0, 29.0]    32
(69.0, 78.0]    31
(88.0, 98.0]    29
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
(-0.1, 10.0]    28
(10.0, 20.0]    25
(39.0, 49.0]    23
(59.0, 69.0]    23
(78.0, 88.0]    23
(29.0, 39.0]    20
(49.0, 59.0]    16
Name: count, dtype: int64
```

Auffalend ist, dass die Altersbereiche nicht gleich viele Jahre enthalten, sondern mit `20.0, 29.0` und `69.0, 78.0` zwei Bereiche nur 9 Jahre umfassen. Dies hängt damit zusammen, dass der Altersumfang nur von `0`bis `98` reicht:

[7]:
```
df.min()
```

[7]:
```
Age    0
dtype: int64
```

[8]:
```
df.max()
```

[8]:
```
Age    98
dtype: int64
```

Mit pandas.qcut wird die Menge hingegen in Bereiche unterteilt, die annähernd gleich groß sind:

[9]:
```
cats = pd.qcut(ages, 10, precision=0)
```

[10]:
```
pd.value_counts(cats)
```

[10]:
```
(-1.0, 9.0]     28
(18.0, 26.0]    26
(36.0, 46.0]    26
(62.0, 72.0]    26
(79.0, 92.0]    26
(26.0, 36.0]    25
(46.0, 62.0]    25
(92.0, 98.0]    24
(9.0, 18.0]     22
(72.0, 79.0]    22
Name: count, dtype: int64
```

Wollen wir gewährleisten, dass jede Altersgruppe tatsächlich genau zehn Jahre umfasst, können wir dies mit pandas.Categorical direkt angeben:

[11]:
```
age_groups = ["{0} - {1}".format(i, i + 9) for i in range(0, 109, 10)]
cats = pd.Categorical(age_groups)

cats.categories
```

[11]:
```
Index(['0 - 9', '10 - 19', '100 - 109', '20 - 29', '30 - 39', '40 - 49',
       '50 - 59', '60 - 69', '70 - 79', '80 - 89', '90 - 99'],
      dtype='object')
```

Für die Gruppierung wird nun pandas.cut verwendet:

[12]:
```
df["Age group"] = pd.cut(df.Age, range(0, 111, 10), right=False, labels=cats)

df
```

[12]:
```
     Age Age group
0     60   60 - 69
1     16   10 - 19
2     75   70 - 79
3     82   80 - 89
4     40   40 - 49
..   ...       ...
245   16   10 - 19
246   35   30 - 39
247   21   20 - 29
248   26   20 - 29
249   87   80 - 89

[250 rows x 2 columns]
```

2.4.12 Kombinieren und Zusammenführen von Datensätzen

Die in pandas-Objekten enthaltenen Daten können auf verschiedene Weise miteinander kombiniert werden:

- pandas.merge verbindet Zeilen in DataFrames basierend auf einem oder mehreren Schlüsseln. Diese Funktion ist von SQL oder anderen relationalen Datenbanken vertraut, da sie Datenbank-Join-Operationen implementiert.
- pandas.concat verkettet oder *stapelt* Objekte entlang einer Achse.
- Die Instanzmethoden pandas.DataFrame.combine_first oder pandas.Series.combine_first ermöglichen das Zusammenfügen von sich überschneidenden Daten, um fehlende Werte in einem Objekt mit Werten aus einem anderen zu ergänzen.
- Mit pandas.merge_asof könnt ihr zeitreihenbasierte *Window Joins* zwischen DataFrame-Objekten durchführen.

Datenbankähnliche DataFrame-Joins

Merge- oder Join-Operationen kombinieren Datensätze durch die Verknüpfung von Zeilen mit einem oder mehreren Schlüsseln. Diese Operationen sind besonders wichtig in relationalen, SQL-basierten Datenbanken. Die Merge-Funktion in pandas ist der Haupteinstiegspunkt für die Anwendung dieser Algorithmen auf eure Daten.

[1]:
```
import pandas as pd
```

[2]:
```
books = pd.DataFrame(
    {"Language": ["en", "en", "de", "fr", "de", "de", "en"], "Range": range(7)}
)

updates = pd.DataFrame({"Language": ["de", "en", "pt"], "Range": range(3)})
```

[3]:
```
books
```

[3]:
```
  Language  Range
0       en      0
1       en      1
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2       de      2
3       fr      3
4       de      4
5       de      5
6       en      6
```

[4]:
```
updates
```

[4]:
```
  Language  Range
0       de      0
1       en      1
2       pt      2
```

Dies ist ein Beispiel für eine 1:n-Beziehung; die Daten in `df1` haben mehrere Zeilen mit den Bezeichnungen `de` und `en`, während `df2` nur eine Zeile für jeden Wert in der Schlüsselspalte hat. Wenn wir `merge` mit diesen Objekten aufrufen, erhalten wir:

[5]:
```
pd.merge(books, updates)
```

[5]:
```
  Language  Range
0       en      1
```

Hinweis:

Beachtet, dass ich nicht angegeben habe, über welche Spalte die Verknüpfung erfolgen soll. Wenn diese Information nicht angegeben wird, verwendet `merge` die sich überschneidenden Spaltennamen als Schlüssel. Es ist jedoch eine gute Praxis, dies explizit anzugeben:

[6]:
```
pd.merge(books, updates, on="Language")
```

[6]:
```
  Language  Range_x  Range_y
0       en        0        1
1       en        1        1
2       en        6        1
3       de        2        0
4       de        4        0
5       de        5        0
```

Wenn die Spaltennamen in jedem Objekt unterschiedlich sind, könnt ihr sie separat angeben:

[7]:
```
books = pd.DataFrame(
    {"Language": ["en", "en", "de", "fr", "de", "de", "en"], "Range": range(7)}
)
updates = pd.DataFrame({"Language": ["de", "en", "es"], "Range": range(3)})

pd.merge(books, updates, left_on="Language", right_on="Language")
```

[7]:
```
  Language  Range_x  Range_y
0       en        0        1
1       en        1        1
2       en        6        1
3       de        2        0
4       de        4        0
5       de        5        0
```

Fie Werte `fr` und `es` und die zugehörigen Daten fehlen im Ergebnis. Standardmäßig führt `merge` einen *Inner Join* durch; die Schlüssel im Ergebnis sind die Schnittmenge bzw. die gemeinsame Menge in beiden Tabellen. Andere mögliche Optionen sind *Left Join*, *Right Join* und *Outer Join*. *Outer Join* nimmt die Vereinigung der Schlüssel und kombiniert den Effekt der Anwendung von *Left Join* und *Right Join*:

```
[8]: pd.merge(books, updates, how="outer")
```

```
[8]:   Language  Range
0        en      0
1        en      1
2        de      2
3        fr      3
4        de      4
5        de      5
6        en      6
7        de      0
8        es      2
```

Verschiedene *Join*-Typen mit `how`-Argument

Option	Verhalten
`how='inner'`	verwendet nur die in beiden Tabellen beobachteten Schlüsselkombinationen
`how='left'`	verwendet alle in der linken Tabelle gefundenen Schlüsselkombinationen
`how='right'`	verwendet alle in der rechten Tabelle gefundenen Schlüsselkombinationen
`how='outer'`	verwendet alle in beiden Tabellen beobachteten Schlüsselkombinationen zusammen

n:n-Beziehungen bilden das kartesische Produkt der übereinstimmenden Schlüssel, z.B.:

```
[9]: pd.merge(books, updates, on="Language", how="left")
```

```
[9]:   Language  Range_x  Range_y
0        en         0       1.0
1        en         1       1.0
2        de         2       0.0
3        fr         3       NaN
4        de         4       0.0
5        de         5       0.0
6        en         6       1.0
```

Da es drei `en`-Zeilen im linken DataFrame und einen im rechten DataFrame gab, gibt es drei `en`-Zeilen im Ergebnis. Die Join-Methode wirkt sich nur auf die eindeutigen Schlüsselwerte aus, die im Ergebnis erscheinen:

Um mehrere Schlüssel zusammenzuführen, übergebt eine Liste von Spaltennamen:

```
[10]: books = pd.DataFrame(
          {
              "Title": ["Jupyter Tutorial", "Jupyter Tutorial", "PyViz Tutorial"],
              "Language": ["de", "en", "de"],
              "Range": [1, 2, 3],
          }
      )

      updates = pd.DataFrame(
          {
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "Title": [
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        "Language": ["de", "de", "de", "en"],
        "Range": [4, 5, 6, 7],
    }
)

pd.merge(books, updates, on=["Title", "Language"], how="outer")
```

[10]:

	Title	Language	Range_x	Range_y
0	Jupyter Tutorial	de	1.0	4.0
1	Jupyter Tutorial	en	2.0	NaN
2	PyViz Tutorial	de	3.0	5.0
3	Python Basics	de	NaN	6.0
4	Python Basics	en	NaN	7.0

2.4.13 Gruppenoperationen

Mit `groupby` ist ein Prozess gemeint, der einen oder mehrere der folgenden Schritte umfasst:

- **Split** teilt die Daten in Gruppen nach bestimmten Kriterien auf
- **Apply** wendet eine Funktion unabhängig auf jede Gruppe an
- **Combine** kombiniert die Ergebnisse in einer Datenstruktur

In der ersten Phase des Prozesses werden die in einem pandas-Objekt enthaltenen Daten, sei es eine Series, ein DataFrame oder etwas anderes, in Gruppen aufgeteilt, die auf einem oder mehreren Schlüsseln basieren. Die Aufteilung wird auf einer bestimmten Achse eines Objekts durchgeführt. Ein DataFrame kann zum Beispiel nach seinen Zeilen (`axis=0`) oder seinen Spalten (`axis=1`) gruppiert werden. Danach wird auf jede Gruppe eine Funktion angewendet, die einen neuen Wert erzeugt. Schließlich werden die Ergebnisse all dieser Funktionsanwendungen in einem Ergebnisobjekt kombiniert. Die Form des Ergebnisobjekts hängt normalerweise davon ab, was mit den Daten gemacht wird.

Jeder Gruppierungsschlüssel kann viele Formen annehmen, und die Schlüssel müssen nicht alle vom gleichen Typ sein: * Eine Liste oder ein Array von Werten, die die gleiche Länge wie die zu gruppierende Achse haben * Ein Wert, der einen Spaltennamen in einem DataFrame angibt * Ein Dict oder eine Series, die eine Entsprechung zwischen den Werten auf der Achse, die gruppiert wird, und den Gruppennamen darstellt * Eine Funktion, die auf dem Achsenindex oder den einzelnen Beschriftungen im Index aufgerufen wird

Hinweis

Die drei letztgenannten Methoden sind Abkürzungen, um ein Array von Werten zu erzeugen, die für die Aufteilung des Objekts verwendet werden.

Macht euch keine Sorgen, wenn dies alles abstrakt erscheint. Im Laufe dieses Kapitels werde ich viele Beispiele für all diese Methoden geben. Für den Anfang hier ein kleiner Tabellendatensatz als DataFrame:

[1]:
```
import pandas as pd
import numpy as np
```

[2]:
```
df = pd.DataFrame(
    {
        "Title": [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            None,
            "Python Basics",
            "Python Basics",
        ],
        "Language": ["de", "en", "de", None, "de", "en"],
        "2021-12": [19651, 4722, 2573, None, 525, 157],
        "2022-01": [30134, 3497, 4873, None, 427, 85],
        "2022-02": [33295, 4009, 3930, None, 276, 226],
    }
)

df
```

[2]:

	Title	Language	2021-12	2022-01	2022-02
0	Jupyter Tutorial	de	19651.0	30134.0	33295.0
1	Jupyter Tutorial	en	4722.0	3497.0	4009.0
2	PyViz Tutorial	de	2573.0	4873.0	3930.0
3	None	None	NaN	NaN	NaN
4	Python Basics	de	525.0	427.0	276.0
5	Python Basics	en	157.0	85.0	226.0

Angenommen, ihr möchtet den Summe der Spalte `02/2022` anhand der Beschriftungen von `Title` berechnen. Es gibt mehrere Möglichkeiten, dies zu tun. Eine davon ist der Zugriff auf `02/2022` und der Aufruf von `groupby` mit der Spalte (einer Series) in `Title`:

[3]:
```
grouped = df["2022-02"].groupby(df["Title"])

grouped
```

[3]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x10f466a50>
```

Diese `grouped`-Variable ist nun ein spezielles `SeriesGroupBy`-Objekt. Es hat noch nichts berechnet, außer einigen Zwischendaten über den Gruppenschlüssel `df['Title']`. Die Idee ist, dass dieses Objekt über alle Informationen verfügt, die benötigt werden, um eine Operation auf jede der Gruppen anzuwenden. Zur Berechnung der Gruppenmittelwerte können wir beispielsweise die Methode `sum` des `GroupBy`-Objekts aufrufen:

[4]:
```
grouped.sum()
```

[4]:
```
Title
Jupyter Tutorial    37304.0
PyViz Tutorial       3930.0
Python Basics         502.0
Name: 2022-02, dtype: float64
```

Später werde ich mehr darüber erklären, was passiert, wenn ihr `.sum()` aufruft. Wichtig ist hier, dass die Daten (eine Reihe) durch Aufteilung der Daten auf den Gruppenschlüssel aggregiert wurden, wodurch eine neue Reihe entsteht, die nun durch die eindeutigen Werte in der Spalte `Title` indiziert ist. Der resultierende Index ist `Title`, weil `groupby(df['Title']` dies tat.

Hätten wir stattdessen mehrere Arrays als Liste übergeben, würden wir etwas anderes erhalten:

[5]:
```
sums = df["2021-12"].groupby([df["Language"], df["Title"]]).sum()

sums
```

[5]:
```
Language  Title
de        Jupyter Tutorial    19651.0
          PyViz Tutorial       2573.0
          Python Basics         525.0
en        Jupyter Tutorial     4722.0
          Python Basics         157.0
Name: 2021-12, dtype: float64
```

Hier haben wir die Daten anhand von zwei Schlüsseln gruppiert, und die resultierende Reihe hat nun einen hierarchischen Index, der aus den beobachteten eindeutigen Schlüsselpaaren besteht:

[6]:
```
sums.unstack()
```

[6]:
```
Title     Jupyter Tutorial  PyViz Tutorial  Python Basics
Language
de                 19651.0          2573.0          525.0
en                  4722.0             NaN          157.0
```

Häufig befinden sich die Gruppierungsinformationen in demselben DataFrame wie die Daten, die ihr bearbeiten möchtet. In diesem Fall könnt ihr Spaltennamen (egal ob es sich um Zeichenketten, Zahlen oder andere Python-Objekte handelt) als Gruppenschlüssel übergeben:

[7]:
```
df.groupby("Title").sum()
```

[7]:
```
                Language  2021-12  2022-01  2022-02
Title
Jupyter Tutorial     deen  24373.0  33631.0  37304.0
PyViz Tutorial         de   2573.0   4873.0   3930.0
Python Basics        deen    682.0    512.0    502.0
```

Hierbei fällt auf, dass das Ergebnis keine Spalte `Language` enthält. Da es sich bei `df['Language']` nicht um numerische Daten handelt, stört sie im Tabellenlayout und wird daher automatisch aus dem Ergebnis ausgeschlossen. Standardmäßig werden alle numerischen Spalten aggregiert.

[8]:
```
df.groupby(["Title", "Language"]).sum()
```

[8]:
```
                          2021-12  2022-01  2022-02
Title            Language
Jupyter Tutorial de        19651.0  30134.0  33295.0
                 en         4722.0   3497.0   4009.0
PyViz Tutorial   de         2573.0   4873.0   3930.0
Python Basics    de          525.0    427.0    276.0
                 en          157.0     85.0    226.0
```

Unabhängig vom Ziel der Verwendung von `groupby` ist eine allgemein nützliche `groupby`-Methode `size`, die eine Serie mit den Gruppengrößen zurückgibt:

[9]:
```
df.groupby(["Language"]).size()
```

[9]:
```
Language
de    3
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
en    2
dtype: int64
```

Hinweis

Alle fehlenden Werte in einem Gruppenschlüssel werden standardmäßig aus dem Ergebnis ausgeschlossen. Dieses Verhalten kann deaktiviert werden, indem `dropna=False` an `groupby` übergeben wird.

[10]:
```
df.groupby("Language", dropna=False).size()
```

[10]:
```
Language
de     3
en     2
NaN    1
dtype: int64
```

[11]:
```
df.groupby(["Title", "Language"], dropna=False).size()
```

[11]:
```
Title             Language
Jupyter Tutorial  de          1
                  en          1
PyViz Tutorial    de          1
Python Basics     de          1
                  en          1
NaN               NaN         1
dtype: int64
```

Iteration über Gruppen

Das von `groupby` zurückgegebene Objekt unterstützt Iteration und erzeugt eine Folge von 2-Tupeln, die den Gruppennamen zusammen mit dem Datenpaket enthalten. Betrachten wir das Folgende:

[12]:
```
for name, group in df.groupby("Title"):
    print(name)
    print(group)
```

```
Jupyter Tutorial
              Title Language  2021-12  2022-01  2022-02
0  Jupyter Tutorial       de  19651.0  30134.0  33295.0
1  Jupyter Tutorial       en   4722.0   3497.0   4009.0
PyViz Tutorial
            Title Language  2021-12  2022-01  2022-02
2  PyViz Tutorial       de   2573.0   4873.0   3930.0
Python Basics
           Title Language  2021-12  2022-01  2022-02
4  Python Basics       de    525.0    427.0    276.0
5  Python Basics       en    157.0     85.0    226.0
```

Bei mehreren Schlüsseln ist das erste Element des Tupels ein Tupel von Schlüsselwerten:

[13]:
```
for (i1, i2), group in df.groupby(["Title", "Language"]):
    print((i1, i2))
    print(group)
```

```
('Jupyter Tutorial', 'de')
              Title Language  2021-12  2022-01  2022-02
0  Jupyter Tutorial       de  19651.0  30134.0  33295.0
('Jupyter Tutorial', 'en')
              Title Language  2021-12  2022-01  2022-02
1  Jupyter Tutorial       en   4722.0   3497.0   4009.0
('PyViz Tutorial', 'de')
             Title Language  2021-12  2022-01  2022-02
2  PyViz Tutorial       de   2573.0   4873.0   3930.0
('Python Basics', 'de')
           Title Language  2021-12  2022-01  2022-02
4  Python Basics       de    525.0    427.0    276.0
('Python Basics', 'en')
           Title Language  2021-12  2022-01  2022-02
5  Python Basics       en    157.0     85.0    226.0
```

Als nächstes wollen wir ein `dict` der Daten als Einzeiler ausgeben:

[14]:
```
books = dict(list(df.groupby("Title")))

books
```

[14]:
```
{'Jupyter Tutorial':              Title Language  2021-12  2022-01  2022-02
 0  Jupyter Tutorial       de  19651.0  30134.0  33295.0
 1  Jupyter Tutorial       en   4722.0   3497.0   4009.0,
 'PyViz Tutorial':             Title Language  2021-12  2022-01  2022-02
 2  PyViz Tutorial       de   2573.0   4873.0   3930.0,
 'Python Basics':            Title Language  2021-12  2022-01  2022-02
 4  Python Basics       de    525.0    427.0    276.0
 5  Python Basics       en    157.0     85.0    226.0}
```

Standardmäßig gruppiert `groupby` auf `axis=0`, aber ihr könnt auch auf jeder der anderen Achsen gruppieren. Zum Beispiel könnten wir die Spalten unseres Beispiels `df` hier nach `dtype` gruppieren wie folgt:

[15]:
```
df.dtypes
```

[15]:
```
Title        object
Language     object
2021-12     float64
2022-01     float64
2022-02     float64
dtype: object
```

[16]:
```
grouped = df.groupby(df.dtypes, axis=1)
```

[17]:
```
for dtype, group in grouped:
    print(dtype)
    print(group)
```

```
float64
   2021-12  2022-01  2022-02
0  19651.0  30134.0  33295.0
1   4722.0   3497.0   4009.0
2   2573.0   4873.0   3930.0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3      NaN      NaN      NaN
4    525.0    427.0    276.0
5    157.0     85.0    226.0
object
              Title Language
0  Jupyter Tutorial       de
1  Jupyter Tutorial       en
2    PyViz Tutorial       de
3              None     None
4     Python Basics       de
5     Python Basics       en
```

Auswählen einer Spalte oder Untergruppe von Spalten

Die Indizierung eines `GroupBy`-Objekts, das aus einem DataFrame mit einem Spaltennamen oder einem Array von Spaltennamen erstellt wurde, hat den Effekt einer Spaltenunterteilung für die Aggregation. Dies bedeutet, dass:

[18]:
```
df.groupby("Title")["2021-12"]
df.groupby("Title")[["2022-01"]]
```

[18]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x13e402250>
```

sind vereinfachte Schreibweisen für:

[19]:
```
df["2021-12"].groupby(df["Title"])
df[["2022-01"]].groupby(df["Title"])
```

[19]:
```
<pandas.core.groupby.generic.DataFrameGroupBy object at 0x11f043590>
```

Insbesondere bei großen Datensätzen kann es wünschenswert sein, nur einige Spalten zu aggregieren. Um zum Beispiel im vorhergehenden Datensatz die Summe nur für die Spalte `01/2022` zu berechnen und das Ergebnis als DataFrame zu erhalten, könnten wir schreiben:

[20]:
```
df.groupby(["Title", "Language"])[["2022-01"]].sum()
```

[20]:
```
                          2022-01
Title            Language
Jupyter Tutorial de        30134.0
                 en         3497.0
PyViz Tutorial   de         4873.0
Python Basics    de          427.0
                 en           85.0
```

Das von dieser Indizierungsoperation zurückgegebene Objekt ist ein gruppierter DataFrame, wenn eine Liste oder ein Array übergeben wird, oder eine gruppierte Serie, wenn nur ein einzelner Spaltenname als Skalar übergeben wird:

[21]:
```
series_grouped = df.groupby(["Title", "Language"])["2022-01"]

series_grouped
```

[21]:
```
<pandas.core.groupby.generic.SeriesGroupBy object at 0x13e416710>
```

[22]:
```
series_grouped.sum()
```

[22]:
```
Title             Language
Jupyter Tutorial  de          30134.0
                  en           3497.0
PyViz Tutorial    de           4873.0
Python Basics     de            427.0
                  en             85.0
Name: 2022-01, dtype: float64
```

Gruppierung mit `dicts` und `Series`

Gruppierungsinformationen können auch in einer anderen Form als einem Array vorliegen:

[23]:
```
df.iloc[2:3, [2, 3]] = np.nan
```

Angenommen, ich habe eine Gruppenkorrespondenz für die Spalten und möchte die Spalten nach Gruppen zusammenfassen:

[24]:
```
mapping = {"2021-12": "Dec 2021", "2022-01": "Jan 2022", "2022-02": "Feb 2022"}
```

Nun könnte aus diesem `dict` ein Array konstruiert werden, um es an `groupby` zu übergeben, aber stattdessen können wir auch einfach das `dict` übergeben:

[25]:
```
by_column = df.groupby(mapping, axis=1)

by_column.sum()
```

[25]:

	Dec 2021	Feb 2022	Jan 2022
0	19651.0	33295.0	30134.0
1	4722.0	4009.0	3497.0
2	0.0	3930.0	0.0
3	0.0	0.0	0.0
4	525.0	276.0	427.0
5	157.0	226.0	85.0

Die gleiche Funktionalität gilt für `Series`, die als eine Abbildung mit fester Größe betrachtet werden können:

[26]:
```
map_series = pd.Series(mapping)

map_series
```

[26]:
```
2021-12    Dec 2021
2022-01    Jan 2022
2022-02    Feb 2022
dtype: object
```

[27]:
```
df.groupby(map_series, axis=1).count()
```

[27]:

	Dec 2021	Feb 2022	Jan 2022
0	1	1	1
1	1	1	1
2	0	1	0
3	0	0	0
4	1	1	1
5	1	1	1

Gruppieren mit Funktionen

Die Verwendung von Python-Funktionen ist im Vergleich zu einem `Dict` oder einer `Series` eine allgemeinere Methode zur Definition einer Gruppenzuordnung. Jede Funktion, die als Gruppenschlüssel übergeben wird, wird einmal pro Indexwert aufgerufen, wobei die Rückgabewerte als Gruppennamen verwendet werden. Betrachtet konkret den Beispiel-DataFrame aus dem vorherigen Abschnitt, das die Titel als Indexwerte enthält. Angenommen, Wenn ihr nach der Länge der Namen gruppieren wollt, könnt ihr zwar ein Array mit den Längen der Strings berechnen, aber es ist einfacher, die Funktion `len` zu übergeben:

[28]:
```
df = pd.DataFrame(
    [
        [19651, 30134, 33295],
        [4722, 3497, 4009],
        [2573, 4873, 3930],
        [525, 427, 276],
        [157, 85, 226],
    ],
    index=[
        "Jupyter Tutorial",
        "Jupyter Tutorial",
        "PyViz Tutorial",
        "Python Basics",
        "Python Basics",
    ],
    columns=["2021-12", "2022-01", "2022-02"],
)
```

[29]:
```
df.groupby(len).count()
```

[29]:
```
    2021-12  2022-01  2022-02
13        2        2        2
14        1        1        1
16        2        2        2
```

Das Mischen von Funktionen mit Arrays, Dicts oder Series ist kein Problem, da alles intern in Arrays umgewandelt wird:

[30]:
```
languages = ["de", "en", "de", "de", "en"]
```

[31]:
```
df.groupby([len, languages]).count()
```

[31]:
```
       2021-12  2022-01  2022-02
13 de        1        1        1
   en        1        1        1
14 de        1        1        1
16 de        1        1        1
   en        1        1        1
```

Gruppierung nach Indexebenen

Eine letzte praktische Funktion für hierarchisch indizierte Datensätze ist die Möglichkeit der Aggregation anhand einer der Indexebenen einer Achse. Schauen wir uns ein Beispiel an:

[32]:
```
version_hits = [
    [19651, 0, 30134, 0, 33295, 0],
    [4722, 1825, 3497, 2576, 4009, 3707],
    [2573, 0, 4873, 0, 3930, 0],
    [None, None, None, None, None, None],
    [525, 0, 427, 0, 276, 0],
    [157, 0, 85, 0, 226, 0],
]
df = pd.DataFrame(
    version_hits,
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            None,
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", None, "de", "en"],
    ],
    columns=[
        ["2021-12", "2021-12", "2022-01", "2022-01", "2022-02", "2022-02"],
        ["latest", "stable", "latest", "stable", "latest", "stable"],
    ],
)
df.columns.names = ["Month", "Version"]

df
```

[32]:

Month		2021-12		2022-01		2022-02	
Version		latest	stable	latest	stable	latest	stable
Jupyter Tutorial	de	19651.0	0.0	30134.0	0.0	33295.0	0.0
	en	4722.0	1825.0	3497.0	2576.0	4009.0	3707.0
PyViz Tutorial	de	2573.0	0.0	4873.0	0.0	3930.0	0.0
NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Python Basics	de	525.0	0.0	427.0	0.0	276.0	0.0
	en	157.0	0.0	85.0	0.0	226.0	0.0

[33]:
```
df.groupby(level="Month", axis=1).sum()
```

[33]:

Month		2021-12	2022-01	2022-02
Jupyter Tutorial	de	19651.0	30134.0	33295.0
	en	6547.0	6073.0	7716.0
PyViz Tutorial	de	2573.0	4873.0	3930.0
NaN	NaN	0.0	0.0	0.0
Python Basics	de	525.0	427.0	276.0
	en	157.0	85.0	226.0

2.4.14 Aggregation

Aggregationen beziehen sich auf jede Datentransformation, die skalare Werte aus Arrays erzeugt. In den vorangegangenen Beispielen wurden mehrere von ihnen verwendet, darunter `count` und `sum`. Ihr fragt euch nun vielleicht, was passiert, wenn ihr `sum()` auf ein `GroupBy`-Objekt anwendet. Für viele gängige Aggregationen, wie die in der folgenden Tabelle, gibt es optimierte Implementierungen. Sie sind jedoch nicht auf diesen Satz von Methoden beschränkt.

Funktionsname	Beschreibung
`any`, `all`	Gibt `True` zurück, wenn einer (ein oder mehrere Werte) oder alle Nicht-NA-Werte „truthy“ sind
`count`	Anzahl der Nicht-NA-Werte
`cummin`, `cummax`	Kumuliertes Minimum und Maximum der Nicht-NA-Werte
`cumsum`	Kumulative Summe der Nicht-NA-Werte
`cumprod`	Kumulatives Produkt von Nicht-NA-Werten
`first`, `last`	Erste und letzte Nicht-NA-Werte
`mean`	Mittelwert der Nicht-NA-Werte
`median`	Arithmetischer Median der Nicht-NA-Werte
`min`, `max`	Minimum und Maximum der Nicht-NA-Werte
`nth`	Abrufen des n-ten größten Wertes
`ohlc`	Berechnung von vier *Open-high-low-close*-Statistiken für zeitreihenähnliche Daten
`prod`	Produkt der Nicht-NA-Werte
`quantile`	Berechnet das Stichprobenquantil
`rank`	Ordinale Ränge von Nicht-NA-Werten, wie beim Aufruf von Series.rank
`sum`	Summe der Nicht-NA-Werte
`std`, `var`	Standardabweichung und Varianz der Stichprobe

Ihr könnt eigene Aggregationen verwenden und zusätzlich jede Methode aufrufen, die auch für das gruppierte Objekt definiert ist. Zum Beispiel wählt die `Series`-Methode `nsmallest` die kleinste angeforderte Anzahl von Werten aus den Daten aus.

Obwohl `nsmallest` nicht explizit für `GroupBy` implementiert ist, können wir es dennoch mit einer nicht optimierten Implementierung verwenden. Intern zerlegt `GroupBy` die `Series`, ruft `df.nsmallest(n)` für jeden Teil auf und fügt diese Ergebnisse dann im Ergebnisobjekt zusammen:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.DataFrame(
    {
        "Title": [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            None,
            "Python Basics",
            "Python Basics",
        ],
        "2021-12": [30134, 6073, 4873, None, 427, 95],
        "2022-01": [33295, 7716, 3930, None, 276, 226],
        "2022-02": [19651, 6547, 2573, None, 525, 157],
    }
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
)

df
```

[2]:
```
              Title  2021-12  2022-01  2022-02
0  Jupyter Tutorial  30134.0  33295.0  19651.0
1  Jupyter Tutorial   6073.0   7716.0   6547.0
2    PyViz Tutorial   4873.0   3930.0   2573.0
3              None      NaN      NaN      NaN
4     Python Basics    427.0    276.0    525.0
5     Python Basics     95.0    226.0    157.0
```

[3]:
```
grouped = df.groupby("Title")
```

[4]:
```
grouped["2022-01"].nsmallest(1)
```

[4]:
```
Title
Jupyter Tutorial  1    7716.0
PyViz Tutorial    2    3930.0
Python Basics     5     226.0
Name: 2022-01, dtype: float64
```

Um eine eigene Aggregationsfunktionen zu verwenden, übergebt eine beliebige Funktion, die ein Array aggregiert, an die Methode `aggregate` oder `agg`:

[5]:
```
def range(arr):
    return arr.max() - arr.min()


grouped.agg(range)
```

[5]:
```
                 2021-12  2022-01  2022-02
Title
Jupyter Tutorial  24061.0  25579.0  13104.0
PyViz Tutorial        0.0      0.0      0.0
Python Basics       332.0     50.0    368.0
```

Ihr werdet feststellen, dass einige Methoden wie `describe` ebenfalls funktionieren, auch wenn es sich dabei streng genommen nicht um Aggregationen handelt:

[6]:
```
grouped.describe()
```

[6]:
```
                 2021-12                                                  \
                   count     mean           std     min       25%      50%
Title
Jupyter Tutorial     2.0  18103.5  17013.696262  6073.0  12088.25  18103.5
PyViz Tutorial       1.0   4873.0           NaN  4873.0   4873.00   4873.0
Python Basics        2.0    261.0    234.759451    95.0    178.00    261.0

                              2022-01           ...                     \
                       75%      max   count     mean  ...       75%      max
Title                                                 ...
Jupyter Tutorial  24118.75  30134.0     2.0  20505.5  ...  26900.25  33295.0
PyViz Tutorial     4873.00   4873.0     1.0   3930.0  ...   3930.00   3930.0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Python Basics       344.00    427.0     2.0    251.0  ...    263.50    276.0

                 2022-02                                                \
                   count     mean          std     min     25%      50%
Title
Jupyter Tutorial     2.0  13099.0  9265.927261  6547.0  9823.0  13099.0
PyViz Tutorial       1.0   2573.0          NaN  2573.0  2573.0   2573.0
Python Basics        2.0    341.0   260.215295   157.0   249.0    341.0


                      75%      max
Title
Jupyter Tutorial  16375.0  19651.0
PyViz Tutorial     2573.0   2573.0
Python Basics       433.0    525.0

[3 rows x 24 columns]
```

Hinweis

Benutzerdefinierte Aggregationsfunktionen sind im Allgemeinen viel langsamer als die optimierten Funktionen in der obigen Tabelle. Dies liegt daran, dass bei der Erstellung der Zwischendatensätze für die Gruppe ein gewisser Mehraufwand entsteht (Funktionsaufrufe, Umordnung von Daten).

Spaltenweise zusätzliche Funktionen

Wie wir bereits gesehen haben, ist das Aggregieren einer `Series` oder aller Spalten eines `DataFrame` eine Frage der Verwendung von `aggregate` (oder `agg`) mit der gewünschten Funktion oder des Aufrufs einer Methode wie `mean` oder `std`. Es kommt jedoch häufiger vor, dass gleichzeitig mit einer anderen Funktion je nach Spalte oder mit mehreren Funktionen aggregiert werden soll.

[7]:
```
grouped.agg("mean")
```

[7]:
```
                  2021-12  2022-01  2022-02
Title
Jupyter Tutorial  18103.5  20505.5  13099.0
PyViz Tutorial     4873.0   3930.0   2573.0
Python Basics       261.0    251.0    341.0
```

Wenn ihr stattdessen eine Liste von Funktionen oder Funktionsnamen übergebt, erhaltet ihr einen `DataFrame` mit Spaltennamen aus den Funktionen zurück:

[8]:
```
grouped.agg(["mean", "std", range])
```

[8]:
```
                  2021-12                          2022-01                \
                     mean           std    range     mean           std
Title
Jupyter Tutorial  18103.5  17013.696262  24061.0  20505.5  18087.084356
PyViz Tutorial     4873.0           NaN      0.0   3930.0           NaN
Python Basics       261.0    234.759451    332.0    251.0     35.355339

                        2022-02
                   range     mean          std    range
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Title
Jupyter Tutorial  25579.0  13099.0  9265.927261  13104.0
PyViz Tutorial        0.0   2573.0          NaN      0.0
Python Basics        50.0    341.0   260.215295    368.0
```

Hier haben wir `agg` eine Liste von Aggregationsfunktionen übergeben, die unabhängig voneinander für die Datengruppen ausgewertet werden sollen.

Ihr braucht die Namen, die `GroupBy` den Spalten gibt, nicht zu akzeptieren; insbesondere haben Lambda-Funktionen den Namen `<lambda>`, was ihre Identifizierung erschwert. Wenn ihr eine Liste von Tupels übergebt, wird das erste Element jedes Tuples als Spaltenname im `DataFrame` verwendet:

[9]:
```
grouped.agg(
    [("Mittelwert", "mean"), ("Standardabweichung", "std"), ("Bereich", range)]
)
```

[9]:
```
                    2021-12                                  2022-01  \
                Mittelwert Standardabweichung  Bereich Mittelwert
Title
Jupyter Tutorial    18103.5       17013.696262  24061.0    20505.5
PyViz Tutorial       4873.0                NaN      0.0     3930.0
Python Basics         261.0         234.759451    332.0      251.0

                                          2022-02                    \
                Standardabweichung  Bereich Mittelwert Standardabweichung
Title
Jupyter Tutorial       18087.084356  25579.0    13099.0        9265.927261
PyViz Tutorial                  NaN      0.0     2573.0                NaN
Python Basics             35.355339     50.0      341.0         260.215295


                  Bereich
Title
Jupyter Tutorial  13104.0
PyViz Tutorial        0.0
Python Basics       368.0
```

Bei einem `DataFrame` habt ihr die Möglichkeit, eine Liste von Funktionen anzugeben, die auf alle Spalten oder auf verschiedene Funktionen pro Spalte angewendet werden. Nehmen wir an, wir möchten die gleichen drei Statistiken für die Spalten berechnen:

[10]:
```
stats = ["count", "mean", "max"]
evaluations = grouped.agg(stats)

evaluations
```

[10]:
```
                  2021-12                    2022-01                    2022-02  \
                    count     mean      max   count     mean      max   count
Title
Jupyter Tutorial        2  18103.5  30134.0       2  20505.5  33295.0       2
PyViz Tutorial          1   4873.0   4873.0       1   3930.0   3930.0       1
Python Basics           2    261.0    427.0       2    251.0    276.0       2
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                    mean      max
Title
Jupyter Tutorial  13099.0  19651.0
PyViz Tutorial     2573.0   2573.0
Python Basics       341.0    525.0
```

Wie ihr sehen könnt, hat der resultierende `DataFrame` hierarchische Spalten, genauso wie ihr sie bekommen würdet, wenn ihr jede Spalte separat aggregieren und pandas.concat verwenden würdet, um die Ergebnisse zusammenzufügen, indem ihr die Spaltennamen als Schlüsselargument verwendet:

[11]:
```
evaluations["2021-12"]
```

[11]:
```
                  count     mean      max
Title
Jupyter Tutorial      2  18103.5  30134.0
PyViz Tutorial        1   4873.0   4873.0
Python Basics         2    261.0    427.0
```

Wie zuvor kann eine Liste von Tupeln mit benutzerdefinierten Namen übergeben werden:

[12]:
```
tuples = [("Mittelwert", "mean"), ("Varianz", np.var)]

grouped[["2021-12", "2022-01"]].agg(tuples)
```

[12]:
```
                   2021-12                2022-01
                Mittelwert      Varianz Mittelwert      Varianz
Title
Jupyter Tutorial    18103.5  289465860.5    20505.5  327142620.5
PyViz Tutorial       4873.0          NaN     3930.0          NaN
Python Basics         261.0      55112.0      251.0       1250.0
```

Nehmen wir nun an, dass potenziell verschiedene Funktionen auf eine oder mehrere der Spalten angewendet werden sollen, dann übergeben wir dazu ein `dict` an `agg`, das eine Zuordnung von Spaltennamen zu einer der Funktionsspezifikationen enthält:

[13]:
```
grouped.agg({"2021-12": "mean", "2022-01": np.var})
```

[13]:
```
                  2021-12      2022-01
Title
Jupyter Tutorial  18103.5  327142620.5
PyViz Tutorial     4873.0          NaN
Python Basics       261.0       1250.0
```

[14]:
```
grouped.agg({"2021-12": ["min", "max", "mean", "std"], "2022-01": "sum"})
```

[14]:
```
                 2021-12                                  2022-01
                     min      max     mean           std      sum
Title
Jupyter Tutorial  6073.0  30134.0  18103.5  17013.696262  41011.0
PyViz Tutorial    4873.0   4873.0   4873.0           NaN   3930.0
Python Basics       95.0    427.0    261.0    234.759451    502.0
```

Aggregierte Daten ohne Zeilenindizes zurückgeben

In allen bisherigen Beispielen werden die aggregierten Daten mit einem Index zurückgegeben. Da dies nicht immer erwünscht ist, könnt ihr dieses Verhalten in den meisten Fällen deaktivieren, indem ihr `as_index=False` an `groupby` übergebt:

[15]:
```
grouped.agg([range], as_index=False).mean()
```

[15]:
```
2021-12  8131.000000
2022-01  8543.000000
2022-02  4490.666667
dtype: float64
```

Durch die Verwendung der Methode `as_index=False` werden einige unnötige Berechnungen vermieden. Natürlich ist es jederzeit möglich, das Ergebnis wieder mit Index zu erhalten, indem `reset_index` für das Ergebnis aufgerufen wird.

2.4.15 Apply

Die am allgemeinsten einsetzbare `GroupBy`-Methode ist `apply`. Sie teilt das zu bearbeitende Objekt auf, ruft die übergebene Funktion auf jedem Teil auf und versucht dann, die Teile miteinander zu verketten.

Nehmen wir an, wir wollen die fünf größten `hit`-Werte nach Gruppen auswählen. Hierzu schreiben wir zunächst eine Funktion, die die Zeilen mit den größten Werten in einer bestimmten Spalte auswählt:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.DataFrame(
    {
        "2021-12": [30134, 6073, 4873, None, 427, 95],
        "2022-01": [33295, 7716, 3930, None, 276, 226],
        "2022-02": [19651, 6547, 2573, None, 525, 157],
    },
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", "en", "de", "en"],
    ],
)
df.index.names = ["Title", "Language"]

df
```

[2]:
```
                          2021-12  2022-01  2022-02
Title           Language
Jupyter Tutorial de        30134.0  33295.0  19651.0
                 en         6073.0   7716.0   6547.0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
PyViz Tutorial   de          4873.0   3930.0   2573.0
                 en             NaN      NaN      NaN
Python Basics    de           427.0    276.0    525.0
                 en            95.0    226.0    157.0
```

[3]:
```
def top(df, n=5, column="2021-12"):
    return df.sort_values(by=column, ascending=False)[:n]


top(df, n=3)
```

[3]:
```
                          2021-12  2022-01  2022-02
Title            Language
Jupyter Tutorial de         30134.0  33295.0  19651.0
                 en          6073.0   7716.0   6547.0
PyViz Tutorial   de          4873.0   3930.0   2573.0
```

Wenn wir nun z.B. nach Titeln gruppieren und `apply` mit dieser Funktion aufrufen, erhalten wir Folgendes:

[4]:
```
grouped_titles = df.groupby("Title")

grouped_titles.apply(top)
```

[4]:
```
                          2021-12  2022-01  2022-02
Title            Language
Jupyter Tutorial de         30134.0  33295.0  19651.0
                 en          6073.0   7716.0   6547.0
PyViz Tutorial   de          4873.0   3930.0   2573.0
                 en             NaN      NaN      NaN
Python Basics    de           427.0    276.0    525.0
                 en            95.0    226.0    157.0
```

Was ist hier passiert? Die obere Funktion wird für jede Zeilengruppe des DataFrame aufgerufen, und dann werden die Ergebnisse mit pandas.concat zusammengefügt, wobei die Teile mit den Gruppennamen gekennzeichnet werden. Das Ergebnis hat daher einen hierarchischen Index, dessen innere Ebene Indexwerte aus dem ursprünglichen DataFrame enthält.

Wenn ihr eine Funktion an `apply` übergebt, die andere Argumente oder Schlüsselwörter benötigt, könnt ihr diese nach der Funktion übergeben:

[5]:
```
grouped_titles.apply(top, n=1)
```

[5]:
```
                                        2021-12  2022-01  2022-02
Title            Title            Language
Jupyter Tutorial Jupyter Tutorial de     30134.0  33295.0  19651.0
PyViz Tutorial   PyViz Tutorial   de      4873.0   3930.0   2573.0
Python Basics    Python Basics    de       427.0    276.0    525.0
```

Wir haben nun die grundlegende Verwendungsweise von `apply` gesehen. Was innerhalb der übergebenen Funktion geschieht, ist sehr vielseitig und bleibt euch überlassen; sie muss nur ein pandas-Objekt oder einen Einzelwert zurückgeben. Im Folgend werden wir daher hauptsächlich Beispielen zeigen, die euch Anregungen geben können, wie ihr verschiedene Probleme mit `groupby` lösen könnt.

Zunächst vergegenwärtigen wir uns nochmal an `describe`, aufgerufen über dem `GroupBy`-Objekt:

[6]:
```
result = grouped_titles.describe()

result
```

[6]:
```
                 2021-12                                                \
                   count     mean           std     min       25%      50%
Title
Jupyter Tutorial     2.0  18103.5  17013.696262  6073.0  12088.25  18103.5
PyViz Tutorial       1.0   4873.0           NaN  4873.0   4873.00   4873.0
Python Basics        2.0    261.0    234.759451    95.0    178.00    261.0

                                 2022-01           ...                    \
                       75%      max   count     mean  ...       75%      max
Title                                                  ...
Jupyter Tutorial  24118.75  30134.0     2.0  20505.5  ...  26900.25  33295.0
PyViz Tutorial     4873.00   4873.0     1.0   3930.0  ...   3930.00   3930.0
Python Basics       344.00    427.0     2.0    251.0  ...    263.50    276.0

                 2022-02                                              \
                   count     mean          std     min     25%      50%
Title
Jupyter Tutorial     2.0  13099.0  9265.927261  6547.0  9823.0  13099.0
PyViz Tutorial       1.0   2573.0          NaN  2573.0  2573.0   2573.0
Python Basics        2.0    341.0   260.215295   157.0   249.0    341.0


                      75%      max
Title
Jupyter Tutorial  16375.0  19651.0
PyViz Tutorial     2573.0   2573.0
Python Basics       433.0    525.0

[3 rows x 24 columns]
```

Wenn ihr innerhalb von `GroupBy` eine Methode wie `describe` aufruft, ist dies eigentlich nur eine Abkürzung für:

[7]:
```
f = lambda x: x.describe()

grouped_titles.apply(f)
```

[7]:
```
                            2021-12       2022-01       2022-02
Title
Jupyter Tutorial count      2.000000      2.000000      2.000000
                 mean   18103.500000  20505.500000  13099.000000
                 std    17013.696262  18087.084356   9265.927261
                 min     6073.000000   7716.000000   6547.000000
                 25%    12088.250000  14110.750000   9823.000000
                 50%    18103.500000  20505.500000  13099.000000
                 75%    24118.750000  26900.250000  16375.000000
                 max    30134.000000  33295.000000  19651.000000
PyViz Tutorial   count      1.000000      1.000000      1.000000
                 mean    4873.000000   3930.000000   2573.000000
                 std             NaN           NaN           NaN
                 min     4873.000000   3930.000000   2573.000000
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                  25%     4873.000000   3930.000000   2573.000000
                  50%     4873.000000   3930.000000   2573.000000
                  75%     4873.000000   3930.000000   2573.000000
                  max     4873.000000   3930.000000   2573.000000
Python Basics     count      2.000000      2.000000      2.000000
                  mean     261.000000    251.000000    341.000000
                  std      234.759451     35.355339    260.215295
                  min       95.000000    226.000000    157.000000
                  25%      178.000000    238.500000    249.000000
                  50%      261.000000    251.000000    341.000000
                  75%      344.000000    263.500000    433.000000
                  max      427.000000    276.000000    525.000000
```

Unterdrückung der Gruppenschlüssel

In den vorangegangenen Beispielen habr ihr gesehen, dass das resultierende Objekt einen hierarchischen Index hat, der aus den Gruppenschlüsseln zusammen mit den Indizes der einzelnen Teile des ursprünglichen Objekts gebildet wird. Ihr können dies deaktivieren, indem ihr `group_keys=False` an `groupby` übergebt:

[8]:
```
grouped_lang = df.groupby("Language", group_keys=False)

grouped_lang.apply(top)
```

[8]:
```
                          2021-12  2022-01  2022-02
Title            Language
Jupyter Tutorial de        30134.0  33295.0  19651.0
PyViz Tutorial   de         4873.0   3930.0   2573.0
Python Basics    de          427.0    276.0    525.0
Jupyter Tutorial en         6073.0   7716.0   6547.0
Python Basics    en           95.0    226.0    157.0
PyViz Tutorial   en            NaN      NaN      NaN
```

Quantil- und Bucket-Analyse

Wie bereits in *Diskretisierung und Gruppierung* beschrieben, verfügt pandas über einige Werkzeuge, insbesondere `cut` und `qcut`, um Daten in Buckets mit Bins eurer Wahl oder nach Stichprobenquantilen aufzuteilen. Kombiniert man diese Funktionen mit `groupby`, kann man bequem eine Bucket- oder Quantilanalyse für einen Datensatz durchführen. Betrachtet einen einfachen Zufallsdatensatz und eine gleich lange Bucket-Kategorisierung mit `cut`:

[9]:
```
df2 = pd.DataFrame(
    {
        "data1": np.random.randn(1000),
        "data2": np.random.randn(1000)
    }
)
quartiles = pd.cut(df2.data1, 4)

quartiles[:10]
```

[9]:
```
0     (-0.0817, 1.433]
1     (-0.0817, 1.433]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2     (-0.0817, 1.433]
3    (-1.597, -0.0817]
4       (1.433, 2.948]
5     (-0.0817, 1.433]
6    (-1.597, -0.0817]
7     (-0.0817, 1.433]
8    (-1.597, -0.0817]
9    (-1.597, -0.0817]
Name: data1, dtype: category
Categories (4, interval[float64, right]): [(-3.118, -1.597] < (-1.597, -0.0817] < (-0.
↪0817, 1.433] < (1.433, 2.948]]
```

Das von `cut` zurückgegebene `Categorical`-Objekt kann direkt an `groupby` übergeben werden. Wir könnten also eine Reihe von Gruppenstatistiken für die Quartile wie folgt berechnen:

[10]:
```
def stats(group):
    return pd.DataFrame(
        {
            "min": group.min(),
            "max": group.max(),
            "count": group.count(),
            "mean": group.mean(),
        }
    )


grouped_quart = df2.groupby(quartiles)

grouped_quart.apply(stats)
```

[10]:
```
                             min       max  count      mean
data1
(-3.118, -1.597]  data1 -3.111451 -1.605312     53 -2.005040
                  data2 -2.032455  1.733728     53 -0.061672
(-1.597, -0.0817] data1 -1.589185 -0.081825    431 -0.714370
                  data2 -2.882329  3.902974    431  0.073316
(-0.0817, 1.433]  data1 -0.077836  1.408519    441  0.554646
                  data2 -2.853656  2.558197    441 -0.068681
(1.433, 2.948]    data1  1.442869  2.948015     75  1.882466
                  data2 -1.941170  3.242524     75  0.156225
```

Dies waren Buckets gleicher Länge; um Buckets gleicher Größe auf der Grundlage von Stichprobenquantilen zu berechnen, können wir `qcut` verwenden. Ich übergebe `labels=False`, um nur Quantilzahlen zu erhalten:

[11]:
```
quartiles_samp = pd.qcut(df2.data1, 4, labels=False)
grouped_quart_samp = df2.groupby(quartiles_samp)

grouped_quart_samp.apply(stats)
```

[11]:
```
                  min       max  count      mean
data1
0     data1 -3.111451 -0.726183    250 -1.287129
      data2 -2.882329  2.629725    250  0.021474
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
1     data1 -0.726057 -0.013731    250 -0.372047
      data2 -2.708554  3.902974    250  0.088167
2     data1 -0.013457  0.659397    250  0.276518
      data2 -2.853656  2.180024    250 -0.091263
3     data1  0.660929  2.948015    250  1.269152
      data2 -2.527610  3.242524    250  0.020658
```

Daten mit gruppenspezifischen Werten auffüllen

Wenn ihr fehlende Daten bereinigt, werdet ihr in einigen Fällen Datenbeobachtungen mit `dropna` ersetzen, aber in anderen Fällen möchtet ihr vielleicht die Nullwerte (`NA`) mit einem festen Wert oder einem aus den Daten abgeleiteten Wert auffüllen. `fillna` ist das richtige Werkzeug dafür; hier fülle ich zum Beispiel die Nullwerte mit dem Mittelwert auf:

[12]:
```
s = pd.Series(np.random.randn(8))
s[::3] = np.nan

s
```

[12]:
```
0         NaN
1   -0.004095
2   -0.525244
3         NaN
4    1.362079
5   -1.416516
6         NaN
7    0.891944
dtype: float64
```

[13]:
```
s.fillna(s.mean())
```

[13]:
```
0    0.061634
1   -0.004095
2   -0.525244
3    0.061634
4    1.362079
5   -1.416516
6    0.061634
7    0.891944
dtype: float64
```

Hier sind einige Beispieldaten zu meinen Tutorials, die in deutsch- und englischsprachige Ausgaben unterteilt sind:

Angenommen, ihr möchtet, dass der Füllwert je nach Gruppe variiert. Diese Werte können vordefiniert werden, und da die Gruppen ein internes Namensattribut `name` haben, könnt ihr dieses mit `apply` verwenden:

[14]:
```
fill_values = {"de": 10632, "en": 3469}
fill_func = lambda g: g.fillna(fill_values[g.name])

df.groupby("Language").apply(fill_func)
```

[14]:
```
                                    2021-12  2022-01  2022-02
Language Title            Language
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
de       Jupyter Tutorial de        30134.0  33295.0  19651.0
         PyViz Tutorial   de         4873.0   3930.0   2573.0
         Python Basics    de          427.0    276.0    525.0
en       Jupyter Tutorial en         6073.0   7716.0   6547.0
         PyViz Tutorial   en         3469.0   3469.0   3469.0
         Python Basics    en           95.0    226.0    157.0
```

Ihr könnt auch die Daten gruppieren und `apply` mit einer Funktion zu verwenden, die `fillna` für jedes Datenpaket aufruft:

[15]:
```
fill_mean = lambda g: g.fillna(g.mean())

df.groupby("Language").apply(fill_mean)
```

[15]:
```
                                  2021-12  2022-01  2022-02
Language Title            Language
de       Jupyter Tutorial de        30134.0  33295.0  19651.0
         PyViz Tutorial   de         4873.0   3930.0   2573.0
         Python Basics    de          427.0    276.0    525.0
en       Jupyter Tutorial en         6073.0   7716.0   6547.0
         PyViz Tutorial   en         3084.0   3971.0   3352.0
         Python Basics    en           95.0    226.0    157.0
```

Gruppierter gewichteter Durchschnitt

Da Operationen zwischen Spalten in einem `DataFrame` oder zwei `Series` möglich sind, können wir z.B. den gruppengewichteten Durchschnitt berechnen:

[16]:
```
df3 = pd.DataFrame(
    {
        "category": ["de", "de", "de", "de", "en", "en", "en", "en"],
        "data": np.random.randint(100000, size=8),
        "weights": np.random.rand(8),
    }
)

df3
```

[16]:
```
  category   data   weights
0       de  11386  0.748662
1       de  15461  0.524961
2       de  95386  0.460069
3       de  95307  0.067965
4       en  13249  0.646899
5       en  77389  0.313475
6       en  98805  0.651772
7       en  80871  0.782137
```

Der nach Kategorien gewichtete Gruppendurchschnitt würde dann lauten:

[17]:
```
grouped_cat = df3.groupby("category")
get_wavg = lambda g: np.average(g["data"], weights=g["weights"])
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
grouped_cat.apply(get_wavg)
```

[17]:
```
category
de    37189.285575
en    67026.662870
dtype: float64
```

Korrelation

Eine interessante Aufgabe könnte darin bestehen, einen `DataFrame` zu berechnen, der aus den prozentualen Veränderungen besteht.

Zu diesem Zweck erstellen wir zunächst eine Funktion, die die paarweise Korrelation der Spalte `2021-12` mit den nachfolgenden Spalten berechnet:

[18]:
```
corr = lambda x: x.corrwith(x["2021-12"])
```

Als nächstes berechnen wir die prozentuale Veränderung:

[19]:
```
pcts = df.pct_change().dropna()
```

Schließlich gruppieren wir diese prozentualen Änderungen nach Jahr, das aus jeder Zeilenbeschriftung mit einer einzeiligen Funktion extrahiert werden kann, die das Attribut Jahr jeder Datumsbeschriftung zurückgibt:

[20]:
```
by_language = pcts.groupby("Language")

by_language.apply(corr)
```

[20]:
```
          2021-12   2022-01  2022-02
Language
de            1.0  1.000000  1.00000
en            1.0  0.699088  0.99781
```

[21]:
```
by_language.apply(lambda g: g["2021-12"].corr(g["2022-01"]))
```

[21]:
```
Language
de    1.000000
en    0.699088
dtype: float64
```

Performance-Probleme mit `apply`

Da die `apply`-Methode typischerweise auf jeden einzelnen Wert in einer `Series` wirkt, wird die Funktion für jeden Wert einmal aufgerufen. Wenn ihr tausende Werte habt, wird die Funktion auch tausende Male aufgerufen. Dadurch werden die schnellen Vektorisierungen von pandas ignoriert sofern ihr keine NumPy-Funktionen verwendet, und langsames Python verwendet. Zum Beispiel haben wir zuvor die Daten nach Titel gruppiert und dann unsere `top`-Methode mit `apply` aufgerufen. Messen wir hierfür die Zeit:

[22]:
```
%%timeit
grouped_titles.apply(top)
```

```
562 µs ± 14.5 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Wir können dasselbe Ergebnis auch ohne `apply` erhalten indem wir unserer Methode `top` den DataFrame übergeben:

[23]:
```
%%timeit
top(df)
```

```
45.2 µs ± 455 ns per loop (mean ± std. dev. of 7 runs, 10,000 loops each)
```

Diese Berechnung ist 18 mal schneller.

Optimieren von `apply` mit Cython

Nicht immer lässt sich jedoch für `apply`so einfach eine Alternative finden. Numerische Operationen wie unsere `top`-Methode lässt sich jedoch mit Cython schneller machen. Um Cython in Jupyyter zu nutzen, verwenden wir die folgende *IPython-Magie*:

[24]:
```
%load_ext Cython
```

Dann können wir unsere `top`-Funktion mit Cython definieren:

[25]:
```
%%cython
def top_cy(df, n=5, column="2021-12"):
    return df.sort_values(by=column, ascending=False)[:n]
```

[26]:
```
%%timeit
grouped_titles.apply(top_cy)
```

```
571 µs ± 4.62 µs per loop (mean ± std. dev. of 7 runs, 1,000 loops each)
```

Damit haben wir noch nicht wirklich viel gewonnen. Weitere Optimierungsmöglichkeiten wären nun, dass wir mit `cpdef` den Typ im Cython-Code definieren. Dafür müssten wir jedoch unsere Methode umbauen, da dann kein `DataFrame` mehr übergeben werden kann.

2.4.16 Pivot-Tabellen und Kreuztabellen

Eine Pivot-Tabelle ist ein Werkzeug zur Datenzusammenfassung, das häufig in Tabellenkalkulationsprogrammen und anderer Datenanalysesoftware zu finden ist. Sie fasst eine Tabelle mit Daten nach einem oder mehreren Schlüsseln zusammen und ordnet die Daten in einem Rechteck an, wobei einige der Gruppenschlüssel entlang der Zeilen und einige entlang der Spalten angeordnet sind. Pivot-Tabellen in Python mit pandas werden durch die *groupby*-Funktion in Kombination mit Umformungsoperationen unter Verwendung *hierarchischer Indizierung* ermöglicht. DataFrame hat eine pivot_table-Methode, und es gibt auch eine Top-Level-Funktion pandas.pivot_table. `pivot_table` bietet nicht nur eine bequeme Schnittstelle zu `groupby`, sondern kann auch Teilsummen (`margins`) hinzufügen.

Nehmen wir an, wir wollten eine Tabelle mit Gruppenmittelwerten (der Standardaggregationstyp von `pivot_table`) berechnen, die nach Titel und Sprache in den Zeilen geordnet ist:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
df = pd.DataFrame(
    {
        "2021-12": [30134, 6073, 4873, None, 427, 95],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "2022-01": [33295, 7716, 3930, None, 276, 226],
        "2022-02": [19651, 6547, 2573, None, 525, 157],
    },
    index=[
        [
            "Jupyter Tutorial",
            "Jupyter Tutorial",
            "PyViz Tutorial",
            "PyViz Tutorial",
            "Python Basics",
            "Python Basics",
        ],
        ["de", "en", "de", None, "de", "en"],
    ],
)

df.index.names = ["Title", "Language"]
```

[3]:
```
df.pivot_table(index=["Title", "Language"])
```

[3]:
```
                          2021-12  2022-01  2022-02
Title            Language
Jupyter Tutorial de        30134.0  33295.0  19651.0
                 en         6073.0   7716.0   6547.0
PyViz Tutorial   de         4873.0   3930.0   2573.0
Python Basics    de          427.0    276.0    525.0
                 en           95.0    226.0    157.0
```

Das hätte man auch direkt mit `groupby` machen können.

Nehmen wir nun an, wir wollen den Durchschnitt jeden Monats nehmen und zusätzlich nach `Title` gruppieren. Ich werde `Title` und `Language` in die Tabellenspalten und die Monate in die Zeilen setzen:

[4]:
```
df.pivot_table(columns=["Title", "Language"])
```

[4]:
```
Title    Jupyter Tutorial         PyViz Tutorial Python Basics
Language               de      en             de           de     en
2021-12           30134.0  6073.0         4873.0        427.0   95.0
2022-01           33295.0  7716.0         3930.0        276.0  226.0
2022-02           19651.0  6547.0         2573.0        525.0  157.0
```

Wir können diese Tabelle um Teilsummen erweitern, indem wir `margins=True` übergeben. Dies hat den Effekt, dass alle Zeilen- und Spaltenbeschriftungen hinzugefügt werden, wobei die entsprechenden Werte die Gruppenstatistiken für alle Daten innerhalb einer einzelnen Ebene sind:

[5]:
```
df.pivot_table(columns=["Title", "Language"], margins=True)
```

[5]:
```
Title    Jupyter Tutorial                  PyViz Tutorial          \
Language               de      en      All             de     All
2021-12           30134.0  6073.0  18103.5         4873.0  4873.0
2022-01           33295.0  7716.0  20505.5         3930.0  3930.0
2022-02           19651.0  6547.0  13099.0         2573.0  2573.0

Title    Python Basics
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Language            de     en    All
2021-12          427.0   95.0  261.0
2022-01          276.0  226.0  251.0
2022-02          525.0  157.0  341.0
```

In diesem Fall sind die Werte für `All` die Mittelwerte.

Um eine andere Aggregationsfunktion als `mean` zu verwenden, übergebt diese an das Schlüsselwortargument `aggfunc`. Mit `sum` erhaltet ihr beispielsweise die Summe:

```
[6]: df.pivot_table(columns=["Title", "Language"], aggfunc=sum, margins=True)
```

```
[6]: Title    Jupyter Tutorial                  PyViz Tutorial          \
     Language               de      en      All             de     All
     2021-12           30134.0  6073.0  36207.0         4873.0  4873.0
     2022-01           33295.0  7716.0  41011.0         3930.0  3930.0
     2022-02           19651.0  6547.0  26198.0         2573.0  2573.0

     Title    Python Basics
     Language            de     en    All
     2021-12          427.0   95.0  522.0
     2022-01          276.0  226.0  502.0
     2022-02          525.0  157.0  682.0
```

Wenn einige Kombinationen leer (oder anderweitig `NA`) sind, könnt ihr `fill_value` übergeben:

```
[7]: df.pivot_table(columns=["Title", "Language"], aggfunc=sum, margins=True, fill_value=0)
```

```
[7]: Title    Jupyter Tutorial             PyViz Tutorial       Python Basics  \
     Language               de    en    All             de   All            de
     2021-12             30134  6073  36207           4873  4873           427
     2022-01             33295  7716  41011           3930  3930           276
     2022-02             19651  6547  26198           2573  2573           525

     Title
     Language   en  All
     2021-12    95  522
     2022-01   226  502
     2022-02   157  682
```

`pivot_table`-Optionen:

Funktionsname	Beschreibung
`values`	Spaltenname(n) zum Aggregieren; standardmäßig werden alle numerischen Spalten aggregiert
`index`	Spaltennamen oder andere Gruppenschlüssel, die in den Zeilen der resultierenden Pivot-Tabelle gruppiert werden sollen
`columns`	Spaltennamen oder andere Gruppenschlüssel, die in den Spalten der resultierenden Pivot-Tabelle gruppiert werden sollen
`aggfunc`	Aggregationsfunktion oder Liste von Funktionen (standardmäßig `mean`); kann eine beliebige Funktion sein, die in einem `groupby`-Kontext gültig ist
`fill_value`	ersetzt fehlende Werte in der Ergebnistabelle
`dropna`	wenn `True`, werden Spalten, deren Einträge alle `NA` sind, nicht berücksichtigt
`margins`	fügt Zeilen-/Spalten-Zwischensummen und Gesamtsummen ein (Standardeinstellung: `False`)
`margins_na`	Name, der für die Zeilen-/Spaltenbeschriftung verwendet wird, wenn `margins=True` übergeben wird, Standardwert ist `All`.
`observed`	Bei kategorialen Gruppenschlüsseln werden bei `True` nur die beobachteten Kategoriewerte in den Schlüsseln angezeigt und nicht alle Kategorien

Kreuztabellen

Eine Kreuztabelle ist ein Spezialfall einer Pivot-Tabelle, die die Häufigkeit von Gruppen berechnet. Wollen wir z.B. im Rahmen einer Analyse diese Daten vielleicht ermitteln, welcher Titel in welcher Sprache erschienen ist, so könnten wir dafür `pivot_table` verwenden, aber die Funktion pandas.crosstab ist bequemer.

Hierfür setzen wir zunächst die bestehenden Index zurück:

[8]:
```
df.reset_index(inplace=True)
```

[9]:
```
pd.crosstab(df.Title, df.Language)
```

[9]:
```
Language          de  en
Title
Jupyter Tutorial   1   1
PyViz Tutorial     1   0
Python Basics      1   1
```

Die ersten beiden Argumente für `crosstab` können jeweils entweder ein Array oder eine Series oder eine Liste von Arrays sein.

Mit `margins=True` können wir uns auch die Summen der Spalten und Zeilen sowie die Gesamtsumme berechnen lassen:

[10]:
```
pd.crosstab(df.Title, df.Language, margins=True)
```

[10]:
```
Language          de  en  All
Title
Jupyter Tutorial   1   1    2
PyViz Tutorial     1   0    1
Python Basics      1   1    2
All                3   2    5
```

2.4.17 dtype konvertieren

Manchmal passen die pandas-Datentypen nicht wirklich gut. Dies kann z.B. auf Serialisierungsformate zurückzuführen sein, die keine Typinformationen enthalten. Manchmal solltet ihr jedoch den Typ auch ändern, um eine bessere Performance zu erzielen – entweder mehr Manipulationsmöglichkeiten oder weniger Speicherbedarf. In den folgenden Beispielen werden wir verschiedene Konvertierungen einer `Series` vornehmen:

[1]:
```
import numpy as np
import pandas as pd
```

[2]:
```
s = pd.Series(np.random.randn(7))
```

[3]:
```
s
```

[3]:
```
0   -0.540728
1   -1.803588
2    2.709375
3    0.020922
4   -1.233492
5    1.809115
6   -1.310579
dtype: float64
```

Automatische Konvertierung

pandas.Series.convert_dtypes versucht, eine `Series` in einen Typ zu konvertieren, der `NA` unterstützt. Im Fall unserer `Series` wird der Typ von `float64` in `Float64` geändert:

[4]:
```
s.convert_dtypes()
```

[4]:
```
0   -0.540728
1   -1.803588
2    2.709375
3    0.020922
4   -1.233492
5    1.809115
6   -1.310579
dtype: Float64
```

Bedauerlicherweise habe ich jedoch mit `convert_dtypes` kaum Kontrolle darüber, in welchen Datentyp konvertiert wird. Daher bevorzuge ich pandas.Series.astype:

[5]:
```
s.astype("Float32")
```

[5]:
```
0   -0.540728
1   -1.803589
2    2.709375
3    0.020922
4   -1.233492
5    1.809115
6   -1.310579
dtype: Float32
```

Die Verwendung des richtigen Typs kann Speicherplatz einsparen. Üblich ist ein 8 Byte breiter Datentyp, also `int64` oder `float64`. Wenn ihr einen schmaleren Typ verwenden könnt, reduziert dies den Speicherverbrauch deutlich, sodass

ihr mehr Daten verarbeiten könnt. Ihr könnt NumPy verwenden, um die Grenzen von Integer- und Float-Typen zu überprüfen:

```
[6]: np.iinfo("int64")
[6]: iinfo(min=-9223372036854775808, max=9223372036854775807, dtype=int64)

[7]: np.finfo("float32")
[7]: finfo(resolution=1e-06, min=-3.4028235e+38, max=3.4028235e+38, dtype=float32)

[8]: np.finfo('float64')
[8]: finfo(resolution=1e-15, min=-1.7976931348623157e+308, max=1.7976931348623157e+308,␣
     ↪dtype=float64)
```

Speicherverbrauch

Um den Speicherverbrauch der `Series` zu berechnen, könnt ihr pandas.Series.nbytes verwenden um den Speicher, der von den Daten verwendet wird, zu ermitteln. pandas.Series.memory_usage erfasst darüberhinaus auch den Indexspeicher und den Datentyp. Mit `deep=True` lässt sich auch der Speicherverbrauch auf Systemebene ermitteln.

```
[9]: s.nbytes
[9]: 56

[10]: s.astype("Float32").nbytes
[10]: 35

[11]: s.memory_usage()
[11]: 188

[12]: s.astype("Float32").memory_usage()
[12]: 167

[13]: s.memory_usage(deep=True)
[13]: 188
```

String- und Kategorietypen

Die Methode pandas.Series.astype kann auch numerische Reihen in Zeichenketten umwandeln, wenn ihr `str` übergebt. Beachtet den `dtype` im folgenden Beispiel:

```
[14]: s.astype(str)
[14]: 0     -0.5407280521417943
      1     -1.8035884918817433
      2       2.709375360462432
      3    0.020922480171874997
      4     -1.2334923058043816
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
5       1.809115347105466
6     -1.3105793583289318
dtype: object
```

[15]:
```
s.astype(str).memory_usage()
```

[15]:
```
188
```

[16]:
```
s.astype(str).memory_usage(deep=True)
```

[16]:
```
661
```

Zur Konvertierung in einen kategorialen Typ könnt ihr `'category'` als Typ übergeben:

[17]:
```
s.astype(str).astype("category")
```

[17]:
```
0     -0.5407280521417943
1     -1.8035884918817433
2       2.709375360462432
3    0.020922480171874997
4     -1.2334923058043816
5       1.809115347105466
6     -1.3105793583289318
dtype: category
Categories (7, object): ['-0.5407280521417943', '-1.2334923058043816', '-1.
↪3105793583289318', '-1.8035884918817433', '0.020922480171874997', '1.809115347105466',
↪'2.709375360462432']
```

Eine kategoriale `Series` ist nützlich für String-Daten und kann zu großen Speichereinsparungen führen. Das liegt daran, dass pandas bei der Konvertierung in kategoriale Daten nicht länger Python-Strings für jeden Wert verwendet, sondern sich wiederholende Werte nicht dupliziert werden. Ihr habt immer noch alle Funktionen des `str`-Attributs, aber ihr spart viel Speicherplatz wenn ihr viele doppelte Werte habt und steigert die Leistung, da ihr nicht so viele String-Operationen durchführen müsst.

[18]:
```
s.astype("category").memory_usage(deep=True)
```

[18]:
```
495
```

Geordnete Kategorien

Um geordnete Kategorien zu erstellen, müsst ihr einen eigenen pandas.CategoricalDtype definieren:

[19]:
```
from pandas.api.types import CategoricalDtype


sorted = pd.Series(sorted(set(s)))
cat_dtype = CategoricalDtype(categories=sorted, ordered=True)

s.astype(cat_dtype)
```

[19]:
```
0   -0.540728
1   -1.803588
2    2.709375
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3    0.020922
4   -1.233492
5    1.809115
6   -1.310579
dtype: category
Categories (7, float64): [-1.803588 < -1.310579 < -1.233492 < -0.540728 < 0.020922 < 1.
↪809115 < 2.709375]
```

[20]:
```
s.astype(cat_dtype).memory_usage(deep=True)
```

[20]:
```
495
```

In der folgenden Tabelle sind die Typen aufgeführt, die ihr an `astype` übergeben könnt.

Datentyp	Beschreibung
`str`, `'str'`	in Python-String konvertieren
`'string'`	in Pandas-String konvertieren mit `pandas.NA`
`int`, `'int'`, `'int64'`	in NumPy `int64` konvertieren
`'int32'`, `'uint32'`	in NumPy `int32` konvertieren
`'Int64'`	in pandas `Int64` konvertieren mit `pandas.NA`
`float`, `'float'`, `'float64'`	in Floats konvertieren
`'category'`	in `CategoricalDtype` konvertieren mit `pandas.NA`

Umwandlung in andere Datentypen

Die Methode pandas.Series.to_numpy oder die Eigenschaft pandas.Series.values liefert uns ein NumPy-Array mit Werten, und pandas.Series.to_list gibt eine Python-Liste mit Werten zurück. Warum solltet ihr das wollen? pandas-Objekte sind meist viel benutzerfreundlicher und der Code lässt sich leichter lesen. Zudem werden Python-Listen sehr viel langsamer verarbeitet werden können. Mit pandas.Series.to_frame könnt ihr ggf. einen DataFrame mit einer einzigen Spalte erzeugen:

[21]:
```
s.to_frame()
```

[21]:
```
          0
0 -0.540728
1 -1.803588
2  2.709375
3  0.020922
4 -1.233492
5  1.809115
6 -1.310579
```

Auch die Funktion pandas.to_datetime kann hilfreich sein um in pandas um Werte in Datum und Uhrzeit zu konvertieren.

KAPITEL 3

Daten lesen, speichern und bereitstellen

Einen Überblick zu öffentlichen Repositories mit Forschungsdaten erhaltet ihr z.B. in *Open-Data*.

Neben spezifischen Python-Bibliotheken zum Zugriff auf *Entfernte Dateisysteme* und *Geodaten* stellen wir euch *Serialisierungsformate* und drei Werkzeuge genauer vor:

- *pandas IO tools*
- *httpx*
- *Intake*

Siehe auch:

Scrapy
: Framework zum Extrahieren von Daten aus Websites als JSON-, CSV- oder XML-Dateien.

Pattern
: Python-Modul zum Data Mining, Verarbeitung natürlicher Sprache, ML und Netzwerkanalyse

Web Scraping Reference
: Übersicht zu Web Scraping mit Python

Zum Speichern von relationalen Daten, Python-Objekten und Geodaten stellen wir euch *PostgreSQL*, *SQLAlchemy* und *PostGIS* vor.

Als nächstes zeigen wir euch, wie ihr die Daten über ein *Application Programming Interface (API)* bereitstellt.

Mit *DVC* stellen wir euch ein Werkzeug vor, das euch Datenprovenienz erlaubt. Damit vollzieht ihr die Herkunft und den Entstehungsweg von Daten nach.

Im Anschluß lernt ihr im nächsten Kapitel noch einige Good Practices und hilfreiche Python-Pakete zum *Bereinigen und Validieren von Daten* kennen.

3.1 Open-Data

Einen Überblick über öffentliche Repositories mit Forschungsdaten erhaltet ihr z.B. in:

- Registry of research data repositories (re3data)
- Awesome Public Datasets
- Public APIs
- Machine learning datasets
- Roboflow Computer Vision Datasets
- DBpedia
- World Health Data Platform/Global Health Observatory
- UNICEF Data
- IATI Registry
- World Bank Open Data
- Open Data Inception
- EU Open Data Portal Data
- GovData.de
- US Census Bureau
- data.gov
- Google Dataset Search
- Google Public Data Search
- Registry of Open Data on AWS
- Yelp Open Dataset
- Kaggle Datasets
- OpenDataMonitor
- Open Data Impact Map
- CKAN
- UC Irvine Machine Learning Repository
- Hugging Face Datasets
- Dataverse Project
- Open Data Kit
- LODUM University of Münster‘s Open Data initiative
- freeCodeCamp open-data
- Reddit Datasets Community

3.2 pandas IO tools

pandas verfügt über eine Reihe von Funktionen zum Lesen von Tabellendaten als DataFrame-Objekt, darunter

Funktion	Beschreibung
pandas.read_csv	lädt *CSV*-Daten aus einer Datei, einer URL oder einem dateiähnlichen Objekt; üblicherweise wird ein Komma als Trennzeichen verwendet
pandas.read_fwf	liest FWF (fixed-width files), also Daten im Spaltenformat mit fester Breite
pandas.read_clipbo	liest Daten aus der Zwischenablage und übergibt sie an `read_csv`; U.A. (unter anderem) nützlich für die Konvertierung von Tabellen aus Webseiten
pandas.read_excel	liest Tabellendaten aus einer *Excel* XLS- oder XLSX-Datei
pandas.read_hdf	liest HDF5 (Hierarchical Data Format)-Dateien
pandas.read_html	liest alle Tabellen aus dem angegebenen *HTML*-Dokument
pandas.read_json	liest Daten aus einer *JSON*-Datei
pandas.read_feather	liest das Feather-Binärdateiformat
pandas.read_orc	liest Apache ORC (Optimized Row Columnar)-Binärdaten
pandas.read_parque	liest das Apache Parquet-Binärdateiformat
pandas.read_pickle	liest ein beliebiges Objekt, das im Python-*Pickle*-Format gespeichert ist
pandas.read_sas	liest einen SAS (Statistical Analysis System)-Datensatz
pandas.read_spss	liest eine von SPSS erstellte Datendatei
pandas.read_sql	liest die Ergebnisse einer SQL-Abfrage (mit *SQLAlchemy*) als pandas DataFrame
pandas.read_sql_tab	liest eine ganze SQL-Tabelle (mit *SQLAlchemy*) als pandas DataFrame (entspricht einer Abfrage, die alles in dieser Tabelle mit `read_sql` auswählt)
pandas.read_stata	liest einen Datensatz aus dem Stata-Dateiformat

Siehe auch:

pandas I/O API
: Die pandas I/O API ist eine Sammlung von `reader`-Funktionen, die ein pandas-Objekt zurückgeben. Meist stehen auch entsprechende `write`-Methoden zur Verfügung.

Zunächst werde ich einen Überblick über einige dieser Funktionen geben, die dazu gedacht sind, Text- und Exceldaten in einen pandas-DataFrame zu konvertieren: *CSV*, *JSON* und *Excel*. Dabei lassen sich die optionalen Argumente für diese Funktionen in folgende Kategorien einteilen:

Indizierung
: Können eine oder mehrere Spalten den zurückgegebenen DataFrame erschließen, und ob die Spaltennamen aus der Datei, den von euch angegebenen Argumenten oder gar nicht abgerufen werden sollen.

Typinferenz und Datenkonvertierung
: Dazu gehören die benutzerdefinierten Wertkonvertierungen und die benutzerdefinierte Liste der Markierungen

für fehlende Werte.

Parsen von Datum und Uhrzeit
Dies umfasst die Kombinationsfähigkeit, einschließlich der Kombination von Datums- und Zeitinformationen, die über mehrere Spalten verteilt sind, in einer einzigen Spalte im Ergebnis.

Iteration
Unterstützung für die Iteration über Teile von sehr großen Dateien.

Probleme mit unsauberen Daten
Überspringen von Zeilen oder Fußzeilen, Kommentaren oder anderen Kleinigkeiten wie numerischen Daten mit durch Kommas getrennte Tausender.

Da Daten in der realen Welt sehr unübersichtlich sein können, haben einige der Datenladefunktionen (insbesondere `read_csv`) im Laufe der Zeit eine lange Liste optionaler Argumente angehäuft. Die Online-Dokumentation von pandas enthält viele Beispiele für die einzelnen Funktionen.

Einige dieser Funktionen, wie `pandas.read_csv`, führen eine Typinferenz durch, da die Datentypen der Spalten nicht Teil des Datenformats sind. Das bedeutet, dass ihr nicht unbedingt angeben müsst, welche Spalten numerisch, integer, boolesch oder string sind. Bei anderen Datenformaten wie HDF5, ORC und Parquet sind die Datentypinformationen hingegen bereits in das Format eingebettet.

3.3 Serialisierungsformate

Datenserialisierung konvertiert strukturierte Daten in ein Format, das geteilt und gespeichert werden kann. Bevor ihr jedoch die Daten serialisiert, müsst ihr Euch entscheiden, wie die Daten strukturiert werden sollen – flach oder verschachtelt. Der Unterschied zwischen den beiden Stilen seht ihr an folgendem Beispiel:

Flacher JSON-Stil:

```
{
  "id"          :  "veit",
  "first_name"  :  "Veit",
  "last_name"   :  "Schiele",
}
```

Verschachtelter JSON-Stil:

```
{
  "veit" : {
    "first_name"  :  "Veit",
    "last_name"   :  "Schiele",
  },
}
```

3.3.1 Datenserialisierung

Wenn die Daten flach serialisiert werden sollen, bietet Python zwei Funktionen an:

repr

repr() gibt eine druckbare Repräsentation der Eingabe aus, z.B.:

[1]:
```
data = {"id": "veit", "first_name": "Veit", "last_name": "Schiele"}

print(repr(data))
```

```
{'id': 'veit', 'first_name': 'Veit', 'last_name': 'Schiele'}
```

[2]:
```
with open("data.txt", "w") as f:
    f.write(repr(data))
```

ast.literal_eval

Die ast.literal_eval()-Funktion parst und analysiert den Python-Datentyp eines Ausdrucks. Unterstützte Datentypen sind Zeichenketten, Zahlen, Tupel, Listen, Dictionaries und None.

[3]:
```
import ast


with open("data.txt", "r") as f:
    d = ast.literal_eval(f.read())

print(d)
```

```
{'id': 'veit', 'first_name': 'Veit', 'last_name': 'Schiele'}
```

3.3.2 CSV

Überblick

Unterstützung von Datenstrukturen	--	CSV wird zum Speichern von Tabellendaten verwendet, ist aber im Gegensatz zu anderen hier besprochenen Serialisierungsformaten nicht für verschachtelte Daten geeignet.
Standardisierung	--	CSV ist nicht gut standardisiert: weder das Encoding noch die Trennung der Zelleninhalte (Komma, Semikolon etc.).
Schema IDL	--	Nein
Sprachunterstützung	++	Das CSV-Format wird in fast jeder Programmiersprache gut unterstützt. Ein csv-Modul ist in der Python-Standardbibliothek enthalten und pandas läd eine CSV-Datei direkt als `Dataframe`. Auch wenn CSV das einzige hier besprochene Format ist, das gut von Tabellenkalkulationen wie Excel unterstützt wird, solltet ihr strukturiertere Excel-Dateien direkt einlesen, z.B. mit pandas read_excel.
Menschliche Lesbarkeit	+-	CSV ist speziell bei Ganz- oder Dezimalzahlen mit gleicher Zeichenlänge gut lesbar. In allen anderen Fällen kann es schwierig werden, die entsprechenden Spalten zu identifizieren.
Geschwindigkeit	+	CSV kann sehr schnell serialisiert und deserialisiert werden.
Dateigröße	++	Nur *Protocol Buffers (Protobuf)* sollte kompakter sein.

Beispiel

iris.csv

```
5.1,0.222222222,3.5,0.625,1.4,0.06779661,0.2,0.041666667,setosa
4.9,0.166666667,3,0.416666667,1.4,0.06779661,0.2,0.041666667,setosa
4.7,0.111111111,3.2,0.5,1.3,0.050847458,0.2,0.041666667,setosa
4.6,0.083333333,3.1,0.458333333,1.5,0.084745763,0.2,0.041666667,setosa
5,0.194444444,3.6,0.666666667,1.4,0.06779661,0.2,0.041666667,setosa
...
```

Siehe auch:

- **RFC 4180**

CSV-Beispiel

[1]:
```
import pandas as pd
```

Nach dem Import von pandas lesen wir zunächst mit `read_csv` eine CSV-Datei ein:

[2]:
```
pd.read_csv("https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/
↪save-data/books.csv")
```

[2]:
```
      Python basics  en  Veit Schiele  BSD-3-Clause  2021-10-28
0  Jupyter Tutorial  en  Veit Schiele  BSD-3-Clause  2019-06-27
1  Jupyter Tutorial  de  Veit Schiele  BSD-3-Clause  2020-10-26
2    PyViz Tutorial  en  Veit Schiele  BSD-3-Clause  2020-04-13
```

Wie ihr seht, hat diese Datei keine Kopfzeile. Um dem DataFrame eine Kopfzeile zu geben, habt ihr mehrere Möglichkeiten. Ihr könnt pandas erlauben, Standard-Spaltennamen zuzuweisen, oder ihr könnt die Namen auch selbst festlegen:

[3]:
```
pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
    header=None,
)
```

[3]:
```
                  0   1             2             3           4
0     Python basics  en  Veit Schiele  BSD-3-Clause  2021-10-28
1  Jupyter Tutorial  en  Veit Schiele  BSD-3-Clause  2019-06-27
2  Jupyter Tutorial  de  Veit Schiele  BSD-3-Clause  2020-10-26
3    PyViz Tutorial  en  Veit Schiele  BSD-3-Clause  2020-04-13
```

[4]:
```
pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
    names=[
        "Titel",
        "Sprache",
        "Autor*innen",
        "Lizenz",
        "Veröffentlichungsdatum",
    ],
)
```

[4]:
```
              Titel Sprache   Autor*innen        Lizenz Veröffentlichungsdatum
0     Python basics       en  Veit Schiele  BSD-3-Clause              2021-10-28
1  Jupyter Tutorial       en  Veit Schiele  BSD-3-Clause              2019-06-27
2  Jupyter Tutorial       de  Veit Schiele  BSD-3-Clause              2020-10-26
3    PyViz Tutorial       en  Veit Schiele  BSD-3-Clause              2020-04-13
```

Angenommen, ihr möchtet, dass die Spalte `Autor*innen` der Index des zurückgegebenen DataFrame ist. Ihr könnt entweder angeben, dass ihr die Spalte bei Index 3 oder mit dem Namen `Autor*innen` haben möchtet, indem ihr das Argument `index_col` verwendet:

[5]:
```
pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    index_col=["Autor*innen"],
    names=[
        "Titel",
        "Sprache",
        "Autor*innen",
        "Lizenz",
        "Veröffentlichungsdatum",
    ],
)
```

[5]:
```
                        Titel Sprache        Lizenz Veröffentlichungsdatum
Autor*innen
Veit Schiele     Python basics      en  BSD-3-Clause             2021-10-28
Veit Schiele  Jupyter Tutorial      en  BSD-3-Clause             2019-06-27
Veit Schiele  Jupyter Tutorial      de  BSD-3-Clause             2020-10-26
Veit Schiele    PyViz Tutorial      en  BSD-3-Clause             2020-04-13
```

Für den Fall, dass ihr einen hierarchischen Index aus mehreren Spalten bilden wollt, übergebt eine Liste von Spaltennummern oder -namen:

[6]:
```
pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
    index_col=[2, 0],
    names=[
        "Titel",
        "Sprache",
        "Autor*innen",
        "Lizenz",
        "Veröffentlichungsdatum",
    ],
)
```

[6]:
```
                             Sprache        Lizenz Veröffentlichungsdatum
Autor*innen  Titel
Veit Schiele Python basics         en  BSD-3-Clause             2021-10-28
             Jupyter Tutorial      en  BSD-3-Clause             2019-06-27
             Jupyter Tutorial      de  BSD-3-Clause             2020-10-26
             PyViz Tutorial        en  BSD-3-Clause             2020-04-13
```

In manchen Fällen hat eine Tabelle kein festes Trennzeichen, sondern verwendet mehrere Leerzeichen oder ein anderes Muster zur Trennung von Feldern. Angenommen, eine Datei sieht folgendermaßen aus:

[7]:
```
list(open("books.txt"))
```

[7]:
```
['   Titel             Sprache  Autor*innen   Lizenz        Veröffentlichungsdatum\n',
 '1  Python basics     en       Veit Schiele  BSD-3-Clause  2021-10-28\n',
 '2  Jupyter Tutorial  en       Veit Schiele  BSD-3-Clause  2019-06-27\n',
 '3  Jupyter Tutorial  de       Veit Schiele  BSD-3-Clause  2020-10-26\n',
 '4  PyViz Tutorial    en       Veit Schiele  BSD-3-Clause  2020-04-13\n']
```

In solchen Fällen könnt ihr einen regulären Ausdruck als Trennzeichen für `read_csv` übergeben. Dies kann durch den regulären Ausdruck `\s\s+` ausgedrückt werden, also haben wir dann:

[8]:
```
pd.read_csv("books.txt", sep="\s\s+", engine="python")
```

[8]:
```
              Titel Sprache   Autor*innen        Lizenz Veröffentlichungsdatum
1     Python basics       en  Veit Schiele  BSD-3-Clause              2021-10-28
2  Jupyter Tutorial       en  Veit Schiele  BSD-3-Clause              2019-06-27
3  Jupyter Tutorial       de  Veit Schiele  BSD-3-Clause              2020-10-26
4    PyViz Tutorial       en  Veit Schiele  BSD-3-Clause              2020-04-13
```

Da es einen Spaltennamen weniger als die Anzahl der Datenzeilen gab, folgert `read_csv`, dass in diesem Fall die erste Spalte der Index des DataFrame sein sollte.

Die Parser-Funktionen haben viele zusätzliche Argumente, die euch helfen, die große Vielfalt der auftretenden Ausnahmedateiformate zu handhaben. So könnt ihr beispielsweise mit `skiprows` einzelne Zeilen einer Datei überspringen:

[9]:
```
pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
    skiprows=[2],
    names=[
        "Titel",
        "Sprache",
        "Autor*innen",
        "Lizenz",
        "Veröffentlichungsdatum",
    ],
)
```

[9]:
```
              Titel Sprache   Autor*innen        Lizenz Veröffentlichungsdatum
0     Python basics       en  Veit Schiele  BSD-3-Clause              2021-10-28
1  Jupyter Tutorial       en  Veit Schiele  BSD-3-Clause              2019-06-27
2    PyViz Tutorial       en  Veit Schiele  BSD-3-Clause              2020-04-13
```

Der Umgang mit fehlenden Werten ist ein wichtiger und häufig komplizierter Teil beim Parsen von Daten. Fehlende Daten sind normalerweise entweder nicht vorhanden (leerer String) oder durch einen Platzhalter gekennzeichnet. Standardmäßig verwendet Pandas eine Reihe von häufig vorkommenden Platzhalter, wie `NA` und `NULL`:

[10]:
```
df = pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
    names=[
        "Titel",
        "Sprache",
        "Autor*innen",
        "Lizenz",
        "Veröffentlichungsdatum",
        "doi",
    ],
)

df
```

[10]:
```
              Titel Sprache   Autor*innen        Lizenz  \
0     Python basics       en  Veit Schiele  BSD-3-Clause
1  Jupyter Tutorial       en  Veit Schiele  BSD-3-Clause
2  Jupyter Tutorial       de  Veit Schiele  BSD-3-Clause
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3    PyViz Tutorial      en  Veit Schiele  BSD-3-Clause

  Veröffentlichungsdatum  doi
0             2021-10-28  NaN
1             2019-06-27  NaN
2             2020-10-26  NaN
3             2020-04-13  NaN
```

[11]:
```
df.isna()
```

[11]:
```
   Titel  Sprache  Autor*innen  Lizenz  Veröffentlichungsdatum   doi
0  False    False        False   False                   False  True
1  False    False        False   False                   False  True
2  False    False        False   False                   False  True
3  False    False        False   False                   False  True
```

Die Option `na_values` kann entweder eine Liste oder eine Reihe von Strings annehmen, um fehlende Werte zu berücksichtigen:

[12]:
```
df = pd.read_csv(
    "https://raw.githubusercontent.com/veit/python-basics-tutorial-de/main/docs/save-
↪data/books.csv",
    na_values=["BSD-3-Clause"],
    names=[
        "Titel",
        "Sprache",
        "Autor*innen",
        "Lizenz",
        "Veröffentlichungsdatum",
        "doi",
    ],
)

df
```

[12]:
```
             Titel Sprache   Autor*innen  Lizenz Veröffentlichungsdatum  doi
0     Python basics       en  Veit Schiele     NaN             2021-10-28  NaN
1  Jupyter Tutorial       en  Veit Schiele     NaN             2019-06-27  NaN
2  Jupyter Tutorial       de  Veit Schiele     NaN             2020-10-26  NaN
3    PyViz Tutorial       en  Veit Schiele     NaN             2020-04-13  NaN
```

Die häufigsten Argumente der Funktion `read_csv`:

Argument	Beschreibung
`path`	Zeichenfolge, die den Speicherort im Dateisystem, eine URL oder ein dateiähnliches Objekt angibt
`sep` oder `delim`	Zeichenfolge oder regulärer Ausdruck zum Trennen der Felder in jeder Zeile
`heade`	Zeilennummer, die als Spaltennamen zu verwenden ist; Standardwert ist `0`, also die erste Zeile, sollte aber `None` sein, wenn es keine Kopfzeile gibt
`index`	index_col Zeilennummern oder -namen, die als Zeilenindex im Ergebnis verwendet werden sollen; kann ein einzelner Name/eine einzelne Zahl oder eine Liste von ihnen für einen hierarchischen Index sein
`names`	Liste der Spaltennamen
`skipr`	Anzahl der zu ignorierenden Zeilen am Anfang der Datei oder Liste der Zeilennummern beginnend bei `0`, die übersprungen werden sollen
`na_va`	Folge von Werten, die durch NA ersetzt werden sollen
`comme`	Zeichen, um Kommentare vom Zeilenende abzutrennen
`parse`	Versuch, Daten mit datetime zu parsen; standardmäßig `False`. Wenn `True`, wird versucht, alle Spalten zu parsen. Andernfalls kann eine Liste von Spaltennummern oder -namen angegeben werden, die analysiert werden sollen. Wenn das Element der Liste ein Tupel oder eine Liste ist, werden mehrere Spalten miteinander kombiniert und in ein Datum umgewandelt, z.B. wenn Datum und Uhrzeit auf zwei Spalten aufgeteilt sind
`keep_`	wenn Spalten zum Parsen des Datums kombiniert werden, werden die kombinierten Spalten beibehalten; Standardeinstellung: `False`.
`conve`	Dict, das die Spaltennummer der Namen enthält, die auf Funktionen abgebildet werden, z.B. würde `{'Titel': f}` die Funktion `f` auf alle Werte in der Spalte `Titel` anwenden
`dayfi`	beim Parsen potenziell mehrdeutiger Datumsangaben als internationales Format behandeln, z.B. `28/6/2021` → `28. Juni 2021`; standardmäßig `False`
`date_`	zu verwendende Funktion zum Parsen von Datumsangaben
`nrows`	Anzahl der zu lesenden Zeilen vom Anfang der Datei
`itera`	Rückgabe eines `TextFileReader`-Objekts zum stückweisen Einlesen der Datei; dieses Objekt kann auch mit der `with`-Anweisung verwendet werden
`chunk`	Für die Iteration die Größe der Datenblöcke.
`skip_`	Anzahl der Zeilen, die am Ende der Datei ignoriert werden sollen
`verbo`	gibt verschiedene Informationen zur Parser-Ausgabe aus, z.B. die Anzahl der fehlenden Werte in nichtnumerischen Spalten.
`encod`	Textkodierung für Unicode, z.B. `utf-8` für UTF-8-kodierten Text
`squee`	wenn die geparsten Daten nur eine Spalte enthalten, wird eine Serie zurückgegeben
`thous`	Trennzeichen für Tausender, z.B. `,` oder `.`

Stückweises Einlesen von Textdateien

Wenn ihr sehr große Dateien verarbeiten wollt, könnt ihr auch nur einen kleinen Teil einer Datei einlesen oder durch kleinere Teile einer Datei iterieren.

Bevor wir uns eine große Datei ansehen, verringern wir mit `options.display.max_rows` die Anzahl der angezeigten Zeilen:

[13]:
```
pd.options.display.max_rows = 10
```

[14]:
```
pd.read_csv("example.csv")
```

[14]:
```
            Date      Mon.     Tues.      Wed.     Thurs.      Fri.      Sat.  \
0     1996-01-01  0.129453 -0.023836  1.121460  1.698286 -0.598506  1.042221
1     1996-01-02 -0.094021 -0.727942  0.698641 -1.198040  1.927505  1.147445
2     1996-01-03 -0.560857  0.145222 -0.990202  1.200214  0.717339  1.117095
3     1996-01-04 -0.169755 -0.677391 -1.533519 -0.343477 -0.109705  1.038236
4     1996-01-05  1.344705 -1.817261  0.460991 -0.839633  0.265814  0.477659
...          ...       ...       ...       ...       ...       ...       ...
9127  2020-12-27 -0.881800 -0.074270 -0.351769  1.381641 -0.049548  1.664180
9128  2020-12-28 -0.143386  0.198217 -1.243861  1.196576  1.338166 -0.212333
9129  2020-12-29  0.398787 -0.848786  1.791707 -1.167592 -0.033881 -0.285559
9130  2020-12-30  0.587846  0.411580  1.150380  0.444638 -1.093577  0.605456
9131  2020-12-31  0.736350  0.436292 -0.260171 -0.066066 -0.328324 -0.586792

          Sun.
0    -0.726412
1    -1.134103
2    -1.793565
3    -0.799088
4     0.636383
...        ...
9127 -1.032204
9128 -0.023131
9129 -0.323477
9130  1.463345
9131 -1.204582

[9132 rows x 8 columns]
```

Wenn ihr nur eine kleine Anzahl von Zeilen lesen wollt (ohne die gesamte Datei zu lesen), könnt ihr dies mit `nrows` angeben:

[15]:
```
pd.read_csv("example.csv", nrows=7)
```

[15]:
```
         Date      Mon.     Tues.      Wed.     Thurs.      Fri.      Sat.  \
0  1996-01-01  0.129453 -0.023836  1.121460  1.698286 -0.598506  1.042221
1  1996-01-02 -0.094021 -0.727942  0.698641 -1.198040  1.927505  1.147445
2  1996-01-03 -0.560857  0.145222 -0.990202  1.200214  0.717339  1.117095
3  1996-01-04 -0.169755 -0.677391 -1.533519 -0.343477 -0.109705  1.038236
4  1996-01-05  1.344705 -1.817261  0.460991 -0.839633  0.265814  0.477659
5  1996-01-06 -0.354445 -0.065182 -1.244963 -0.559732  0.042362 -0.303712
6  1996-01-07  1.460922  0.164412  0.883960 -0.833642  0.001582  1.138469

       Sun.
0 -0.726412
1 -1.134103
2 -1.793565
3 -0.799088
4  0.636383
5  0.067632
6  0.561618
```

Um eine Datei stückweise zu lesen, könnt ihr mit `chunksize` die Anzahl der Zeilen angeben:

[16]:
```
pd.read_csv("example.csv", chunksize=1000)
```

[16]:
```
<pandas.io.parsers.readers.TextFileReader at 0x13295f2d0>
```

Das von `read_csv` zurückgegebene `TextFileReader`-Objekt ermöglicht die Iteration über Teile der Datei entsprechend der `chunksize`. Zum Beispiel können wir über die `example.csv`-Datei iterieren und die Anzahl der Werte in der Spalte `Date` wie folgt aggregieren:

[17]:
```
chunks = pd.read_csv("example.csv", chunksize=1000)

serie = pd.Series([], dtype="float64")
for chunk in chunks:
    values = serie.add(chunk["Date"].value_counts(), fill_value=0)

sorted_values = values.sort_values(ascending=False)
```

[18]:
```
sorted_values[:10]
```

[18]:
```
Date
2020-08-22    1.0
2020-09-07    1.0
2020-08-24    1.0
2020-08-25    1.0
2020-08-26    1.0
2020-08-27    1.0
2020-08-28    1.0
2020-08-29    1.0
2020-08-30    1.0
2020-08-31    1.0
dtype: float64
```

`TextFileReader` verfügt außerdem über eine `get_chunk`-Methode, mit der ihr Stücke beliebiger Größe lesen könnt.

DataFrame und Series als csv-Datei schreiben

Daten können auch in ein mit Trennzeichen versehenes Format exportiert werden. Mit der Methode `pandas.DataFrame.to_csv` können wir die Daten in eine kommagetrennte Datei schreiben:

[19]:
```
df.to_csv("out.csv")
```

Natürlich können auch andere Begrenzungszeichen verwendet werden, z.B. zum Schreiben nach `sys.stdout`, so dass das Textergebnis auf der Konsole und nicht in einer Datei ausgegeben wird:

[20]:
```
import sys
```

[21]:
```
df.to_csv(sys.stdout, sep="|")
```

```
|Titel|Sprache|Autor*innen|Lizenz|Veröffentlichungsdatum|doi
0|Python basics|en|Veit Schiele||2021-10-28|
1|Jupyter Tutorial|en|Veit Schiele||2019-06-27|
2|Jupyter Tutorial|de|Veit Schiele||2020-10-26|
3|PyViz Tutorial|en|Veit Schiele||2020-04-13|
```

Fehlende Werte erscheinen in der Ausgabe als leere Zeichenketten. Möglicherweise möchtet ihr sie durch einen anderen Platzhalter kennzeichnen:

[22]:
```
df.to_csv(sys.stdout, na_rep="NaN")
```

```
,Titel,Sprache,Autor*innen,Lizenz,Veröffentlichungsdatum,doi
0,Python basics,en,Veit Schiele,NaN,2021-10-28,NaN
1,Jupyter Tutorial,en,Veit Schiele,NaN,2019-06-27,NaN
2,Jupyter Tutorial,de,Veit Schiele,NaN,2020-10-26,NaN
3,PyViz Tutorial,en,Veit Schiele,NaN,2020-04-13,NaN
```

Wenn keine anderen Optionen angegeben sind, werden sowohl die Zeilen- als auch die Spaltenbeschriftungen geschrieben. Beides kann deaktiviert werden:

[23]:
```
df.to_csv(sys.stdout, index=False, header=False)
```

```
Python basics,en,Veit Schiele,,2021-10-28,
Jupyter Tutorial,en,Veit Schiele,,2019-06-27,
Jupyter Tutorial,de,Veit Schiele,,2020-10-26,
PyViz Tutorial,en,Veit Schiele,,2020-04-13,
```

Ihr könnt auch nur eine Teilmenge der Spalten schreiben, und zwar in einer von euch gewählten Reihenfolge:

[24]:
```
df.to_csv(
    sys.stdout,
    index=False,
    columns=["Titel", "Sprache", "Autor*innen", "Veröffentlichungsdatum"],
)
```

```
Titel,Sprache,Autor*innen,Veröffentlichungsdatum
Python basics,en,Veit Schiele,2021-10-28
Jupyter Tutorial,en,Veit Schiele,2019-06-27
Jupyter Tutorial,de,Veit Schiele,2020-10-26
PyViz Tutorial,en,Veit Schiele,2020-04-13
```

Arbeiten mit dem csv-Modul von Python

Die meisten Formen von Tabellendaten können mit Funktionen wie `pandas.read_csv` geladen werden. In einigen Fällen kann jedoch auch eine manuelle Bearbeitung erforderlich sein. Es ist nicht ungewöhnlich, eine Datei mit einer oder mehreren fehlerhaften Zeilen zu erhalten, die `read_csv` zum Scheitern bringen. Für jede Datei mit einem einstelligen Begrenzungszeichen könnt ihr das in Python eingebaute csv-Modul verwenden. Um es zu verwenden, übergebt eine offene Datei oder ein dateiähnliches Objekt an `csv.reader`:

[25]:
```
import csv


f = open("out.csv")
reader = csv.reader(f)

for line in reader:
    print(line)
```

```
['', 'Titel', 'Sprache', 'Autor*innen', 'Lizenz', 'Veröffentlichungsdatum', 'doi']
['0', 'Python basics', 'en', 'Veit Schiele', '', '2021-10-28', '']
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
['1', 'Jupyter Tutorial', 'en', 'Veit Schiele', '', '2019-06-27', '']
['2', 'Jupyter Tutorial', 'de', 'Veit Schiele', '', '2020-10-26', '']
['3', 'PyViz Tutorial', 'en', 'Veit Schiele', '', '2020-04-13', '']
```

Dialekte

csv-Dateien gibt es in vielen verschiedenen Varianten. Das Python csv-Modul kommt bereits mit drei verschiedenen Dialekten:

Parameter	excel	excel-tab	unix
delimiter	','	'\t'	','
quotechar	'"'	'"'	'"'
doublequote	True	True	True
skipinitialspace	False	False	False
lineterminator	'\r\n'	'\r\n'	'\n'
quoting	csv.QUOTE_MINIMAL	csv.QUOTE_MINIMAL	csv.QUOTE_ALL
escapechar	None	None	None

Ihr könnt damit auch euer eigenes Format definieren mit einem anderen Trennzeichen, einer anderen Zeichenkettenkonvention oder einem anderen Zeilenendezeichen. Hierfür empfiehlt sich die Registrierung eines eigenen Dialekts. Mögliche Optionen und Funktionen von `csv.register_dialect` sind:

Argument	Beschreibung
delimit	Ein-Zeichen-Zeichenfolge zur Trennung von Feldern; Standardwert ist `,`.
lineter	Zeilenabschlusszeichen zum Schreiben; Standardwert ist `\r\n`. Reader ignoriert dies und erkennt plattformübergreifende Zeilenbegrenzer.
quotech	Anführungszeichen für Felder mit Sonderzeichen (wie ein Trennzeichen); Standardwert ist ".
quoting	Zitier-Konvention. Zu den Optionen gehören `csv.QUOTE_ALL` – alle Felder zitieren, `csv.QUOTE_MINIMAL` – nur Felder mit Sonderzeichen wie dem Begrenzungszeichen, `csv.QUOTE_NONNUMERIC` und `csv.QUOTE_NONE` – keine Zitate. Der Standardwert ist `QUOTE_MINIMAL`.
skipini	Leerzeichen nach jedem Begrenzungszeichen ignorieren; Standardwert ist `False`.
doubleq	bei `True` werden Anführhuntszeichen innerhalb eines Feldes verdoppelt.
escapec	Zeichenkette, um das Trennzeichen zu umgehen, wenn `quoting` auf `csv.QUOTE_NONE` gesetzt ist; standardmäßig deaktiviert.

[26]:
```
csv.register_dialect(
    "my_csv_dialect",
    lineterminator="\n",
    delimiter=",",
    quotechar="'",
    quoting=csv.QUOTE_MINIMAL,
)
```

Nun kann die CSV-Datei geöffnet werden mit:

[27]:
```
with open("out.csv") as f:
    reader = csv.reader(f, "my_csv_dialect")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    for line in reader:
        print(line)
```

```
['', 'Titel', 'Sprache', 'Autor*innen', 'Lizenz', 'Veröffentlichungsdatum', 'doi']
['0', 'Python basics', 'en', 'Veit Schiele', '', '2021-10-28', '']
['1', 'Jupyter Tutorial', 'en', 'Veit Schiele', '', '2019-06-27', '']
['2', 'Jupyter Tutorial', 'de', 'Veit Schiele', '', '2020-10-26', '']
['3', 'PyViz Tutorial', 'en', 'Veit Schiele', '', '2020-04-13', '']
```

Dann können wir ein Dict mit Datenspalten erstellen, indem wir Dict Comprehensions verwenden und mit zip über die Werte aus `Values` iterieren. Beachtet dabei, dass dies bei großen Dateien viel Speicherplatz benötigt, da hierfürdie Zeilen in Spalten umgewandelt werden:

[28]:
```
with open("out.csv") as f:
    reader = csv.reader(f, "my_csv_dialect")
    lines = list(reader)
    header, values = lines[0], lines[1:]
    data_dict = {h: v for h, v in zip(header, zip(*values))}

data_dict
```

[28]:
```
{'': ('0', '1', '2', '3'),
 'Titel': ('Python basics',
  'Jupyter Tutorial',
  'Jupyter Tutorial',
  'PyViz Tutorial'),
 'Sprache': ('en', 'en', 'de', 'en'),
 'Autor*innen': ('Veit Schiele',
  'Veit Schiele',
  'Veit Schiele',
  'Veit Schiele'),
 'Lizenz': ('', '', '', ''),
 'Veröffentlichungsdatum': ('2021-10-28',
  '2019-06-27',
  '2020-10-26',
  '2020-04-13'),
 'doi': ('', '', '', '')}
```

Um Dateien mit Trennzeichen manuell zu schreiben, könnt ihr `csv.writer` verwenden. Er akzeptiert ein offenes, beschreibbares Dateiobjekt und die gleichen Dialekt- und Formatoptionen wie `csv.reader`:

[29]:
```
with open("new.csv", "w") as f:
    writer = csv.writer(f, "my_csv_dialect")
    writer.writerow(("", "Titel", "Sprache", "Autor*innen"))
    writer.writerow(("1", "Python basics", "en", "Veit Schiele"))
    writer.writerow(("2", "Jupyter Tutorial", "en", "Veit Schiele"))
```

[30]:
```
list(open("new.csv"))
```

[30]:
```
[',Titel,Sprache,Autor*innen\n',
 '1,Python basics,en,Veit Schiele\n',
 '2,Jupyter Tutorial,en,Veit Schiele\n']
```

3.3.3 JSON

Überblick

Unterstützung von Datenstrukturen	+-	JSON unterstützt Array- und Map- oder Objectstrukturen vieler verschiedener Datentypen einschließlich Zeichenfolgen, Zahlen, Boolesche Werte, Null usw., aber keine Datumsformate. Jedoch unterstützt JSON auch nicht alle Datentypen von JavaScript: `NaN` und `Infinity` werden zu `null`. Bitte beachtet auch, dass die JSON keine Kommentare unterstützt und ihr gegebenenfalls darum herumarbeiten müsst, z.B. mit einem `__comment__` Schlüssel/Wert-Paar.
Standardisierung	+	JSON hat einen formal streng typisierten Standard siehe auch **RFC 8259**). Jedoch enthalten JSON-Daten auch einige Fallstricke aufgrund der Mehrdeutigkeit der JSON-Specifikationen: *Ein JSON-Parser MUSS alle Texte akzeptieren, die der JSON-Grammatik entsprechen* (**RFC 7159**). und *Eine Implementierung kann Grenzen für die Größe der akzeptierten Texte setzen. Eine Implementierung kann Grenzen für die maximale Verschachtelungstiefe festlegen. Eine Implementierung kann Grenzen für den Bereich und die Genauigkeit von Zahlen festlegen. Eine Implementierung kann Grenzen für die Länge und den Zeicheninhalt von Zeichenketten festlegen* (**RFC 7158#9**). Unglücklicherweise gibt es weder eine Referenzimplementierung noch eine offizielle Testsuite, die das erwartete Verhalten zeigen würden – immerhin gibt zumindest JSON_Checker einige Hinweise.
Schema IDL	+-	Zum Teil mit JSON Schema Proposal, JSON Encoding Rules (JER), Kwalify, Rx, JSON-LD oder JMESPath. Immerhin gibt es viele verschiedene Validatoren.
Sprachunterstützung	++	Das JSON-Format wird sehr gut von den meisten Programmiersprachen unterstützt. Die Datenstrukturen von JSON sind nahe an den Objekten der meisten Sprachen, z.B. kann ein Python `dict` einfach als JSON-`object` und eine Python `list` einfach als JSON-`array` dargestellt werden.
Menschliche Lesbarkeit	+-	JSON ist ein menschlich lesbares Serialisierungsformat, aber es unterstützt keine Kommentare.
Geschwindigkeit	++	JSON ist eines der menschlich lesbaren Serialisierungsformate, die schnell zu serialisieren und zu deserialisieren sind.
Dateigröße	+-	Die Dateigröße von JSON liegt im Mittelfeld, ähnlich *YAML* und *TOML*.

Siehe auch:

- JC – JSON Convert
- fx
- gron

- python-json-patch

Beispiel

Antwort der *OSM-Nominatim-API*:

```
[
    {
        'place_id': 234847916,
        'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/
↪copyright',
        'osm_type': 'relation',
        'osm_id': 131761,
        'boundingbox': ['52.5200695', '52.5232601', '13.4103097', '13.4160798'],
        'lat': '52.521670650000004',
        'lon': '13.413278026558228',
        'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
        'class': 'highway',
        'type': 'pedestrian',
        'importance': 0.6914982526373583
    },
    {
        'place_id': 53256307,
        'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/
↪copyright',
        'osm_type': 'node',
        'osm_id': 4389211800,
        'boundingbox': ['52.5231653', '52.5232653', '13.414475', '13.414575'],
        'lat': '52.5232153',
        'lon': '13.414525',
        'display_name': 'Alexanderplatz, Alexanderstraße, Mitte, Berlin, 10178,␣
↪Deutschland',
        'class': 'highway',
        'type': 'bus_stop',
        'importance': 0.22100000000000003,
        'icon': 'https://nominatim.openstreetmap.org/images/mapicons/transport_bus_stop2.
↪p.20.png'
    },
    {
        'place_id': 90037579,
        'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://osm.org/
↪copyright',
        'osm_type': 'way',
        'osm_id': 23853138,
        'boundingbox': ['52.5214702', '52.5217276', '13.4037885', '13.4045026'],
        'lat': '52.5215991',
        'lon': '13.404112295159964',
        'display_name': 'Alexander Plaza, 1, Rosenstraße, Mitte, Berlin, 10178,␣
↪Deutschland',
        'class': 'tourism',
        'type': 'hotel',
        'importance': 0.11100000000000002,
        'icon': 'https://nominatim.openstreetmap.org/images/mapicons/accommodation_
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪hotel2.p.20.png'
    }
]
```

JSON-Beispiel

JSON (kurz für *JavaScript Object Notation*) hat sich zu einem der Standardformate für die Übermittlung von Daten per HTTP-Anfrage zwischen Webbrowsern und anderen Anwendungen entwickelt. Hier ist ein Beispiel:

JSON ähnelt Python-Code, mit Ausnahme des Nullwerts `null` und dem Verbot von Kommas am Ende von Listen. Die Grundtypen sind Objekte (Dicts), Arrays (Listen), Zeichenketten, Zahlen, Boolesche Werte und `null`. Alle Schlüssel eines Objekts müssen Zeichenketten sein. Es gibt mehrere Python-Bibliotheken zum Lesen und Schreiben von JSON-Daten. Ich werde hier json aus der Python-Standardbibliothek verwenden. Um einen JSON-String in die Python-Form zu konvertieren, verwende ich `json.loads`:

[1]:
```
>>> import json


f = open("books.json")
data = json.load(f)

for i in data:
    print(i)
```

```
{'Titel': 'Python basics', 'Sprache': 'en', 'Autor*innen': 'Veit Schiele', 'Lizenz':
↪'BSD-3-Clause', 'Veröffentlichungsdatum': '2021-10-28'}
{'Titel': 'Jupyter Tutorial', 'Sprache': 'en', 'Autor*innen': 'Veit Schiele', 'Lizenz':
↪'BSD-3-Clause', 'Veröffentlichungsdatum': '2019-06-27'}
{'Titel': 'Jupyter Tutorial', 'Sprache': 'de', 'Autor*innen': 'Veit Schiele', 'Lizenz':
↪'BSD-3-Clause', 'Veröffentlichungsdatum': '2020-10-26'}
{'Titel': 'PyViz Tutorial', 'Sprache': 'en', 'Autor*innen': 'Veit Schiele', 'Lizenz':
↪'BSD-3-Clause', 'Veröffentlichungsdatum': '2020-04-13'}
```

json.dumps hingegen konvertiert ein Python-Objekt zurück nach JSON:

[2]:
```
json.dumps(data)
```

[2]:
```
'[{"Titel": "Python basics", "Sprache": "en", "Autor*innen": "Veit Schiele", "Lizenz":
↪"BSD-3-Clause", "Ver\\u00f6ffentlichungsdatum": "2021-10-28"}, {"Titel": "Jupyter␣
↪Tutorial", "Sprache": "en", "Autor*innen": "Veit Schiele", "Lizenz": "BSD-3-Clause",
↪"Ver\\u00f6ffentlichungsdatum": "2019-06-27"}, {"Titel": "Jupyter Tutorial", "Sprache":
↪ "de", "Autor*innen": "Veit Schiele", "Lizenz": "BSD-3-Clause", "Ver\\
↪u00f6ffentlichungsdatum": "2020-10-26"}, {"Titel": "PyViz Tutorial", "Sprache": "en",
↪"Autor*innen": "Veit Schiele", "Lizenz": "BSD-3-Clause", "Ver\\u00f6ffentlichungsdatum
↪": "2020-04-13"}]'
```

Wie ihr ein JSON-Objekt oder eine Liste von Objekten in einen DataFrame oder eine andere Datenstruktur für die Analyse konvertiert, bleibt euch überlassen. Praktischerweise könnt ihr eine Liste von Dicts (die zuvor JSON-Objekte waren) an den DataFrame-Konstruktor übergeben:

[3]:
```
import pandas as pd
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
df = pd.DataFrame(data)

df
```

[3]:

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

pandas.read_json kann JSON-Datensätze in bestimmten Anordnungen automatisch in eine Serie oder einen DataFrame umwandeln, z.B.:

```
[4]: pd.read_json("books.json")
```

[4]:

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

Die Standardoptionen für `pandas.read_json` gehen davon aus, dass jedes Objekt im JSON-Array eine Zeile in der Tabelle ist.

Wenn ihr Daten aus pandas in JSON exportieren wollt, könnt ihr pandas.DataFrame.to_json verwenden:

```
[5]: print(df.to_json())
```

```
{"Titel":{"0":"Python basics","1":"Jupyter Tutorial","2":"Jupyter Tutorial","3":"PyViz␣
↪Tutorial"},"Sprache":{"0":"en","1":"en","2":"de","3":"en"},"Autor*innen":{"0":"Veit␣
↪Schiele","1":"Veit Schiele","2":"Veit Schiele","3":"Veit Schiele"},"Lizenz":{"0":"BSD-
↪3-Clause","1":"BSD-3-Clause","2":"BSD-3-Clause","3":"BSD-3-Clause"},"Ver\
↪u00f6ffentlichungsdatum":{"0":"2021-10-28","1":"2019-06-27","2":"2020-10-26","3":"2020-
↪04-13"}}
```

```
[6]: print(df.to_json(orient="records"))
```

```
[{"Titel":"Python basics","Sprache":"en","Autor*innen":"Veit Schiele","Lizenz":"BSD-3-
↪Clause","Ver\u00f6ffentlichungsdatum":"2021-10-28"},{"Titel":"Jupyter Tutorial",
↪"Sprache":"en","Autor*innen":"Veit Schiele","Lizenz":"BSD-3-Clause","Ver\
↪u00f6ffentlichungsdatum":"2019-06-27"},{"Titel":"Jupyter Tutorial","Sprache":"de",
↪"Autor*innen":"Veit Schiele","Lizenz":"BSD-3-Clause","Ver\u00f6ffentlichungsdatum":
↪"2020-10-26"},{"Titel":"PyViz Tutorial","Sprache":"en","Autor*innen":"Veit Schiele",
↪"Lizenz":"BSD-3-Clause","Ver\u00f6ffentlichungsdatum":"2020-04-13"}]
```

3.3.4 Excel

pandas unterstützt auch das Lesen von Tabellendaten, die in Dateien von Excel 2003 (und höher) gespeichert sind, entweder mit der Klasse `ExcelFile` oder der Funktion `pandas.read_excel`. Intern verwenden diese Werkzeuge die Zusatzpakete xlrd und openpyxl, um XLS- bzw. XLSX-Dateien zu lesen. Diese müssen separat von pandas mit pipenv installiert werden.

Um `ExcelFile` zu verwenden, erstellt eine Instanz, indem ihr einen Pfad zu einer xls- oder xlsx-Datei übergebt:

[1]:
```
import pandas as pd
```

[2]:
```
xlsx = pd.ExcelFile("library.xlsx")
```

Ihr könnt dann die Sheets der Datei anzeigen mit:

[3]:
```
xlsx.sheet_names
```

[3]:
```
['books']
```

[4]:
```
books = pd.read_excel(xlsx, "books")

books
```

[4]:

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

Wenn ihr mehrere Sheets einer Datei einlest, ist es schneller, die Excel-Datei zu erstellen, aber ihr könnt auch einfach den Dateinamen an `pandas.read_excel` übergeben:

[5]:
```
books = pd.read_excel("library.xlsx", "books")

books
```

[5]:

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

Um pandas-Daten im Excel-Format zu schreiben, müsst ihr zunächst einen `ExcelWriter` erstellen und dann mit pandas.DataFrame.to_excel Daten in diesen schreiben:

[6]:
```
writer = pd.ExcelWriter("library.xlsx")

books.to_excel(writer, "books")
writer.close()
```

Ihr könnt auch einen Dateipfad an `to_excel` übergeben und so den `ExcelWriter` umgehen:

[7]:
```
books.to_excel("library.xlsx")
```

3.3.5 XML/HTML

Übersicht

Unterstützung für Datenstrukturen	++	XML ist sehr flexibel, da jedes Element Attribute und beliebige Kindelemente haben kann.
Standardisierung	++	XML ist gut stadnardisiert, die Spezifikation findet ihr unter https://www.w3.org/TR/xml/. XML unterstützt sowohl DOM-Parser als auch streaming SAX-Parser.
Schema-IDL	++	XML schema, RELAX NG
Sprachunterstützung	+	Wird in allen wichtigen Sprachen unterstützt, üblicherweise mit integrierten Bibliotheken.
Menschliche Lesbarkeit	+-	XML ist ein lesbares Serialisierungsprotokoll. Ein Nachteil vom XML ist die Ausführlichket, insbesondere die beschreibenden End-Tags.
Geschwindigkeit	+	XML ist ziemlich schnell obwohl es normalerweise langsamer als JSON ist.
Dateigröße	--	XML ist im Vergleich am größten.

Beispiel

Quellcode 1: books.xml

```xml
<?xml version="1.0"?>

<!--
SPDX-FileCopyrightText: 2022 Veit Schiele

SPDX-License-Identifier: BSD-3-Clause
-->

<catalog>
   <book id="1">
      <title>Python basics</title>
      <language>en</language>
      <author>Veit Schiele</author>
      <license>BSD-3-Clause</license>
      <date>2021-10-28</date>
   </book>
   <book id="2">
      <title>Jupyter Tutorial</title>
      <language>en</language>
      <author>Veit Schiele</author>
      <license>BSD-3-Clause</license>
      <date>2019-06-27</date>
   </book>
   <book id="3">
      <title>Jupyter Tutorial</title>
      <language>de</language>
      <author>Veit Schiele</author>
      <license>BSD-3-Clause</license>
      <date>2020-10-26</date>
   </book>
   <book id="4">
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
      <title>PyViz Tutorial</title>
      <language>en</language>
      <author>Veit Schiele</author>
      <license>BSD-3-Clause</license>
      <date>2020-04-13</date>
   </book>
</catalog>
```

Siehe auch:

- Home
- Specification
- Validator
- The XML FAQ

XML/HTML-Beispiele

HTML

Python verfügt über zahlreiche Bibliotheken zum Lesen und Schreiben von Daten in den allgegenwärtigen HTML- und XML-Formaten. Beispiele sind *lxml*, *Beautiful Soup* und html5lib. Während lxml im Allgemeinen vergleichsweise viel schneller ist, können die anderen Bibliotheken besser mit fehlerhaften HTML- oder XML-Dateien umgehen.

pandas hat eine eingebaute Funktion, `read_html`, die Bibliotheken wie lxml, html5lib und Beautiful Soup verwendet, um automatisch Tabellen aus HTML-Dateien als DataFrame-Objekte zu parsen. Diese müssen zusätzlich installiert werden. Mit *Spack* könnt ihr lxml, BeautifulSoup und html5lib in eurem Kernel bereitstellen:

```
$ spack env activate python-311
$ spack install py-lxml py-beautifulsoup4~html5lib~lxml py-html5lib
```

Alternativ könnt ihr BeautifulSoup auch mit anderen Paketmanagern installieren, z.B.

```
$ pipenv install lxml beautifulsoup4 html5lib
```

Um zu zeigen, wie das funktioniert, verwende ich eine HTML-Datei der Wikipedia, die einen Überblick über verschiedene Serialisierungsformate gibt.

[1]:
```
import pandas as pd


tables = pd.read_html(
    "https://en.wikipedia.org/wiki/Comparison_of_data-serialization_formats"
)
```

Die Funktion `pandas.read_html` hat eine Reihe von Optionen, aber standardmäßig sucht sie nach allen Tabellendaten, die in `<table>`-Tags enthalten sind, und versucht, diese zu analysieren. Das Ergebnis ist eine Liste von DataFrame-Objekten:

[2]:
```
len(tables)
```

[2]:
```
3
```

[3]:
```
formats = tables[0]

formats.head()
```

[3]:
```
            Name                                  Creator-maintainer  \
0     Apache Avro                          Apache Software Foundation
1  Apache Parquet                          Apache Software Foundation
2           ASN.1                                      ISO, IEC, ITU-T
3         Bencode  Bram Cohen (creator) BitTorrent, Inc. (maintai...
4            Binn                                    Bernardo Ramos

         Based on Standardized?[definition needed]  \
0               -                                No
1               -                                No
2               -                               Yes
3               -                   De facto as BEP
4  JSON (loosely)                                No

                                    Specification  \
0                         Apache Avro™ Specification
1                                    Apache Parquet
2  ISO/IEC 8824 / ITU-T X.680 (syntax) and ISO/IE...
3          Part of BitTorrent protocol specification
4                                Binn Specification

                                    Binary?  \
0                                       Yes
1                                       Yes
2       BER, DER, PER, OER, or custom via ECN
3  Except numbers and delimiters, being ASCII
4                                       Yes

                     Human-readable? Supports references?e Schema-IDL?  \
0                            Partialg                     -    Built-in
1                                  No                   No           -
2  XER, JER, GSER, or custom via ECN                  Yesf    Built-in
3                                  No                   No          No
4                                  No                   No          No

                         Standard APIs Supports zero-copy operations
0  C, C#, C++, Java, PHP, Python, Ruby                             -
1                      Java, Python, C++                           No
2                                      -                          OER
3                                     No                           No
4                                     No                          Yes
```

Von hier aus können wir einige *Datenbereinigungen und -analysen* vornehmen, wie z.B. die Anzahl der verschiedenen Schema-IDL:

[4]:
```
formats["Schema-IDL?"].value_counts()
```

[4]:
```
Schema-IDL?
No
↪  15
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Yes                                                                                 ␣
↪    5
Built-in                                                                            ␣
↪    4
Schema WD                                                                           ␣
↪    1
Partial (Kwalify, Rx, built-in language type-defs)                                  ␣
↪    1
XML schema, RELAX NG                                                                ␣
↪    1
WSDL, XML schema                                                                    ␣
↪    1
Partial (JSON Schema Proposal, other JSON schemas/IDLs)                             ␣
↪    1
?                                                                                   ␣
↪    1
Ion schema                                                                          ␣
↪    1
Partial (JSON Schema Proposal, ASN.1 with JER, Kwalify, Rx, Itemscript Schema), JSON-LD ␣
↪    1
-                                                                                   ␣
↪    1
XML schema                                                                          ␣
↪    1
XML Schema                                                                          ␣
↪    1
Partial (Signature strings)                                                         ␣
↪    1
CDDL                                                                                ␣
↪    1
Schema-IDL?                                                                         ␣
↪    1
Name: count, dtype: int64
```

XML

pandas hat eine Funktion `read_xml`, wodurch das Lesen von XML-Dateien sehr einfach wird:

[5]:
```
pd.read_xml("books.xml")
```

[5]:

	id	title	language	author	license	date
0	1	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	2	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	3	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	4	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

lxml

Alternativ kann auch zunächst `lxml.objectify` zum Parsen von XML-Dateien verwendet werden. Dabei erhalten wir mit `getroot` einen Verweis auf den Wurzelknoten der XML-Datei:

[6]:
```
from lxml import objectify


parsed = objectify.parse(open("books.xml"))
root = parsed.getroot()
```

[7]:
```
books = []

for element in root.book:
    data = {}
    for child in element.getchildren():
        data[child.tag] = child.pyval
    books.append(data)
```

[8]:
```
pd.DataFrame(books)
```

[8]:

	title	language	author	license	date
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

BeautifulSoup

[1]:
```
import requests


url = "https://de.wikipedia.org/wiki/Liste_der_Stra%C3%9Fen_und_Pl%C3%A4tze_in_Berlin-
↪Mitte"
r = requests.get(url)
```

1. Installieren:

   Mit *Spack* könnt ihr BeautifulSoup in eurem Kernel bereitstellen:

   ```
   $ spack env activate python-38
   $ spack install py-beautifulsoup4@4.10.0%gcc@11.2.0~html5lib~lxml
   ```

   Alternativ könnt ihr BeautifulSoup auch mit anderen Paketmanagern installieren, z.B.

   ```
   $ pipenv install beautifulsoup4
   ```

2. Mit `r.content` können wir uns das HTML der Seite ausgeben lassen.
3. Als nächstes müssen wir diesen String mit BeautifulSoup in eine Python-Darstellung der Seite zerlegen:

[2]:
```
from bs4 import BeautifulSoup
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
soup = BeautifulSoup(r.content, "html.parser")
```

4. Um den Code zu strukturieren, erstellen wir eine neue Funktion `get_dom` (**D**ocument **O**bject **M**odel), die den gesamten vorhergehenden Code einschließt:

[3]:
```
def get_dom(url):
    r = request.get(url)
    r.raise_for_status()
    return BeautifulSoup(r.content, "html.parser")
```

Das Herausfiltern einzelner Elemente kann z.B. über CSS-Selektoren erfolgen. Diese können in einer Website ermittelt werden, indem ihr z.B. in Firefox mit der rechten Maustaste auf eine der Tabellenzellen in der ersten Spalte der Tabelle klickt. Im sich nun öffnenden *Inspector* könnt ihr das Element erneut mit der rechten Maustaste anklicken und dann *Copy* → *CSS Selector* auswählen. In der Zwischenablage befindet sich dann z.B. `table.wikitable:nth-child(13) > tbody:nth-child(2) > tr:nth-child(1)`. Diesen *CSS-Selector* bereinigen wir nun, da wir weder nach dem 13. Kindelement der Tabelle `table.wikitable` noch dem 2. Kindelement in `tbody` filtern wollen sondern nur nach der 1. Spalte innerhalb von `tbody`.

Schließlich lassen wir uns mit `limit=3` in diesem Notebook exemplarisch nur die ersten drei Ergebnisse anzeigen:

[4]:
```
links = soup.select(
    "table.wikitable > tbody > tr > td:nth-child(1) > a", limit=3
)

print(links)
```

```
[<a href="/wiki/Ackerstra%C3%9Fe_(Berlin)" title="Ackerstraße (Berlin)">Ackerstraße</a>,
↪<a href="/wiki/Alexanderplatz" title="Alexanderplatz">Alexanderplatz</a>, <a href="/
↪wiki/Almstadtstra%C3%9Fe" title="Almstadtstraße">Almstadtstraße</a>]
```

Wir wollen jedoch nicht den gesamten HTML-Link, sondern nur dessen Textinnhalt:

[5]:
```
for content in links:
    print(content.text)
```

```
Ackerstraße
Alexanderplatz
Almstadtstraße
```

Siehe auch

- Beautiful Soup Documentation

3.3.6 YAML

Übersicht

Unterstützung für Datenstrukturen	++	YAML, kurz für *YAML Ain't Markup Language*, unterstützt die meisten Datentypen, einschließlich Zeichenfolgen, Ganzzahlen, Gleitkommazahlen und Datumsangaben. YAML unterstützt sogar Referenzen und externe Daten.
Standardisation	+	YAML ist ein stark typisierter formaler Standard, aber es ist schwierig, Schema-Validatoren zu finden.
Schema-IDL	+-	Teilweise mit Kwalify, Rx und integrierten Sprachtypdefinitionen.
Language support	+-	Es gibt Bibliotheken für die beliebtesten Sprachen.
Human readability	+	Grundlegendes YAML ist wirklich einfach zu lesen, aber die Komplexität von YAML kann Leser stark verwirren.
Speed	--	YAML kann nur langsam serialisiert und deserialisiert werden.
File size	+-	YAML liegt im mittleren Bereich ähnlich wie *JSON* und *TOML*.

Siehe auch:

- Home
- Specification
- YAML Validator
- StrictYAML
- What YAML features does StrictYAML remove?
- noyaml.com

Beispiel

CITATION.cff:

```
# YAML 1.2
---
cff-version: 1.1.0
message: If you use this software, please cite it as below.
authors:
  - family-names: Druskat
    given-names: Stephan
    orcid: https://orcid.org/0000-0003-4925-7248
title: "My Research Software"
version: 2.0.4
doi: 10.5281/zenodo.1234
date-released: 2017-12-18
```

Ihr könnt euch YAML-Dateien als Python-Dictionaries ausgeben lassen mit:

```
[1]: import yaml
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
with open("CITATION.cff", "r") as file:
    cite = yaml.safe_load(file)
    print(cite)
```

```
{'cff-version': '1.1.0', 'message': 'If you use this software, please cite it as below.',
↪ 'authors': [{'family-names': 'Druskat', 'given-names': 'Stephan', 'orcid': 'https://
↪orcid.org/0000-0003-4925-7248'}], 'title': 'My Research Software', 'version': '2.0.4',
↪'doi': '10.5281/zenodo.1234', 'date-released': datetime.date(2017, 12, 18)}
```

3.3.7 TOML

Überblick

Unterstützung für Datenstrukturen	+	TOML (Tom's Obvious, Minimal Language) unterstützt die meisten Datenstrukturen, einschließlich Zeichenfolgen, Ganzzahlen, Gleitkommazahlen und Datumsangaben, jedoch keine Referenzen wie *YAML*.
Standardisation	++	TOML ist ein formaler, stark typisierter Standard.
Schema-IDL	+-	Teilweise mit JSON Schema Everywhere
Language support	++	TOML ist ein relativ junges Serialisierungsformat und findet noch nicht so breite Unterstützung wie JSON, CSV oder XML in den verschiedenen Programmiersprachen.
Human readability	++	Eines der Hauptziele von TOML war es, sehr einfach zu lesen zu sein.
Speed	+-	TOML kann mit mittlerer Geschwindigkeit verarbeitet werden.
File size	-	Nur *XML/HTML* ist weniger kompakt.

Beispiel

pyproject.toml

```
[tool.black]
line-length = 79

[tool.isort]
atomic=true
force_grid_wrap=0
include_trailing_comma=true
lines_after_imports=2
lines_between_types=1
multi_line_output=3
not_skip="__init__.py"
use_parentheses=true

known_first_party="jupyter-tutorial"
known_third_party=["mpi4py", "numpy", "requests"]
```

Ihr benötigt das Python-Paket toml, um TOML-Dateien in Python-Dictionaries umwandeln zu können. Anschließend könnt ihr TOML-Dateien laden, z.B. mit:

```
import toml


config = toml.load("pyproject.toml")
```

Siehe auch:

- Home
- GitHub
- Wiki
- What is wrong with TOML?
- An INI critique of TOML

Beispiel

pyproject.toml

```
[tool.black]
line-length = 79

[tool.isort]
atomic=true
force_grid_wrap=0
include_trailing_comma=true
lines_after_imports=2
lines_between_types=1
multi_line_output=3
not_skip="__init__.py"
use_parentheses=true

known_first_party=["MY_FIRST_MODULE", "MY_SECOND_MODULE"]
known_third_party=["mpi4py", "numpy", "requests"]
```

Für Python < 3.11 benötigt ihr das Python-Paket toml, um TOML-Dateien in Python-Dictionaries umwandeln zu können. Anschließend könnt ihr TOML-Dateien laden, z.B. mit:

[1]:
```
import toml


config = toml.load("pyproject.toml")

config
```

[1]:
```
{'tool': {'black': {'line-length': 79},
  'isort': {'atomic': True,
   'force_grid_wrap': 0,
   'include_trailing_comma': True,
   'lines_after_imports': 2,
   'lines_between_types': 1,
   'multi_line_output': 3,
   'not_skip': '__init__.py',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  'use_parentheses': True,
  'known_first_party': ['MY_FIRST_MODULE', 'MY_SECOND_MODULE'],
  'known_third_party': ['mpi4py', 'numpy', 'requests']}}}
```

3.3.8 Pickle

Überblick

Unterstützung von Datenstrukturen	+-	Pickle wird zum Speichern von Python-Objektstrukturen wie `list` oder `dict` in einem Byte-Stream verwendet. Im Gegensatz zu `marshal` werden bereits serialisierte Objekte getrackt, sodass spätere Verweise nicht erneut serialisiertt werden. Auch rekursive Objekte sind möglich.
Standardisierung	++	Pickle ist definiert in den Python Enhancement Proposals 307, 3154 und 574.
Schema IDL	--	Nein
Sprachunterstützung	--	Python-spezifisch
Menschliche Lesbarkeit	+-	Pickle ist ein binäres Serialisierungsformat, das jedoch einfach mit Python gelesen werden kann.
Geschwindigkeit	+-	Das Pickle-Format kann von Python meist schnell serialisiert und deserialisiert werden; siehe jedoch Don't pickle your data.
Dateigröße	++	Kompaktes Binärformat, das jedoch noch weiter komprimiert werden kann, s. Data Compression and Archiving.

Siehe auch:

pickle – Python object serialization
: Dokumentation des `pickle`-Moduls

shelve – Python object persistence
: Indizierte Datenbanken von `pickle`-Objekten

Uwe Korn: The implications of pickling ML models
: Alternativen zu `pickle` für ML-Modelle

Ned Batchelder: Pickle's nine flaws
: Nachteile von `pickle` und Alternativen

Pickle-Beispiele

Python-`pickle`-Modul

In diesem Beispiel wollen wir mit dem Python pickle-Modul das foldende Dict im Pickle-Format speichern:

[1]:
```
pyviz = {
    "Titel": "PyViz Tutorial",
    "Sprache": "de",
    "Autor*innen": "Veit Schiele",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    "Lizenz": "BSD-3-Clause",
    "Veröffentlichungsdatum": "2020-04-13",
}
```

[2]:
```
import pickle
```

[3]:
```
with open("pyviz.pkl", "wb") as f:
    pickle.dump(pyviz, f, pickle.HIGHEST_PROTOCOL)
```

Nun lesen wir die Pickle-Datei wieder ein:

[4]:
```
with open("pyviz.pkl", "rb") as f:
    pyviz = pickle.load(f)

print(pyviz)
```

```
{'Titel': 'PyViz Tutorial', 'Sprache': 'de', 'Autor*innen': 'Veit Schiele', 'Lizenz':
↪'BSD-3-Clause', 'Veröffentlichungsdatum': '2020-04-13'}
```

Auf diese Weise können wir Python-Objekte einfach persistent speichern.

Warnung

`pickle` kann nur als kurzfristiges Speicherformat empfohlen werden. Das Problem ist, dass nicht garantiert wird, dass das Format im Laufe der Zeit stabil bleibt; ein heute gepickeltes Objekt lässt sich möglicherweise mit einer späteren Version der Bibliothek nicht mehr entpickeln.

pandas

Alle Pandas-Objekte haben eine `to_pickle`-Methode, die Daten im Pickle-Format auf die Festplatte schreibt:

[5]:
```
import pandas as pd


books = pd.read_pickle("books.pkl")

books
```

[5]:

	Titel	Sprache	Autor*innen	Lizenz	Veröffentlichungsdatum
0	Python basics	en	Veit Schiele	BSD-3-Clause	2021-10-28
1	Jupyter Tutorial	en	Veit Schiele	BSD-3-Clause	2019-06-27
2	Jupyter Tutorial	de	Veit Schiele	BSD-3-Clause	2020-10-26
3	PyViz Tutorial	en	Veit Schiele	BSD-3-Clause	2020-04-13

pandas-Objekte haben alle eine `to_pickle`-Methode, die die Daten im Pickle-Format auf die Festplatte schreibt:

[6]:
```
books.to_pickle("books.pkl")
```

3.3.9 Protocol Buffers (Protobuf)

Überblick

Unterstützung für Datenstrukturen	+	Protobuf erlaubt euch Datenstrukturen in `*.proto`-Dateien zu definieren. Dabei unterstützt Protobuf viele primitive Datentypen, die zu verschachtelten Datentypen kombiniert werden können.
Standardisierung	+-	Protobuf ist ein stark typisierter flexibler Standard.
Schema-IDL	++	Eingabauter IDL-Compiler
Sprachunterstützung	++	Das protobuf-Format wird gut von den meisten Programmiersprachen unterstützt.
Menschliche Lesbarkeit	--	Protobuf ist nicht für Menschen lesbar.
Geschwindigkeit	++	Protobuf ist sehr schnell, vor allem in C++.
Dateigröße	++	Protobuf ist das kompakteste Format.

Siehe auch:

- Home
- GitHub
- Language Guide (proto3)
- Buf
  - Home
  - Docs
  - GitHub
- *gRPC*

3.3.10 Andere Formate

Apache Avro
: Ein kompaktes und schnelles Binärdatenformat.

  Siehe auch:

  - Specification

BSON
: Abkürzung für *Binary JSON*. Ein binäres Datenformat hauptsächlich für *MongoDB*

  Siehe auch:

  - Specification
  - MongoDB Extended JSON
  - bsondump

Cap'n Proto
: Ein schnelles Datenaustauschformat.

  Siehe auch:

- GitHub

JSON5
Eine Obermenge von JSON durch Einfügen von Zeichenfolgen mit mehreren Zeilen Escape-Zeichen, Hexadezimalzahlen, Kommentaren usw.

Siehe auch:

- PyPI

HOCON
Abkürzung für *Human-Optimized Config Object Notation*. Eine JSON-Obermenge mit Kommentaren, mehrzeiligen Zeichenfolgen usw.

Siehe auch:

- GitHub
- Play framework configuration file syntax and features

MessagePack
Ein effizientes binäres Serialisierungsformat, das von *Redis*-Skripten unterstützt wird.

Siehe auch:

- Specification
- GitHub

SDLang
Abkürzung für *Simple Declarative Language*. Stellt Daten in einer XML-ähnlichen Struktur in Textform dar.

Siehe auch:

- Language Guide
- GitHub

XDR (:rfc:`4506`)
Abkürzung für *External Data Representation Standard*. Nützlich zum Übertragen von Daten zwischen verschiedenen Computerarchitekturen.

3.4 Intake

Intake erleichtert das Auffinden, Untersuchen, Laden und Verbreiten von Daten. Es ist damit nicht nur für die Datenwissenschaften und das Data-Engineering interessant, sondern auch für das Anbieten von Daten.

Siehe auch:

- Docs
- GitHub
- Intake: Taking the Pain out of Data Access
- Intake: Parsing Data from Filenames and Paths
- Intake: Discovering and Exploring Data in a Graphical Interface
- Accessing Remote Data with a Generalized File System
- Intake: Caching Data on First Read Makes Future Analysis Faster

3.4.1 Intake installieren

Anforderungen

Für die Verwendung von *intake.gui* müssen aktuelle Versionen von Bokeh2.0 und Panel verfügbar sein.

Installation

Intake lässt sich einfach für euren Jupyter-Kernel installieren mit:

```
$ pipenv install intake
```

Katalog mit Beispieldaten erstellen

Für die nachfolgenden Beispiele benötigen wir einige Datensätze, die wir erstellen mit:

```
$ pipenv run intake example
Creating example catalog...
  Writing us_states.yml
  Writing states_1.csv
  Writing states_2.csv

To load the catalog:
    >>> import intake
    >>> cat = intake.open_catalog('us_states.yml')
```

3.4.2 Intake für die Datenwissenschaften

Intake erleichtert das Laden vieler verschiedener Formate und Typen. Um einen vollständigen Überblick zu erhalten, schaut euch das Plugin Directory und das Intake Project Dashboard an. Intake überführt die Daten dann in übliche Speicherformate wie Pandas DataFrames, Numpy-Arrays oder Python-Listen. Anschließend sind sie leicht durchsuchbar und auch für verteilte Systeme zugänglich. Sollte euch ein Plugin fehlen, könnt ihr auch selbst welche estellen, wie in Making Drivers beschrieben.

Laden einer Datenquelle

Im Folgenden werden wir zwei csv-Datensätze lesen und in einen Intake-Katalog überführen.

[1]:
```
import intake


ds = intake.open_csv("states_*.csv")

print(ds)
```

```
<intake.source.csv.CSVSource object at 0x7fc5575ccd50>
```

Mit der open_*-Funktion von Intake lassen sich verschiedenen Datenquellen einlesen. Je nach Datenformat oder Dienst lassen sich unterschiedliche Argmuente verwenden.

Konfigurieren des Suchpfades für Datenquellen

Intake überprüft die Intake-Konfigurationsdatei nach `catalog_patund` die Umgebungsvariable `"INTAKE_PATH"` auf eine durch Doppelpunkte getrennte Liste von Pfaden bzw. Semikolon in Windows, um nach Katalogdateien zu suchen. Beim Import `intake` werden alle Einträge aus allen Katalogen angezeigt, auf die als Teil eines globalen Katalogs von `intake.cat` referenziert werden.

Daten lesen

Intake liest Daten in Container verschiedener Formate:

- Tabellen in Pandas DataFrames
- Mehrdimensionale Array in Numpy Arrays
- Semistrukturierte Daten in Python-Listen von Objekten, üblicherweise Dictionaries

Um herauszufinden, in welchem Containerformat Intake die Daten vorhält, könnt ihr das `container`-Attribut verwenden:

[2]:
```
ds.container
```

[2]:
```
'dataframe'
```

Neben `dataframe` kann das Ergebnis auch `ndarray` oder `python` sein.

[3]:
```
df = ds.read()

df.head()
```

[3]:
```
        state        slug code               nickname  \
0     Alabama     alabama   AL      Yellowhammer State
1      Alaska      alaska   AK       The Last Frontier
2     Arizona     arizona   AZ  The Grand Canyon State
3    Arkansas    arkansas   AR       The Natural State
4  California  california   CA            Golden State

                  website admission_date  admission_number capital_city  \
0  http://www.alabama.gov     1819-12-14                22   Montgomery
1       http://alaska.gov     1959-01-03                49       Juneau
2          https://az.gov     1912-02-14                48      Phoenix
3     http://arkansas.gov     1836-06-15                25  Little Rock
4       http://www.ca.gov     1850-09-09                31   Sacramento

                       capital_url  population  population_rank  \
0      http://www.montgomeryal.gov     4833722               23
1            http://www.juneau.org      735132               47
2          https://www.phoenix.gov     6626624               15
3        http://www.littlerock.org     2959373               32
4  http://www.cityofsacramento.org    38332521                1

                                  constitution_url  \
0  http://alisondb.legislature.state.al.us/alison...
1  http://www.legis.state.ak.us/basis/folioproxy...
2             http://www.azleg.gov/Constitution.asp
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3  http://www.arkleg.state.ar.us/assembly/Summary...
4           http://www.leginfo.ca.gov/const-toc.html

                                      state_flag_url  \
0  https://cdn.civil.services/us-states/flags/ala...
1  https://cdn.civil.services/us-states/flags/ala...
2  https://cdn.civil.services/us-states/flags/ari...
3  https://cdn.civil.services/us-states/flags/ark...
4  https://cdn.civil.services/us-states/flags/cal...

                                      state_seal_url  \
0  https://cdn.civil.services/us-states/seals/ala...
1  https://cdn.civil.services/us-states/seals/ala...
2  https://cdn.civil.services/us-states/seals/ari...
3  https://cdn.civil.services/us-states/seals/ark...
4  https://cdn.civil.services/us-states/seals/cal...

                                       map_image_url  \
0  https://cdn.civil.services/us-states/maps/alab...
1  https://cdn.civil.services/us-states/maps/alas...
2  https://cdn.civil.services/us-states/maps/ariz...
3  https://cdn.civil.services/us-states/maps/arka...
4  https://cdn.civil.services/us-states/maps/cali...

                           landscape_background_url  \
0  https://cdn.civil.services/us-states/backgroun...
1  https://cdn.civil.services/us-states/backgroun...
2  https://cdn.civil.services/us-states/backgroun...
3  https://cdn.civil.services/us-states/backgroun...
4  https://cdn.civil.services/us-states/backgroun...

                             skyline_background_url  \
0  https://cdn.civil.services/us-states/backgroun...
1  https://cdn.civil.services/us-states/backgroun...
2  https://cdn.civil.services/us-states/backgroun...
3  https://cdn.civil.services/us-states/backgroun...
4  https://cdn.civil.services/us-states/backgroun...

                         twitter_url  \
0    https://twitter.com/alabamagov
1        https://twitter.com/alaska
2                                NaN
3   https://twitter.com/arkansasgov
4  https://twitter.com/cagovernment

                                      facebook_url
0             https://www.facebook.com/alabamagov
1  https://www.facebook.com/AlaskaLocalGovernments
2                                              NaN
3           https://www.facebook.com/Arkansas.gov
4                                              NaN
```

[4]:
```
for chunk in ds.read_chunked():
    print("Chunk: %d" % len(chunk))
```

```
Chunk: 24
Chunk: 26
```

[5]:
```
ddf = ds.to_dask()

ddf.head()
```

[5]:
```
        state        slug code                nickname  \
0     Alabama     alabama   AL      Yellowhammer State
1      Alaska      alaska   AK       The Last Frontier
2     Arizona     arizona   AZ  The Grand Canyon State
3    Arkansas    arkansas   AR       The Natural State
4  California  california   CA             Golden State

                  website admission_date  admission_number capital_city  \
0  http://www.alabama.gov     1819-12-14                22   Montgomery
1      http://alaska.gov     1959-01-03                49       Juneau
2         https://az.gov     1912-02-14                48      Phoenix
3    http://arkansas.gov     1836-06-15                25  Little Rock
4       http://www.ca.gov     1850-09-09                31   Sacramento

                     capital_url  population  population_rank  \
0     http://www.montgomeryal.gov     4833722               23
1           http://www.juneau.org      735132               47
2         https://www.phoenix.gov     6626624               15
3       http://www.littlerock.org     2959373               32
4  http://www.cityofsacramento.org    38332521                1

                                 constitution_url  \
0  http://alisondb.legislature.state.al.us/alison...
1  http://www.legis.state.ak.us/basis/folioproxy...
2             http://www.azleg.gov/Constitution.asp
3  http://www.arkleg.state.ar.us/assembly/Summary...
4          http://www.leginfo.ca.gov/const-toc.html

                                    state_flag_url  \
0  https://cdn.civil.services/us-states/flags/ala...
1  https://cdn.civil.services/us-states/flags/ala...
2  https://cdn.civil.services/us-states/flags/ari...
3  https://cdn.civil.services/us-states/flags/ark...
4  https://cdn.civil.services/us-states/flags/cal...

                                    state_seal_url  \
0  https://cdn.civil.services/us-states/seals/ala...
1  https://cdn.civil.services/us-states/seals/ala...
2  https://cdn.civil.services/us-states/seals/ari...
3  https://cdn.civil.services/us-states/seals/ark...
4  https://cdn.civil.services/us-states/seals/cal...

                                     map_image_url  \
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
0  https://cdn.civil.services/us-states/maps/alab...
1  https://cdn.civil.services/us-states/maps/alas...
2  https://cdn.civil.services/us-states/maps/ariz...
3  https://cdn.civil.services/us-states/maps/arka...
4  https://cdn.civil.services/us-states/maps/cali...

                           landscape_background_url  \
0  https://cdn.civil.services/us-states/backgroun...
1  https://cdn.civil.services/us-states/backgroun...
2  https://cdn.civil.services/us-states/backgroun...
3  https://cdn.civil.services/us-states/backgroun...
4  https://cdn.civil.services/us-states/backgroun...

                             skyline_background_url  \
0  https://cdn.civil.services/us-states/backgroun...
1  https://cdn.civil.services/us-states/backgroun...
2  https://cdn.civil.services/us-states/backgroun...
3  https://cdn.civil.services/us-states/backgroun...
4  https://cdn.civil.services/us-states/backgroun...

                       twitter_url  \
0    https://twitter.com/alabamagov
1        https://twitter.com/alaska
2                               NaN
3   https://twitter.com/arkansasgov
4  https://twitter.com/cagovernment

                                    facebook_url
0             https://www.facebook.com/alabamagov
1  https://www.facebook.com/AlaskaLocalGovernments
2                                              NaN
3           https://www.facebook.com/Arkansas.gov
4                                              NaN
```

[6]:
```
cat = intake.open_catalog("us_states.yml")
```

[7]:
```
list(cat)
```

[7]:
```
['states']
```

[8]:
```
cat.states.to_dask()[["state", "slug"]].head()
```

[8]:
```
        state        slug
0     Alabama     alabama
1      Alaska      alaska
2     Arizona     arizona
3    Arkansas    arkansas
4  California  california
```

[9]:
```
cat.states(csv_kwargs={"header": None, "skiprows": 1}).read().head()
```

[9]:
```
           0           1   2                          3                          4   \
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
0     Alabama     alabama  AL      Yellowhammer State  http://www.alabama.gov
1      Alaska      alaska  AK       The Last Frontier       http://alaska.gov
2     Arizona     arizona  AZ  The Grand Canyon State          https://az.gov
3    Arkansas    arkansas  AR       The Natural State     http://arkansas.gov
4  California  california  CA            Golden State       http://www.ca.gov

           5   6            7                                 8         9   10  \
0  1819-12-14  22   Montgomery      http://www.montgomeryal.gov   4833722  23
1  1959-01-03  49       Juneau            http://www.juneau.org    735132  47
2  1912-02-14  48      Phoenix          https://www.phoenix.gov   6626624  15
3  1836-06-15  25  Little Rock        http://www.littlerock.org   2959373  32
4  1850-09-09  31   Sacramento  http://www.cityofsacramento.org  38332521   1

                                              11  \
0  http://alisondb.legislature.state.al.us/alison...
1  http://www.legis.state.ak.us/basis/folioproxy...
2             http://www.azleg.gov/Constitution.asp
3  http://www.arkleg.state.ar.us/assembly/Summary...
4          http://www.leginfo.ca.gov/const-toc.html

                                              12  \
0  https://cdn.civil.services/us-states/flags/ala...
1  https://cdn.civil.services/us-states/flags/ala...
2  https://cdn.civil.services/us-states/flags/ari...
3  https://cdn.civil.services/us-states/flags/ark...
4  https://cdn.civil.services/us-states/flags/cal...

                                              13  \
0  https://cdn.civil.services/us-states/seals/ala...
1  https://cdn.civil.services/us-states/seals/ala...
2  https://cdn.civil.services/us-states/seals/ari...
3  https://cdn.civil.services/us-states/seals/ark...
4  https://cdn.civil.services/us-states/seals/cal...

                                              14  \
0  https://cdn.civil.services/us-states/maps/alab...
1  https://cdn.civil.services/us-states/maps/alas...
2  https://cdn.civil.services/us-states/maps/ariz...
3  https://cdn.civil.services/us-states/maps/arka...
4  https://cdn.civil.services/us-states/maps/cali...

                                              15  \
0  https://cdn.civil.services/us-states/backgroun...
1  https://cdn.civil.services/us-states/backgroun...
2  https://cdn.civil.services/us-states/backgroun...
3  https://cdn.civil.services/us-states/backgroun...
4  https://cdn.civil.services/us-states/backgroun...

                                              16  \
0  https://cdn.civil.services/us-states/backgroun...
1  https://cdn.civil.services/us-states/backgroun...
2  https://cdn.civil.services/us-states/backgroun...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3  https://cdn.civil.services/us-states/backgroun...
4  https://cdn.civil.services/us-states/backgroun...

                                 17  \
0    https://twitter.com/alabamagov
1        https://twitter.com/alaska
2                               NaN
3   https://twitter.com/arkansasgov
4  https://twitter.com/cagovernment

                                                 18
0              https://www.facebook.com/alabamagov
1  https://www.facebook.com/AlaskaLocalGovernments
2                                              NaN
3            https://www.facebook.com/Arkansas.gov
4                                              NaN
```

3.4.3 Intake-GUI : Erkunden von Daten in einer grafischen Oberfläche

Intake-GUI wurde neu implementiert, sodass sie nicht nur in Jupyter-Notebooks, sondern auch in anderen Webanwendungen bereitgestellt werden kann. Er zeigt den Inhalt aller installierten Kataloge an und ermöglicht die Auswahl lokaler und entfernter Kataloge sowie das Durchsuchen und Auswählen von Einträgen aus diesen.

Intake unterstützt die Arbeitsteilung zwischen den Bereichen Data-Engineering, die Daten kuratieren, verwalten und verbreiten, und den Datenwissenschaften, die Daten analysieren und visualisieren ohne wissen zu müssen, wie sie gespeichert sind.

Das Intake-GUI basiert auf Panel wobei das Bedienfeld eine zusammensetzbare Dashboard-Lösung für die Anzeige von Plots, Bildern, Tabellen, Texten sowie Widgets bietet. Panel funktioniert sowohl in einem Jupyter-Notebook als auch in einer eigenständigen Tornado-Anwendung.

Aus Sicht des Data-Engineering bedeutet dies, dass die Aufnahme-GUI an einem Endpunkt bereitgestellt und als Datenexplorationswerkzeug für Datenbenutzer verwenden werden kann. Dies bedeutet auch, dass es einfach ist, die GUI anzupassen und neu zu organisieren, um euer eigenes Logo einzufügen, Teile davon in euren eigenen Anwendungen wiederzuverwenden oder neue Funktionen hinzuzufügen.

Zukünftig soll Intake-GUI auch die Eingabe von Benutzerparametern erlauben sowie das Bearbeiten und Speichern von Katalogen.

[1]:
```
import intake


intake.gui
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[1]:
```
Row
    [0] Column(margin=(25, 0, 0, 0), width=50)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        [0] PNG(str, align='center')
    [1] Column(max_width=1600, name='GUI', width_policy='max')
        [0] Row
            [0] Column
                [0] Column(name='Select Catalog')
                    [0] Markdown(str, max_height=40)
                    [1] MultiSelect(min_width=200, options={'builtin': <Intake c...},␣
↪size=9, value=[<Intake catalog: b...], width_policy='min')
                [1] Row(name='Controls')
                    [0] Toggle(name='', width=50)
                    [1] Button(name='-', width=50)
                    [2] Toggle(name='', width=50)
            [1] Column
                [0] Column(name='Select Data Source')
                    [0] Markdown(str, max_height=40)
                    [1] MultiSelect(min_width=200, size=9, width_policy='min')
                [1] Row(name='Controls')
                    [0] Toggle(disabled=True, name='', width=50)
                    [1] Toggle(disabled=True, name='', width=50)
            [2] Column(height=240, name='Description', width_policy='max')
                [0] Markdown(None, max_height=40)
                [1] Str(str, css_classes=['scrolling'], height=200, sizing_mode='stretch_
↪width')
        [1] Row(height_policy='min', max_width=1600, name='Search', width_policy='max')
        [2] Column(max_width=1600, name='Add Catalog', width_policy='max')
        [3] Column(name='Plot', width_policy='max')
```

Das GUI enthält drei Hauptbereiche:

1. eine Liste von Katalogen. Der standardmäßig angezeigte *builtin*-Katalog enthält im System installierte Datensätze, genau wie `intake.cat`.
2. eine Liste der Quellen im aktuell ausgewählten Katalog.
3. eine Beschreibung der aktuell ausgewählten Quelle.

Ad 1: Kataloge

Aktuell wird in der Liste der Kataloge kein Katalog angezeigt. Unter den drei Hauptbereichen befinden sich jedoch drei Schaltflächen, mit denen Kataloge hinzugefügt, entfernt oder durchsucht werden können.

Die Schaltflächen sind auch über die API verfügbar, z.B. für *Add Catalog* mit:

[2]:
```
intake.gui.add("./us_crime/us_crime.yaml")
```

Remote-Kataloge sind z.B. verfügbar unter

- https://s3.amazonaws.com/earth-data/UCMerced_LandUse/catalog.yml
- https://raw.githubusercontent.com/ContinuumIO/anaconda-package-data/master/catalog/anaconda_package_data.yaml
- https://raw.githubusercontent.com/pangeo-data/pangeo-datastore/master/intake-catalogs/master.yaml

Ad 2. Quellen

Durch die Auswahl einer Quelle aus der Liste wird der Beschreibungstext auf der linken Seite der Benutzeroberfläche aktualisiert.

Auch dieser ist über die API verfügbar:

[3]:
```
intake.gui.sources
```

[3]:
```
[name: us_crime
 container: dataframe
 plugin: ['csv']
 description: US Crime data [UCRDataTool](https://www.ucrdatatool.gov/Search/Crime/State/
↪StatebyState.cfm)
 direct_access: forbid
 user_parameters: []
 metadata:
   plots:
     line_example:
       kind: line
       y: ['Robbery', 'Burglary']
       x: Year
     violin_example:
       kind: violin
       y: ['Burglary rate', 'Larceny-theft rate', 'Robbery rate', 'Violent Crime rate']
       group_label: Type of crime
       value_label: Rate per 100k
       invert: True
 args:
   urlpath: {{ CATALOG_DIR }}/data/crime.csv]
```

Dieser besteht aus einer Liste regulärer Intake-Datenquelleinträge. Um uns die ersten Einträge anzuschauen, können wir folgendes eingeben:

[4]:
```
source = intake.gui.sources[0]

source.to_dask().head()
```

[4]:
```
   textasciitildeYear  Population  Violent crime total  \
0        1960   179323175               288460
1        1961   182992000               289390
2        1962   185771000               301510
3        1963   188483000               316970
4        1964   191141000               364220

   Murder and nonnegligent Manslaughter  Legacy rape /1  Revised rape /2  \
0                                  9110           17190              NaN
1                                  8740           17220              NaN
2                                  8530           17550              NaN
3                                  8640           17650              NaN
4                                  9360           21420              NaN

   Robbery  Aggravated assault  Property crime total  Burglary  ...  \
0   107840              154320               3095700    912100  ...
1   106670              156760               3198600    949600  ...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2   110860              164570               3450700    994300  ...
3   116470              174210               3792500   1086400  ...
4   130390              203050               4200400   1213200  ...

   Violent Crime rate  Murder and nonnegligent manslaughter rate  \
0               160.9                                        5.1
1               158.1                                        4.8
2               162.3                                        4.6
3               168.2                                        4.6
4               190.6                                        4.9

   Legacy rape rate /1  Revised rape rate /2  Robbery rate  \
0                  9.6                   NaN          60.1
1                  9.4                   NaN          58.3
2                  9.4                   NaN          59.7
3                  9.4                   NaN          61.8
4                 11.2                   NaN          68.2

   Aggravated assault rate  Property crime rate  Burglary rate  \
0                     86.1               1726.3          508.6
1                     85.7               1747.9          518.9
2                     88.6               1857.5          535.2
3                     92.4               2012.1          576.4
4                    106.2               2197.5          634.7

   Larceny-theft rate  Motor vehicle theft rate
0              1034.7                     183.0
1              1045.4                     183.6
2              1124.8                     197.4
3              1219.1                     216.6
4              1315.5                     247.4

[5 rows x 22 columns]
```

[5]:
```
source.gui
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[5]:
```
Row(name='Source')
```

[6]:
```
intake.gui.source.description
```

[6]:
```
Column(height=240, name='Description', width_policy='max')
    [0] Markdown(str, max_height=40)
    [1] Str(str, css_classes=['scrolling'], height=200, sizing_mode='stretch_width')
```

[7]:
```
cat = intake.open_catalog("./us_crime/us_crime.yaml")

cat.gui
```

[7]:
```
Row
    [0] Column(margin=(25, 0, 0, 0), width=50)
        [0] PNG(str, align='center')
    [1] Column(max_width=1600, name='Entries', width_policy='max')
        [0] Row
            [0] Column
                [0] Column(name='Select Data Source')
                    [0] Markdown(str, max_height=40)
                    [1] MultiSelect(min_width=200, options=OrderedDict([('us_crime',␣
↪...]), size=9, value=[name: us_crime
container:...], width_policy='min')
                [1] Row(name='Controls')
                    [0] Toggle(name='', width=50)
                    [1] Toggle(disabled=True, name='', width=50)
            [1] Column(height=240, name='Description', width_policy='max')
                [0] Markdown(str, max_height=40)
                [1] Str(str, css_classes=['scrolling'], height=200, sizing_mode='stretch_
↪width')
        [1] Column(name='Plot', width_policy='max')
```

[8]:
```
us_crime = cat.gui.sources[0]
```

[9]:
```
intake.output_notebook()

us_crime.plot.bivariate(
    "Burglary rate",
    "Property crime rate",
    legend=False,
    width=500,
    height=400
) * us_crime.plot.scatter(
    "Burglary rate",
    "Property crime rate",
    color="black",
    size=15,
    legend=False,
) + us_crime.plot.table(
    [
        "Burglary rate",
        "Property crime rate"
    ],
    width=350,
    height=350
)
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

[9]:
```
:Layout
   .Overlay.I :Overlay
      .Bivariate.I :Bivariate   [Burglary rate,Property crime rate]   (Density)
      .Scatter.I   :Scatter   [Burglary rate]   (Property crime rate)
   .Table.I   :Table   [Burglary rate,Property crime rate]
```

Ad 3. Quellansicht

Sobald Kataloge geladen sind und die gewünschten Quellen ausgewählt wurden, sind diese unter dem Attribut `intake.gui.sources` verfügbar. Jeder Quelleneintrag verfügt über Methoden und kann wie jeder Katalogeintrag als Datenquelle geöffnet werden. Für *Source: UCMerced_LandUse_by_landuse* sieht der Eintrag beispielsweise so aus:

```
name: UCMerced_LandUse_by_landuse
container: None
plugin: []
description: All images matching given landuse from UCMerced_LandUse/Image.
direct_access: forbid
user_parameters: [{'name': 'landuse', 'description': 'which landuse to collect', 'type':
↪'str', 'default': 'airplane'}]
metadata:
args:
  urlpath: s3://earth-data/UCMerced_LandUse/Images/{{ landuse }}/{{ landuse }}{id:2d}.tif
  storage_options:
    anon: True
  concat_dim: id
  coerce_shape: [256, 256]
```

Unter der Liste der Quellen befindet sich eine Reihe von Schaltflächen zum Erschließen der ausgewählten Datenquelle: Dabei öffnet **Plot** ein Unterfenster zum Anzeigen der vordefinierten (d.h. der in yaml angegebenen) Plots für die ausgewählte Quelle.

Siehe auch

- GUI

3.4.4 Intake für das Data-Engineering

Intake unterstützt das Data-Engineering beim Bereitstellen von Daten und bei der Spezifikation der Datenquellen, der Verteilung der Daten, der Parametrisierung der Benutzeroptionen etc. Dies erleichtert im anschließenden Data-Engineering das einfache Erschließen der Daten, da die möglichen Optionen bereits im Katalog angegeben sind.

[1]:
```
import hvplot.pandas
import intake


intake.output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

Intake-Datensätze werden mit sog. Treibern geladen, von denen einige schon mit dem Intake-Paket mitkommen, andere jedoch als Plugin nachgeladen werden müssen. Die verfügbaren Treiber könnt ihr euch folgendermaßen anzeigen lassen:

[2]:
```
list(intake.registry)
```

[2]:
```
['parquet',
 'alias',
 'catalog',
 'csv',
 'intake_remote',
 'json',
 'jsonl',
 'ndzarr',
 'numpy',
 'textfiles',
 'tiled',
 'tiled_cat',
 'yaml_file_cat',
 'yaml_files_cat',
 'zarr_cat']
```

Jedem dieser Treiber ist eine `intake.open_*`-Funktion zugeordnet. Es ist auch möglich, auf Treiber mit dem vollständig qualifizierten Namen (z.B. `package.submodule.DriverClass`) zu verweisen. Im folgenden Beispiel konzentrieren wir uns jedoch auf den `csv`-Treiber, der in der Standard-Intake-Installation enthalten ist.

Im Allgemeinen besteht der erste Schritt zum Schreiben eines Katalogeintrags darin, die entsprechende `open_*` Funktion zum Erstellen eines DataSource-Objekts zu verwenden:

[3]:
```
source = intake.open_csv(
    "https://timeseries.weebly.com/uploads/2/1/0/8/21086414/sea_ice.csv"
)
```

Die obige Angabe hat nun ein DataSource-Objekt erstellt, aber noch nicht überprüft, ob tatsächlich auf die Daten zugegriffen werden kann. Um zu testen, ob das Laden wirklich erfolgreich war, kann die Quelle selbst erschlossen (`source.discover`) oder gelesen (`source.read`) werden:

[4]:
```
source.discover()
```

[4]:
```
{'dtype': {'Time': 'object', 'Arctic': 'float64', 'Antarctica': 'float64'},
 'shape': (None, 3),
 'npartitions': 1,
 'metadata': {}}
```

[5]:
```
df = source.read()
df.head()
```

[5]:
```
      Time  Arctic  Antarctica
0  1990M01   12.72        3.27
1  1990M02   13.33        2.15
2  1990M03   13.44        2.71
3  1990M04   12.16        5.10
4  1990M05   10.84        7.37
```

Nachdem wir festgestellt haben, das sich die Daten wie gewünscht laden lassen, wollen wir uns die Daten visuell erschließen:

[6]:
```
df.hvplot(
    kind="line", x="Time", y=["Arctic", "Antarctica"], width=700, height=500
)
```

[6]:
```
:NdOverlay   [Variable]
   :Curve   [Time]   (value)
```

Wir können nun eine Quelle korrekt laden und erhalten auch eine grafische Ausgabe zur Erschließung der Daten. Dieses Rezept können wir uns nun in der YAML-Syntax ausgeben lassen mit:

[7]:
```
print(source.yaml())
```

```
sources:
  csv:
    args:
      urlpath: https://timeseries.weebly.com/uploads/2/1/0/8/21086414/sea_ice.csv
    description: ''
    driver: intake.source.csv.CSVSource
    metadata: {}
```

Schließlich können wir eine YAML-Datei erstellen, die dieses Rezept mit einer zusätzlichen Beschreibung und dem getesteten Diagramm enthält:

[8]:
```
%%writefile sea.yaml
sources:
    sea_ice:
      args:
        urlpath: "https://timeseries.weebly.com/uploads/2/1/0/8/21086414/sea_ice.csv"
      description: "Polar sea ice cover"
      driver: csv
      metadata:
        plots:
          basic:
            kind: line
            x: Time
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            y: [Arctic, Antarctica]
            width: 700
            height: 500
```

```
Overwriting sea.yaml
```

Um zu überprüfen, ob die YAML-Datei auch funktioniert, können wir sie erneut laden und versuchen, damit zu arbeiten:

[9]:
```
cat = intake.open_catalog("sea.yaml")
```

[10]:
```
cat.sea_ice.plot.basic()
```

[10]:
```
:NdOverlay   [Variable]
   :Curve   [Time]   (value)
```

Der Katalog scheint funktionsfähig und kann nun freigegeben werden. Der einfachste Weg, einen Intake-Katalog freizugeben, besteht darin, ihn an einem Ort abzulegen, an dem er von eurer Zielgruppe gelesen werden kann. In diesem Tutorial, das in einem Git-Repo gespeichert ist, kann dies die URL der Datei im Repo sein. Alles, was ihr für eure Benutzer freigeben müsst, ist die URL des Katalogs. Ihr könnt dies selbst ausprobieren mit:

[11]:
```
cat = intake.open_catalog(
    "https://raw.githubusercontent.com/veit/Python4DataScience-de/main/docs/data-
↪processing/intake/sea.yaml"
)
```

[12]:
```
cat.sea_ice.read().head()
```

[12]:

	Time	Arctic	Antarctica
0	1990M01	12.72	3.27
1	1990M02	13.33	2.15
2	1990M03	13.44	2.71
3	1990M04	12.16	5.10
4	1990M05	10.84	7.37

Hinweis

Dieser Katalog ist auch wieder eine DataSource-Instanz, d.h., dass ihr aus anderen Katalogen darauf verweisen und so eine Hierarchie von Datenquellen aufbauen könnt. Beispielsweise verfügt ihr über einen Stamm- oder Hauptkatalog, der auf mehrere andere Kataloge verweist, die jeweils Einträge eines bestimmten Typs enthalten. und das Ganze kann z.B. mit Intake-GUI durchsucht werden. Auf diese Weise hat die Datenerfassung insgesamt eine Struktur, die das Navigieren zum richtigen Datensatz vereinfacht. Ihr könnt sogar separate Hierarchien haben, die auf dieselben Daten verweisen.

[13]:
```
print(cat.yaml())
```

```
sources:
  sea:
    args:
      path: https://raw.githubusercontent.com/veit/Python4DataScience-de/main/docs/data-
↪processing/intake/sea.yaml
    description: ''
    driver: intake.catalog.local.YAMLFileCatalog
    metadata: {}
```

3.5 httpx

httpx vereinfacht HTML-Anfragen gegenüber der Python-Standardbibliothek urllib.request.

3.5.1 httpx-Installation und Beispielanwendung

Installation

Für die Kommunikation mit solchen REST-APIs ist die httpx-Bibliothek hilfreich. Mit Spack könnt ihr httpx in eurem Kernel bereitstellen:

```
$ spack env activate python-311
$ spack install py-httpx
```

Alternativ könnt ihr httpx auch mit anderen Paketmanagern installieren, z.B.

```
$ pipenv install httpx
```

Beispiel OSM Nominatim-API

In diesem Beispiel holen wir unsere Daten von der OpenStreetMap Nominatim-API. Diese ist erreichbar über die URL `https://nominatim.openstreetmap.org/search?`. Um z.B. Informationen über das Berlin Congress Center in Berlin im JSON-Format zu erhalten, sollte die URL `https://nominatim.openstreetmap.org/search.php?q=Alexanderplatz+Berlin&format=json` angegeben werden, und wenn ihr euch den entsprechenden Kartenausschnitt anzeigen lassen wollt, so müsst ihr einfach nur `&format=json` weglassen

Anschließend definieren wir die Such-URL und die Parameter. Nominatim erwartet mindestens die folgenden beiden Parameter

Schlüssel	Werte
`q`	Adressabfrage, die folgende Spezifikationen erlaubt: `street`, `city`, `county`, `state`, `country` und `postalcode`.
`format`	Format, in dem die Daten zurückgegeben werden. Möglich Werte sind `html`, `xml`, `json`, `jsonv2`, `geojson` und `geocodejson`.

Die Abfrage kann dann gestellt werden mit:

[1]:
```
import httpx


search_url = "https://nominatim.openstreetmap.org/search?"
params = {
    "q": "Alexanderplatz, Berlin",
    "format": "json",
}
r = httpx.get(search_url, params=params)
```

[2]:
```
r.status_code
```

[2]:
```
200
```

[3]:
```
r.json()
```

[3]:
```
[{'place_id': 159000335,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
  'osm_type': 'way',
  'osm_id': 783052052,
  'lat': '52.5219814',
  'lon': '13.413635717448294',
  'class': 'place',
  'type': 'square',
  'place_rank': 25,
  'importance': 0.47149825263735834,
  'addresstype': 'square',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']},
 {'place_id': 159254539,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
  'osm_type': 'node',
  'osm_id': 3908141014,
  'lat': '52.5215661',
  'lon': '13.4112804',
  'class': 'railway',
  'type': 'station',
  'place_rank': 30,
  'importance': 0.43609907778808027,
  'addresstype': 'railway',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Dircksenstraße, Mitte, Berlin, 10179, Deutschland',
  'boundingbox': ['52.5165661', '52.5265661', '13.4062804', '13.4162804']},
 {'place_id': 159367604,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
  'osm_type': 'way',
  'osm_id': 346206374,
  'lat': '52.5216214',
  'lon': '13.4131913',
  'class': 'highway',
  'type': 'pedestrian',
  'place_rank': 26,
  'importance': 0.10000999999999993,
  'addresstype': 'road',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5216214', '52.5216661', '13.4131913', '13.4131914']},
 {'place_id': 159038218,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
  'osm_type': 'way',
  'osm_id': 301241483,
  'lat': '52.5227066',
  'lon': '13.415336',
  'class': 'highway',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  'type': 'primary',
  'place_rank': 26,
  'importance': 0.10000999999999993,
  'addresstype': 'road',
  'name': 'Alexanderstraße',
  'display_name': 'Alexanderstraße, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5226699', '52.5227698', '13.4152008', '13.4154146']}]
```

Es werden drei verschiedene Orte gefunden, der Platz, eine Bushaltestelle und ein Hotel. Um nun weiter filtern zu können, können wir uns nur den bedeutendsten Ort anzeigen lassen:

[4]:
```
params = {"q": "Alexanderplatz, Berlin", "format": "json", "limit": "1"}
r = httpx.get(search_url, params=params)

r.json()
```

[4]:
```
[{'place_id': 159000335,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
  'osm_type': 'way',
  'osm_id': 783052052,
  'lat': '52.5219814',
  'lon': '13.413635717448294',
  'class': 'place',
  'type': 'square',
  'place_rank': 25,
  'importance': 0.47149825263735834,
  'addresstype': 'square',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']}]
```

Clean Code

Nachdem wir nun wissen, dass der Code funktioniert, wollen wir alles in eine saubere und flexible Funktion umwandeln.

Um sicherzustellen, dass die Interaktion erfolgreich war, verwenden wir die Methode `raise_for_status` von `httpx`, die eine Exception auslöst, wenn der HTTP-Statuscode nicht `200 OK` ist:

[5]:
```
r.raise_for_status()
```

[5]:
```
<Response [200 OK]>
```

Da wir die Lastgrenzen der Nominatim-API nicht überschreiten möchten, werden wir unsere Anforderungen mit der Funktion `time.sleep` verzögern:

[6]:
```
from time import sleep


sleep(1)
r.json()
```

[6]:
```
[{'place_id': 159000335,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  'osm_type': 'way',
  'osm_id': 783052052,
  'lat': '52.5219814',
  'lon': '13.413635717448294',
  'class': 'place',
  'type': 'square',
  'place_rank': 25,
  'importance': 0.47149825263735834,
  'addresstype': 'square',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']}]
```

Als nächstes deklarieren wir die Funktion selbst. Als Argumente benötigen wir die Adresse, das Format, das Limit der zurückzugebenden Objekte mit dem Standardwert 1 und weitere `kwargs` (**k**ey**w**ord **arg**ument**s**), die als Parameter übergeben werden:

[7]:
```
def nominatim_search(address, format="json", limit=1, **kwargs):
    """Thin wrapper around the Nominatim search API.
    For the list of parameters see
    https://nominatim.org/release-docs/develop/api/Search/#parameters
    """
    search_url = "https://nominatim.openstreetmap.org/search?"
    params = {"q": address, "format": format, "limit": limit, **kwargs}
    r = httpx.get(search_url, params=params)
    # Raise an exception if the status is unsuccessful
    r.raise_for_status()

    sleep(1)
    return r.json()
```

Nun können wir die Funktion ausprobieren, z.B. mit

[8]:
```
nominatim_search("Alexanderplatz, Berlin")
```

[8]:
```
[{'place_id': 159000335,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. http://osm.org/copyright',
  'osm_type': 'way',
  'osm_id': 783052052,
  'lat': '52.5219814',
  'lon': '13.413635717448294',
  'class': 'place',
  'type': 'square',
  'place_rank': 25,
  'importance': 0.47149825263735834,
  'addresstype': 'square',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']}]
```

Caching

Falls innerhalb einer Session immer wieder dieselben Abfragen gestellt werden sollen,ist es sinnvoll, diese Daten nur einmal abzurufen und wiederzuverwenden. In Python können wir `lru_cache` aus der `functools`-Standardbibliothek von Python verwenden. `lru_cache` speichert die N letzten Anfragen (**L**east **R**ecent **U**sed) und sobald das Limit überschritten wird, werden die ältesten Werte verworfen. Um dies für die Methode `nominatim_search` zu verwenden, müsst ihr lediglich einen Import und einen *Decorator* defnieren:

[9]:
```
from functools import lru_cache


@lru_cache(maxsize=1000)
def nominatim_search(address, format="json", limit=1, **kwargs):
    """..."""
```

`lru_cache` speichert die Ergebnisse jedoch nur während einer Session. Wenn ein Skript wegen einem Timeout oder einer Exception beendet wird, sind die Ergebnisse verloren. Sollen die Daten dauerhafter gespeichert werden, können Tools wie joblib oder python-diskcache verwendet werden.

3.5.2 Modul erstellen

Es ist nicht sehr praktisch, Jupyter jedes Mal zu starten und alle Zellen des *httpx-Notebooks* zu durchlaufen, nur um die Funktionen verwenden zu können. Stattdessen sollten wir unsere Funktionen in einem separaten Modul speichern, wie in nominatim.py:

1. Hierfür habe ich in Jupyter an derselben Stelle, wie diese Notebooks eine neue Textdatei erstellt, ihr den Namen `nominatim.py` gegeben.
2. Anschließend habe ich die Importe, die Methode `nominatim_search` und deren Decorator `lru_cache` hineinkopiert und die Datei gespeichert.
3. Nun können wir zu unserem Notebook zurückkehren und den Code aus dieser Datei importieren und unsere Suchen ausführen:

[1]:
```
from nominatim import nominatim_search
```

[2]:
```
nominatim_search("Alexanderplatz, Berlin, Germany")
```

[2]:
```
[{'place_id': 261767431,
  'licence': 'Data © OpenStreetMap contributors, ODbL 1.0. https://www.openstreetmap.org/
↪copyright',
  'osm_type': 'way',
  'osm_id': 783052052,
  'lat': '52.5219814',
  'lon': '13.413635717448294',
  'class': 'place',
  'type': 'square',
  'place_rank': 25,
  'importance': 0.47149825263735834,
  'addresstype': 'square',
  'name': 'Alexanderplatz',
  'display_name': 'Alexanderplatz, Mitte, Berlin, 10178, Deutschland',
  'boundingbox': ['52.5201457', '52.5238113', '13.4103097', '13.4160801']}]
```

Das Auslagern des Codes von Notebooks in Module erleichtert nicht nur dessen Wiederverwendbarkeit, es macht auch die Notebooks lesbarer.

Damit der Code jedoch funktioniert, muss sich nominatim.py im selben Ordner wie ein Jupyter-Notizbuch befinden. Wenn Sie dieses Modul von einer anderen Stelle aus aufrufen möchten, müsste die Pfadangabe im `import`geändert werden. In diesem Fall sollte besser ein eigenes Paket erstellt werden, wie dies in *Paketierung* beschrieben ist.

3.6 Entfernte Dateisysteme

boto3
: S3

azure-storage-blob
: Azure

pydrive2
: Google Drive

paramiko
: SSH

PyArrow
: HDFS

3.7 Geodaten

Rasterio
: liest und schreibt GeoTIFF und andere Formen von Raster-Datasets

Geospatial Data Abstraction Library (GDAL)
: bietet eine Low-Level-, aber leistungsfähigere API zum Lesen und Schreiben hunderter Datenformate.

satpy
: Einfach zu verwendende API für Satellitenbildformate von Sensoren wie MODIS, Sentinel-2 etc.

sentinelsat
: Suchen und Herunterladen von Copernicus Sentinel-Satellitenbildern über Befehlszeile oder Python.

fiona
: liest und schreibt `*shp`- und `*json`-Daten und viele weitere Formate.

pyproj
: Python-Schnittstelle zu PROJ, einer Bibliothek für kartografische Projektionen und Koordinatentransformationen.

pyModis
: Sammlung von Python-Skripten zum Herunterladen und Mosaizieren von MODIS-Daten.

Arcpy
: wird von Esri ArcGIS verwendet, um geographische Daten zu analysieren, zu konvertieren und zu verwalten sowie Karten automatisiert zu erstellen.

RSGISLib
: oder *The Remote Sensing and GIS Software Library* ist eine Reihe von Fernerkundungswerkzeugen für die Rasterverarbeitung und -analyse.

pgeocode
: wird für die Abfrage von GPS-Koordinaten und Gemeindenamen anhand von Postleitzahlen, Entfernungen zwischen Postleitzahlen sowie allgemeine Entfernungsabfragen verwendet.

3.8 PostgreSQL

3.8.1 Grundfunktionen

ACID-konform
: ACID (**A** tomicity, **C** onsistency, **I** solation, **D** urability) ist eine Reihe von Eigenschaften, die Datenbanktransaktionen erfüllen sollten um auch im Störungsfall weiterhin die Gültigkeit der Daten gewährleisten zu können.

SQL:2011
: Mit temporal_tables wird auch der SQL-Standard ISO/IEC 9075:2011 erfüllt, u.a. durch:

  - Time Period definitions
  - Valid time tables
  - Transaction time tables (system-versioned tables) with time-sliced and sequenced queries

Datentypen
: Folgende Datentypen werden out of the box unterstützt:

  - primitiven Datentypen: Integer, Numeric, String, Boolean
  - Strukturierte Datentypen: Date/Time, Array, Range, UUID
  - Dokumenttypen: JSON/JSONB, XML, Key-value (Hstore)
  - Geometrische Datentypen: Point, Line, Circle, Polygon
  - Anpassungen: Composite, Custom Types

Transactional Data Definition Language (DDL)
: Transactional DDL wird via Write-Ahead Logging realisiert. Dabei sind auch große Änderungen möglich, nicht jedoch *add* und *drop* von Datenbanken und Tabellen:

```
$ psql mydb
mydb=# DROP TABLE IF EXISTS foo;
NOTICE: table "foo" does not exist
DROP TABLE
mydb=# BEGIN;
BEGIN
mydb=# CREATE TABLE foo (bar int);
CREATE TABLE
mydb=# INSERT INTO foo VALUES (1);
INSERT 0 1
mydb=# ROLLBACK;
ROLLBACK
mydb=# SELECT * FROM foo;
ERROR: relation "foo" does not exist
```

Concurrent Index
: PostgreSQL kann Indizes erstellen ohne Schreibzugriffe auf Tabellen sperren zu müssen.

  Siehe auch:

  Building Indexes Concurrently

Erweiterungen
PostgreSQL kann einfach erweitert werden. Das mit dem Quellcode gelieferte contrib/-Verzeichnis enthält verschiedene Erweiterungen, die in Appendix F beschrieben sind. Andere Erweiterungen sind unabhängig entwickelt worden, wie z.B. PostGIS oder Slony-I.

Common Table Expression
WITH Queries (Common Table Expressions) unterteilt komplexe Anfragen in einfachere Anfragen, z.B.:

```
WITH regional_insolation AS (
    SELECT region, SUM(amount) AS total_insolation
    FROM orders
    GROUP BY region
), top_regions AS (
    SELECT region
    FROM regional_insolation
    WHERE total_insolation > (SELECT SUM(total_insolation)/10 FROM regional_
↪insolation)
)
```

Zudem gibt es auch noch einen `RECURSIVE`-Modifier, der die `WITH`-Abfrage auf seine eigene Ausgabe verweist. Im folgenden ein Beispiel zum Summieren der Zahlen von 1 bis 100:

```
WITH RECURSIVE t (n) AS (
    WERTE (1)
  UNION ALL
    SELECT n + 1 FROM t WO <100
)
SELECT sum (n) FROM t;
```

Multi-Version Concurrency Control (MVCC)
Multi-Version Concurrency Control erlaubt, dass zwei oder mehr Sessions gleicheitig auf dieselben Daten zugreifen ohne dabei die Integrität der Daten zu gefährden.

Cross Platform
PostgreSQL läuft auf gängigen CPU-Architekturen wie `x86`, `PowerPC`, `Sparc`, `ARM`, `MIPS` oder `PA-RISC`. Auch die meisten Betriebssysteme werden unterstützt: `Linux`, `Windows`, `FreeBSD`, `OpenBSD`, `NetBSD`, `Mac OS`, `AIX`, `HP/UX` und `Solaris`.

Siehe auch:

explain.depesz.com
Web-App, die PostgreSQLs EXPLAIN- und ANALYZE -Anweisungen visualisiert.

Foreign Data Wrappers (FDW)

2003 wurde SQL erweitert um SQL/MED (*SQL Management of External Data*). PostgreSQL 9.1 unterstützte dies *read-only*, 9.3 dann auch schreibend. Seitdem sind eine Reihe von Foreign Data Wrappers (FDW) für PostgreSQL entwickelt worden.

Im Folgenden nur eine kleine Auswahl der bekanntesten FDW:

Bemerkung: Beachtet bitte, dass die meisten dieser Wrapper nicht offiziell von der PostgreSQL Global Development Group (PGDG) unterstützt werden.

Generische SQL-Wrapper

ODBC
Nativer ODBC FDW für PostgreSQL 9.5

- GitHub

Multicorn
Multicorn erleichtert die Entwicklung von FDWs. So verwendet z.B. SQLAlchemy Multicorn um seine Daten in PostgreSQL zu speichern.

- GitHub
- PGXN
- Docs

VirtDB
Nativer Zugang zu VirtDB (SAP ERP, Oracle RDBMS)

- GitHub

Spezifische SQL-Wrapper

postgres_fdw
Mit postgres_fdw kann auf Daten aus anderen PostgreSQL-Servern zugegriffen werden.

- Git
- PGXN
- Docs

Oracle
FDW für Oracle-Datenbanken

- GitHub
- PGXN
- Docs

MySQL
FDW für MySQL ab PostgrSQL9.3

- GitHub
- PGXN

SQLite
FDW für SQLite3

- GitHub
- PGXN
- Docs

NoSQL-Database-Wrappers

Cassandra
FDW für Cassandra

- GitHub
- rankactive

Neo4j
FWD für Neo4j, die auch eine Cypher-Funktion für PostgreSQL bereitstellt

- GitHub
- Docs

Redis
FDW für Redis

- GitHub

Riak
FDW für Riak

- GitHub

File-Wrappers

CSV
Offizielle Erweiterung für PostgreSQL 9.1

- Git
- Docs

JSON
FDW für JSON-Dateien

- GitHub
- Beispiel

XML
FDW für XML-Dateien

- GitHub
- PGXN

Geo Wrappers

GDAL/OGR
FDW für den GDAL/OGR-Treiber einschließlich Datenbanken wie Oracle und SQLite sowie Dateiformate wie MapInfo, CSV, Excel, OpenOffice, OpenStreetMap PBF und XML.

- GitHub

Geocode/GeoJSON
Eine Sammlung von FDWs für PostGIS

- GitHub

Open Street Map PBF
FDW für Open Street Map PBF

- GitHub

Generische Web-Wrappers

ICAL
FDW für ICAL

- GitHub
- Docs

IMAP
FDW für das *Internet Message Access Protocol (IMAP)*

- Docs

RSS
FDQ für RSS-Feeds

- Docs

Siehe auch:

- PostgreSQL Wiki
- PGXN-Website

Prozedurale Programmiersprachen

Mit PostgreSQL können benutzerdefinierte Funktionen außer in SQL und C auch in anderen Sprachen geschrieben werden.

In der Standarddistribution von PostgreSQL stehen derzeit vier prozedurale Sprachen zur Verfügung:

- PL/pgSQL
- PL/Tcl
- PL/Perl
- PL/Python

Es sind zusätzliche prozedurale Programmiersprachen verfügbar, die jedoch nicht in der Kerndistribution enthalten sind:

- PL/Java
- PL/Lua
- PL/R
- PL/sh
- PL/v8

Siehe auch:

External Procedural Languages

Zusätzlich können andere Sprachen definiert werden, s. Writing A Procedural Language Handler.

DB-API 2.0

Die Python-API für Datenbank-Konnektoren ist einfach zu bedienen und zu verstehen. Die beiden wesentlichen Konzepte sind:

Connection

Connection Objects erlauben die folgenden Methoden:

`connect(parameters...)`
öffnet die Verbindung zur Datenbank

`.close()`
schließt die Verbindung zur Datenbank

`.commit()`
überträgt die ausstehende Transaktion zur Datenbank

`.rollback()`
Diese Methode ist optional da nicht alle Datenbanken das Zurückrollen von Transaktionen erlauben.

`.cursor ()`
Rückgabe eines neuen Corsor-Objekts über die Verbindung

Beispiel:

```
import driver

conn = driver.connect(database='example',
                      host='localhost',
                      port=5432)

try:
    # create the cursor
    # use the cursor
except Exception:
    conn.rollback()
else:
    conn.commit()
    conn.close()
```

Cursor

Cursorobjekte werden zum Verwalten des Kontexts einer `.fetch*()`-Methode verwendet.

Dabei sind Cursor, die in derselben *Connection* erstellt werden, nicht isoliert voneinander.

Es gibt zwei Attribute für Cursor-Objekte:

`.description`
enthält die folgenden sieben Elemente:

1. `name`
2. `type_code`
3. `display_size`
4. `internal_size`
5. `precision`
6. `scale`
7. `null_ok`

Die ersten beiden Elemente (`name` und `type_code`) sind obligatorisch, die anderen fünf sind optional und werden auf `None` gesetzt, wenn keine sinnvollen Werte angegeben werden können.

`.rowcount`
gibt die Anzahl der Zeilen an, die der letzte Aufruf von `.execute*()` mit `SELECT`, `UPDATE` oder `INSERT` ergab.

Beispiel:

```
cursor = conn.cursor()
cursor.execute(
    """
    SELECT column1, column2
    FROM tableA
    """
)
for column1, column2 in cursor.fetchall():
    print(column1, column2)
```

Siehe auch:

PEP 249 – Python Database API Specification v2.0

Psycopg

Pycopg ist ein PostgreSQL-Adapter, der auf der C-Bibliothek für PostgreSQL libpq basiert. Er bietet U.A.:

- DB-API-2.0-Kompatibilität
- Multithreading bei Thread Safety
- Connections pooling um einen Cache von bestehenden Datenbankverbindungen für Anfragen verwenden zu können.
- Asynchronous und Coroutines support
- Adaptation der Python Typen in SQL

Installation

Ihr könnt psycopg2 mit Spack installieren, z.B. mit

```
$ spack env activate python-311
$ spack install py-psycopg2
```

Objektrelationale Abbildung

«Objektrelationale Abbildung (englisch object-relational mapping, ORM) ist eine Technik der Softwareentwicklung, mit der ein in einer objektorientierten Programmiersprache geschriebenes Anwendungsprogramm seine Objekte in einer relationalen Datenbank ablegen kann.»[1]

Im einfachsten Fall werden Klassen auf Tabellen abgebildet, wobei jedes Objekt einer Tabellenzeile entspricht und jedem Attribut eine Tabellenspalte.

Um Vererbungshierarchien abzubilden gibt es im Wesentlichen drei verschiedene Verfahren:

[1] Wikipedia: Objektrelationale Abbildung

Single Table
: Dabei wird eine Tabelle pro Vererbungshierarchie angelegt, wobei alle Attribute der Basisklasse und aller davon abgeleiteten Klassen in einer gemeinsamen Tabelle gespeichert wird.

Joined Table* oder *Class Table
: Dabei wird eine Tabelle je Unterklasse angelegt und für jede davon abgeleitete Unterklasse eine weitere Tabelle.

Table per Class* oder *Concrete Table
: Dabei werden die Attribute der abstrakten Basisklasse in die Tabellen für die konkreten Unterklassen mit aufgenommen. Hierbei ist es jedoch nicht möglich, mit einer Abfrage Instanzen verschiedener Klassen zu ermitteln.

SQLAlchemy

SQLAlchemy ist ein Python-SQL-Toolit und objektrelationaler Mapper.

SQLAlchemy ist bekannt für sein ORM, wobei es verschiedene Muster für das objektrelationale Mapping bereitstellt, wobei Klassen auf verschiedene Weise auf die Datenbank abgebildet werden können. Das Objektmodell und das Datenbankschema sind von Anfang an sauber entkoppelt.

SQLAlchemy unterscheidet sich grundlegend von anderen ORMs, da SQL und Details der Objekt-Relation nicht wegabstrahiert werden: alle Prozesse werden als eine Zusammenstellung einzelner Tools dargestellt.

SQLAlchemy unterstützt neben PostgreSQL auch weitere Dialekte relationaler Datenbanken:

Dialekte	Python-Pakete	import	Dokumentation
postgresql	psycopg2-binary	psycopg2	Installation
mysql	mysqlclient	MySQLdb	README
mssql	pyodbc	pyodbc	Wiki
oracle	cx_oracle	cx_Oracle	cx_Oracle

Datenbankverbindung

```
from sqlalchemy import create_engine


engine = create_engine("postgresql:///example", echo=True)
```

Datenmodell

```
from sqlalchemy import Column, ForeignKey, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import relationship


Base = declarative_base()


class Address(Base):
    __tablename__ = "address"
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    id = Column(Integer, primary_key=True)
    street = Column(String)
    zipcode = Column(String)
    country = Column(String, nullable=False)


class Contact(Base):
    __tablename__ = "contact"

    id = Column(Integer, primary_key=True)


firstname = Column(String, nullable=False)
lastname = Column(String, nullable=False)
email = Column(String, nullable=False)
address_id = Column(Integer, ForeignKey(Address.id), nullable=False)
address = relationship("Address")
```

Tabellen erstellen

```
Base.metadata.create_all(engine)
```

Create Session

```
session = Session(engine)
address = Address(street="Birnbaumweg 10", zipcode="79115", country="Germany")

contact = Contact(
    firstname="Veit",
    lastname="Schiele",
    email="veit@cusy.io",
    address=address
)

session.add(contact)
session.commit()
```

Read

```
contact = session.query(Contact).filter_by(email="veit@cusy.io").first()
print(contact.firstname)

contacts = session.query(Contact).all()

for contact in contacts:
    print(contact.firstname)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
contacts = session.query(Contact).filter_by(email="veit@cusy.io").all()

for contact in contacts:
    print(contact.firstname)
```

Update

```
contact = session.query(Contact).filter_by(email="veit@cusy.io").first()
contact.email = "info@veit-schiele.de"
session.add(contact)
session.commit()
```

Delete

```
contact = (
    session.query(Contact).filter_by(email="info@veit-schiele.de").first()
)
session.delete(contact)
session.commit()
```

Erweiterungen

SQLAlchemy-Continuum
: Versionierungs- und Revisionserweiterung für SQLAlchemy

SQLAlchemy-Utc
: SQLAlchemy-Typ zum Speichern von *datetime.datetime*-Werten

SQLAlchemy-Utils
: Verschiedene Utility-Funktionen, neue Datentypen und Hilfsprogramme für SQLAlchemy

DEPOT
: Framework zur einfachen Speicherung und Bereitstellung von Dateien in Webanwendungen

SQLAlchemy-ImageAttach
: SQLAlchemy-Erweiterung zum Anhängen von Bildern an Entitätsobjekte

SQLAlchemy-Searchable
: Im Volltext durchsuchbare Modelle für SQLAlchemy

Siehe auch:

- Awesome SQLAlchemy

Alembic

Alembic basiert auf SQLAlchemy und dient als Datenbankmigrationswerkzeug mit den folgenden Funktionen:

- `ALTER`-Anweisungen an eine Datenbank ausgeben um die Strukturen von Tabellen und anderen Konstrukten zu ändern
- System zum Erstellen von Migrationsskripten. Optional kann auch die Reihenfolge der Schritte für das Downgrade angegeben werden.
- Die Skripte werden in einer bestimmten Reihenfolge ausgeführt.

Siehe auch:

Auto Generating Migrations

Migrationsumgebung erstellen

Das Migration Environment ist ein Verzeichnis, das für eine bestimmte Anwendung spezifisch ist. Sie wird mit dem `ini`-Befehl von Alembic erstellt und anschließend zusammen mit dem Quellcode der Anwendung verwaltet.

```
$ cd myrproject
$ alembic init alembic
Creating directory /path/to/myproject/alembic...done
Creating directory /path/to/myproject/alembic/versions...done
Generating /path/to/myproject/alembic.ini...done
Generating /path/to/myproject/alembic/env.py...done
Generating /path/to/myproject/alembic/README...done
Generating /path/to/myproject/alembic/script.py.mako...done
Please edit configuration/connection/logging settings in
'/path/to/myproject/alembic.ini' before proceeding.
```

Die Struktur eines solchen Migrationsumgebung kann später dann z.B. so aussehen:

```
myproject/
└── alembic
    ├── alembic.ini
    ├── env.py
    ├── README
    ├── script.py.mako
    └── versions
        ├── 2b1ae634e5cd_add_order_id.py
        ├── 3512b954651e_add_account.py
        └── 3adcc9a56557_rename_username_field.py
```

Vorlagen

Alembic erhält eine Reihe von Vorlagen, die mit `list` angezeigt werden können:

```
$ alembic list_templates
Available templates:

generic - Generic single-database configuration.
multidb - Rudimentary multi-database configuration.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
pylons - Configuration that reads from a Pylons project environment.

Templates are used via the 'init' command, e.g.:

  alembic init --template pylons ./scripts
```

ini-Datei konfigurieren

Die mit der `generic`-Vorlage erstellte Datei sieht folgendermaßen aus:

```
# A generic, single database configuration.

[alembic]
# path to migration scripts
script_location = alembic

# template used to generate migration files
# file_template = %%(rev)s_%%(slug)s

# timezone to use when rendering the date
# within the migration file as well as the filename.
# string value is passed to dateutil.tz.gettz()
# leave blank for localtime
# timezone =

# max length of characters to apply to the
# "slug" field
#truncate_slug_length = 40

# set to 'true' to run the environment during
# the 'revision' command, regardless of autogenerate
# revision_environment = false

# set to 'true' to allow .pyc and .pyo files without
# a source .py file to be detected as revisions in the
# versions/ directory
# sourceless = false

# version location specification; this defaults
# to alembic/versions.  When using multiple version
# directories, initial revisions must be specified with --version-path
# version_locations = %(here)s/bar %(here)s/bat alembic/versions

# the output encoding used when revision files
# are written from script.py.mako
# output_encoding = utf-8

sqlalchemy.url = driver://user:pass@localhost/dbname

# Logging configuration
[loggers]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
keys = root,sqlalchemy,alembic

[handlers]
keys = console

[formatters]
keys = generic

[logger_root]
level = WARN
handlers = console
qualname =

[logger_sqlalchemy]
level = WARN
handlers =
qualname = sqlalchemy.engine

[logger_alembic]
level = INFO
handlers =
qualname = alembic

[handler_console]
class = StreamHandler
args = (sys.stderr,)
level = NOTSET
formatter = generic

[formatter_generic]
format = %(levelname)-5.5s [%(name)s] %(message)s
datefmt = %H:%M:%S
```

%(here)s
: Ersetzungsvariable zum Erstellen absoluter Pfade

file_template
: Dies ist das Namensschema, das zum Generieren neuer Migrationsdateien verwendet wird. Zu den verfügbaren Variablen gehören:

  %%(rev)s
  : Revision-ID

  %%(slug)s
  : Verkürzte Revisionsnachricht

  %%(year)d, %%(month).2d, %%(day).2d, %%(hour).2d, %%(minute).2d, %%(second).2d
  : Erstellungszeitpunkt

Erstellen eines Migrationsskripts

Eine neue Revision kann erstellt werden mit:

```
$ alembic revision -m "create account table"
Generating /path/to/yourproject/alembic/versions/1975ea83b712_create_account_table.py...
↪done
```

Die Datei `1975ea83b712_create_account_table.py` sieht dann folgendermaßen aus:

```
"""create account table

Revision ID: 1975ea83b712
Revises:
Create Date: 2018-12-08 11:40:27.089406

"""

# revision identifiers, used by Alembic.
revision = "1975ea83b712"
down_revision = None
branch_labels = None

import sqlalchemy as sa

from alembic import op


def upgrade():
    pass


def downgrade():
    pass
```

down_revision
: Variable, die Alembic mitteilt, in welcher Reihenfolge die Migrationen ausgeführt werden sollen, z.B.:

```
# revision identifiers, used by Alembic.
revision = "ae1027a6acf"
down_revision = "1975ea83b712"
```

upgrade, downgrade
: z.B.:

```
def upgrade():
    op.create_table(
        "account",
        sa.Column("id", sa.Integer, primary_key=True),
        sa.Column("name", sa.String(50), nullable=False),
        sa.Column("description", sa.Unicode(200)),
    )
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
def downgrade():
    op.drop_table("account")
```

`create_table()` und `drop_table()` sind Alembic-Direktiven. Einen Überblick über alle Alembic-Direktiven erhaltet ihr in der Operation Reference.

Ausführen von Migration

Erste Migration:

```
$ alembic upgrade head
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
INFO  [alembic.context] Running upgrade None -> 1975ea83b712
```

Wir können auch direkt auf Revisionsnummern verweisen:

```
$ alembic upgrade ae1
```

Auch relative Migrationen können angestoßen werden:

```
$ alembic upgrade +2
```

oder:

```
$ alembic downgrade -1
```

oder:

```
$ alembic upgrade ae10+2
```

Informationen anzeigen

Aktuelle Version

```
$ alembic current
INFO  [alembic.context] Context class PostgresqlContext.
INFO  [alembic.context] Will assume transactional DDL.
Current revision for postgresql://scott:XXXXX@localhost/test: 1975ea83b712 ->␣
↪ae1027a6acf (head), Add a column
```

Historie

```
$ alembic history --verbose

Rev: ae1027a6acf (head)
Parent: 1975ea83b712
Path: /path/to/yourproject/alembic/versions/ae1027a6acf_add_a_column.py

    add a column

    Revision ID: ae1027a6acf
    Revises: 1975ea83b712
    Create Date: 2014-11-20 13:02:54.849677

Rev: 1975ea83b712
Parent: <base>
Path: /path/to/yourproject/alembic/versions/1975ea83b712_add_account_table.py

    create account table

    Revision ID: 1975ea83b712
    Revises:
    Create Date: 2014-11-20 13:02:46.257104
```

Die Historie kann auch spezifischer angezeigt werden:

```
$ alembic history -r1975ea:ae1027
```

oder:

```
$ alembic history -r-3:current
```

oder:

```
$ alembic history -r1975ea:
```

ipython-sql

ipython-sql führt die `%sql` oder `%%sql`-Magics für iPython und Jupyter Notebooks ein.

Installation

Ihr könnt ipython-sql einfach in eurem Jupyter-Kernel installieren mit:

```
$ pipenv install ipython-sql
```

Erste Schritte

1. Zunächst wird ipython-sql in eurem Notebook aktiviert mit

```
In [1]: %load_ext sql
```

2. Für die Verbindung zur Datenbank wird die SQLAlchemy URL verwendet:

```
In [2]: %sql postgresql://
```

3. Anschließend könnt ihr eine Tabelle erstellen, z.B.:

```
In [3]: %%sql postgresql://
   ....: CREATE TABLE accounts (login, name, email)
   ....: INSERT INTO accounts VALUES ('veit', 'Veit Schiele', veit@example.org);
```

4. Die Inhalte der Tabelle `accounts` könnt ihr abfragen mit

```
In [4]: result = %sql select * from accounts
```

Konfiguration

Abfrageergebnisse werden als Liste geladen sodass sehr große Datenmengen den Arbeitsspeicher belegen können. Üblicherweise gibt es keine automatische Begrenzung, mit `Autolimit` lässt sich jedoch die Ergebnismenge limitieren.

Bemerkung: `displaylimit` begrenzt nur die Menge der angezeigten Ergebnisse, nicht jedoch den Speicherbedarf.

Mit `%config SqlMagic` könnt ihr Euch die aktuelle Konfiguration ausgeben lassen:

```
In [5]: %config SqlMagic
SqlMagic options
--------------
SqlMagic.autocommit=<Bool>
    Current: True
    Set autocommit mode
SqlMagic.autolimit=<Int>
    Current: 0
    Automatically limit the size of the returned result sets
SqlMagic.autopandas=<Bool>
    Current: False
    Return pandas DataFrames instead of regular result sets
...
```

Bemerkung: Wenn `autopandas` auf `True` gesetzt wurde, wird `displaylimit` nicht angewendet. In diesem Fall kann die `max_rows`-Option von pandas verwendet werden wie in der pandas-Dokumentation beschrieben.

---

pandas

Wenn pandas installiert ist, kann die `DataFrame`-Methode verwendet werden:

```
In [6]: result = %sql SELECT * FROM accounts

In [7]: dataframe = result.DataFrame()

In [8]: %sql --persist dataframe

In [9]: %sql SELECT * FROM dataframe;
```

--persist
: Argument mit dem Namen eines `DataFrame`-Objekts, erstellt aus diesem einen Tabellennahmen in der Datenbank.

--append
: Argument um in einer vorhandenen Tabelle Zeilen mit diesem namen hinzuzufügen.

PostgreSQL-Funktionen

Meta-Befehle von `psql` lassen sich auch in ipython-sql verwenden:

-l, --connections
: listet alle aktiven Verbindungen auf

-x, --close *SESSION-NAME*
: schließt benannte Verbindung

-c, --creator *CREATOR-FUNCTION*
: gibt die Creator-Funktion für eine neue Verbindung an

-s, --section *SECTION-NAME*
: gibt Abschnitt von `dsn_file` an, der in einer Verbindung verwendet werden soll

-p, --persist
: erstellt aus einem benannten DataFrame eine Tabelle in der Datenbank

--append
: ähnlich wie `--persist`, die Inhalte werden jedoch an die Tabelle angehängt

-a, --connection_arguments "*{connection arguments}*"
: gibt ein Dict von Verbindungsargumenten an, die an den SQL-Treiber übergeben werden

-f, --file *PATH*
: führt SQL aus der Datei unter diesem Pfad aus

Siehe auch:

- pgspecial

Warnung: Da ipython-sql `---`-Optionen wie z.B. `--persist` verarbeitet, und gleichzeitig `--` als SQL-Kommentar akzeptiert, muss der Parser einige Annahmen treffen: so wird z.B. `--persist is great` in der ersten Zeile als Argument und nicht als Kommentar verarbeitet.

PostGIS

PostGIS ist eine Erweiterung für PostgreSQL, die geografische Objekte und Funktionen umfasst. Die Erweiterung implementiert u.a. die Simple-Feature-Access-Spezifikation des Open Geospatial Consortium. Obwohl PostgreSQL bereits Geometrietypen unterstützt, sind diese jedoch für geographische Aufgaben ungenügend. Daher erstellt PostGIS eigene Datentypen, die besser für geographische Aufgaben geeignet sind. Im Einzelnen werden folgende Geometrietypen unterstützt:

- OpenGIS mit Well-Known Text und Well-Known Binary
- Extended Well-Known Text und Extended Well-Known Binary zusätzlich mit Höheninformationen und/oder Messwerten
- SQL/MM mit Circularstring, Compoundcurve, Curvepolygon, Multicurve und Multisurface

GEOS hingegen enthält die zahlreichen räumlichen Funktionen und Operatoren für geographische Daten.

pgRouting schließlich enthält Routing-Funktionen auf Basis von PostGIS.

Im OpenStreetMap-Projekt wird PostGIS zum Rendern von Karten mit Mapnik verwendet.

PostGIS installieren

Für Ubuntu 22.04 könnt ihr PostGIS einfach installieren mit:

code-block:: console

    $ sudo apt install postgis

Anschließend könnt ihr PostGIS aktivieren.

1. Wechseln zum PostgreSQL-User:

```
$ sudo -i -u postgres
```

2. Testuser und Datenbank erstellen:

```
$ createuser postgis
$ createdb postgis_db -O postgis
```

3. Verbindung zur Datenbank herstellen:

```
$ psql -d postgis_db
psql (14.5 (Ubuntu 14.5-0ubuntu0.22.04.1))
Type "help" for help.
```

4. Aktivieren der PostGIS-Erweiterung in der Datenbank:

```
ppostgis_db = # CREATE EXTENSION postgis;
CREATE EXTENSION
```

5. Überprüfen, ob PostGIS funktioniert:

```
    postgis_db=# SELECT PostGIS_version();
            postgis_version
---------------------------------------
 3.2 USE_GEOS=1 USE_PROJ=1 USE_STATS=1
(1 row)
```

Siehe auch:

- PostGIS Installation

Optimieren von PostgreSQL für GIS-Datenbankobjekte

In der Standardinstallation ist PostgreSQL sehr zurückhaltend konfiguriert um auf möglichst vielen Systemen lauffähig zu sein. GIS-Datenbankobjekte sind jedoch im Vergleich zu Textdaten groß. Daher sollte PostgreSQL so konfiguriert werden, dass sie mit diesen Objekten besser funktioniert. Hierfür konfigurieren wir die Datei `/etc/postgresql/14/main/postgresql.conf` folgendermaßen:

1. `shared_buffer` sollte auf ca. 75% des gesamten Arbeitsspeichers geändert werden, jedoch `128kB` nie unterschreiten:

```
shared_buffers = 768MB
```

2. `work_mem` sollte auf mindestens 16MB erhöht werden:

```
work_mem = 16MB
```

3. `maintenance_work_mem` sollte auf 128MB erhöht werden:

```
maintenance_work_mem = 128MB
```

4. Schließlich sollte noch `random_page_cost` auf `2.0` gesetzt werden.

```
random_page_cost = 2.0
```

Damit die Änderungen übernommen werden, sollte PostgreSQL neu gestartet werden:

```
$ sudo service postgresql restart
```

Laden von Geodaten

Nun Laden wir einige Geodaten in unsere Datenbank, damit wir uns mit den Tools und Prozessen zum Abrufen dieser Daten vertraut machen können.

Natural Earth bietet eine großartige Quelle für Basisdaten für die ganze Welt in verschiedenen Maßstäben. Und sas Beste ist, dass diese Daten gemeinfrei sind:

1. Herunterladen der Daten

```
$ mkdir nedata
$ cd !$
cd nedata
$ wget https://www.naturalearthdata.com/http//www.naturalearthdata.com/download/110m/
↪cultural/ne_110m_admin_0_countries.zip
```

1. Entpacken der Datei

```
$ sudo apt install unzip
$ unzip ne_110m_admin_0_countries.zip
Archive:  ne_110m_admin_0_countries.zip
  inflating: ne_110m_admin_0_countries.README.html
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
extracting: ne_110m_admin_0_countries.VERSION.txt
extracting: ne_110m_admin_0_countries.cpg
 inflating: ne_110m_admin_0_countries.dbf
 inflating: ne_110m_admin_0_countries.prj
 inflating: ne_110m_admin_0_countries.shp
 inflating: ne_110m_admin_0_countries.shx
```

1. Laden in unsere `postgis_db`-Datenbank

   Die Dateien `.dbf`, `.prj`, `.shp` und `.shp` bilden ein sog. ShapeFile, ein beliebtes Geodaten-Datenformat, das von der GIS-Software verwendet wird. Um dies in unsere Datenbank zu laden, benötigen wir zusätzlich GDAL, die *Geospatial Data Abstraction Library*. Wenn wir GDAL installieren, erhalten wir auch OGR, *OpenGIS Simple Features Reference Implementation*, eine Vektordaten-Übersetzungsbibliothek, mit der wir das Shapefile in Daten übersetzen können.

   1. GDAL kann un einfach mit dem Paketmanager installiert werden:

      ```
      $ sudo apt install gdal-bin
      ```

   2. Anschließend wechseln wir in den `postgresql`-User:

      ```
      $ sudo -i -u postgres
      ```

   3. Nun konvertieren wir das Shapefile mit `ogr2ogr` und importieren es in unsere Datenbank:

      ```
      $ ogr2ogr -f PostgreSQL PG:dbname=postgis_db -progress \
          -nlt PROMOTE_TO_MULTI \
          /srv/jupyter/nedata/ne_110m_admin_0_countries.shp
      0...10...20...30...40...50...60...70...80...90...100 - done.
      ```

      -f PostgreSQL
      gibt an, dass das Ziel eine PostgreSQL-Datenbank ist

      PG:dbname=postgis_db
      gibt den PostgreSQL-Datenbanknamen an. Neben dem Namen können so auch weitere Optionen angegeben werden, allgemein:

      ```
      PG:"dbname='db_ename' host='addr' port='5432' user='x' password='y'"
      ```

      -progress
      gibt einen Fortschrittsbalken aus

      -nlt PROMOTE_TO_MULTI
      gibt an, dass alle Objekttypen als Multipolygone in die Datenbank geladen werden sollen

      /home/veit/nedata/ne_110m_admin_0_countries.shp
      gibt den Pfad zur Eingabedatei an

      Siehe auch:

      - ogr2ogr

   4. Überprüfen des Imports mit `ogrinfo`

      ```
      $ ogrinfo -so PG:dbname=postgis_db ne_110m_admin_0_countries
      Output
      INFO: Open of `PG:dbname=postgis_db'
      ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
      using driver `PostgreSQL' successful.

Layer name: ne_110m_admin_0_countries
Geometry: Multi Polygon
Feature Count: 177
...
```

5. Alternativ können wir uns auch einzelne Tabellen auflisten lassen:

```
$ psql -d postgis_db
postgis_db=# \dt
                  List of relations
 Schema |           Name             | Type  |  Owner
--------+----------------------------+-------+----------
 public | ne_110m_admin_0_countries | table | postgres
 public | spatial_ref_sys           | table | postgres
(2 rows)
```

6. Schließlich können wir uns bei der Datenbank abmelden mit

```
psql> \q
```

Siehe auch:

- PostGIS Reference

Datenbank-Sicherheit

Datenbank-Berechtigungen

Das PostgreSQL-Login per Superuser `postgres` sollte immer nur über Unix-Domain-Sockets und über `localhost` erlaubt sein. Der Zugang mit Peer-Authentifizierung in der pg_hba.conf-Datei kann hingegen gewährt werden:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
local   all             postgres                                peer
host    all             all             10.23.42.1/24           scram-sha-256
```

Die Datenbank sollte vom Datenbankadministrator angelegt und anschließend so konfiguriert werden, dass sich nicht jeder (`PUBLIC`) damit verbinden kann:

```
CREATE DATABASE myapp;
REVOKE ALL ON myapp FROM PUBLIC;
```

Damit kann sich nur noch der Superuser mit der Datenbank `myapp` verbinden.

Passwörter speichern

Passwörter sollten niemals im Klartext, also z.B. auch nicht in einer `.env`-Datei gespeichert werden. Beim Speichern und Übermitteln von Passwörtern sollte dies immer mit Salts versehen sein. Für PostgreSQL gibt es hierfür die Erweiterung pgcrypto, die einfach aktiviert werden kann mit

```
CREATE EXTENSION pgcrypto;
```

Daher sollten bereits beim Anlegen sichere Passwörter vergeben werden, die anschließend z.B. in Vault o.ä. gespeichert werden:

```
CREATE ROLE myapp_users;
CREATE ROLE myapp_reader IN ROLE myapp_users LOGIN PASSWORD '...';
CREATE ROLE myapp_writer IN ROLE myapp_users LOGIN PASSWORD '...';
```

Anschließend erhalten dann User mit der Rolle `myapp_users` zunächst `CONNECT`-Rechte und dann `myapp_reader` Leserechte und `myapp_writer` Schreibrechte:

```
GRANT CONNECT ON DATABASE to myapp_users;
GRANT SELECT ON diagnosis_key TO myapp_reader;
GRANT INSERT ON diagnosis_key TO myapp_writer;
```

Der User `myapp_reader` kann damit jedoch alle Daten auf einmal lesen. Auch dies ist ein Angriffspunkt, der besser durch eine Funktion beschnitten wird:

```
CREATE OR REPLACE FUNCTION get_key_data(in_id UUID)
    RETURNS JSONB
    AS 'SELECT key_data FROM diagnosis_key WHERE id = in_id;'
    LANGUAGE sql SECURITY DEFINER SET search_path = :schema, pg_temp;
```

Anschließend wird die Funktion `myapp_owner` zugewiesen, `myapp_reader` und `myapp_writer` die Berechtigungen entzogen und schließlich die Ausführung der Funktion `myapp_reader` erlaubt:

```
ALTER FUNCTION get_key_data(UUID) OWNER TO myapp_owner;
REVOKE ALL ON FUNCTION get_key_dataUUID) FROM PUBLIC;
GRANT EXECUTE ON FUNCTION get_key_data(UUID) TO myapp_reader;
```

Damit kann `myapp_reader` also nur noch einen einzelnen Datensatz lesen.

id

Die `id` sollte nicht als `serial`, `bigserial` o.ä. realisiert werden. Hochzählende Zahlen könnten von Angreifern leicht erraten werden. Daher ist der UUIDv4-Datentyp deutlich besser geeignet. In PostgreSQL könnt ihr UUIDv4 generieren mit der uuid-ossp-Erweiterung oder für PostgreSQL9.4 auch der pgcrypto-Erweiterung:

```
CREATE EXTENSION "uuid-ossp";
CREATE TABLE diagnosis_key (
  id uuid primary key default uuid_generate_v4() NOT NULL,
  ...
);
```

oder

```
CREATE EXTENSION "pgcrypto";
CREATE TABLE diagnosis_key (
  id uuid primary key default gen_random_uuid() NOT NULL,
  ...
);
```

Zeitstempel

Gelegentlich werden Datum und Zeit als `bigint`, also als Zahl, gespeichert, und dies obwohl es auch einen `TIMESTAMP`-Datentyp gibt. Dies hätte den Vorteil, dass dann auch einfach mit ihnen gerechnet werden kann, also z.B.:

```
SELECT age(submission_timestamp);
SELECT submission_timestamp - '1 day'::interval;
```

Außerdem könnten die Daten nach einer bestimmten Zeit gelöscht werden, z.B. nach dreißig Tagen mit:

```
DELETE FROM diagnosis_key WHERE age(submission_timestamp) > 30;
```

Das Löschen kann noch beschleunigt werden, wenn für jeden Tag mit der PostgreSQL-Erweiterung pg_partman eine eigene Partition erstellt wird.

Siehe auch:

- Veil2 – Relational Security for Postgres
- PostgreSQL Secure Monitoring (Posemo)

PostgreSQL-Performance

Ihr solltet nicht mit *MVCC – Multiversion Concurrency Control* beginnen, wenn ihr eure PostgreSQL-Datenbank optimieren wollt: viele Verbesserungen können sehr viel einfacher vorgenommen werden da weder Transaktionslogs noch große Linux Kernel Page Sizes verantwortlich sein dürften. Üblicherweise beginnen wir mit zwei Metriken, die sehr gut die Performance eurer Datenbanken anzeigen können:

Cache- und Index-Trefferquote

Cache-Trefferquote (engl.: *cache hit ratio*)
: Prozentsatz der Zeit, in der die Daten aus dem Arbeitsspeicher statt aus dem Festplattenspeicher bereitgestellt werden können. Für eine Web-Anweundung mit vielen kleinen Anfragen empfehle ich ca. 99%.

```
SELECT
  'index hit rate' AS name,
  (sum(idx_blks_hit)) / nullif(sum(idx_blks_hit + idx_blks_read),0) AS ratio
FROM pg_statio_user_indexes
UNION ALL
SELECT
 'table hit rate' AS name,
  sum(heap_blks_hit) / nullif(sum(heap_blks_hit) + sum(heap_blks_read),0) AS ratio
FROM pg_statio_user_tables;
```

Falls die Cache-Trefferquote zu gering sein sollte, könnt ihr einfach den Arbeitsspeicher erhöhen.

Index-Trefferquote (engl.: *index hit ratio*)
Häufigkeit der Verwendung der Indizes.

```
SELECT relname,
  CASE idx_scan
    WHEN 0 THEN 'Insufficient data'
    ELSE (100 * idx_scan / (seq_scan + idx_scan))::text
  END percent_of_times_index_used,
  n_live_tup rows_in_table
FROM
  pg_stat_user_tables
ORDER BY
  n_live_tup DESC;
      relname        | percent_of_times_index_used | rows_in_table
---------------------+-----------------------------+---------------
account               | 11                          |          5409
activity              | 69                          |         58276
application           | 93                          |          5345
...
```

Üblicherweise sollten bei uns nicht mehr als 10.000 Datensätze in einer Tabelle und der Prozentsatz des verwendeten Index größer als 90% sein.

In unserem Beispiel sehen wir, dass der `account`-Tabelle relevante Indizes fehlen, da nur in 11% der Anfragen ein Index genutzt wird. Auch der `activity`-Tabelle fehlen einige passende Indizes, sie hat jedoch auch sehr viele Datensätze, sodass es sinnvoll sein dürfte, sie in mehrere Tabellen aufzuteilen.

Nicht-verwendete Indizes bereinigen

Nicht-verwendete Indizes führen beim Schreiben der Datensätze zu einem langsameren Durchsatz ohne dass dadurch Abfragen schneller würden.

```
SELECT
  schemaname || '.' || relname AS table,
  indexrelname AS index,
  pg_size_pretty(pg_relation_size(i.indexrelid)) AS index_size,
  idx_scan as index_scans
FROM pg_stat_user_indexes ui
JOIN pg_index i ON ui.indexrelid = i.indexrelid
WHERE NOT indisunique AND idx_scan < 50 AND pg_relation_size(relid) > 5 * 8192
ORDER BY pg_relation_size(i.indexrelid) / nullif(idx_scan, 0) DESC NULLS FIRST,
pg_relation_size(i.indexrelid) DESC;
```

Indizes ohne Verwendung können einfach entfernt werden. Schwieriger wird die Entscheidung hingegen bei Indizes, die nur sehr selten verwendet werden: hier muss eine Abwägung zwischen der Schreib- und Abfragegeschwindigkeit getroffen werden.

Nicht-verwendete Daten bereinigen

Auch wenn PostgreSQL sehr vielfältige Daten aufnehmen kann, ist das jedoch nicht immer sinnvoll. Tabellen wie `messages`, `logs` und `events` haben eine gute Chance, den Großteil des Speichers zu beanspruchen ohne dass die Datenbankanwendung unmittelbar davon profitiert: wenn diese Daten vielmehr der Fehleranalyse dienen, sollten sie außerhalb der Datenbank gespeichert und regelmäßig rotiert werden.

Abfrageleistung mit `pg_stat_statements` analysieren

pg_stat_statements zeichnet Abfragen auf und führt eine Reihe von Statistiken dazu. So lassen wir uns in regelmäßigen Abständen anzeigen, welche Abfragen im Durchschnitt am langsamsten sind und welche das System am stärksten belasten:

```
SELECT
  (total_time / 1000 / 60) as total_minutes,
  (total_time/calls) as average_time,
  query
FROM pg_stat_statements
ORDER BY 1 DESC
LIMIT 50;
total_time         |     avg_time      |                              query
-------------------+-------------------+-----------------------------------------------------
↪----------
 295.761165833319 | 10.1374053278061  | SELECT id FROM account WHERE email LIKE ?
 219.138564283326 | 80.24530822355305 | SELECT * FROM account WHERE user_id = ? AND␣
↪current = True
...
```

Üblich sollten Antwortzeiten von ~1ms und in wenigen Fällen ~4–5ms sein. Um mit der Performance-Optimierung zu beginnen, wägen wir meist zwischen der Gesamtzeit und der Durchschnittszeit ab, sodass wir in obigem Beispiel vermutlich mit der zweiten Zeile beginnen würden da wir hier die größeren Einsparmöglichkeiten sehen. Um eine genauere Vorstellung von der Abfrage zu erhalten, analysieren wir sie genauer mit:

```
EXPLAIN ANALYZE
SELECT *
FROM account
WHERE user_id = 123
  AND current = True
                                                                                  QUERY PLAN
-------------------------------------------------------------------------------------------------------------
↪-----------------------------------------------------------------
 Aggregate  (cost=4690.88..4690.88 rows=1 width=0) (actual time=519.288..519.289 rows=1␣
↪loops=1)
   ->  Nested Loop  (cost=0.00..4690.66 rows=433 width=0) (actual time=15.302..519.076␣
↪rows=213 loops=1)
         ->  Index Scan using idx_account_userid on account  (cost=0.00..232.52 rows=23␣
↪width=4) (actual time=10.143..62.822 rows=1 loops=8)
               Index Cond: (user_id = 123)
               Filter: current
               Rows Removed by Filter: 14
 Total runtime: 219.428 ms
(1 rows)
```

Wir sehen also, dass zwar ein Index verwendet wird, jedoch werden 15 verschiedene Zeilen daraus abgerufen von denen dann 14 wieder verworfen werden. Um dies zu optimieren, würden wir einen bedingten oder zusammengesetzten Index erstellen. Im ersten Fall müsste `current = true` erfüllt sein, im zweiten Fall würde ein Composite-Index mit beiden Werten erstellt werden. Ein bedingter Index ist in der Regel sinnvoller bei einem kleinen Satz von Werten, während der Composite-Index bei größeren Sätzen von Werten vorteilhafter ist. In unserem Beispiel dürfte klar ein bedingter Index sinnvoller sein. Diesen können wir erstellen mit:

```
CREATE INDEX CONCURRENTLY idx_account_userid_current ON account(user_id) WHERE current =␣
↪True;
```

Nun müsste sich auch der Query-Plan verbessern:

```
EXPLAIN ANALYZE
SELECT *
FROM account
WHERE user_id = 123
  AND current = True

                                                                                    QUERY PLAN
-------------------------------------------------------------------------------------------------------------
↪-----------------------------------------------------
 Aggregate  (cost=4690.88..4690.88 rows=1 width=0) (actual time=519.288..519.289 rows=1␣
↪loops=1)
     ->  Index Scan using idx_account_userid_current on account  (cost=0.00..232.52␣
↪rows=23 width=4) (actual time=10.143..62.822 rows=1 loops=8)
           Index Cond: ((user_id = 123) AND (current = True))
 Total runtime: .728 ms
(1 rows)
```

pgMonitor

pgMonitor ist eine Umgebung um den Zustand und die Leistung eines PostgreSQL-Cluster zu visualisieren. Es kombiniert eine Reihe von Werkzeugen, um die Erfassung wichtiger Metriken zu erleichtern, darunter:

- Anzahl der Verbindungen
- Datenbankgröße
- Replikationsverzögerung
- Transaktionsumlauf
- Zusätzlicher Speicherplatz, der von euren Tabellen und Indizes belegt wird
- CPU, Speicher, I/O und Betriebszeit

Es kombiniert mehrere Open-Source-Softwarepakete, um eine robuste PostgreSQL-Überwachungsumgebung zu schaffen, einschließlich:

PostgreSQL Exporter
: Datenexport zu Prometheus, der das Sammeln von Metriken von jedem PostgreSQL-Server 9.1 unterstützt.

Prometheus
: sammelt Metriken und ist in hohem Maße anpassbar.

Grafana
: visualisiert Daten in vielen verschiedenen Arten von Diagrammen und Graphen.

Siehe auch:

- pgexporter

Installation und Konfiguration

Installations- und Konfigurationsanleitungen für die verschiedenen Pakete werden bereitgestellt:

1. PostgreSQL Exporter
2. Prometheus
3. Grafana

pganalyze

pganalyze analysiert die Abfragepläne (Query Plans) von PostgreSQL. Aktuell sammelt er Informationen über

- Schema mit Tabellen (Spalten, Constraints, Trigger-Definitionen) und Indizees
- Statistiken zu Tabellen Indizees, Datenbanken und Anfragen (Queries)
- Betriebssystem (OS, RAM, Storage)

  Siehe auch:

  - GitHub
  - Dokumentation

Installation

1. Erstellen eines Monitoring-User für pganalyze:

```
CREATE USER pganalyze WITH PASSWORD '...' CONNECTION LIMIT 5;
GRANT pg_monitor TO pganalyze;
CREATE SCHEMA pganalyze;
GRANT USAGE ON SCHEMA pganalyze TO pganalyze;
REVOKE ALL ON SCHEMA public FROM pganalyze;
CREATE OR REPLACE FUNCTION pganalyze.get_stat_replication() RETURNS SETOF pg_stat_
↪replication AS
$$ /* pganalyze-collector */ SELECT * FROM pg_catalog.pg_stat_replication;
$$ LANGUAGE sql VOLATILE SECURITY DEFINER;
```

2. Überprüfen der Verbindung:

```
PGPASSWORD=...  psql -h localhost -d mydb -U pganalyze
```

3. Aktivieren der `pg_stat_statements`:

```
ALTER SYSTEM SET shared_preload_libraries = 'pg_stat_statements';
```

4. Neustart des PostgreSQL-Daemon:

```
$ sudo service postgresql restart
```

5. Überprüfen von `pg_stat_statements`:

```
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;
SELECT calls, query FROM pg_stat_statements LIMIT 1;
 calls | query
-------+-------
     8 | SELECT * FROM t WHERE field = ?
(1 row)
```

6. Installieren des *Collector*:

```
$ curl -L https://packages.pganalyze.com/pganalyze_signing_key.asc | sudo apt-key␣
↪add -
$ echo "deb [arch=amd64] https://packages.pganalyze.com/ubuntu/bionic/ stable main"␣
↪| sudo tee /etc/apt/sources.list.d/pganalyze_collector.list
$ sudo apt-get update
$ sudo apt-get install pganalyze-collector
```

7. Erstellen des API-Schlüssel

   Für den nächsten Schritt benötigt ihr den pganalyze `api_key`. Diesen könnt ihr erstellen auf unter der Site https://app.pganalyze.com/

8. Konfigurieren des *Collector*:

```
[pganalyze]
api_key: ...

[server]
db_host: 127.0.0.1
db_port: 5432
db_name: postgres, *
db_username: pganalyze
db_password: ...
```

9. Testen der *Collector*-Konfiguration:

```
$ sudo pganalyze-collector --test --reload
```

Siehe auch:

- Installation Guide

Log-Analyse

Um die lokalen Log-Dateien kontinuierlich zu überwachen, zu klassifizieren und statistisch auszuwerten, muss `db_log_location` in `pganalyze-collector.conf` angegeben werden. `pganalyze-collector` bietet eine Hilfe zum Auffinden der Log-Dateien:

```
$ pganalyze-collector --discover-log-location
```

Die Ausgabe kann dann z.B. so aussehen:

```
db_log_location = /var/log/postgresql/postgresql-12-main.log
```

Nachdem dieses Ergebnis in der `pganalyze-collector.conf`-Konfigurationsdatei eingetragen wurde, könnt ihr diese testen mit:

```
$ pganalyze-collector --test
```

Das Ergebnis kann dann z.B. so aussehen:

```
2021/02/06 06:40:06 I [server1] Testing statistics collection...
2021/02/06 06:40:07 I [server1] Test submission successful (15.8 KB received)
2021/02/06 06:40:07 I [server1] Testing local log tailing...
2021/02/06 06:40:13 I [server1] Log test successful
2021/02/06 06:40:13 I Re-running log test with reduced privileges of "pganalyze" user␣
↪(uid = 107, gid = 113)
2021/02/06 06:40:13 I [server1] Testing local log tailing...
2021/02/06 06:40:19 I [server1] Log test successful
```

Wenn der Test erfolgreich verlief, muss der *Collector* neu gestartet werden damit die Konfiugration wirksam wird:

```
$ systemctl restart pganalyze-collector
```

3.9 NoSQL-Datenbanken

Bisher gibt es keine einheitliche Definition von NoSQL, die meisten NoSQL-Datenbanksysteme haben jedoch meist folgendes gemeinsam:

- kein relationales Datenmodell
- verteilte und horizontale Skalierbarkeit
- keine oder schwache Schemarestriktionen
- einfache API
- kein *ACID*, sondern *Eventual Consistency* oder *BASE* als Konsistenzmodell

NoSQL-Datenbanken lassen sich untergliedern in

3.9.1 Schlüssel-Werte-Datenbanksysteme

Schlüssel-Werte-Datenbanken, auch Key Value Stores genannt, speichern *Schlüssel/Wert-Paare*.

Datenbanksysteme

Schlüssel/Wert-Datenbanksysteme sind z.B. Riak, Cassandra, Redis und MongoDB.

Home	Riak	Cassandra	Redis	MongoDB
GitHub	basho/riak	apache/cassandra	redis/redis	mongodb/mongo
Docs	docs.riak.com	cassandra.apache.org/doc/	redis.io/documentation	docs.mongodb.com
Anwendungsgebiete	Session storage, Log data, Sensor data, CMS	Georedundanz, hohe Schreibgeschwindigkeit, demokratische Peer-to-peer (P2P)-Architektur, Daten mit definierter Lebenszeit	Session Cache, Full Page Cache (FPC), Queues, Pub/Sub	IoT, Mobile apps, CMS, einfache Geodaten, …
Entwicklungssprache	Erlang	Java	ANSI C	C++
Lizenzen	Apache License 2.0	Apache License 2.0	Redis Source Available License v2, Server-Side Public License v1	Server Side Public License
Datenmodell	Im Wesentlichen *Schlüssel/Wert-Paar*	*Column Family* entsprechen Tabellen, *Keyspaces* Datenbanken; keine logische Struktur, kein Schema	Schlüssel werden als Zeichenkette gespeichert, Werte als Zeichenkette, Hashes, Listen, Sets und sortierten Sets	Flexibles Schema mit denormalisiertem Modell
Query-Langauge	Keyfilter, *MapReduce*, Link-Walking, keine Ad-hoc Queries möglich	Cassandra Query Language (CQL)		jQuery, *MapReduce*
Transaktionen, Nebenläufigkeit	*ACID*	*Eventual Consistency*	in-memory, asynchron on disc mit *Append Only File Mode*	*Two-phase locking (2PL)*
Replikation, Skalierung	Multi-Master-Replikation	SimpleStrategy, NetworkTopologyStrategy und OldNetworkTopologyStrategy	Master-N-Slaves-Replikation, Sharding mittels *consistent hashing*	Master-Slave-Replikation, Auto-Sharding
Anmerkungen		Siehe auch Scylla, eine Cassandra-kompatible Reimplementierung in C.	Siehe auch: **KeyDB** ein Fork mit Multithreading **Redict** ein Fork, lizenziert unter LGPL-3.0 **Valkey** ein Fork der Linux Foundation	*BSON* mit einre maximalen Dokumentengröße von 16 MB.

3.9.2 Spaltenorientierte Datenbanksysteme

Spaltenorientierte Datenbanken, auch Wide Column Stores genannt, speichern Daten mehrerer Einträge zusammen mit einem Zeitstempel in Spalten. Spalten mit ähnlichen oder verwandten Inhalten können in einer *Column Family* zusammengefasst werden.

Datenbanksysteme

Beispiele für spaltenorientierte Datenbanksysteme sind *Cassandra*, *Hypertable* und *HBase*.

Home	Cassandra	Hypertable	HBase
GitHub	apache/cassandra	vicaya/hypertable	apache/hbase
Docs	cassandra.apache.org/doc/	hypertable.com/documentation	hbase.apache.org/book.html
Anwendungsgebiete	Georedundanz, hohe Schreibgeschwindigkeit, demokratische Peer-to-peer (P2P)-Architektur, Daten mit definierter Lebenszeit	Das Bigtable-Design von Hypertable löst horizontale Skalierungsprobleme durch ein verteiltes Speichersystem für strukturierte Daten.	IoT, fraud detection, recommendation engines
Entwicklungssprache	Java	C++	Java
Lizenzen	Apache License 2.0	GPL-3.0 License	Apache-2.0 License
Datenmodell	*Column Family* entsprechen Tabellen, *Keyspaces* Datenbanken; keine logische Struktur, kein Schema	Zuordnungstabellen (engl. Associative arrays)	In Regionen unterteilte Tabellen
Query-Langauge	Cassandra Query Language (CQL)	Hypertable Query Language (HQL)	Java Client API, Thrift/REST API
Transaktionen, Nebenläufigkeit	*Eventual Consistency*	*MVCC – Multiversion Concurrency Control*	*ACID* je Zeile, *MVCC – Multiversion Concurrency Control*
Replikation, Skalierung	SimpleStrategy, NetworkTopologyStrategy und OldNetworkTopologyStrategy	Replikation auf Dateisystem- Ebene	Master-Slave-Replikation
Anmerkungen		basiert auf verteilten Dateisystemen wie Apache Hadoop, DFS oder GlusterFS	

3.9.3 Dokumentenorientierte Datenbanksysteme

Ein Dokument in diesem Zusammenhang ist eine strukturierte Zusammenstellung bestimmter Daten. Die Daten eines Dokuments werden als *Schlüssel/Wert-Paar* gespeichert, wobei der Wert auch eine Liste oder ein Array sein kann.

Datenbanksysteme

Dokumentenorientierte Datenbanksysteme sind z.B. MongoDB, CouchDB, Riak, OrientDB und ArangoDB.

Home	MongoDB	CouchDB	Riak	OrientDB	ArangoDB
GitHub	mongodb/mongo	apache/couchdb	basho/riak	orientechnologies/orientdb	arangodb/arangodb
Docs	docs.mongodb.co	docs.couchdb.org	docs.riak.com	www.orientdb.cor	arangodb.com/documentation/
Anwendungsgebiete	IoT, Mobile apps, CMS, einfache Geodaten, …	Mobile, CRM, CMS, …	Session storage, Log data, Sensor data, CMS	Stammdatenverwaltung, soziale Netzwerke, Time Series, Key Value, Chat, Verkehrsmanagement	Fraud Detection, IoT, Identitätsmanagement, E-Commerce, Netzwerk, Logistik, CMS
Entwicklungssprache	C++	Erlang	Erlang	Java	C++, JavaScript
Lizenzen	Server Side Public License	Apache License 2.0	Apache License 2.0	Apache License 2.0	Apache License 2.0
Datenmodell	Flexibles Schema mit denormalisiertem Modell	Flexibles Schema	Im Wesentlichen *Schlüssel/Wert-Paar*	Multi-Model	Multi-Model: Dokumente, Graphen und *Schlüssel/Wert-Paar*
Query-Langauge	jQuery, *MapReduce*	REST, *MapReduce*	Keyfilter, *MapReduce*, Link-Walking, keine Ad-hoc Queries möglich	Extended SQL, Gremlin	ArangoDB Query Language (AQL)
Transaktionen, Nebenläufigkeit	*Two-phase locking (2PL)*	• *Two-phase locking (2PL)*, • einzelner Server: *ACID*, • verteilte Systeme: *BASE*	*ACID*	*ACID*	*ACID*, *MVCC – Multiversion Concurrency Control*
Replikation, Skalierung	Master-Slave-Replikation, Auto-Sharding	Master-Master-Replikation	Multi-Master-Replikation	Multi-Master-Replikation, Sharding	Master-Slave-Replikation, Sharding
Anmerkungen	*BSON* mit einer maximalen Dokumentengröße von 16 MB.				

3.9.4 Graphdatenbanksysteme

Graphdatenbanken sind spezialisiert auf vernetzte Informationen und möglichst einfache und effiziente *Graph traversal*.

Graphenmodell

Ein Graph besteht aus einer Menge an Knoten und Kanten. Graphen werden genutzt, um eine Vielfalt an Problemen durch Knoten, Kanten und ihren Beziehungen darzustellen, z.B. in Navigationssystemen, in denen die Wege in Form von Graphen gespeichert werden.

Graph traversal

Graph traversal wird meist zur Suche von Knoten verwendet. Es gibt verschiedene Algorithmen für solche Suchanfragen in einem Graphen, die sich grob einteilen lassen in

- Breiten- und Tiefensuche (engl: breadth-first search, BFS und depth-first search, DFS)

  Die Breitensuche beginnt mit allen Nachbarknoten des Startknotens. Im nächsten Schritt werden dann die Nachbarn der Nachbarn durchsucht. Die Pfadlänge erhöht sich mit jeder Iteration.

  Die Tiefensuche verfolgt einen Pfad solange, bis ein Knoten ohne ausgehende Kanten gefunden wird. Der Pfad wird anschließend zurückverfolgt bis zu einem Knoten, der noch weitere ausgehende Kanten hat. Dort wird die Suche dann fortgesetzt.

- Algorithmische Traversierung

  Beispiele für die algorithmische Traversierung sind

  - Hamiltonweg (Traveling Salesman)
  - Eulerweg
  - Dijkstra-Algorithmus

- Randomisierte Traversierung

  Der Graph wird nicht nach einem bestimmten Schema durchlaufen, sondern der nächste Knoten wird zufällig ausgewählt. Dadurch kann vor allem bei großen Graphen wesentlich schneller ein Suchergebnis präsentiert werden, dieses ist jedoch nicht immer das beste.

Datenbanksysteme

Typische Graphdatenbanken sind Neo4j, OrientDB InfiniteGraph und ArangoDB.

Home	Neo4j	OrientDB	InfiniteGraph	ArangoDB
GitHub	neo4j/neo4j	orientechnologies/orientdb		arangodb/arangodb
Docs	neo4j.com/docs/	orientdb.org/docs/	InfiniteGraph Tutorials	arangodb.com/documentation/
Anwendungsgebiete	CMS, Soziale Netzwerke, GIS-Systeme, ERP, ...	Stammdatenverwaltung, soziale Netzwerke, Time Series, Key Value, Verkehrsmanagement	Erweiterung von Objectivity/DB-Installationen	Fraud Detection, IoT, Identitätsmanagement, E-Commerce, Netzwerk, Logistik, CMS
Entwicklungssprache	Java	Java	Java	C++, JavaScript
Lizenzen	AGPL u. kommerziell	Apache License 2.0	kommerziell	Apache License 2.0
Datenmodell	*Property-Graph-Modell*	Multi-Model	*Property-Graph-Modell*	Multi-Model: Dokumente, Graphen und *Schlüssel/Wert-Paar*
Query-Langauge	REST, Cypher, Gremlin	Extended SQL, Gremlin	Traverser API, PQL	ArangoDB Query Language (AQL)
Transaktionen, Nebenläufigkeit	• *Two-phase locking (2PL)*, • einzelner Server: *ACID*, • verteilte Systeme: *BASE*	*ACID*	*ACID*	*ACID*, *MVCC – Multiversion Concurrency Control*
Replikation, Skalierung	Master-Slave mit Master Failover	Multi-Master-Replikation, Sharding	Objectivity/DB, keine *Graphpartitionierung*	Master-Slave-Replikation, Sharding
Anmerkungen			InfiniteGraph ist eine, auf dem *Objektdatenbanksysteme* Objectivity/DB aufsetzende Graphdatenbank, wobei die Objekte durch Kanten verbunden werden. Hierbei sind auch mehrfache und bidirektionale Kanten erlaubt. Iteratoren entsprechen dem *Graph traversal*.	

Siehe auch:

- Apache TinkerPop Home
- TinkerPop Documentation
- github.com/apache/tinkerpop

- Practical Gremlin – An Apache TinkerPop Tutorial
- gremlinpython

3.9.5 Objektdatenbanksysteme

Viele Programmiersprachen legen eine objektorientierte Programmierung nahe und daher erscheint die Speicherung dieser Objekte natürlich. Daher liegt nahe, den kompletten Prozess von der Implementierung bis zur Speicherung einheitlich und einfach zu gestalten. Im Einzelnen sind die Vorteile:

Natürliche Modellierung und Repräsentation von Problemen
: Probleme lassen sich auf eine Weise modellieren, die der menschlichen Denkweise sehr nahe kommen.

Übersichtlicher, lesbarer und verständlicher
: Die Daten und die auf diesen operierenden Funktionen werden zu einer Einheit zusammengefasst, wodurch die Programme übersichtlicher, lesbarer und verständlicher werden.

Modular und wiederverwendbar
: Programmteile lassen sich einfach und flexibel wiederverwenden.

Erweiterbar
: Programme lassen sich einfach erweitern und an geänderte Anforderungen anpassen.

Object-relational impedance mismatch

Objektorientierte Programmierung und relationale Datenhaltung sind aus verschiedenen Gründen problematisch. So ist ein wichtiges Konzept der OOP zur Umsetzung komplexer Modelle die Vererbung. Im relationalen Paradigma gibt es jedoch nichts vergleichbares. Zur Umwandlung entsprechender Klassenhierarchien in ein relationales Modell wurden objekt-relationale Mapper, ORM entwickelt, wie z.B. *SQLAlchemy*. Prinzipiell gibt es zwei verschiedene Ansätze für ein ORM, wobei in beiden Fällen eine Tabelle für eine Klasse angelegt wird:

Vertikale Partitionierung
: Die Tabelle enthält nur die Attribute der entsprechenden Klasse sowie einen Fremdschlüssel für die Tabelle der Oberklasse. Für jedes Objekt wird ein dann ein Eintrag in der zur Klasse gehörenden Tabelle sowie in den Tabellen aller Oberklassen angelegt. Beim Zugriff müssen die Tabellen mit Joins verbunden werden, wodurch es bei komplexen Modellen zu erheblichen Performance-Verlusten kommen kann.

Horizontale Partitionierung
: Jede Tabelle enthält die Attribute der zugehörigen Klasse sowie aller Oberklassen. Bei einer Änderung der Oberklasse müssen dann jedoch die Tabellen aller abgeleiteten Klassen ebenfalls aktualisiert werden.

Grundsätzlich müssen bei der Kombination von OOP und relationaler Datenhaltung immer zwei Datenmodelle erstellt werden. Dies macht diese Architektur deutlich komplexer, fehleranfälliger und in der Wartung aufwändiger.

Datenbanksysteme

Beispiele für Objektdatenbanksysteme sind ZODB und Objectivity/DB.

Home	ZODB	Objectivity/DB
GitHub	zopefoundation/ZODB	
Docs	www.zodb.org/en/latest/tutorial.h	Objectivity/DB Basics Tutorial
Anwendungsgebiete	Plone, Pyramid, BTrees, volatile Daten	IoT, Telekommunikation, Netzwerktechnik
Entwicklungssprache	Python	Java
Lizenzen	Zope Public License (ZPL) 2.1	kommerziell
Datenmodell	PersistentList, PersistentMapping, BTree	Objects, References, Relationships, Indexes, Trees und Collections
Query-Langauge		Objectivity/DB predicate query language
Transaktionen, Nebenläufigkeit	*ACID*	*ACID*
Replikation, Skalierung	ZODB Replication Services (ZRS)	Quorum basierte synchrone Replikation
Anmerkungen		

3.9.6 XML-Datenbanksysteme

XML-Datenbanken sind in der Lage, XML-Dokumente gegen ein XML-Schema oder eine DTD zu validieren. Darüberhinaus unterstützen sie mindestens *XPATH*, *XQuery* und *XSLT*.

Datenbanksysteme

Beispiele für XML-Datenbanksysteme sind eXist und MonetDB.

Home	eXist	MonetDB	BaseX
GitHub	eXist-db/exist	MonetDB/MonetDB	BaseXdb/basex
Docs	exist-db.org/exist/apps/doc/docu	www.monetdb.org/Documentation	docs.basex.org
Anwendungsgebiete	CMS	CMS, Date-Warehouse, Data mining	CMS
Entwicklungssprache	Java	C	Java
Lizenzen	LGPL-2.1 License	Mozilla Public License 2.0	BSD-3-Clause License
Datenmodell	XML	XML, Spaltenorientierte Datenstruktur	XML, Geo-Daten
Query-Langauge	*XQuery*, *XPATH*	SQL	*XQuery*, *XPATH*
Transaktionen, Nebenläufigkeit		*Optimistic Concurrency*	*ACID*, XQuery Locks
Replikation, Skalierung	Master-Slave-Replikation	Transaktionsreplikation	
Anmerkungen		Mit R können Analysen direkt auf Datenbankebene durchgeführt werden.	

Bedeutende Konzepte und Technologien von NoSQL-Datenbanken sind

- *MapReduce*
- *CAP-Theorem*
- *Eventual Consistency* und *BASE*
- *Konsistente Hashfunktion*
- *MVCC – Multiversion Concurrency Control*
- *Vektoruhr*
- *Paxos*

3.10 Application Programming Interface (API)

APIs können genutzt werden um die Daten bereitstellen zu können. Mit *FastAPI* steht euch eine Bibliothek zur Verfügung, die basierend auf OpenAPI und JSON Schema APIs und Dokumentationen generieren kann. *gRPC* ist hingegen ein modernes Open-Source-RPC-Framework, das HTTP/2 und QUIC verwendet.

Um das Design eurer API festzulegen, könnt ihr euch an Zalandos API Styleguide orientieren. Später könnt ihr dann mit Zally automatisiert die Qualität eurer API überprüfen. Darüberhinaus könnt ihr auch eure eigenen Regeln für Zally definieren, siehe Rule Development Manual.

Siehe auch:

- REST API Design – Resource Modeling
- Richardson Maturity Model – steps toward the glory of REST
- Irresistible APIs – Designing web APIs that developers will love
- REST in Practice
- Build APIs You Won't Hate
- Representational State Transfer (REST)

3.10.1 FastAPI

FastAPI ist ein Web-Framework zum Erstellen von APIs mit auf Python 3.6+ basierenden Type-Hints.

Hauptmerkmale sind:

- sehr hohe Leistung dank pydantic für den Datenteil und Starlette für den Web-Teil
- schnell und einfach zu codieren
- Validierung für die meisten Python-Datentypen, einschließlich
  - JSON-Objekte (`dict`)
  - JSON-Array (`list`)
  - String (`str`), definiert die minimale und maximale Länge
  - Zahlen (`int`, `float`) mit Min- und Max-Werten usw.
  - URLs
  - E -Mail mit python-email-validator
  - UUID

  - … und andere
- robuster, produktionsreifer Code mit automatischer interaktiver Dokumentation
- basierend auf den offenen Standards für APIs: OpenAPI (früher bekannt als Swagger) und JSON Schema

Siehe auch:

- Home
- GitHub

Installation

Anforderungen

```
$ pipenv install fastapi
Adding fastapi to Pipfile's [packages]...
✓ Installation Succeeded
Locking [dev-packages] dependencies...
✓ Success!
Locking [packages] dependencies...
✓ Success!
...
```

Optionale Anforderungen

Für die Produktion benötigt ihr außerdem einen ASGI-Server wie uvicorn:

```
$ pipenv install uvicorn
Adding uvicorn to Pipfile's [packages]...
✓ Installation Succeeded
Locking [dev-packages] dependencies...
✓ Success!
Locking [packages] dependencies...
✓ Success!
Updated Pipfile.lock (051f02)!
...
```

Pydantic kann die folgenden optionalen Abhängigkeiten verwenden:

ujson
: für schnelleres JSON-Parsing.

email_validator
: zur E-Mail-Validierung.

Starlette kann die folgenden optionalen Abhängigkeiten verwenden:

httpx
: wenn ihr den `TestClient` verwenden wollt.

aiofiles
: wenn ihr `FileResponse` oder `StaticFiles` verwenden wollt.

jinja2
: wenn ihr die Standard-Template-Konfiguration verwenden wollt.

python-multipart
: wenn ihr das Parsen von Formularen mit `request.form()` unterstützen wollt.

itsdangerous
: erforderlich für die Unterstützung von `SessionMiddleware`.

pyyaml
: für die Unterstützung von Starlette's `SchemaGenerator`.

graphene
: für die Unterstützung von `GraphQLApp`.

ujson
: wenn ihr `UJSONResponse` verwenden wollt.

orjson
: wenn ihr `ORJSONResponse` verwenden wollt.

Sie können installiert werden, z.B. mit:

```
$ pipenv install fastapi[ujson]
```

Alternativ könnt ihr alle installieren mit:

```
$ pipenv install fastapi[all]
```

Beispiel

1. Erstellen

Erstellt die Datei `main.py` mit diesem Inhalt:

```
from typing import Optional

from fastapi import FastAPI


app = FastAPI()


@app.get("/")
def read_root():
    return {"Hello": "World"}


@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}
```

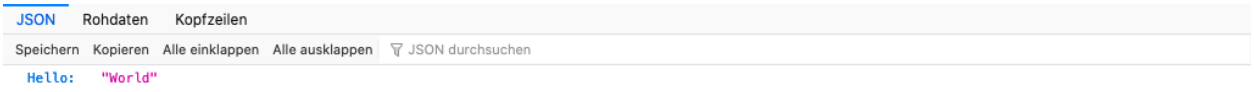
2. Ausführen

Startet den Server mit dem folgenden Aufruf (hier wieder am Beispiel von uvicorn):

```
$ pipenv run uvicorn main:app --reload
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [89155] using statreload
INFO:     Started server process [89164]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

3. Überprüfen

Öffnet euren Webbrowser unter http://127.0.0.1:8000/ und ihr werdet folgendes sehen:



Ihr erhaltet auch eine interaktive API-Dokumentation, die vom Swagger UI unter http://127.0.0.1:8000/docs bereitgestellt wird:

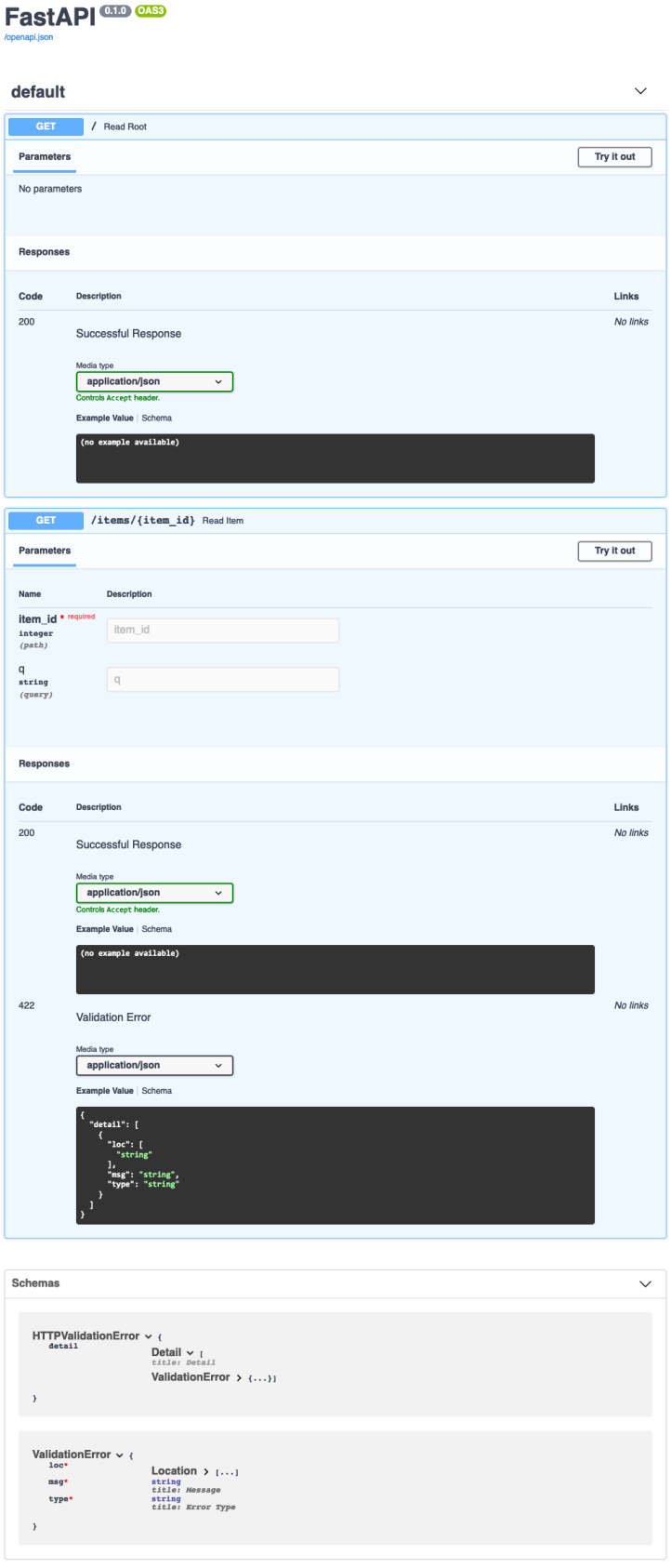
Ihr erhaltet auch eine alternative automatische Dokumentation von ReDoc unter http://127.0.0.1:8000/redoc:

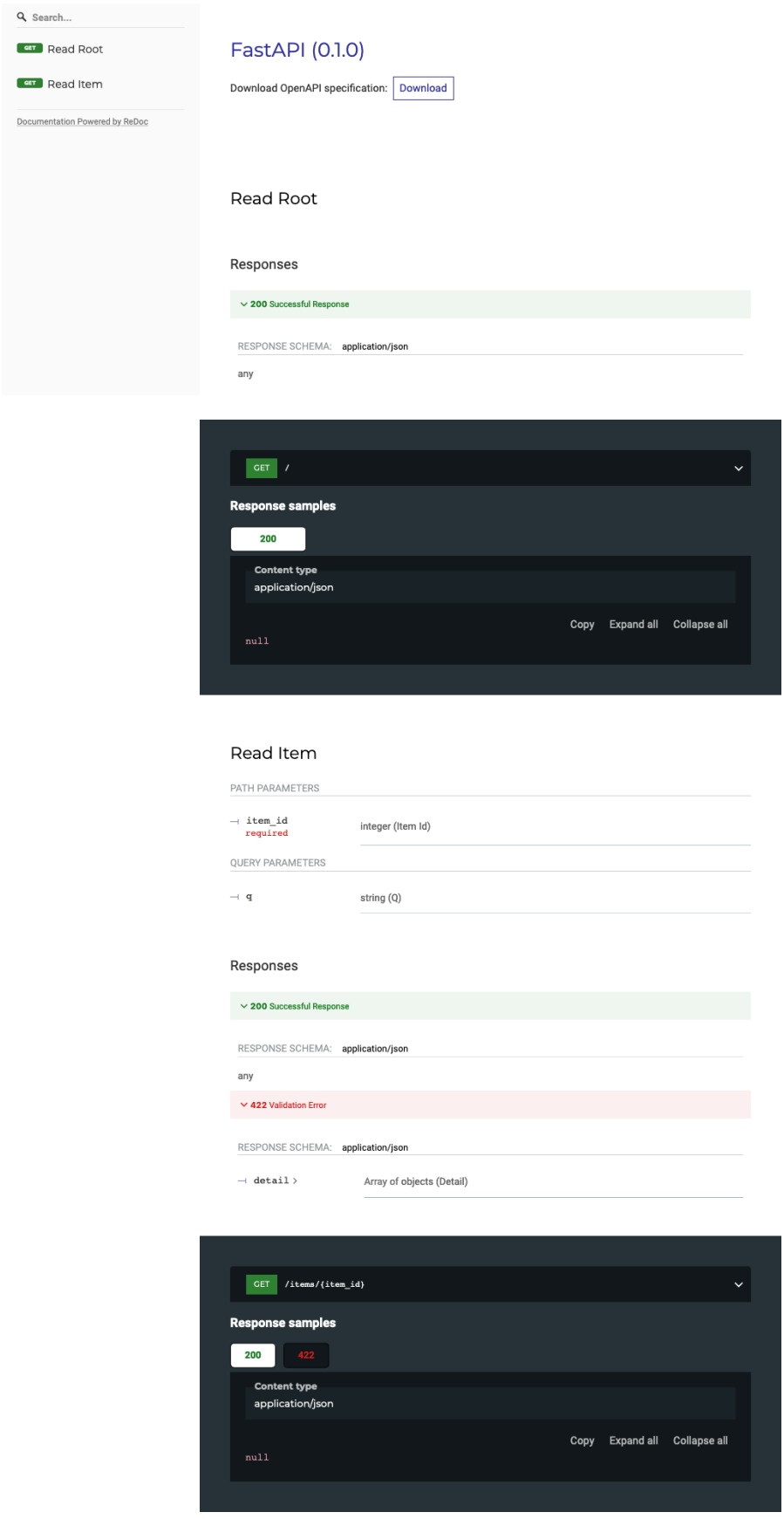
4. Aktualisierungen

Jetzt ändern wir die Datei `main.py` um einen Text von einer `PUT`-Anfrage zu erhalten:

```
from typing import Optional

from pydantic import BaseModel

from fastapi import FastAPI


app = FastAPI()


class Item(BaseModel):
    name: str
    price: float
    is_offer: Optional[bool] = None


@app.get("/")
def read_root():
    return {"Hello": "World"}
```

(Fortsetzung auf der nächsten Seite)


FastAPI
0.1.0
OAS3
/openapi.json
default
GET
/ Read Root
Parameters
Try it out
No parameters
Responses
Code
Description
Links
200
Successful Response
No links
Media type
application/json
Controls Accept header.
Example Value | Schema
(no example available)
GET
/items/{item_id} Read Item
Parameters
Try it out
Name
Description
item_id * required
integer
(path)
item_id
q
string
(query)
q
Responses
Code
Description
Links
200
Successful Response
No links
Media type
application/json
Controls Accept header.
Example Value | Schema
(no example available)
422
Validation Error
No links
Media type
application/json
Example Value | Schema
{
"detail": [
{
"loc": [
"string"
],
"msg": "string",
"type": "string"
}
]
}
Schemas
HTTPValidationError
detail
Detail
title: Detail
ValidationError
ValidationError
loc*
msg*
type*
Location
string
title: Message
string
title: Error Type


Search...
GET Read Root
GET Read Item
Documentation Powered by ReDoc
FastAPI (0.1.0)
Download OpenAPI specification: Download
Read Root
Responses
200 Successful Response
RESPONSE SCHEMA: application/json
any
GET /
Response samples
200
Content type
application/json
Copy Expand all Collapse all
null
Read Item
PATH PARAMETERS
item_id
required
integer (Item Id)
QUERY PARAMETERS
q
string (Q)
Responses
200 Successful Response
RESPONSE SCHEMA: application/json
any
422 Validation Error
RESPONSE SCHEMA: application/json
detail >
Array of objects (Detail)
GET /items/{item_id}
Response samples
200
422
Content type
application/json
Copy Expand all Collapse all
null

(Fortsetzung der vorherigen Seite)

```
@app.get("/items/{item_id}")
def read_item(item_id: int, q: Optional[str] = None):
    return {"item_id": item_id, "q": q}


@app.put("/items/{item_id}")
def update_item(item_id: int, item: Item):
return {"item_name": item.name, "item_id": item_id}
```

Der Server lädt die Datei automatisch neu, da wir den uvicorn-Aufruf in Schritt 2 mit der Option `--reload` ausgeführt haben.

Auch die interaktive API-Dokumentation zeigt nun den neuen Body mit `PUT`. Wenn ihr auf die Schaltfläche *Try it out* (aus der Abbildung in Schritt 3) klickt und einen Wert für den Parameter `item_id` angebt, wird beim Klick auf die *Execute*-Schaltfläche der Parameter vom Browser an das API übertragen und die Antwort auf dem Bildschirm angezeigt. Nachfolgend seht ihr die Ausgabe für den beispielhaft eingetragenen Wert `1234`:

```
{
  "item_name": "string",
  "item_id": 1234
}
```

Erweiterungen

Administration

SQLAlchemy Admin for Starlette/FastAPI

Flexible Admin-Schnittstelle für *SQLAlchemy*-Modelle

Piccolo Admin

Einfache, aber leistungsstarke Admin-Oberfläche über Piccolo-Tabellen, mit der ihr eure Daten leicht hinzufügen, bearbeiten und filtern könnt

Authentifizierung

AuthX

Gebrauchsfertige und anpassbare Authentifizierungen und Oauth2-Management

FastAPI Security

Authentifizierung und Autorisierung

FastAPI simple security

Auf API-Schlüsseln basierendes Sicherheitspaket, das fokussiert ist auf die einfache Nutzung

FastAPI Users

Fügt schnell ein anpassungsfähiges Registrierungs- und Authentifizierungssystem hinzu

ORMs

FastAPI-SQLAlchemy

Einfache Integration zwischen FastAPI, *SQLAlchemy* und Anwendung

FastAPIwee

Einfache Möglichkeit, eine REST-API auf der Grundlage von PeeWee-Modellen zu erstellen

GINO

Leichtgewichtiger asynchroner ORM, der auf SQLAlchemy Core für Python *asyncio* aufbaut und PostgreSQL mit asyncpg, und MySQL mit aiomysql unterstützt (→ Beispiel)

ORM

async ORM, der auf SQLAlchemy Core, Databases und TypeSystem aufbaut

ormar

Asynchroner Mini-ORM, mit dem ihr nur ein Set von Modellen pflegen und ggf. mit *Alembic* migrieren müsst (→ Beispiel); zudem wird er unterstützt von fastapi-users, fastapi-crudrouter und fastapi-pagination.

Piccolo

Schneller, benutzerfreundlicher ORM und Query Builder, der Asyncio unterstützt (→ Beispiele)

Prisma Client Python
Aufbauend auf dem TypeScript ORM Prisma mit Unterstützung von PostgreSQL, MySQL, SQLite, MongoDB und SQL Server (→ Beispiel)

Tortoise ORM
Einfach zu bedienender Asyncio-ORM, inspiriert von Django (→ Beispiele); Aerich ist ein Datenbankmigrationswerkzeug für Tortoise ORM

SQLModel
Bibliothek für die Interaktion von SQL-Datenbanken mit Python-Objekten

SQL Query Builders

FastAPI Filter
Querystring-Filter für die Api-Endpunkte und die Swagger-Benutzeroberfläche. Die unterstützten Backends sind *SQLAlchemy* und *MongoEngine*.

asyncpgsa
Python-Wrapper um asyncpg für die Verwendung mit *SQLAlchemy*

Databases

Einfache Asyncio-Unterstützung für die Datenbanktreiber asyncpg, aiopg, aiomysql, asyncmy und aiosqlite

ODMs

Beanie

Asynchroner Python-Objekt-Dokumenten-Mapper (ODM) für MongoDB, basierend auf Motor und Pydantic

MongoEngine

Python Object-Document Mapper für die Arbeit mit MongoDB

ODMantic

Asynchroner ODM (Object Document Mapper) für MongoDB basierend auf Python-Type-Hints und pydantic

Code-Generatoren

fastapi-code-generator

Code-Generator erstellt eine FastAPI-Anwendung aus einer OpenAPI-Datei, wobei datamodel-code-generator zum Generieren des pydantic-Modells verwendet wird

FastAPI-based API Client Generator

mypy- und IDE-freundlicher API-Client aus einer OpenAPI-Spezifikation unter Verwendung des OpenAPI Generator

Dienstprogramme

Caching

FastAPI Cache

Leichtgewichtiges Cache-System

fastapi-cache

Caching von FastAPI-Antworten und Funktionsergebnissen, mit Backends, die *redis*, *memcache* und *dynamodb* unterstützen

E-Mail

Fastapi-mail

Leichtes Mailsystem zum Versenden von E-Mails und Anhängen, einzeln oder auch in großen Mengen

GraphQL

Strawberry GraphQL

Python GraphQL Bibliothek basierend auf Datenklassen

Logging

ASGI Correlation ID middleware

Middleware zum Laden oder Erzeugen von Korrelations-IDs für jede eingehende Anfrage

starlette context

Middleware für Starlette, die euch ermöglicht, die Kontextdaten einer Anfrage zu speichern und darauf zuzugreifen

Prometheus

Prometheus FastAPI Instrumentator
Konfigurierbarer und modularer Prometheus-Instrumentator

starlette_exporter
Prometheus-Exportprogramm für Starlette und FastAPI

Starlette Prometheus
Prometheus-Integration für Starlette

Templating

fastapi-jinja
Integration der Jinja-Template-Sprache

fastapi-chameleon
Integration der Template-Sprache Chameleon

Paginierung

FastAPI Pagination

Einfach zu verwendende Paginierung für FastAPI mit Integration u.a. in sqlalchemy, gino, databases und ormar

Websockets

fastapi-socketio

Einfache Integration von socket.io in eure FastAPI-Anwendung

FastAPI Websocket Pub/Sub

Schneller und dauerhafter Pub/Sub-Kanal über Websockets

FASTAPI Websocket RPC

Schneller und dauerhafter bidirektionaler JSON RPC Kanal über Websockets

Andere Tools

Pydantic-SQLAlchemy
Erzeugen von Pydantic-Modellen aus SQLAlchemy-Modellen

Fastapi Camelcase
Bereitstellung einer Klasse von Request- und Response-Bodies für FastAPI

fastapi_profiler
FastAPI-Middleware basierend auf pyinstrument zur Leistungsüberprüfung

fastapi-versioning
API-Versionierung für FastAPI-Webanwendungen

Jupter Notebook REST API
Jupyter-Notebooks als REST-API-Endpunkt ausführen

manage-fastapi
Projektgenerator und -manager für FastAPI

msgpack-asgi
Automatisches Aushandeln von MessagePack-Inhalten in ASGI-Anwendungen

fastapi-plugins
Produktionsreife Plugins für das FastAPI-Framework, u.a. für das Caching mit memcached oder Redis, Scheduler, Konfiguration und Logging

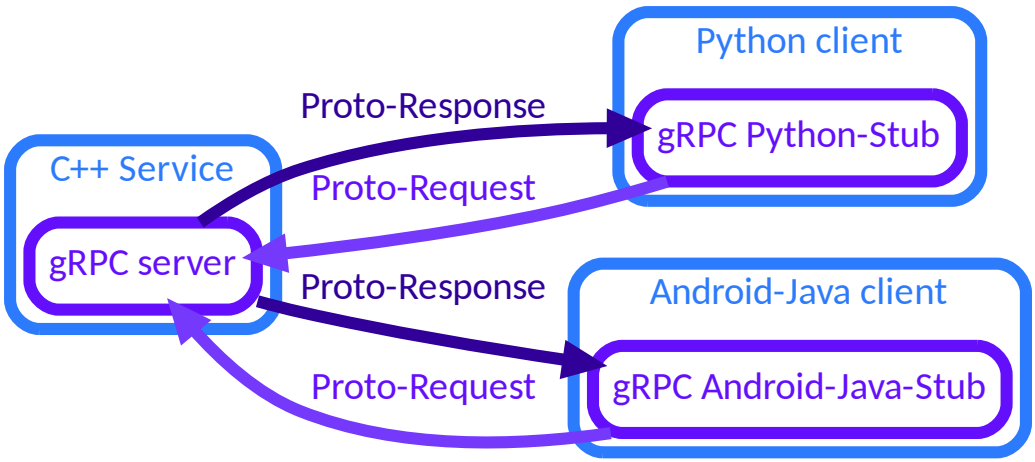
fastapi-serviceutils
Optimiertes Logging, Exception Handling und Konfigurieren

3.10.2 gRPC

gRPC ist ein modernes Open-Source-RPC (Remote Procedure Call)-Framework. Standardmäßig verwendet gRPC *Protocol Buffers (Protobuf)* als Interface Definition Language (IDL) zur Beschreibung sowohl des Interfaces als auch der Struktur der *Payload Messages*. In gRPC kann eine Clientanwendung direkt eine Methode auf einer entfernten Serveranwendung aufrufen als wäre es ein lokales Objekt, sodass verteilte Anwendungen und Dienste einfacher erstellt werden können. Wie in vielen RPC-Systemen basiert gRPC auf der Idee, einen Service zu definieren und die Methoden anzugeben, die mit ihren Parametern und Rückgabetypen aus der Ferne aufgerufen werden können. Der Server implementiert dieses Interface, um die Client-Aufrufe zu verarbeiten. Für den Client wurde ein sog. *Stub* generiert, der dieselben Methoden wie der Server bereitstellt.

Im folgenden die wesentlichen Design-Prinzipien von gRPC:

- gRPC kann auf allen gängigen Entwicklungsplattformen und in vielen verschiedenen Sprachen erstellt werden.
- Es ist auf Geräten mit geringer CPU- und Speicherfähigkeiten funktionsfähig sein, so neben Android[1]- und iOS-Geräten auch auf MicroPython-Boards und in Browsern[2][3].
- Es ist lizenziert unter Apache License 2.0 und nutzt offene Standards wie z.B. HTTP/2 und Quick UDP Internet Connections (QUIC).
- gRPC ist interoperabel und kann daher z.B. auch im LoRaWan (Long Range Wide Area Network) genutzt werden.
- Die einzelnen Layer können unabhängig voneinander entwickelt werden. So kann z.B. der Transport-Layer (OSI-Layer 4) unabhängig vom Application-Layer (OSI-Layer 7) entwickelt werden.
- gRPC unterstützt unterschiedliche Serialisierungsformate, u.a. *Protocol Buffers (Protobuf)*, *JSON*[4], *XML/HTML* und Thrift)
- Asynchrone und synchrone (blockierende) Verarbeitung werden in den meisten Sprachen unterstützt.
- Das Streaming von Nachrichten in einem einzelnen RPC-Aufruf wird unterstützt.
- gRPC erlaubt Protokollerweiterungen für Sicherheit, Integritätsprüfung, Lastausgleich, Failover etc.



Ausgehend von einer Schnittstellendefinition in einer `.proto`-Datei bietet gRPC Protocol-Compiler-Plugins, die clientseitige und serverseitige APIs generieren. Das gRPC-Protokoll gibt abstrakt die Kommunikation zwischen Clients und Servern an:

1. Zuerst wird der Stream vom Client mit einem obligatorischen `Call Header` gestartet
   1. gefolgt von optionalen `Initial-Metadata`
   2. gefolgt von optionalen `Payload Messages`.

   Die Inhalte von `Call Header` und `Initial Metadata` werden als HTTP/2-Headers mit `HPACK` komprimiert.
2. Der Server antwortet mit optionalen `Initial-Metadata`

[1] gRPC in Android Java
[2] gRPC-Web is Generally Available
[3] gRPC-Web Client Runtime Library
[4] gRPC + JSON

1. gefolgt von `Payload Messages`
2. und schließlich mit obligatorischem `Status` und optionalen `Status-Metadata`.

Payload-Nachrichten werden in einen Byte-Stream serialisiert, der in HTTP/2-Frames fragmentiert ist. `Status` und `Trailing-Metadata` werden als HTTP/2-Trailing-Headers gesendet.

Im Gegensatz zu *FastAPI* kann die gRPC-API jedoch nicht einfach auf der Kommandozeile mit cURL getestet werden. Ggf. könnt ihr jedoch grpcurl verwenden. Dies setzt jedoch voraus, dass der gRPC-Server das GRPC Server Reflection Protocol unterstützt. Üblicherweise sollte *Reflection* jedoch nur in der Entwicklungsphase zur Verfügung stehen. Dann könnt ihr jedoch `grpcurl` aufrufen, z.B. mit:

```
$ grpcurl localhost:9111 list
```

Siehe auch:

- Home
- GitHub
- gRPC Blog

---

gRPC-Beispiel

Standardmäßig verwendet gRPC *Protocol Buffers (Protobuf)* zur Serialisierung von Daten, obwohl sie auch mit anderen Datenformaten wie JSON funktionieren.

Definieren der Datenstruktur

Der erste Schritt bei der Arbeit mit Protocol-Buffers besteht darin, die Struktur für die Daten zu definieren, die Sie in einer `.proto`-Datei serialisieren möchten. Protocol-Buffers-Daten sind als *Nachrichten* strukturiert, wobei jede Nachricht ein kleiner logischer Datensatz ist, der eine Reihe von Name-Wert-Paaren enthält, die *fields* genannt werden. `accounts.proto` ist ein einfaches Beispiel hierfür:

Quellcode 2: accounts.proto

```
// SPDX-FileCopyrightText: 2021 Veit Schiele
//
// SPDX-License-Identifier: BSD-3-Clause

syntax = "proto3";

message Account {
```

Warnung: Beachtet bitte, dass ihr üblicherweise **nicht** einfach `uint32` für User- oder Group-IDs verwenden solltet, da diese viel zu einfach zu erraten wären. Hierfür könnt ihr z.B. eine **RFC 4122**-konforme Implementierung verwenden. Eine entsprechende Protobuf-Konfiguration findet ihr in `rfc4122.proto`.

Nachdem ihr eure Datenstruktur definiert habt, könnt ihr das Protocol-Buffer-Compiler-Protokoll `protoc` verwenden, um Deskriptoren in eurer bevorzugten Sprache zu erzeugen. Diese bietet einfache Zugriffsfunktionen für jedes Feld sowie Methoden zur Serialisierung der gesamten Struktur. Wenn eure Sprache z.B. Python ist, werden beim Ausführen des Compilers für das obige Beispiel Deklaratoren generiert, die ihr dann in eurer Anwendung zum Einpflegen, Serialisieren und Abrufen von Protocol-Buffer-Nachrichten verwenden könnt.

Definieren des gRPC-Dienstes

gRPC-Dienste werden ebenfalls in den `.proto`-Dateien definiert, wobei die RPC-Methodenparameter und Rückgabetypen als Protocol-Buffer-Nachrichten angegeben werden:

Quellcode 3: accounts.proto

```
  uint32 account_id = 1;
  string account_name = 2;
}

message CreateAccountRequest {
  string account_name = 1;
}

message CreateAccountResult {
  Account account = 1;
}

message GetAccountsRequest {
  repeated Account account = 1;
}

message GetAccountsResult {
  Account account = 1;
}

service Accounts {
  rpc CreateAccount (CreateAccountRequest) returns (CreateAccountResult);
  rpc GetAccounts (GetAccountsRequest) returns (stream GetAccountsResult);
}
```

Generieren des gRPC-Codes

```
$ pipenv install grpcio grpcio-tools
$ pipenv run python -m grpc_tools.protoc --python_out=. --grpc_python_out=. accounts.
↪proto
```

Dies erzeugt zwei Dateien:

accounts_pb2.py
: enthält Klassen für die in `accounts.proto` definierten Messages.

accounts_pb2_grpc.py
: enthält die definierten Klassen `AccountsStub` für den Aufruf von RPCs, `AccountsServicer` für die API-Definition des Services und eine Funktion `add_AccountsServicer_to_server` für den Server.

Server erstellen

Hierfür schreiben wir die Datei `accounts_server.py`:

Quellcode 4: accounts_server.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import logging

from concurrent import futures

import accounts_pb2 as accounts_messages
import accounts_pb2_grpc as accounts_service
import grpc


class AccountsService(accounts_service.AccountsServicer):
    def CreateAccount(self, request, context):
        metadata = dict(context.invocation_metadata())
        print(metadata)
        account = accounts_messages.Account(
            account_name=request.account_name, account_id=1
        )
        return accounts_messages.CreateAccountResult(account=account)

    def GetAccounts(self, request, context):
        for account in request.account:
            account = accounts_messages.Account(
                account_name=account.account_name,
                account_id=account.account_id,
            )
            yield accounts_messages.GetAccountsResult(account=account)


def serve():
    server = grpc.server(futures.ThreadPoolExecutor(max_workers=10))
    accounts_service.add_AccountsServicer_to_server(AccountsService(), server)
    server.add_insecure_port("[::]:8081")
    server.start()
    server.wait_for_termination()


if __name__ == "__main__":
    logging.basicConfig()
    serve()
```

Client erstellen

Hierfür schreiben wir `accounts_client.py`:

Quellcode 5: accounts_client.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import logging

import accounts_pb2 as accounts_messages
import accounts_pb2_grpc as accounts_service
import grpc


def run():
    channel = grpc.insecure_channel("localhost:8081")
    stub = accounts_service.AccountsStub(channel)
    response = stub.CreateAccount(
        accounts_messages.CreateAccountRequest(account_name="tom"),
    )
    print("Account created:", response.account.account_name)


if __name__ == "__main__":
    logging.basicConfig()
    run()
```

Client und Server starten

1. Starten des Server:

   ```
   $ pipenv run python accounts_server.py
   ```

2. Starten des Client von einem anderen Terminal aus:

   ```
   $ pipenv run python accounts_client.py
   Account created: tom
   ```

gRPC testen

pytest-grpc

gRPC lässt sich automatisiert testen z.B. mit pytest-grpc.

1. Zunächst installieren wir pytest-grpc:

   ```
   $ pipenv install pytest-grpc
   Installing pytest-grpc...
   ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Adding pytest-grpc to Pipfile's [packages]...
✓ Installation Succeeded
...
```

2. Dann erstellen wir für unser *gRPC-Beispiel* ein Test Fixture mit:

Quellcode 6: tests/test_accounts.py

```
# SPDX-License-Identifier: BSD-3-Clause

from pathlib import Path

import grpc
import pytest

from accounts_pb2 import CreateAccountRequest, GetAccountsRequest


@pytest.fixture(scope="module")
def grpc_add_to_server():
    from accounts_pb2_grpc import add_AccountsServicer_to_server

    return add_AccountsServicer_to_server


@pytest.fixture(scope="module")
def grpc_servicer():
    from accounts_server import AccountsService

    return AccountsService()


@pytest.fixture(scope="module")
```

Siehe auch:

- pytest fixtures

3. Anschließend können wir Tests schreiben, z.B.:

```
    return AccountsStub(grpc_channel)


def test_create_account(grpc_stub):
    value = "test-data"
    nl = "\n"
    request = CreateAccountRequest(account_name=value)
    response = grpc_stub.CreateAccount(request)
```

4. Auch die Authentifizierung lässt sich testen, z.B. mit:

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
    response = accounts_server.GetAccounts(request)

    assert response.name == f"test-{request.name}"


@pytest.fixture(scope="module")
def grpc_server(_grpc_server, grpc_addr, my_ssl_key_path, my_ssl_cert_path):
    """
    Overwrites default `grpc_server` fixture with ssl credentials
    """
    credentials = grpc.ssl_server_credentials(
        [(my_ssl_key_path.read_bytes(), my_ssl_cert_path.read_bytes())]
    )

    _grpc_server.add_secure_port(grpc_addr, server_credentials=credentials)
    _grpc_server.start()
    yield _grpc_server
    _grpc_server.stop(grace=None)


@pytest.fixture(scope="module")
def my_channel_ssl_credentials(my_ssl_cert_path):
    # If we're using self-signed certificate it's necessarily to pass root␣
↪certificate to channel
    return grpc.ssl_channel_credentials(
        root_certificates=my_ssl_cert_path.read_bytes()
    )


@pytest.fixture(scope="module")
def grpc_channel(my_channel_ssl_credentials, create_channel):
    """
    Overwrites default `grpc_channel` fixture with ssl credentials
    """
    with create_channel(my_channel_ssl_credentials) as channel:
        yield channel


@pytest.fixture(scope="module")
def grpc_authorized_channel(my_channel_ssl_credentials, create_channel):
    """
    Channel with authorization header passed
    """
    grpc_channel_credentials = grpc.access_token_call_credentials("some_token")
    composite_credentials = grpc.composite_channel_credentials(
        my_channel_ssl_credentials, grpc_channel_credentials
    )
    with create_channel(composite_credentials) as channel:
        yield channel
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
@pytest.fixture(scope="module")
def my_authorized_stub(grpc_stub_cls, grpc_channel):
    """
    Stub with authorized channel
    """
    return grpc_stub_cls(grpc_channel)
```

5. Anschließend können wir gegen einen realen gRPC-Server testen mit:

```
$ pipenv run pytest --fixtures tests/
```

oder direkt gegen den Python-Code:

```
$ pipenv run pytest --fixtures tests/ --grpc-fake-server
============================= test session starts ==============================
platform darwin -- Python 3.7.3, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: /Users/veit/cusy/trn/Python4DataScience/docs/data/grpc
plugins: grpc-0.8.0
collected 2 items

tests/test_accounts.py .F                                                [100%]
...
```

Siehe auch:

- GitHub
- Beispiel

Wireshark

Wireshark ist ein Open-Source-Tool zur Analyse von Netzwerkprotokollen. Im Folgenden zeigen wir Euch, wie ihr den gRPC- und den Protobuf-Dissectors verwenden könnt. Sie erleichtern Euch das Zerlegen (Dekodieren) von gRPC-Nachrichten, die im *Protobuf*- oder *JSON*-Format serialisiert sind. Zudem könnt ihr damit das Server-, Client- und bidirektionales gRPC-Streaming analysieren.

Bemerkung: Üblicherweise kann Wireshark nur gRPC-Messages im Klartext analysieren. Für das Sezieren von TLS-Session benötigt Wireshark den geheimen Schlüssel, deren Export jedoch zum heutigen Zeitpunkt nur Go gRPC unterstützt[1].

Siehe auch:

- Analyzing gRPC messages using Wireshark

---

[1] How to Export TLS Master keys of gRPC

3.11 Glossar

ACID
ACID ist ein Akronym für **A**tomicity **C**onsistency **I**solation **D**urability. Sie gelten als Voraussetzung für die Verlässlichkeit von Datenbanktransaktionen.

Atomarität
Eine Transaktion ist eine Folge von Datenbankoperationen, die entweder vollständig oder gar nicht ausgeführt werden.

Konsistenz
Transaktion, die nach Beendigung einen konsistenten Zustand hinterlässt. Dabei werden vor Abschluss der Transaktion die im Datenbankschema definierten Integritätsbedingungen überprüft.

Isolation
Nebenläufige Transaktionen dürfen sich nicht beeinflussen. Realisiert wird dies üblicherweise mit *Locking*, das die Nebenläufigkeit einschränkt.

Durability
Daten müssen nach erfolgreicher Transaktion dauerhaft in der Datenbankgespeichert bleiben und kann z.B. durch das Schreiben eines Transaktionslogs sichergestellt werden.

BASE
BASE ist ein Akronym für **B**asically **A**vailable, **S**oft State, **E**ventually Consistent und als Gegenbegriff zu *ACID* entstanden.

Dabei wird ein sehr optimistischer Konsistenzbegriff verwendet, der ohne *Locking* auskommt. Locks sind in mehrerlei Hinsicht problematisch, da Zugriffe nicht möglich sind, solange Datensätze durch andere Transaktionen gesperrt sind. Zudem ist die Übereinkunft zum Setzen eines Locks bereits sehr aufwändig.

Konsistenz der Daten wird als ein Zustand betrachtet, der irgendwann erreicht werden kann. Dies ist die Idee der *Eventual Consistency*.

Konkurrierende Zugriffe werden bei BASE durch *MVCC – Multiversion Concurrency Control* vermieden. Es gibt jedoch eine große Bandbreite von Lösungen für die verschiedenen verteilten Datenbanksysteme:

- Causal Consistency

  ist der Konsistenz in *ACID* vergleichbar.
- Read Your Writes
- Session Consistency
- Monotonic Read Consistency
- Monotonic Write Consistency

CAP-Theorem
CAP ist ein Akronym für **C**onsistency, **A**vailability (Verfügbarkeit) und **P**artition Tolerance (Ausfalltoleranz). Die Erkenntnisse des CAP-Theorems spielen bei der Auswahl eines verteilten Datenbanksystems eine zentrale Rolle.

Das CAP-Theorem besagt, dass in verteilen Systemen die drei Anforderungen Konsistenz, Verfügbarkeit und Ausfalltoleranz nicht vollständig vereinbar und nur maximal zwei von dreien erreichbar sind. Für jede Anwendung muss daher individuell entschieden werden, ob eine CA-, CP-, AP-Applikation realisiert werden soll.

Cassandra
Cassandra ist ein *Spaltenorientierte Datenbanksysteme*, und wurde ursprünglich von Facebook entwickelt um Suchen im E-Mail-Engang zu optimieren. Heute wird es unter dem Dach der Apache Software Foundation weiterentwickelt.

Das Datenmodell von Cassandra hat weder eine logische Struktur noch ein Schema. Für die Modellierung wird empfohlen *«First write your queries, then model your data»*. Meist wird dann eine *Column Family* für jede erwartete Anfrage erstellt. Dabei werden die Daten zwar denormalisiert, aber jede *Column Family* antwortet auf eine bestimmte Art von Anfragen.

In Cassandra kann für jede Anfrage die Konsistenz angegeben werden. Das ermöglicht, dass spezifische Anfragen sehr konsistent sein können während andere die Konsistenz der Geschwindigkeit opfern. Für die Schreibkonsistenz gibt es z.B. die folgenden vier Ebenen:

ANY
: gewährleistet, dass die Daten in mindestens einem Knoten gespeichert sind.

ONE
: gewährleistet, dass die Daten im Commit-Log von mindestens einer Replica gespeichert sind.

QUORUM
: gewährleistet, dass die Daten in einen Quorum von Replicas gespeichert sind.

ALL
: gewährleistet, dass die Daten auf alle Replicas gespeichert sind.

Cassandra stellt zwei verschiedene APIs zur Verfügung: Thrift und CQL (Cassandra Query Language).

Column Family
: Column Families entsprechen Tabellen in relationalen Datenbanken. Sie gruppieren Spalten gleichen oder ähnlichen Inhalts, z.B.:

```
profile = {
    cusy: {
        name:       "Cusy GmbH",
        email:      "info@cusy.io",
        website:    "cusy.io"
    },
    veit: {
        name:       "Veit Schiele",
        email:      "veit.schiele@cusy.io",
    }
}
```

CouchDB
: CouchDB ein Akronym für **C**luster **o**f **u**nreliable **c**ommodity **h**ardware **D**ata **B**ase. Dabei handelt es sich um ein *Dokumentenorientierte Datenbanksysteme*.

Eventual Consistency
: *»Konsistenz als Zustandsübergang, der irgendwann erreicht wird.«*

  Der Begriff wurde für *BASE* als Alternative zu *ACID* entwickelt.

Graph traversal
: Graph traversal wird meist zur Suche von Knoten verwendet. Es gibt verschiedene Algorithmen für solche Suchanfragen in einem Graphen, die sich grob einteilen lassen in

  - Breiten- und Tiefensuche (engl: breadth-first search, BFS und depth-first search, DFS)

    Die Breitensuche beginnt mit allen Nachbarknoten des Startknotens. Im nächsten Schritt werden dann die Nachbarn der Nachbarn durchsucht. Die Pfadlänge erhöht sich mit jeder Iteration.

    Die Tiefensuche verfolgt einen Pfad solange, bis ein Knoten ohne ausgehende Kanten gefunden wird. Der Pfad wird anschließend zurückverfolgt bis zu einem Knoten, der noch weitere ausgehende Kanten hat. Dort wird die Suche dann fortgesetzt.

- Algorithmische Traversierung

  Beispiele für die algorithmische Traversierung sind

  - Hamiltonweg (Traveling Salesman)
  - Eulerweg
  - Dijkstra-Algorithmus

- Randomisierte Traversierung

  Der Graph wird nicht nach einem bestimmten Schema durchlaufen, sondern der nächste Knoten wird zufällig ausgewählt. Dadurch kann vor allem bei großen Graphen wesentlich schneller ein Suchergebnis präsentieren, dieses ist jedoch nicht immer das beste.

Graphenmodell
Ein Graph besteht aus einer Menge an Knoten und Kanten. Graphen werden genutzt, um eine Vielfalt an Problemen durch Knoten, Kanten und ihren Beziehungen darzustellen, z.B. in Navigationssystemen, in denen die Wege in Form von Graphen gespeichert werden.

Graphpartitionierung
Mit Graphpartitionierung werden Graphen in kleinere Teilgraphen unterteilt. Dabei gibt es jedoch keine mathematisch exakte Methode, um die Anzahl der durchschnittenen Kanten zu minimieren, sondern nur ein paar heuristische Algorithmen, z.B. Clustering-Algorithmen, die stark vernetzte Teilgraphen zu abstrakten Knoten zusammenziehen.

Von sich überlappenden Partitionierung spricht man bei Graphen, die nicht komplett geteilt werden können und in mehreren Teilgraphen existieren.

HBase
HBase ist ein *Spaltenorientierte Datenbanksysteme*, welches auf verteilten Dateisystemen aufbaut und für realtime-Zugriffe auf großen Datenbeständen konzipiert ist.

Hypertable
Hypertable ist ein *Spaltenorientierte Datenbanksysteme* und auf verteilten Dateisystemen basiert. Das Datenmodell ist das einer mehrdimensionalen Tabelle, die mit Schlüsseln durchsucht werden kann. Die erste Dimension ist der sog. *row-key*, die zweite die *Column Family* die dritte Dimension der *Column Qualifier* und die vierte Dimension die Zeit.

Konsistente Hashfunktion
Konsistente Hashfunktionen minimieren die Anzahl der Neuzuordnungen, da bei einer Änderung nicht alle Schlüssel neu zugeordnet werden müssen sondern nur die Größe einer Hash-Tabelle geändert wird.

Konsistenz
Der Zustand einer Datenbank wird als konsistent bezeichnet, wenn die gespeicherten Daten alle Anforderungen for *Semantische Integrität* erfüllen.

Locking
Als Locking bezeichnet man das Sperren von Daten für nebenläufige Transaktionen.

Je nach Art des Zugriffs gibt es unterschiedliche Lock-Verfahren:

- *Optimistic concurrency*
- *Pessimistic locking*
- *Two-phase locking (2PL)*

MapReduce
MapReduce ist ein von Google Inc. 2004 eingeführtes Framework, das für die nebenläufige Berechnungen enorm

großer Datenmengen auf Computerclustern verwendet wird. Es wurde durch die, in der funktionalen Programmierung häufig verwendeten Funktionen *map* und *reduce* inspiriert auch wenn die Semantik von diesen etwas abweicht.

MongoDB
MongoDB ist eine schemafrei *Dokumentenorientierte Datenbanksysteme*, die Dokumente im BSON-Format verwaltet.

MVCC – Multiversion Concurrency Control
MVCC werden kontrolliert konkurrierende Zugriffe auf Datensätze (Lesen, Einfügen, Ändern, Löschen) durch verschiedene, unveränderliche Versionen dieser Datensätze. Die verschiedenen Versionen werden in eine zeitliche Reihenfolge gebracht, indem jede Version auf ihre Vorgängerversion verweist. MVCC hat sich gerade bei *NoSQL-Datenbanken* zu einer zentralen Basistechnologie entwickelt, die es ermöglicht, konkurrierende Zugriffe auch ohne das *Locking* von Datensätzen zu koordinieren.

Optimistic Concurrency
Optimistic Concurrency, auch Optimistisches Locking ist eine Form des *Locking*, die davon ausgeht, dass wenige schreibende Zugriffe auf der Datenbank stattfinden und lesende Zugriffe keine Sperre auslösen. Bei Änderungen wird dann zunächst geprüft, ob der Zeitstempel seit dem Lesen der Daten unverändert geblieben ist.

Paxos
Paxos ist eine Familie von Protokollen zur Herstellung von Konsens in einem Netzwerk unzuverlässiger oder fehlbarer Prozessoren.

Pessimistic Locking
Pessimistic *Locking* geht von vielen Schreibzugriffen auf die Datenbank aus. Daher sperren auch lesende Zugriffe die Daten werden erst wieder freigegeben, wenn die Änderungen gespeichert sind.

Property-Graph-Modell
PGM
Knoten und Kanten bestehen aus Objekten mit darin eingebetteten Eigenschaften (Properties). Es wird nicht nur ein Wert (Label) in einer Kante bzw. einem Knoten gespeichert, sondern ein *Schlüssel/Wert-Paar*.

Riak
Im Wesentlichen ist Riak ein dezentraler *Schlüssel/Wert-Paar* mit einer flexiblen *MapReduce*-Engine.

Redis
Redis ist ein *Schlüssel-Werte-Datenbanksysteme*, die üblicherweise alle Daten im RAM speichert.

Schlüssel/Wert-Paar
Ein Wert ist immer einem bestimmten Schlüssel zugeordnet, der aus einer strukturierten oder willkürlichen Zeichenkette bestehen kann. Diese Schlüssel können in Namensräume und Datenbanken aufgeteilt werden. Die Werte können neben Strings auch Listen, Sets oder Hashes enthalten.

Semantische Integrität
Semantische Integrität ist immer dann gegeben, wenn die Eingaben richtig und in sich stimmig sind. Dann wird auch von konsistenten Daten gesprochen. Ist dies nicht der Fall, sind die Daten inkonsistent. In SQL kann die semantische Integrität mit `TRIGGER` und `CONSTRAINT` überprüft werden.

Two-phase locking (2PL)
Das Zwei-Phasen-Sperrprotokoll unterscheidet zwei Phasen von Transaktionen:

1. Die Wachstumsphase, in welcher Sperren nur gesetzt, aber nicht freigegeben werden dürfen.
2. Die Schrumpfungsphase, in welcher Sperren nur freigegeben, aber nicht angefordert werden dürfen.

Das Zwei-Phasen-Sperrprotokoll kennt dabei drei Sperrzustände:

SLOCK, Shared Lock oder Read-Lock
wird bei lesendem Zugriff auf Daten gesetzt

XLOCK, Exclusive Lock oder Write-Lock
wird bei schreibendem Zugriff auf Daten gesetzt

UNLOCK
hebt die Sperren SLOCK und XLOCK auf.

Vektoruhr
Eine Vektoruhr ist eine Softwarekomponente zum Zuweisen von eindeutigen Zeitstempeln an Nachrichten. Sie erlaubt, den Ereignissen in verteilten Systemen aufgrund eines Zeitstempels eine Kausalordnung zuzuweisen und insbesondere die Nebenläufigkeit von Ereignissen zu ermitteln.

XPATH
XPATH verarbeitet die Baumstruktur eines XML-Dokuments und erzeugt dabei Ausschnitte aus XML-Dokumenten. Um als Ergebnis vollständige XML-Dokumente zu erhalten, müssen diese z.B. mit *XQuery* oder *XSLT* erstellt werden. XPATH ist keine vollständige Abfragesprache, da sie auf Selektionen und Extraktionen beschränkt ist.

XPATH ist ein Bestandteil von *XQuery* seit Version 1.1 und ab Version 2.0 wird XPATH durch *XQuery* erweitert.

XQuery
XQuery steht für *XML Query Language* und ist hauptsächlich eine funktionale Sprache, bei der während einer Abfrage auch verschachtelte Ausdrücke ausgewertet werden können.

XSLT
XSLT ist ein Akronym für **E**xtensible **S**tylesheet **L**anguage **T**ransformation. Mit ihr lassen sich XML-Dokumente transformieren.

KAPITEL 4

Daten bereinigen und validieren

Im Folgenden wollen wir euch einen praktischen Überblick über verschiedene Bibliotheken und Methoden zur Datenbereinigung und -validierung mit Python geben. Dabei verwenden wir neben bekannten Bibliotheken wie NumPy und Pandas auch mehrere kleine, spezialisierte Bibliotheken wie *dedupe*, *fuzzywuzzy*, *voluptuous*, *bulwark*, *tdda* und *hypothesis*. Wir bevorzugen diese leichtgewichtigeren Lösungen gegenüber großen, universellen Systemen wie Great Expectations oder MobyDQ.

4.1 Überblick

Tab. 1: GitHub-Insights

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
fuzzywuzzy	Stars 9.1k	contributors 60	commit activity 0/year	license GPL-2.0
dedupe	Stars 4k	contributors 50	commit activity 30/year	license MIT
Bulwark	Stars 221	contributors 7	commit activity 0/year	license LGPL-3.0
Hypothesis	Stars 7.2k	contributors 283	commit activity 1.5k/year	license not identifiable by github
TDDA	Stars 275	contributors 8	commit activity 3/year	license MIT
Voluptuous	Stars 1.8k	contributors 73	commit activity 21/year	license BSD-3-Clause
scikit-learn	Stars 58k	contributors 409	commit activity 1.3k/year	license BSD-3-Clause
pandera	Stars 3k	contributors 116	commit activity 148/year	license MIT
Validr	Stars 211	contributors 4	commit activity 31/year	license not identifiable by github
marshmallow	Stars 6.9k	contributors 177	commit activity 143/year	license MIT
datacleaner	Stars 1k	contributors 3	commit activity 0/year	license MIT
Probatus	Stars 120	contributors 25	commit activity 31/year	license MIT
popmon	Stars 485	contributors 11	commit activity 50/year	license MIT
Pandas Profiling	Stars 12k	contributors 100	commit activity 115/year	license MIT
pandas-validation	Stars 20	contributors 1	commit activity 0/year	license MIT
PandasSchema	Stars 185	contributors 6	commit activity 0/year	license GPL-3.0
Opulent-Pandas	Stars 9	contributors 1	commit activity 0/year	license Apache-2.0
signpost	Stars 1	contributors 1	commit activity 0/year	license GPL-3.0

4.1.1 Verwalten fehlender Daten mit pandas

Fehlende Daten treten häufig bei Datenanalysen auf. pandas vereinfacht die Arbeit mit fehlenden Daten so weit wie möglich. Zum Beispiel schließen alle *deskriptiven Statistiken* von pandas-Objekten standardmäßig fehlende Daten aus. pandas verwendet für numerische Daten den Fließkommawert `NaN` (*Not a Number*), um fehlende Daten darzustellen.

In pandas wurde eine der Programmiersprache R entlehnte Konvention übernommen und fehlende Daten als `NA` bezeichnet, was für *not available* (engl.: nicht verfügbar) steht. In statistischen Anwendungen können `NA`-Daten entweder Daten sein, die nicht existieren oder die zwar existieren, aber nicht beobachtet wurden (z.B. durch Probleme bei der Datenerfassung). Auch das None-Objekt von Python wird in nicht-numerischen Arrays als `NA` behandelt.

Methoden zum Handling von `NA`-Objekten:

Argument	Beschreibung
dropna	filtert Achsenbeschriftungen auf der Grundlage, ob Werte für die einzelnen Label fehlende Daten aufweisen, wobei unterschiedliche Schwellenwerte für die zu tolerierende Menge fehlender Daten gelten.
fillna	füllt fehlende Daten mit einem Wert oder mit einer Interpolationsmethode wie `ffill'` oder `bfill` auf.
isna	gibt boolesche Werte zurück, die angeben, welche Werte fehlen/NA sind.
notna	negiert `isna` und gibt `True` für nicht-NA-Werte und `False` für NA-Werte zurück.

In diesem Notebook werden einige Möglichkeiten vorgestellt, wie fehlende Daten mit pandas DataFrames verwaltet werden können. Weitere Informationen findet ihr in der Pandas-Dokumentation: Working with missing data und Missing data cookbook.

Siehe auch:

- Dora
- Badfish

[1]:
```
import pandas as pd
```

[2]:
```
df = pd.read_csv("https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/
↪iot_example_with_nulls.csv")
```

1. Überprüfen der Daten

Bei der Bereinigung von Daten für die Analyse ist es oft wichtig, die fehlenden Daten selbst zu analysieren, um Probleme bei der Datenerfassung oder potenzielle Verzerrungen in den Daten aufgrund der fehlenden Daten zu ermitteln. Zunächst lassen wir uns die ersten 20 Datensätze anzeigen:

[3]:
```
df.head(20)
```

[3]:

	timestamp	username	temperature	heartrate	\
0	2017-01-01T12:00:23	michaelsmith	12.0	67	
1	2017-01-01T12:01:09	kharrison	6.0	78	
2	2017-01-01T12:01:34	smithadam	5.0	89	
3	2017-01-01T12:02:09	eddierodriguez	28.0	76	
4	2017-01-01T12:02:36	kenneth94	29.0	62	
5	2017-01-01T12:03:04	bryanttodd	13.0	86	
6	2017-01-01T12:03:51	andrea98	17.0	81	
7	2017-01-01T12:04:35	scott28	16.0	76	
8	2017-01-01T12:05:05	hillpamela	5.0	82	
9	2017-01-01T12:05:41	moorejeffrey	25.0	63	
10	2017-01-01T12:06:21	njohnson	NaN	63	
11	2017-01-01T12:06:53	gsutton	29.0	80	
12	2017-01-01T12:07:41	jessica48	22.0	83	
13	2017-01-01T12:08:08	hornjohn	16.0	73	
14	2017-01-01T12:08:35	gramirez	24.0	73	
15	2017-01-01T12:09:05	schmidtsamuel	NaN	78	
16	2017-01-01T12:09:48	derrick47	NaN	63	
17	2017-01-01T12:10:23	beckercharles	12.0	61	
18	2017-01-01T12:10:57	ipittman	11.0	69	

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
19  2017-01-01T12:11:34       sabrina65         22.0         82

                                    build  latest      note
0   4e6a7805-8faa-2768-6ef6-eb3198b483ac     0.0  interval
1   7256b7b0-e502-f576-62ec-ed73533c9c84     0.0      wake
2   9226c94b-bb4b-a6c8-8e02-cb42b53e9c90     0.0       NaN
3                                     NaN     0.0    update
4   122f1c6a-403c-2221-6ed1-b5caa08f11e0     NaN       NaN
5   0897dbe5-9c5b-71ca-73a1-7586959ca198     0.0  interval
6   1c07ab9b-5f66-137d-a74f-921a41001f4e     1.0       NaN
7   7a60219f-6621-e548-180e-ca69624f9824     NaN  interval
8   a8b87754-a162-da28-2527-4bce4b3d4191     1.0       NaN
9   585f1a3c-0679-0ffe-9132-508933c70343     0.0      wake
10  e09b6001-125d-51cf-9c3f-9cb686c19d02     NaN       NaN
11  607c9f6e-2bdf-a606-6d16-3004c6958436     1.0    update
12  03e1a07b-3e14-412c-3a69-6b45bc79f81c     NaN    update
13                                    NaN     0.0  interval
14                                    NaN     0.0      wake
15  b9890c1e-79d5-8979-63ae-6c08a4cd476a     0.0       NaN
16  b60bd7de-4057-8a85-f806-e6eec1350338     NaN  interval
17  b1dacc73-c8b7-1d7d-ee02-578da781a71e     0.0      test
18  1aef7db8-9a3e-7dc9-d7a5-781ec0efd200     NaN      user
19  8075d058-7dae-e2ec-d47e-58ec6d26899b     1.0       NaN
```

Dann schauen wir uns an, von welchem Datentyp die Spalten sind:

[4]:
```
df.dtypes
```

[4]:
```
timestamp       object
username        object
temperature    float64
heartrate        int64
build           object
latest         float64
note            object
dtype: object
```

Wir können uns auch die Werte und deren Häufigkeit anzeigen lassen, z.B. für die Spalte `note`:

[5]:
```
df.note.value_counts()
```

[5]:
```
note
wake        16496
user        16416
interval    16274
sleep       16226
update      16213
test        16068
Name: count, dtype: int64
```

2. Entfernen aller Nullwerte (einschließlich der Angabe `n/a`)

2.1 ... mit `pandas.read_csv`

pandas.read_csv filtert normalerweise bereits viele Werte heraus, die es als `NA` oder `NaN` erkennt. Weitere Werte können mit `na_values` angegeben werden.

[6]:
```
df = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/iot_example_
↪with_nulls.csv",
    na_values=["n/a"],
)
```

2.2 ... mit `pandas.DataFrame.dropna`

Mit pandas.DataFrame.dropna lassen sich fehlende Werte löschen.

Um den Umfang der Löschungen zu analysieren lassen wir uns den Umfang des DataFrame vor und nach dem Läschen mit pandas.DataFrame.shape anzeigen:

[7]:
```
df.shape
```

[7]:
```
(146397, 7)
```

[8]:
```
df2 = df.dropna().shape
```

Wir würden mit `pandas.DataFrame.dropna` also mehr als 2/3 der Datensätze verlieren.

Im nächsten Versuch wollen wir analysieren, ob ganze Zeilen oder Spalten keine Daten enthalten. Dabei werden mit `how='all'` Zeilen oder Spalten entfernt, die keine Werte enthalten; `axis=1` besagt, dass leere Zeilen entfernt werden sollen.

[9]:
```
df.dropna(how="all", axis=1).shape
```

[9]:
```
(146397, 7)
```

Auch dies bringt uns noch nicht weiter.

2.3 Alle Spalten finden, in denen alle Daten vorhanden sind

[10]:
```
complete_columns = list(df.columns)
```

[11]:
```
complete_columns
```

[11]:
```
['timestamp',
 'username',
 'temperature',
 'heartrate',
 'build',
 'latest',
 'note']
```

2.4 Allte Spalten finden, in denen die meisten Daten vorhanden sind

[12]:
```
list(df.dropna(thresh=int(df.shape[0] * 0.9), axis=1).columns)
```

[12]:
```
['timestamp', 'username', 'heartrate']
```

`thresh` erfordert eine bestimmte Anzahl von NA-Werten, in unserem Fall 90 %, bevor `axis=1` eine Spalte ausblendet.

2.5 Alle Spalten mit fehlenden Daten finden

Mit pandas.DataFrame.isnull können wir fehlende Werte finden und mit pandas.DataFrame.any erfahren wir, ob ein Element gültig ist, normalerweise über einer Spalte.

[13]:
```
incomplete_columns = list(df.columns[df.isnull().any()])
```

[14]:
```
incomplete_columns
```

[14]:
```
['temperature', 'build', 'latest', 'note']
```

Mit `num_missing` können wir uns nun die Anzahl der fehlenden Werte pro Spalte ausgeben lassen:

[15]:
```
for col in incomplete_columns:
    num_missing = df[df[col].isnull() == True].shape[0]
    print("number missing for column {}: {}".format(col, num_missing))
```

```
number missing for column temperature: 32357
number missing for column build: 32350
number missing for column latest: 32298
number missing for column note: 48704
```

Diese Werte können wir uns auch prozentual ausgeben lassen:

[16]:
```
for col in incomplete_columns:
    percent_missing = df[df[col].isnull() == True].shape[0] / df.shape[0]
    print("percent missing for column {}: {}".format(col, percent_missing))
```

```
percent missing for column temperature: 0.22102228870810195
percent missing for column build: 0.22097447352063226
percent missing for column latest: 0.22061927498514314
percent missing for column note: 0.332684412931959
```

2.6 Ersetzen fehlender Daten

Um unsere Änderungen in der Spalte `latest` überprüfen zu können, verwenden wir pandas.Series.value_counts. Die Methode gibt eine Serie zurück, die die Anzahl der eindeutigen Werte enthält:

[17]:
```
df.latest.value_counts()
```

[17]:
```
latest
0.0    75735
1.0    38364
Name: count, dtype: int64
```

Jetzt ersetzen wir die fehlenden Werte in der Spalte `latest` durch `0` mit DataFrame.fillna:

[18]:
```
df.latest = df.latest.fillna(0)
```

[19]:
```
df.latest.value_counts()
```

[19]:
```
latest
0.0    108033
1.0     38364
Name: count, dtype: int64
```

2.7 Ersetzen fehlender Daten durch `backfill`

Damit die Datensätze in ihrer zeitlichen Reihenfolge aufeinanderfolgen, setzen wir zunächst den Index für `timestamp` mit set_index:

[20]:
```
df = df.set_index("timestamp")
```

[21]:
```
df.head(20)
```

[21]:

timestamp	username	temperature	heartrate	build	latest	note
2017-01-01T12:00:23	michaelsmith	12.0	67	4e6a7805-8faa-2768-6ef6-eb3198b483ac	0.0	interval
2017-01-01T12:01:09	kharrison	6.0	78	7256b7b0-e502-f576-62ec-ed73533c9c84	0.0	wake
2017-01-01T12:01:34	smithadam	5.0	89	9226c94b-bb4b-a6c8-8e02-cb42b53e9c90	0.0	NaN
2017-01-01T12:02:09	eddierodriguez	28.0	76	NaN	0.0	update
2017-01-01T12:02:36	kenneth94	29.0	62	122f1c6a-403c-2221-6ed1-b5caa08f11e0	0.0	NaN
2017-01-01T12:03:04	bryanttodd	13.0	86	0897dbe5-9c5b-71ca-73a1-7586959ca198	0.0	interval
2017-01-01T12:03:51	andrea98	17.0	81	1c07ab9b-5f66-137d-a74f-921a41001f4e	1.0	NaN
2017-01-01T12:04:35	scott28	16.0	76	7a60219f-6621-e548-180e-ca69624f9824	0.0	interval
2017-01-01T12:05:05	hillpamela	5.0	82	a8b87754-a162-da28-2527-4bce4b3d4191	1.0	NaN
2017-01-01T12:05:41	moorejeffrey	25.0	63			
2017-01-01T12:06:21	njohnson	NaN	63			
2017-01-01T12:06:53	gsutton	29.0	80			
2017-01-01T12:07:41	jessica48	22.0	83			
2017-01-01T12:08:08	hornjohn	16.0	73			
2017-01-01T12:08:35	gramirez	24.0	73			
2017-01-01T12:09:05	schmidtsamuel	NaN	78			
2017-01-01T12:09:48	derrick47	NaN	63			
2017-01-01T12:10:23	beckercharles	12.0	61			
2017-01-01T12:10:57	ipittman	11.0	69			
2017-01-01T12:11:34	sabrina65	22.0	82			

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2017-01-01T12:05:41  585f1a3c-0679-0ffe-9132-508933c70343     0.0      wake
2017-01-01T12:06:21  e09b6001-125d-51cf-9c3f-9cb686c19d02     0.0       NaN
2017-01-01T12:06:53  607c9f6e-2bdf-a606-6d16-3004c6958436     1.0    update
2017-01-01T12:07:41  03e1a07b-3e14-412c-3a69-6b45bc79f81c     0.0    update
2017-01-01T12:08:08                                  NaN     0.0  interval
2017-01-01T12:08:35                                  NaN     0.0      wake
2017-01-01T12:09:05  b9890c1e-79d5-8979-63ae-6c08a4cd476a     0.0       NaN
2017-01-01T12:09:48  b60bd7de-4057-8a85-f806-e6eec1350338     0.0  interval
2017-01-01T12:10:23  b1dacc73-c8b7-1d7d-ee02-578da781a71e     0.0      test
2017-01-01T12:10:57  1aef7db8-9a3e-7dc9-d7a5-781ec0efd200     0.0      user
2017-01-01T12:11:34  8075d058-7dae-e2ec-d47e-58ec6d26899b     1.0       NaN
```

Anschließend verwenden wir pandas.DataFrame.groupby, um die Datensätze nach `username` zu gruppieren und dann die fehlenden Daten mit der `backfill`-Methode von pandas.core.groupby.DataFrameGroupBy.fillna zu füllen. `limit` definiert die maximale Anzahl aufeinanderfolgender `NaN`-Werte:

[22]:
```
df.temperature = df.groupby("username").temperature.fillna(
    method="backfill", limit=3
)
```

[23]:
```
for col in incomplete_columns:
    num_missing = df[df[col].isnull() == True].shape[0]
    print("number missing for column {}: {}".format(col, num_missing))
```

```
number missing for column temperature: 22633
number missing for column build: 32350
number missing for column latest: 0
number missing for column note: 48704
```

Argumente der Funktion `fillna`:

Argument	Beschreibung
`value`	Skalarwert oder dict-atähnliches Objekt, das zum Auffüllen fehlender Werte verwendet wird
Methode	Interpolation; standardmäßig `ffill`, wenn die Funktion ohne weitere Argumente aufgerufen wird
`axis`	Aufzufüllende Achse; Voreinstellung `axis=0`
`inplace`	ändert das aufrufende Objekt, ohne eine Kopie zu erzeugen
`limit`	Für das Auffüllen in Vorwärts- und Rückwärtsrichtung, maximale Anzahl von aufeinanderfolgenden Perioden zum Auffüllen

4.1.2 Erkennen und Filtern von Ausreißern

Das Filtern oder Transformieren von Ausreißern ist weitgehend eine Frage der Anwendung von Array-Operationen. Betrachtet einen DataFrame mit einigen normal verteilten Daten:

[1]:
```
import numpy as np
import pandas as pd


df = pd.DataFrame(np.random.randn(1000, 4))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
df.describe()
```

[1]:
```
                0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.003064     0.096247     0.056972    -0.020452
std       1.008448     0.976915     1.004594     1.010706
min      -3.208608    -3.209683    -2.950671    -3.386906
25%      -0.708152    -0.599949    -0.647935    -0.680458
50%       0.041351     0.105793     0.060608    -0.023031
75%       0.680191     0.731972     0.751053     0.625897
max       3.226749     3.221513     3.257586     3.604722
```

Angenommen, ihr wollt in einer der Spalten Werte finden, deren absoluter Wert größer als 3 oder kleiner als -3 ist:

[2]:
```
col = df[1]

col[col.abs() > 3]
```

[2]:
```
219   -3.209683
411    3.140193
934    3.221513
Name: 1, dtype: float64
```

Um alle Zeilen auszuwählen, in denen Wert größer 3 oder kleiner -3 in einer der Spalten ist, könnt ihr pandas.DataFrame.any auf einen booleschen DataFrame anwenden, wobei mit `any(axis=1)` üüberprüft wird, ob ein Wert in einer Zeile wahr ist:

[3]:
```
df[(df.abs() > 3).any(axis=1)]
```

[3]:
```
            0         1         2         3
95   3.226749 -1.579194 -0.719229  0.214088
101  0.562001  0.333124  1.559934 -3.215258
219  1.083981 -3.209683 -0.667363 -1.438619
279 -0.494579 -0.864461  1.278015  3.409804
348 -0.090586 -2.045006  0.065500  3.604722
411  1.552122  3.140193  2.438573 -1.334575
467  3.188898 -0.407706 -0.120590  0.630946
615 -0.168051  0.312383  1.491362 -3.386906
675 -3.208608 -0.314907  0.734751 -0.126330
685  1.669532  0.266478 -1.005036  3.066603
928  0.272368  0.535655  3.257586  1.425122
934  0.246128  3.221513  0.310442 -2.481366
945 -3.086749 -2.254744  0.799249  0.689478
```

Auf dieser Grundlage können die Werte begrenzt werden auf ein Intervall zwischen `-3` und `3`. Hierfür verwenden wir die Anweisung `np.sign(df)`, die Werte `1` und `-1` erzeugt, je nachdem, ob die Werte in `df` positiv oder negativ sind:

[4]:
```
df[df.abs() > 3] = np.sign(df) * 3

df.describe()
```

[4]:
```
                0            1            2            3
count  1000.000000  1000.000000  1000.000000  1000.000000
mean      0.002944     0.096095     0.056714    -0.020931
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
std       1.006261     0.975114     1.003805     1.005317
min      -3.000000    -3.000000    -2.950671    -3.000000
25%      -0.708152    -0.599949    -0.647935    -0.680458
50%       0.041351     0.105793     0.060608    -0.023031
75%       0.680191     0.731972     0.751053     0.625897
max       3.000000     3.000000     3.000000     3.000000
```

4.1.3 String-Vergleiche

In diesem Notebook verwenden wir die beliebte Bibliothek für String-Vergleiche fuzzywuzzy. Sie basiert auf der eingebauten Python-Bibliothek difflib. Weitere Informationen zu den verschiedenen verfügbaren Methoden und ihren Unterschieden findet ihr im Blogbeitrag FuzzyWuzzy: Fuzzy String Matching in Python.

Siehe auch:

textacy

1. Installation

Mit *Spack* könnt ihr `fuzzywuzzy` und die optionale `python-levenshtein`-Bibliothek in eurem Kernel bereitstellen:

```
$ spack env activate python-311
$ spack install py-fuzzywuzzy+speedup
```

Alternativ könnt ihr die beiden Bibliotheken auch mit anderen Paketmanagern installieren, z.B.

```
$ pipenv install fuzzywuzzy[speedup]
```

2. Imort

[1]:
```
from fuzzywuzzy import fuzz, process
```

3. Beispiel

[2]:
```
berlin = ["Berlin, Germany", "Berlin, Deutschland", "Berlin", "Berlin, DE"]
```

String-Ähnlichkeit

Die Übereinstimmung der erstem beiden Strings `'Berlin, Germany'` und `'Berlin, Deutschland'` scheint gering:

[3]:
```
fuzz.ratio(berlin[0], berlin[1])
```

[3]:
```
65
```

Partielle String-Ähnlichkeit

Inkonsistente Teilzeichenfolgen sind ein häufiges Problem. Um dies zu umgehen, verwendet fuzzywuzzy eine Heuristik, die als *best partial* bezeichnet wird.

[4]:
```
fuzz.partial_ratio(berlin[0], berlin[1])
```

[4]:
```
60
```

Token-Sortierung

Bei der Token-Sortierung wird die betreffende Zeichenfolge mit einem Token versehen, die Token alphabetisch sortiert und anschließend wieder zu einer Zeichenfolge zusammengefügt, beispielsweise:

[5]:
```
fuzz.ratio(berlin[1], berlin[2])
```

[5]:
```
48
```

[6]:
```
fuzz.token_set_ratio(berlin[1], berlin[2])
```

[6]:
```
100
```

Weitere Informationen

[7]:
```
fuzz.ratio?
```

Extrahieren aus einer Liste

[8]:
```
choices = [
    "Germany",
    "Deutschland",
    "France",
    "United Kingdom",
    "Great Britain",
    "United States",
]
```

[9]:
```
process.extract("DE", choices, limit=2)
```

[9]:
```
[('Deutschland', 90), ('Germany', 45)]
```

[10]:
```
process.extract("Vereinigtes Königreich", choices)
```

[10]:
```
[('United Kingdom', 51),
 ('United States', 41),
 ('Germany', 39),
 ('Great Britain', 35),
 ('Deutschland', 31)]
```

[11]:
```
process.extractOne("frankreich", choices)
```

```
[11]: ('France', 62)
```

```
[12]: process.extractOne("U.S.", choices)
```

```
[12]: ('United States', 86)
```

Bekannte Ports

FuzzyWuzzy wird auch in andere Sprachen portiert! Hier einige bekannte Ports:

- Java: xpresso
- Java: xdrop fuzzywuzzy
- Rust: fuzzyrusty
- JavaScript: fuzzball.js
- C++: tmplt fuzzywuzzy
- C#: FuzzySharp
- Go: go-fuzzywuzzy
- Pascal: FuzzyWuzzy.pas
- Kotlin: FuzzyWuzzy-Kotlin
- R: fuzzywuzzyR

4.1.4 Daten deduplizieren

1. Beispieldaten laden

```
[1]: import pandas as pd
```

```
[2]: customers = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/customer_data_
↪duped.csv",
    encoding="utf-8",
)
```

2. Deduplizieren mit pandas

2.1 Überblick

```
[3]: customers
```

```
[3]:                    name                                                job  \
0      Patricia Schaefer                                Programmer, systems
1          Olivie Dubois  Ingénieur recherche et développement en agroal...
2        Mary Davies-Kirk                           Public affairs consultant
3     Miroslawa Eckbauer                                Dispensing optician
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
4          Richard Bauer                    Accountant, chartered certified
...                  ...                                                ...
2075        Maurice Stey                                 Systems developer
2076     Linda Alexander                               Commrcil horiculuri
2077        Diane Bailly                                        Pharmacien
2078   Jorge Riba Cerdán                                     Hotel manager
2079       Ryan Thompson                                Brewing technologist

                          company                          street_address  \
0                     Estrada-Best                          398 Paul Drive
1                           Moreno                        rue Lucas Benard
2                        Baker Ltd                       Flat 3\nPugh mews
3                      Ladeck GmbH                     Mijo-Lübs-Straße 12
4                     Hoffman-Rocha                     6541 Rodriguez Wall
...                             ...                                    ...
2075  Linke Margraf GmbH & Co. OHG                 Laila-Scheibe-Allee 2/0
2076     Webb, Ballald and Vasquel                          5594 Persn Ciff
2077                         Voisin                          527, rue Dijoux
2078                  Amador-Diego  Rambla de Adriana Barceló 854 Puerta 3
2079                Smith-Sullivan                      136 Rodriguez Point

                           city         state                          email  \
0                 Christianview      Delaware           lambdavid@gmail.com
1     Saint Anastasie-les-Bains            AR   berthelotjacqueline@mahe.fr
2                   Stanleyfurt            ZA    middletonconor@hotmail.com
3                Neubrandenburg        Berlin             sophia01@yahoo.de
4                   Carlosmouth         Texas         tross@jensen-ware.org
...                         ...           ...                           ...
2075                Luckenwalde       Hamburg  gutknechtevelyn@niemeier.com
2076                 Mooneybury      Maryland               ahleythoa@ail.co
2077             Duval-les-Bains            CH               aruiz@reynaud.fr
2078                     Huesca      Asturias     manuelamosquera@yahoo.com
2079             Bradfordborough  North Dakota                lcruz@gmail.com

          user_name
0          ndavidson
1        manonallain
2     colemanmichael
3        romanjunitz
4             adam78
...              ...
2075        dkreusel
2076     kennethrchn
2077      dorothee41
2078       eugenia17
2079         cnewton

[2080 rows x 8 columns]
```

2.2 Datentypen anzeigen

Hierfür verwenden wir pandas.DataFrame.dtypes:

[4]:
```
customers.dtypes
```

[4]:
```
name              object
job               object
company           object
street_address    object
city              object
state             object
email             object
user_name         object
dtype: object
```

2.3 Fehlende Werte ermitteln

pandas.isnull zeigt für ein array-ähnliches Objekt an, ob Werte fehlen:

- `NaN` in numerischen Arrays
- `None` oder `NaN` in Objekt-Arrays
- `NaT` in datetimelike

  Siehe auch:

  - notna für die boolesche Umkehrung von pandas.isna
  - Series.isna für die fehlenden Werte in einer Serie
  - DataFrame.isna für die fehlenden Werte in einem DataFrame
  - Index.isna für die fehlenden Werte in einem Index

[5]:
```
for col in customers.columns:
    print(col, customers[col].isnull().sum())
```

```
name 0
job 0
company 0
street_address 0
city 0
state 0
email 0
user_name 0
```

2.4 Duplizierte Datensätze ermitteln

```
[6]: customers.duplicated()
```

```
[6]: 0       False
1       False
2       False
3       False
4       False
        ...
2075    False
2076    False
2077    False
2078    False
2079    False
Length: 2080, dtype: bool
```

`customers.duplicated()` gibt uns noch nicht den gewünschten Hinweis, ob es doppelte Datensätze gibt. Im Folgenden lassen wir uns alle Datensätze ausgeben, für die `True` zurückgegeben wird:

```
[7]: customers[customers.duplicated()]
```

```
[7]: Empty DataFrame
Columns: [name, job, company, street_address, city, state, email, user_name]
Index: []
```

Offenbar gibt es keine duplizierten Datensätze.

2.5 Duplizierte Daten löschen

Das Löschen duplizierter Datensätze mit `drop_duplicates` sollte demnach nichts ändern und die Anzahl der Datensätze bei 2080 belassen:

```
[7]: customers.drop_duplicates()
```

```
[7]:                    name                                                job  \
0      Patricia Schaefer                            Programmer, systems
1          Olivie Dubois  Ingénieur recherche et développement en agroal...
2        Mary Davies-Kirk                    Public affairs consultant
3     Miroslawa Eckbauer                           Dispensing optician
4          Richard Bauer             Accountant, chartered certified
...                  ...                                             ...
2075        Maurice Stey                          Systems developer
2076     Linda Alexander                       Commrcil horiculuri
2077        Diane Bailly                                Pharmacien
2078   Jorge Riba Cerdán                             Hotel manager
2079       Ryan Thompson                       Brewing technologist

                        company                        street_address  \
0                   Estrada-Best                        398 Paul Drive
1                         Moreno                     rue Lucas Benard
2                     Baker Ltd                     Flat 3\nPugh mews
3                    Ladeck GmbH                    Mijo-Lübs-Straße 12
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
4                    Hoffman-Rocha                       6541 Rodriguez Wall
...                            ...                                      ...
2075  Linke Margraf GmbH & Co. OHG                 Laila-Scheibe-Allee 2/0
2076     Webb, Ballald and Vasquel                          5594 Persn Ciff
2077                         Voisin                          527, rue Dijoux
2078                  Amador-Diego  Rambla de Adriana Barceló 854 Puerta 3
2079                Smith-Sullivan                      136 Rodriguez Point

                           city         state                          email  \
0                  Christianview      Delaware            lambdavid@gmail.com
1     Saint Anastasie-les-Bains            AR    berthelotjacqueline@mahe.fr
2                    Stanleyfurt            ZA     middletonconor@hotmail.com
3                 Neubrandenburg        Berlin              sophia01@yahoo.de
4                    Carlosmouth         Texas         tross@jensen-ware.org
...                          ...           ...                            ...
2075                 Luckenwalde       Hamburg  gutknechtevelyn@niemeier.com
2076                  Mooneybury      Maryland               ahleythoa@ail.co
2077              Duval-les-Bains            CH               aruiz@reynaud.fr
2078                      Huesca      Asturias     manuelamosquera@yahoo.com
2079              Bradfordborough  North Dakota                 lcruz@gmail.com

          user_name
0          ndavidson
1        manonallain
2     colemanmichael
3        romanjunitz
4             adam78
...              ...
2075        dkreusel
2076     kennethrchn
2077      dorothee41
2078       eugenia17
2079         cnewton

[2080 rows x 8 columns]
```

Nun wollen wir nur diejenigen Datensätze löschen, deren `user_name` identisch ist:

[8]:
```
customers.drop_duplicates(["user_name"])
```

[8]:
```
                   name                                                job  \
0      Patricia Schaefer                            Programmer, systems
1          Olivie Dubois  Ingénieur recherche et développement en agroal...
2       Mary Davies-Kirk                    Public affairs consultant
3     Miroslawa Eckbauer                            Dispensing optician
4          Richard Bauer                Accountant, chartered certified
...                  ...                                             ...
2074        Rhonda James                       Recruitment consultant
2076     Linda Alexander                          Commrcil horiculuri
2077        Diane Bailly                                    Pharmacien
2078   Jorge Riba Cerdán                                  Hotel manager
2079       Ryan Thompson                       Brewing technologist
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                        company                            street_address  \
0                  Estrada-Best                              398 Paul Drive
1                        Moreno                            rue Lucas Benard
2                    Baker Ltd                           Flat 3\nPugh mews
3                   Ladeck GmbH                         Mijo-Lübs-Straße 12
4                 Hoffman-Rocha                         6541 Rodriguez Wall
...                         ...                                         ...
2074  Turner, Bradley and Scott                    28382 Stokes Expressway
2076  Webb, Ballald and Vasquel                             5594 Persn Ciff
2077                     Voisin                             527, rue Dijoux
2078               Amador-Diego  Rambla de Adriana Barceló 854 Puerta 3
2079             Smith-Sullivan                         136 Rodriguez Point

                           city          state                          email  \
0                 Christianview       Delaware           lambdavid@gmail.com
1     Saint Anastasie-les-Bains             AR  berthelotjacqueline@mahe.fr
2                   Stanleyfurt             ZA   middletonconor@hotmail.com
3                Neubrandenburg         Berlin             sophia01@yahoo.de
4                    Carlosmouth          Texas         tross@jensen-ware.org
...                          ...            ...                           ...
2074           Port Gabrielaport  New Hampshire          zroberts@hotmail.com
2076                 Mooneybury       Maryland              ahleythoa@ail.co
2077             Duval-les-Bains             CH              aruiz@reynaud.fr
2078                     Huesca       Asturias     manuelamosquera@yahoo.com
2079             Bradfordborough   North Dakota               lcruz@gmail.com

          user_name
0          ndavidson
1        manonallain
2     colemanmichael
3        romanjunitz
4             adam78
...              ...
2074      heathscott
2076     kennethrchn
2077      dorothee41
2078       eugenia17
2079         cnewton

[2029 rows x 8 columns]
```

Dies löschte 51 Datensätze.

3. Dedupe

Alternativ können wir die duplizierte Daten mit der Dedupe-Bibliothek erkennen, die ein flaches neuronales Netzwerk verwendet, um aus einem kleinen Training zu lernen.

Siehe auch

csvdedupe bietet ein Kommandozeilenwerkzeug für Dedupe.

Zudem haben dieselben Entwickler*innen parserator erstellt, mit dem ihr Textfunktionen extrahieren und eure eigenen Textextraktion trainieren könnt.

3.1 Dedupe konfigurieren

Nun definieren wir die Felder, auf die bei der Deduplizierung geachtet werden soll und erstellen ein neues `deduper`-Objekt:

[9]:
```
import os

import dedupe


customers = pd.read_csv(
    "https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/customer_data_
↪duped.csv",
    encoding="utf-8",
)
```

[10]:
```
variables = [
    {"field": "name", "type": "String"},
    {"field": "job", "type": "String"},
    {"field": "company", "type": "String"},
    {"field": "street_address", "type": "String"},
    {"field": "city", "type": "String"},
    {"field": "state", "type": "String", "has_missing": True},
    {"field": "email", "type": "String", "has_missing": True},
    {"field": "user_name", "type": "String"},
]

deduper = dedupe.Dedupe(variables)
```

Wenn der Wert eines Feldes fehlt, sollte dieser fehlende Wert als `None`-Objekt dargestellt werden. Durch `'has_missing': True` wird jedoch ein neues, zusätzliches Feld erstellt, das angibt, ob die Daten vorhanden waren oder nicht, und die fehlenden Daten werden mit Null versehen.

Siehe auch

- Missing Data

[11]:
```
deduper
```

[11]:
```
<dedupe.api.Dedupe at 0x7f03b6d92610>
```

[12]:
```
customers.shape
```

[12]:
```
(2080, 8)
```

4. Trainingsdaten erstellen

[13]:
```
deduper.prepare_training(customers.T.to_dict())
```

prepare_training initialisiert das aktive Lernen mit unseren Daten und, optional, mit vorhandenen Trainingsdaten.

T spiegelt den DataFrame über seine Diagonale, indem Zeilen als Spalten geschrieben werden und umgekehrt. Hierfür wird pandas.DataFrame.transpose verwendet.

5. Aktives Lernen

Mit dedupe.console_label könnt ihr eure Dedupe-Instanz trainieren. Wenn Dedupe ein Datensatzpaar findet, werdet ihr gebeten, es als Duplikat zu kennzeichnen. Ihr könnt hierfür die Tasten `y`, `n` und `u`, um Duplikate zu kennzeichnen. Drückt `f`, wenn ihr fertig seid.

[14]:
```
dedupe.console_label(deduper)
```

```
name : Kenneth Moore
job : Magazine journalist
company : Cross, Bell and Diaz
street_address : 75443 Lindsey Pine
city : Thompsonshire
state : Colorado
email : ashley28@rice.com
user_name : todd72

name : Kenneth Moore
job : Magazine journalist
company : Cross, Bfll anf Diaz
street_address : 753 Lindsey Pine
city : Thompsonshe
state : Colorao
email : ashey28@rice.co
user_name : todd72

0/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished
```

```
y
```

```
name : Frédérique Lejeune-Daniel
job : Technicien chimiste
company : Schmitt
street_address : chemin Denise Ferrand
city : Saint CharlotteVille
state : IE
email : jchretien@costa.com
user_name : joseph60

name : Frédérique Lejeune-Daniel
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
job : Tecce cse
company : Sctmitt
street_address : chemin Denise Ferrand
city : Saint ChalotteVille
state : IE
email : jchretien@costacom
user_name : joseph60

1/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished / (p)revious
```

```
y
```

```
name : Herr Johann Eigenwillig
job : Immigrtion officer
company : Süßebuer Hänel GmbH
street_address : Lanernplatz 0
city : Stadtsteinach
state : Thürinen
email : hemieluie@nock.com
user_name : istoll

name : Herr Johann Eigenwillig
job : Immigration officer
company : Süßebier Hänel GmbH
street_address : Langernplatz 0
city : Stadtsteinach
state : Thüringen
email : haasemarieluise@noack.com
user_name : istoll

2/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished / (p)revious
```

```
y
```

```
name : Dr. Catherine Sutton
job : Engineer, maintenance
company : Ross LLC
street_address : 13689 Morales Centers
city : North Sarah
state : New Mexico
email : lewisnicole@yahoo.com
user_name : clittle

name : Dr. Catherine Sutton
job : Enginee maintenance
company : Ross LLC
street_address : 13689 Morales Centers
city : North Sarah
state : New Mexico
email : ewinicoe@yaoo.com
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
user_name : little

3/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished / (p)revious
```

```
y
```

```
name : Andrés Franco Bravo
job : Photographer
company : Pareja-Fábregas
street_address : Cuesta Margarita Robledo 251 Piso 1
city : Granada
state : Alicante
email : fátimazamora@batlle.com
user_name : losasebastian

name : Andrés Franco Bravo
job : Photographer
company : Pare8a8Fábre8as
street_address : Cuesta Magaita Robledo 251 Piso 1
city : Granada
state : Alicante
email : fáimazamra@balle.cm
user_name : lsasebastian

4/10 positive, 0/10 negative
Do these records refer to the same thing?
(y)es / (n)o / (u)nsure / (f)inished / (p)revious
```

```
f
```

```
Finished labeling
```

Die letzten beiden verglichenen Trainingsdatensätze machen deutlich, dass wir dieses Duplikat mit unserem obigen `drop_duplicates`-Beispiel nicht gelöscht haben – `clittle` und `little` wurden als unterschiedlich erkannt.

Mit Dedupe.train werden die von euch markierten Datensatzpaare zu den Trainingsdaten hinzugefügt und das Matching-Modell aktualisiert.

Mit `index_predicates=True` berücksichtigt die Deduplizierung auch Prädikate, die auf der Indizierung der Daten beruhen.

Wenn ihr fertig seid, speichert eure Trainingsdaten mit Dedupe.write_settings.

[15]:
```python
settings_file = "csv_example_learned_settings"

if os.path.exists(settings_file):
    print("reading from", settings_file)
    with open(settings_file, "rb") as f:
        deduper = dedupe.StaticDedupe(f)
else:
    deduper.train(index_predicates=True)
    with open(settings_file, "wb") as sf:
        deduper.write_settings(sf)
```

```
reading from csv_example_learned_settings
```

Mit dedupe.Dedupe.partition werden Datensätze identifiziert, die sich alle auf dieselbe Entität beziehen, und als Tupel zurückgegeben, die eine Folge von Datensatz-IDs und Konfidenzwerten sind. Weitere Einzelheiten zum Konfidenzwert findet ihr unter dedupe.Dedupe.cluster.

[16]:
```
dupes = deduper.partition(customers.T.to_dict())
```

[17]:
```
dupes
```

[17]:
```
[((136, 1360), (1.0, 1.0)),
 ((298, 1026), (1.0, 1.0)),
 ((354, 858), (1.0, 1.0)),
 ((478, 1119), (1.0, 1.0)),
 ((938, 1890), (1.0, 1.0)),
 ((1785, 1939), (1.0, 1.0)),
 ((0,), (1.0,)),
 ((1,), (1.0,)),
 ((2,), (1.0,)),
 ((3,), (1.0,)),
 ((4,), (1.0,)),
 ...]
```

Wir können uns auch nur einzelne Einträge ausgeben lassen:

[18]:
```
dupes[0]
```

[18]:
```
((136, 1360), (1.0, 1.0))
```

Diese können wir uns dann mit pandas.DataFrame.iloc anzeigen lassen:

[19]:
```
customers.iloc[[136,1360]]
```

[19]:
```
                          name                  job  company  \
136   Frédérique Lejeune-Daniel  Technicien chimiste  Schmitt
1360  Frédérique Lejeune-Daniel            Tecce cse  Sctmitt

             street_address                  city state                email  \
136   chemin Denise Ferrand  Saint CharlotteVille    IE  jchretien@costa.com
1360  chemin Denise Ferrand   Saint ChalotteVille    IE   jchretien@costacom

     user_name
136   joseph60
1360  joseph60
```

4.1.5 Pandas DataFrame-Validierung mit Bulwark

Bulwark ist ein Paket zum eigenschaftsbasierten Testen von pandas-Dataframes. Das Projekt wurde stark von der nicht mehr unterstützten Engarde-Bibliothek beeinflusst.

1. Installation

```
$ pipenv install bulwark
Installing bulwark...
Adding bulwark to Pipfile's [packages]...
✓ Installation Succeeded
Locking [dev-packages] dependencies...
✓ Success!
Updated Pipfile.lock (0d075a)!
```

2. Verwendung

2.1 Überprüfungen

Mit dem bulwark.checks-Modul könnt ihr viele gängige Annahmen überprüfen, z.B.

- `has_columns` überprüft, ob bestimmte Spalten so oder so ähnlich vorhanden und in der richtigen Reihenfolge sind
- `has_dtypes` überprüft die Datentypen von Spalten
- `has_no_infs` überprüft, ob keine numpy.inf im DataFrame vorhanden sind
- `has_no_nans` überprüft, ob es keine numpy.nan im DataFrame vorhanden sind
- `has_set_within_vals` überprüft, ob die in einem dict angegebenen Werte eine Teilmenge der zugehörigen Spalte sind
- `has_unique_index` überprüft, ob der Index eindeutig ist
- `is_monotonic` überprüft, ob Werte einer Spalte aufsteigend oder absteigend sind
- `one_to_many` überprüft, ob zwischen zwei Spalten eine n:1-Beziehung besteht

Die Überprüfungen sind dann sehr simpel, z.B. der Check, ob in der Spalte `pipe` keine `numpy.nan` vorhanden sind mit

```
import bulwark.checks as ck


df.pipe(ck.has_no_nans())
```

2.2 Decorators

Für jeden Check erstellt bulwark.decorators, z.B. `@dc.IsShape((-1, 10))` oder `@dc.IsMonotonic(strict=True)`.

CustomCheck

Ihr könnt auch eure eigenen benutzerdefinierten Funktionen erstellen, z.B.:

[1]:
```
import bulwark.checks as ck
import bulwark.decorators as dc
import numpy as np
import pandas as pd


def len_longer_than(df, l):
    if len(df) <= l:
        raise AssertionError("df is not as long as expected.")
    return df


@dc.CustomCheck(len_longer_than, 10)
def append_a_df(df, df2):
    return pd.concat([df, df2], ignore_index=True)


df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})

append_a_df(df, df2)
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[1], line 21
     18 df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
     19 df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})
---> 21 append_a_df(df, df2)

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
↪decorators.py:81, in CustomCheck.__call__.<locals>.decorated(*args, **kwargs)
     78 df = f(*args, **kwargs)
     79 if self.enabled:
     80     # differs from BaseDecorator
---> 81     ck.custom_check(df, self.check_func, **self.check_func_params)
     82 return df

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
↪checks.py:588, in custom_check(df, check_func, *args, **kwargs)
    576 """Assert that `check(df, *args, **kwargs)` is true.
    577
    578 Args:
   (...)
    585
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    586 """
    587 try:
--> 588     check_func(df, *args, **kwargs)
    589 except AssertionError as e:
    590     msg = "{} is not true.".format(check_func.__name__)

Cell In[1], line 9, in len_longer_than(df, l)
      7 def len_longer_than(df, l):
      8     if len(df) <= l:
----> 9         raise AssertionError("df is not as long as expected.")
     10     return df

AssertionError: len_longer_than is not true.
```

MultiCheck

Mit `MultiCheck` könnt ihr mehrere Tests gleichzeitig ausführen und alle Fehler auf einmal sehen, z.B.:

[2]:
```
@dc.MultiCheck(
    checks={ck.has_no_nans: {"columns": None}, len_longer_than: {"l": 6}},
    warn=False,
)
def append_a_df(df, df2):
    return df.append(df2, ignore_index=True)


df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})

append_a_df(df, df2)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_55151/1437634915.py in ?()
      8
      9 df = pd.DataFrame({"a": [1, 2, 3], "b": [4, 5, 6]})
     10 df2 = pd.DataFrame({"a": [1, np.nan, 3, 4], "b": [4, 5, 6, 7]})
     11
---> 12 append_a_df(df, df2)

~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/bulwark/
↪decorators.py in ?(*args, **kwargs)
     20         @functools.wraps(f)
     21         def decorated(*args, **kwargs):
---> 22             df = f(*args, **kwargs)
     23             if self.enabled:
     24                 self.check_func(df, **self.check_func_params)
     25             return df

/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_55151/1437634915.py in ?(df,␣
↪df2)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
      2     checks={ck.has_no_nans: {"columns": None}, len_longer_than: {"l": 6}},
      3     warn=False,
      4 )
      5 def append_a_df(df, df2):
----> 6     return df.append(df2, ignore_index=True)

~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/pandas/core/
↪generic.py in ?(self, name)
   5985             and name not in self._accessors
   5986             and self._info_axis._can_hold_identifiers_and_holds_name(name)
   5987         ):
   5988             return self[name]
-> 5989         return object.__getattribute__(self, name)

AttributeError: 'DataFrame' object has no attribute 'append'
```

4.1.6 Hypothesis: *Property* -basiertes Testen

In diesem Notebook verwenden wir *Property* -basierte Tests, um Probleme in unserem Code zu finden. Hypothesis ist eine Bibliothek, die Haskells Quickcheck ähnelt. Später lernen wir sie zusammen mit anderen Testbibliotheken noch genauer kennen: Hypothesis. Hypothesis kann auch Mock-Objekte und Tests für Numpy-Datentypen bereitstellen.

1. Importe

[1]:
```
import re

from hypothesis import assume, given
from hypothesis.strategies import emails, integers, tuples
```

2. Bereich finden

[2]:
```
def calculate_range(tuple_obj):
    return max(tuple_obj) - min(tuple_obj)
```

3. Test mit `strategies` und `given`

Mit hypothesis.strategies könnt ihr unterschiedliche Testdaten erstellen. Hierfür beitet Hypothesis Strategien für die meisten Typen und Argumente schränken die Möglichkeiten ein um sie euren Erfordernissen anzupassen. Im Beispiel unten verwenden wir die integers-Strategie, die mit dem Python-Decorator `@given` auf die Funktion angewendet wird. Genauer nimmt er unsere Testfunktion und wandelt sie in eine parametrisierte um sie über weite Bereiche passender Daten auszuführen:

[3]:
```
@given(tuples(integers(), integers(), integers()))
def test_calculate_range(tup):
    result = calculate_range(tup)
    assert isinstance(result, int)
    assert result > 0
```

[4]:
```
test_calculate_range()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[4], line 1
----> 1 test_calculate_range()

Cell In[3], line 2, in test_calculate_range()
      1 @given(tuples(integers(), integers(), integers()))
----> 2 def test_calculate_range(tup):
      3     result = calculate_range(tup)
      4     assert isinstance(result, int)

    [... skipping hidden 1 frame]

Cell In[3], line 5, in test_calculate_range(tup)
      3 result = calculate_range(tup)
      4 assert isinstance(result, int)
----> 5 assert result > 0

AssertionError:
Falsifying example: test_calculate_range(
    tup=(0, 0, 0),
)
```

Nun korrigieren wir den Test mit >= und überprüfen ihn erneut:

[5]:
```
@given(tuples(integers(), integers()))
def test_calculate_range(tup):
    result = calculate_range(tup)
    assert isinstance(result, int)
    assert result >= 0
```

[6]:
```
test_calculate_range()
```

3. Gegen Reguläre Ausdrücke prüfen

Mit regulären Ausrücken (engl.: *regular expressions*) lassen sich Zeichenketten auf bestimmte syntaktische Regeln überprüfen. In Python könnt ihr zum Überprüfen regulärer Ausdrücke re.match verwenden.

> **Hinweis:**
>
> Auf der Website regex101 könnt ihr zunächst eure regulären Ausdrücke ausprobieren.

Als Beispiel versuchen wir, aus E-Mail-Adressen `username` und die `domain` zu ermitteln:

[7]:
```
def parse_email(email):
    result = re.match("(?P<username>\w+).(?P<domain>[\w\.]+)", email).groups()
    return result
```

Nun schreiben wir einen Test `test_parse_email` zum Überprüfen unserer Methode. Als Eingabewerte verwenden wir die emails-Strategie von Hypothesis. Als `result` erwarten wir z.B.:

```
('0', 'A.com')
('F', 'j.EeHNqsx')
...
```

Im Test nehmen wir einerseits an, dass immer zwei Einträge zurückgegeben werden und im zweiten Eintrag ein Punkt (`.`) vorkommt.

[8]:
```
@given(emails())
def test_parse_email(email):
    result = parse_email(email)
    # print(result)
    assert len(result) == 2
    assert '.' in result[1]
```

[9]:
```
test_parse_email()
```

```
---------------------------------------------------------------------------
ExceptionGroup                            Traceback (most recent call last)
Cell In[9], line 1
----> 1 test_parse_email()

Cell In[8], line 2, in test_parse_email()
      1 @given(emails())
----> 2 def test_parse_email(email):
      3     result = parse_email(email)
      4     # print(result)

File ~/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
↪hypothesis/core.py:1374, in given.<locals>.run_test_as_given.<locals>.wrapped_
↪test(*arguments, **kwargs)
   1361         # The dance here is to avoid showing users long tracebacks
   1362         # full of Hypothesis internals they don't care about.
   1363         # We have to do this inline, to avoid adding another
   (...)
   1367         # which will actually appear in tracebacks is as clear as
   1368         # possible - "raise the_error_hypothesis_found".
   1369         the_error_hypothesis_found = e.with_traceback(
   1370             None
   1371             if isinstance(e, BaseExceptionGroup)
   1372             else get_trimmed_traceback()
   1373         )
-> 1374         raise the_error_hypothesis_found
   1376 if not (ran_explicit_examples or state.ever_executed):
   1377     raise SKIP_BECAUSE_NO_EXAMPLES

ExceptionGroup: Hypothesis found 2 distinct failures. (2 sub-exceptions)
```

Mit Hypothesis wurden zwei Beispiele gefunden, die deutlich machen, dass unser regulärer Ausdruck in der `parse_email`-Methode noch nicht hinreichend ist: `0/0@A.ac` und `/@A.ac`. Nachdem wir unseren regulären Ausdruck entsprechend angepasst haben, können wir den Test erneut aufrufen:

[10]:
```
def parse_email(email):
    result = re.match(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "(?P<username>[\.\w\-\!~#$%&\|{}\+\/\^\`\=\*']+).(?P<domain>[\w\.\-]+)",
        email,
    ).groups()
    return result
```

[11]:
```
test_parse_email()
```

4.1.7 TDDA: Test-Driven Data Analysis

TDDA verwendet Dateieingaben (wie NumPy-Arrays oder Pandas DataFrames) und eine Reihe von Einschränkungen (engl.: *constraints*), die als JSON-Datei gespeichert werden.

- Reference Test unterstützt die Erstellung von Referenztests, die entweder auf `unittest` oder `pytest` basieren.
- Constraints wird verwendet, um Constraints aus einem (Pandas)-DataFrame zu ermitteln, sie als JSON auszuschreiben und zu überprüfen, ob Datensätze die Constraints in der Constraints-Datei erfüllen. Es unterstützt auch Tabellen in einer Vielzahl von relationalen Datenbanken.
- Rexpy ist ein Werkzeug zur automatischen Ableitung von regulären Ausdrücken aus einer Spalte in einem Pandas DataFrame oder aus einer (Python)-Liste von Beispielen.

1. Importe

[1]:
```
import pandas as pd
import numpy as np

from tdda.constraints import discover_df, verify_df
```

[2]:
```
df = pd.read_csv("https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/
↪iot_example.csv")
```

2. Daten überprüfen

Mit pandas.DataFrame.sample lassen wir uns die ersten zehn Datensätze anzeigen:

[3]:
```
df.sample(10)
```

[3]:
```
                  timestamp        username  temperature  heartrate  \
123221  2017-02-19T17:27:43  wilsonpatricia           20         60
80684   2017-02-02T18:32:06         vmiller           27         79
108810  2017-02-13T23:50:44    sandraobrien            9         73
47330   2017-01-20T10:00:25           rcline           18         63
25842   2017-01-11T19:32:20        colton99           20         72
31413   2017-01-14T01:06:46        zjimenez           11         89
115347  2017-02-16T14:11:28         alvin94           23         87
138131  2017-02-25T16:45:28       chelsea05            7         66
97693   2017-02-09T13:31:43    thomasknight           18         66
33904   2017-01-15T01:02:00         paulwall            8         66

                                    build  latest   note
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
123221  2452cba2-623a-9aaf-4620-b1a00a707088       0    NaN
80684   877d2246-f54d-2d6b-8991-5698199def39       1   user
108810  2c185e6b-ad8a-bbc3-5b8a-a7f0cff3b3b5       1   user
47330   c8b03e53-db4a-bbb7-1d06-efb20eca17ec       0    NaN
25842   1d99cb10-0189-3bc2-d7e3-4fee382387ef       0  sleep
31413   eb2bfbd3-6a61-cfae-afcc-6879319aebad       1   user
115347  9924687d-959e-99cf-6510-712904df2583       0   wake
138131  f995acb5-fff8-3ff7-0581-152e81988b81       1   user
97693   a9180bc3-90a3-88bf-36e5-a67549147b28       1   user
33904   56b09285-495a-23ef-31f7-489fad16096f       1  sleep
```

Und mit pandas.DataFrame.dtypes lassen wir uns die Datentypen für die einzelnen Spalten anzeigen:

[4]:
```
df.dtypes
```

[4]:
```
timestamp      object
username       object
temperature     int64
heartrate       int64
build          object
latest          int64
note           object
dtype: object
```

3. Erstellen eines *constraints*-Objekt

Mit `discover_constraints` kann ein Vonstraints-Objekt erzeugt werden.

[5]:
```
constraints = discover_df(df)
```

[6]:
```
constraints
```

[6]:
```
<tdda.constraints.base.DatasetConstraints at 0x144fd7250>
```

[7]:
```
constraints.fields
```

[7]:
```
Fields([('timestamp', <tdda.constraints.base.FieldConstraints at 0x15fec8790>),
        ('username', <tdda.constraints.base.FieldConstraints at 0x15fec8bd0>),
        ('temperature',
         <tdda.constraints.base.FieldConstraints at 0x15fec9050>),
        ('heartrate', <tdda.constraints.base.FieldConstraints at 0x15fec9750>),
        ('build', <tdda.constraints.base.FieldConstraints at 0x15fec9b10>),
        ('latest', <tdda.constraints.base.FieldConstraints at 0x15feca390>),
        ('note', <tdda.constraints.base.FieldConstraints at 0x15feca710>)])
```

4. Schreiben der *Constraints* in eine Datei

[8]:
```
with open("ignore-iot_constraints.tdda", "w") as f:
    f.write(constraints.to_json())
```

Wenn wir uns die Datei genauer betrachten können wir erkennen, dass z.B. für die `timestamp`-Spalte eine Zeichenkette mit 19 Zeichen erwartet wird und `temperature` Integer mit Werten von 5–29 erwartet.

[9]:
```
cat ignore-iot_constraints.tdda
```

```
{
    "creation_metadata": {
        "local_time": "2023-08-19 13:13:48",
        "utc_time": "2023-08-19 11:11:48",
        "creator": "TDDA 2.0.09",
        "host": "fay.fritz.box",
        "user": "veit",
        "n_records": 146397,
        "n_selected": 146397
    },
    "fields": {
        "timestamp": {
            "type": "string",
            "min_length": 19,
            "max_length": 19,
            "max_nulls": 0,
            "no_duplicates": true
        },
        "username": {
            "type": "string",
            "min_length": 3,
            "max_length": 21,
            "max_nulls": 0
        },
        "temperature": {
            "type": "int",
            "min": 5,
            "max": 29,
            "sign": "positive",
            "max_nulls": 0
        },
        "heartrate": {
            "type": "int",
            "min": 60,
            "max": 89,
            "sign": "positive",
            "max_nulls": 0
        },
        "build": {
            "type": "string",
            "min_length": 36,
            "max_length": 36,
            "max_nulls": 0,
            "no_duplicates": true
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        },
        "latest": {
            "type": "int",
            "min": 0,
            "max": 1,
            "sign": "non-negative",
            "max_nulls": 0
        },
        "note": {
            "type": "string",
            "min_length": 4,
            "max_length": 8,
            "allowed_values": [
                "interval",
                "sleep",
                "test",
                "update",
                "user",
                "wake"
            ]
        }
    }
}
```

5. Überprüfen von Dataframes

Hierfür lesen wir zunächst eine neue csv-Datei mit Pandas ein und lassen uns dann zehn Datensätze exemplarisch ausgeben:

[10]:
```
new_df = pd.read_csv("https://raw.githubusercontent.com/kjam/data-cleaning-101/master/
↪data/iot_example_with_nulls.csv")

new_df.sample(10)
```

[10]:

	timestamp	username	temperature	heartrate	\
126044	2017-02-20T20:33:42	karenrichards	NaN	79	
54844	2017-01-23T10:00:53	karen37	NaN	76	
15484	2017-01-07T16:36:43	carterjill	24.0	61	
3709	2017-01-02T23:50:12	ebenton	18.0	82	
131978	2017-02-23T05:37:46	cameron67	NaN	87	
61302	2017-01-25T23:57:04	rebecca88	19.0	71	
130493	2017-02-22T15:15:56	cgriffin	29.0	81	
121685	2017-02-19T02:53:16	johnberg	16.0	68	
87150	2017-02-05T08:17:46	kelly71	28.0	68	
138723	2017-02-25T22:22:49	ayoung	22.0	75	

	build	latest	note
126044	284fab65-9fcc-18e4-8838-6e89ac938f77	NaN	NaN
54844	6ea2310d-b136-dfae-3e4a-730cb01a6881	1.0	wake
15484	3b524f97-4a6a-156e-e182-760818cc5c6b	0.0	interval
3709	f23b7b48-0ad1-18b5-c5a8-033d66d47007	1.0	NaN

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
131978  2806ee39-a668-d0f7-44b9-9255188d51df     1.0  interval
61302                                    NaN     NaN  interval
130493                                   NaN     1.0       NaN
121685  e3b81408-7c4f-78b5-c373-2ee086e6dbbf     NaN       NaN
87150   99842995-cc89-638c-9067-a7d92e450097     0.0  interval
138723  dca3e5f6-c05a-c490-8384-4f7cfda4e19a     0.0       NaN
```

Wir sehen mehrere Felder, die als `NaN` ausgegeben werden. Um dies nun systematisch zu analysieren, wenden wir verify_df auf unseren neuen DataFrame an. Dabei gibt `passes` gibt die Anzahl der bestandenen, `failures` die Anzahl der fehlgeschlagenen Constraints zurück.

[11]:
```
v = verify_df(new_df, "ignore-iot_constraints.tdda")
```

[12]:
```
v
```

[12]:
```
<tdda.constraints.pd.constraints.PandasVerification at 0x174786fd0>
```

[13]:
```
v.passes
```

[13]:
```
30
```

[14]:
```
v.failures
```

[14]:
```
3
```

Wir können uns auch anzeigen lassen, in welchen Spalten welche Constraints bestanden und fehlgeschlagen sind:

[15]:
```
print(str(v))
```

```
FIELDS:

timestamp: 0 failures  5 passes  type   min_length   max_length   max_nulls   no_
↪duplicates

username: 0 failures  4 passes  type   min_length   max_length   max_nulls

temperature: 1 failure  4 passes  type   min   max   sign   max_nulls

heartrate: 0 failures  5 passes  type   min   max   sign   max_nulls

build: 1 failure  4 passes  type   min_length   max_length   max_nulls   no_duplicates

latest: 1 failure  4 passes  type   min   max   sign   max_nulls

note: 0 failures  4 passes  type   min_length   max_length   allowed_values

SUMMARY:

Constraints passing: 30
Constraints failing: 3
```

Alternativ können wir uns diese Ergebnisse auch tabellarisch anzeigen lassen:

[16]:
```
v.to_frame()
```

[16]:
```
         field  failures  passes  type   min min_length   max max_length  \
0    timestamp         0       5  True   NaN       True   NaN       True
1     username         0       4  True   NaN       True   NaN       True
2  temperature         1       4  True  True        NaN  True        NaN
3    heartrate         0       5  True  True        NaN  True        NaN
4        build         1       4  True   NaN       True   NaN       True
5       latest         1       4  True  True        NaN  True        NaN
6         note         0       4  True   NaN       True   NaN       True

   sign max_nulls no_duplicates allowed_values
0   NaN      True          True            NaN
1   NaN      True           NaN            NaN
2  True     False           NaN            NaN
3  True      True           NaN            NaN
4   NaN     False          True            NaN
5  True     False           NaN            NaN
6   NaN       NaN           NaN           True
```

4.1.8 Datenvalidierung mit Voluptuous (Schemadefinitionen)

In diesem Notebook verwenden wir Voluptuous, um Schemata für unsere Daten zu definieren. Wir können dann die Schemaprüfung an verschiedenen Stellen unserer Bereinigung verwenden, um sicherzustellen, dass wir die Kriterien erfüllen. Schließllich können wir Ausnahmen für die Schemaüberprüfung verwenden, um unreine oder ungültige Daten zu markieren, beiseite zu legen oder zu entfernen.

Siehe auch

- Validr
- marshmallow

1. Importe

[1]:
```
import logging

from datetime import datetime

import pandas as pd

from voluptuous import ALLOW_EXTRA, All, Range, Required, Schema
from voluptuous.error import Invalid, MultipleInvalid
```

- `Required` markiert den Knoten eines Schemas als erforderlich und gibt optional einen Standardwert an, siehe auch voluptuous.schema_builder.Required.
- `Range` begrenzt den Wert auf einen Bereich, wobei entweder `min` oder `max` weggelassen werden kann; siehe auch voluptuous.validators.Range.
- `ALL` wird für feldübergreifende Validierungen verwendet: prüft die Grundstruktur der Daten in einem ersten Durchgang und erst im zweiten Durchgang wird die feldübergreifende Validierung angewendet; siehe auch voluptuous.validators.All.
- `ALLOW_EXTRA` erlaubt zusätzliche Wörterbuchschlüssel

- `MultipleInvalid` basiert auf `Invalid`, siehe auch voluptuous.error.MultipleInvalid.
- `Invalid` kennzeichnet Daten als ungültig, siehe auch voluptuous.error.Invalid.

2. Logger

[2]:
```
logger = logging.getLogger(0)

logger.setLevel(logging.WARNING)
```

3. Beispieldaten lesen

[3]:
```
sales = pd.read_csv('https://raw.githubusercontent.com/kjam/data-cleaning-101/master/
↪data/sales_data.csv')
```

4. Daten untersuchen

[4]:
```
sales.head()
```

[4]:
```
   Unnamed: 0            timestamp          city  store_id  sale_number  \
0           0  2018-09-10 05:00:45  Williamburgh         6         1530
1           1  2018-09-12 10:01:27    Ibarraberg         1         2744
2           2  2018-09-13 12:01:48   Sarachester         2         1908
3           3  2018-09-14 20:02:19  Caldwellbury        14          771
4           4  2018-09-16 01:03:21     Erikaland        11         1571

   sale_amount         associate
0       1167.0          Gary Lee
1        258.0      Daniel Davis
2        266.0      Michael Roth
3       -108.0  Michaela Stewart
4       -372.0       Mark Taylor
```

[5]:
```
sales.dtypes
```

[5]:
```
Unnamed: 0       int64
timestamp       object
city            object
store_id         int64
sale_number      int64
sale_amount    float64
associate       object
dtype: object
```

5. Schema definieren

In der Spalte `sale_amount` sollen alle Werte zwischen 2,5 and 1450,99 sein.

[6]:
```
schema = Schema(
    {
        Required("sale_amount"): All(float, Range(min=2.50, max=1450.99)),
    },
    extra=ALLOW_EXTRA,
)
```

[7]:
```
error_count = 0
for s_id, sale in sales.T.to_dict().items():
    try:
        schema(sale)
    except MultipleInvalid as e:
        logging.warning(
            "issue with sale: %s (%s) - %s", s_id, sale["sale_amount"], e
        )
        error_count += 1
```

```
WARNING:root:issue with sale: 3 (-108.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 4 (-372.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 5 (-399.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 6 (-304.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 7 (-295.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 10 (-89.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 13 (-303.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 15 (-432.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 19 (-177.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 20 (-154.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 22 (-130.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 23 (1487.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 25 (-145.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 28 (1471.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 31 (-259.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 38 (-241.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 40 (-4.0) - value must be at least 2.5 for dictionary␣
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪value @ data['sale_amount']
WARNING:root:issue with sale: 41 (1581.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 45 (1529.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 46 (-238.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 48 (-284.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 51 (-164.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 55 (-184.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 56 (-304.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 59 (1579.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 60 (-455.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 63 (1551.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 65 (-397.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 69 (-400.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 70 (1482.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 71 (-321.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 74 (-47.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 76 (-68.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 86 (1454.0) - value must be at most 1450.99 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 101 (-213.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 103 (-144.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 104 (-265.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 107 (-349.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 111 (-78.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 112 (-310.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 116 (1570.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 120 (1490.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 123 (-179.0) - value must be at least 2.5 for dictionary␣
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪value @ data['sale_amount']
WARNING:root:issue with sale: 124 (-391.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 129 (1504.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 130 (-91.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 132 (-372.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 141 (1512.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 142 (-449.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 149 (1494.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 152 (-405.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 155 (1599.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 156 (1527.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 157 (-462.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 162 (-358.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 164 (-78.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 167 (-358.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 171 (-391.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 178 (-304.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 180 (-9.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 187 (1475.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 194 (-433.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 195 (-329.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 196 (-147.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 203 (-319.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 206 (-132.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 207 (-20.0) - value must be at least 2.5 for dictionary␣
↪value @ data['sale_amount']
WARNING:root:issue with sale: 209 (1539.0) - value must be at most 1450.99 for␣
↪dictionary value @ data['sale_amount']
WARNING:root:issue with sale: 211 (-167.0) - value must be at least 2.5 for dictionary␣
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪value @ data['sale_amount']
```

Um die Elemente einer Spalte als Schlüssel und die Elemente einer anderen Spalte als Werte verwenden zu können, machen wir einfach die gewünschte Spalte zum Index des DataFrame und transponieren sie mit der Funktion `.T()`; siehe auch pandas.DataFrame.transpose.

[8]:
```
error_count
```

[8]:
```
69
```

Aktuell wissen wir jedoch noch nicht, ob

- wir ein falsch definiertes Schema haben
- möglicherweise negative Werte zurückgegeben oder falsch markiert werden
- höhere Werte kombinierte Einkäufe oder Sonderverkäufe sind

6. Hinzufügen einer benutzerdefinierten Validierung

[9]:
```
def ValidDate(fmt="%Y-%m-%d %H:%M:%S"):
    return lambda v: datetime.strptime(v, fmt)
```

[10]:
```
schema = Schema(
    {
        Required("timestamp"): All(ValidDate()),
    },
    extra=ALLOW_EXTRA,
)
```

[11]:
```
error_count = 0
for s_id, sale in sales.T.to_dict().items():
    try:
        schema(sale)
    except MultipleInvalid as e:
        logging.warning(
            "issue with sale: %s (%s) - %s", s_id, sale["timestamp"], e
        )
        error_count += 1
```

[12]:
```
error_count
```

[12]:
```
0
```

7. Gültige Datumsstrukturen sind noch keine gültigen Daten

[13]:
```
def ValidDate(fmt="%Y-%m-%d %H:%M:%S"):
    def validation_func(v):
        try:
            assert datetime.strptime(v, fmt) <= datetime.now()
        except AssertionError:
            raise Invalid("date is in the future! %s" % v)

    return validation_func
```

[14]:
```
schema = Schema(
    {
        Required("timestamp"): All(ValidDate()),
    },
    extra=ALLOW_EXTRA,
)
```

[15]:
```
error_count = 0
for s_id, sale in sales.T.to_dict().items():
    try:
        schema(sale)
    except MultipleInvalid as e:
        logging.warning(
            "issue with sale: %s (%s) - %s", s_id, sale["timestamp"], e
        )
        error_count += 1
```

[16]:
```
error_count
```

[16]:
```
0
```

4.1.9 Normalisierung und Vorverarbeitung

sklearn.preprocessing kann in vielfältiger Form verwendet werden um Daten zu bereinigen:

- Standardisierung mit StandardScaler, MinMaxScaler, MaxAbsScaler oder RobustScaler.
- Zentrierung von Kernel-Matrizen mit KernelCenterer.
- Nicht-lineare Transformationen mit QuantileTransformer, PowerTransformer
- Normalisierung mit normalize.
- Kodierung kategorischer Merkmale mit OrdinalEncoder, OneHotEncoder.
- Diskretisierung (auch bekannt als Quantisierung oder Binning) mit KBinsDiscretizer.
- Binarisierung von Merkmalen mit Binarizer
- Zurechnen (engl.: *Imputation*) fehlender Werte mit SimpleImputer, IterativeImputer oder KNNImputer wobei die zugerechneten Werte mit MissingIndicator markiert werden können.

Siehe auch

- statsmodels

Beispiel

Im folgenden Beispiel füllen wir Mittelwerte auf und nehmen einige Skalierungen vor:

1. Importe

[1]:
```
from datetime import datetime

import numpy as np
import pandas as pd

from sklearn import preprocessing
from sklearn.impute import SimpleImputer
```

[2]:
```
hvac = pd.read_csv("https://raw.githubusercontent.com/kjam/data-cleaning-101/master/data/
↪HVAC_with_nulls.csv")
```

2. Datenqualität überprüfen

Datentypen mit pandas.DataFrame.dtypes anzeigen

[3]:
```
hvac.dtypes
```

[3]:
```
Date           object
Time           object
TargetTemp    float64
ActualTemp      int64
System          int64
SystemAge     float64
BuildingID      int64
10            float64
dtype: object
```

Dimensionen das DataFrame mit pandas.DataFrame.shape als Tupel zurückgeben

[4]:
```
hvac.shape
```

[4]:
```
(8000, 8)
```

Erste *n* Zeilen mit pandas.DataFrame.head zurückgeben

[5]:
```
hvac.head()
```

[5]:

	Date	Time	TargetTemp	ActualTemp	System	SystemAge	BuildingID	10
0	6/1/13	0:00:01	66.0	58	13	20.0	4	NaN
1	6/2/13	1:00:01	NaN	68	3	20.0	17	NaN
2	6/3/13	2:00:01	70.0	73	17	20.0	18	NaN
3	6/4/13	3:00:01	67.0	63	2	NaN	15	NaN
4	6/5/13	4:00:01	68.0	74	16	9.0	3	NaN

3. Fehlenden Werten den Mittelwert zuschreiben

Hierzu verwenden wir die mean-Strategie von sklearn.impute.SimpleImputer

[6]:
```
imp = SimpleImputer(missing_values=np.nan, strategy="mean")
```

[7]:
```
hvac_numeric = hvac[["TargetTemp", "SystemAge"]]
```

[8]:
```
imp = imp.fit(hvac_numeric.loc[:10])
```

Weiter Infos zu `fit` erhaltet ihr in der Scikit Learn-Dokumentation.

fit_transform wandelt dann die angepassten Daten um:

[9]:
```
transformed = imp.fit_transform(hvac_numeric)
```

[10]:
```
transformed
```

[10]:
```
array([[66.        , 20.        ],
       [67.50773481, 20.        ],
       [70.        , 20.        ],
       ...,
       [67.50773481,  4.        ],
       [65.        , 23.        ],
       [66.        , 21.        ]])
```

[11]:
```
hvac["TargetTemp"], hvac["SystemAge"] = transformed[:,0], transformed[:,1]
```

Nun lassen wir uns die ersten Zeilen mit den geänderten Datensätzen anzeigen.

[12]:
```
hvac.head()
```

[12]:

	Date	Time	TargetTemp	ActualTemp	System	SystemAge	BuildingID	10
0	6/1/13	0:00:01	66.000000	58	13	20.000000	4	NaN
1	6/2/13	1:00:01	67.507735	68	3	20.000000	17	NaN
2	6/3/13	2:00:01	70.000000	73	17	20.000000	18	NaN
3	6/4/13	3:00:01	67.000000	63	2	15.386643	15	NaN
4	6/5/13	4:00:01	68.000000	74	16	9.000000	3	NaN

4. Skalieren

Zur Standardisierung von Datensätzen, die wie standardnormalverteilte Daten aussehen, können wir sklearn.preprocessing.scale verwenden. Damit lassen sich die Faktoren ermitteln, um die ein Wert sich vergrößert oder verkleinert. Dies können wir für die Skalierung der aktuellen Temperatur verwenden.

[13]:
```
hvac["ScaledTemp"] = preprocessing.scale(hvac["ActualTemp"])
```

[14]:
```
hvac["ScaledTemp"].head()
```

[14]:
```
0   -1.293272
1    0.048732
2    0.719733
3   -0.622270
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
4    0.853934
Name: ScaledTemp, dtype: float64
```

sklearn.preprocessing.MinMaxScaler skaliert die Mermale so, dass sie zwischen einem bestimmten Minimal- und Maximalwert liegen, häufig zwischen Null und Eins. Dies hat den Vorteil, dass die Skalierung robuster gegenüber sehr kleinen Standardabweichungen von Merkmalen wird.

[15]:
```
min_max_scaler = preprocessing.MinMaxScaler()
```

[16]:
```
temp_minmax = min_max_scaler.fit_transform(hvac[["ActualTemp"]])
```

[17]:
```
temp_minmax
```

[17]:
```
array([[0.12],
       [0.52],
       [0.72],
       ...,
       [0.56],
       [0.32],
       [0.44]])
```

Nun fügen wir auch `temp_minmax` noch als neue Spalte ein:

[18]:
```
hvac["MinMaxScaledTemp"] = temp_minmax[:,0]
hvac["MinMaxScaledTemp"].head()
```

[18]:
```
0    0.12
1    0.52
2    0.72
3    0.32
4    0.76
Name: MinMaxScaledTemp, dtype: float64
```

[19]:
```
hvac.head()
```

[19]:
```
     Date     Time  TargetTemp  ActualTemp  System  SystemAge  BuildingID  10  \
0  6/1/13  0:00:01   66.000000          58      13  20.000000           4 NaN
1  6/2/13  1:00:01   67.507735          68       3  20.000000          17 NaN
2  6/3/13  2:00:01   70.000000          73      17  20.000000          18 NaN
3  6/4/13  3:00:01   67.000000          63       2  15.386643          15 NaN
4  6/5/13  4:00:01   68.000000          74      16   9.000000           3 NaN

   ScaledTemp  MinMaxScaledTemp
0   -1.293272              0.12
1    0.048732              0.52
2    0.719733              0.72
3   -0.622270              0.32
4    0.853934              0.76
```

4.1.10 Satelitendaten Geostandorten zuordnen

Beispiel: Tracking der Internationalen Raumstation mit Dask

In diesem Notebook werden wir zwei APIs verwenden:

1. Google Maps Geocoder
2. Open Notify API for ISS location

Wir werden sie verwenden, um den ISS-Standort und die nächste Durchlaufzeit in Bezug auf eine Liste von Städten zu verfolgen. Um unsere Diagramme zu erstellen und Daten intelligent zu parallelisieren, verwenden wir Dask, insbesondere *Dask Delayed*.

1. Importe

[1]:
```
import logging
import sys

from datetime import datetime
from math import radians
from operator import itemgetter
from time import sleep

import numpy as np
import requests

from dask import delayed
from sklearn.metrics import DistanceMetric
```

2. Logger

[2]:
```
logger = logging.getLogger()

logger.setLevel(logging.INFO)
```

3. Latitude- und Longitude-Paare aus einer Liste von Städten

s.a. Location APIs

[3]:
```
def get_lat_long(address):
    resp = requests.get(
        "https://eu1.locationiq.org/v1/search.php",
        params={"key": "92e7ba84cf3465", "q": address, "format": "json"},
    )
    if resp.status_code != 200:
        print("There was a problem with your request!")
        print(resp.content)
        return
    data = resp.json()[0]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    return {
        "name": data.get("display_name"),
        "lat": float(data.get("lat")),
        "long": float(data.get("lon")),
    }
```

[4]:
```
get_lat_long('Berlin, Germany')
```

[4]:
```
{'name': 'Berlin, 10117, Germany', 'lat': 52.5170365, 'long': 13.3888599}
```

[5]:
```
locations = []

for city in [
    "Seattle, Washington",
    "Miami, Florida",
    "Berlin, Germany",
    "Singapore",
    "Wellington, New Zealand",
    "Beirut, Lebanon",
    "Beijing, China",
    "Nairobi, Kenya",
    "Cape Town, South Africa",
    "Buenos Aires, Argentina",
]:
    locations.append(get_lat_long(city))
    sleep(2)
```

[6]:
```
locations
```

[6]:
```
[{'name': 'Seattle, King County, Washington, USA',
  'lat': 47.6038321,
  'long': -122.3300624},
 {'name': 'Miami, Miami-Dade County, Florida, USA',
  'lat': 25.7741728,
  'long': -80.19362},
 {'name': 'Berlin, 10117, Germany', 'lat': 52.5170365, 'long': 13.3888599},
 {'name': 'Singapore', 'lat': 1.357107, 'long': 103.8194992},
 {'name': 'Wellington, Wellington City, Wellington, 6011, New Zealand',
  'lat': -41.2887953,
  'long': 174.7772114},
 {'name': 'Beirut, Beirut Governorate, Lebanon',
  'lat': 33.8959203,
  'long': 35.47843},
 {'name': 'Beijing, Dongcheng District, Beijing, 100010, China',
  'lat': 39.906217,
  'long': 116.3912757},
 {'name': 'Nairobi, Kenya', 'lat': -1.2832533, 'long': 36.8172449},
 {'name': 'Cape Town, City of Cape Town, Western Cape, 8001, South Africa',
  'lat': -33.928992,
  'long': 18.417396},
 {'name': 'Autonomous City of Buenos Aires, Comuna 6, Autonomous City of Buenos Aires,␣
↪Argentina',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  'lat': -34.6075682,
  'long': -58.4370894}]
```

4. ISS-Daten abrufen und Durchlaufzeiten der Städte ermitteln

[7]:
```
def get_spaceship_location():
    resp = requests.get("http://api.open-notify.org/iss-now.json")
    location = resp.json()["iss_position"]
    return {
        "lat": float(location.get("latitude")),
        "long": float(location.get("longitude")),
    }
```

[8]:
```
def great_circle_dist(lon1, lat1, lon2, lat2):
    dist = DistanceMetric.get_metric("haversine")
    lon1, lat1, lon2, lat2 = map(np.radians, [lon1, lat1, lon2, lat2])

    X = [[lat1, lon1], [lat2, lon2]]
    kms = 6367
    return (kms * dist.pairwise(X)).max()
```

[9]:
```
def iss_dist_from_loc(issloc, loc):
    distance = great_circle_dist(
        issloc.get("long"), issloc.get("lat"), loc.get("long"), loc.get("lat")
    )
    logging.info("ISS is ~%dkm from %s", int(distance), loc.get("name"))
    return distance
```

[10]:
```
def iss_pass_near_loc(loc):
    resp = requests.get(
        "http://api.open-notify.org/iss-pass.json",
        params={"lat": loc.get("lat"), "lon": loc.get("long")},
    )
    data = resp.json().get("response")[0]
    td = datetime.fromtimestamp(data.get("risetime")) - datetime.now()
    m, s = divmod(int(td.total_seconds()), 60)
    h, m = divmod(m, 60)
    logging.info(
        "ISS will pass near %s in %02d:%02d:%02d", loc.get("name"), h, m, s
    )
    return td.total_seconds()
```

[11]:
```
iss_dist_from_loc(get_spaceship_location(), locations[2])
```

```
INFO:root:ISS is ~12639km from Berlin, 10117, Germany
```

[11]:
```
12639.759939298825
```

[12]:
```
iss_pass_near_loc(locations[2])
```

```
INFO:root:ISS will pass near Berlin, 10117, Germany in 00:25:14
```

[12]:
```
1514.253889
```

5. Erstellen einer `delayed`-Pipeline

[13]:
```
output = []

for loc in locations:
    issloc = delayed(get_spaceship_location)()
    dist = delayed(iss_dist_from_loc)(issloc, loc)
    output.append((loc.get("name"), dist))

closest = delayed(lambda x: sorted(x, key=itemgetter(1))[0])(output)
```

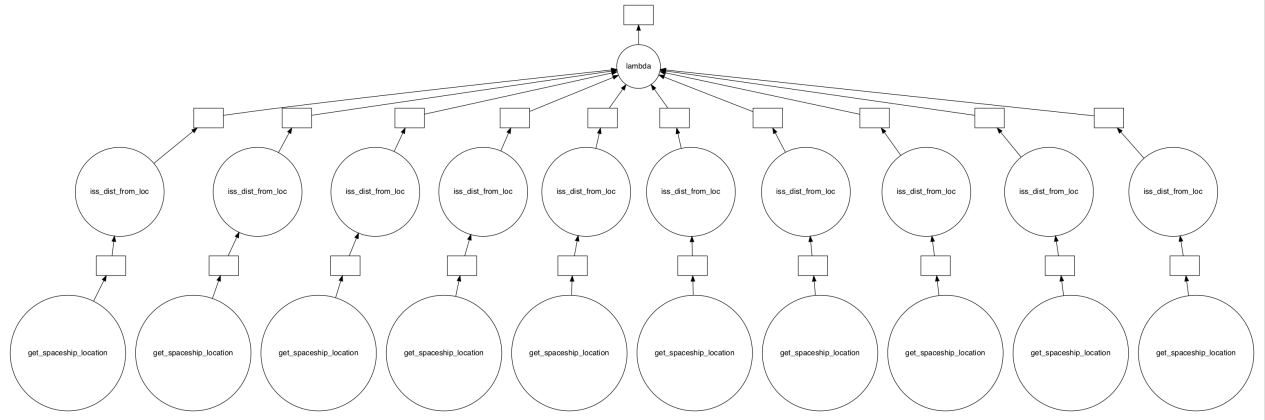
[14]:
```
closest
```

[14]:
```
Delayed('lambda-5ab5a78f-cb72-4168-bce1-f9983fdb8a2e')
```

6. DAG anzeigen

[15]:
```
closest.visualize()
```

[15]:



7. `compute()`

[16]:
```
closest.compute()
```

```
INFO:root:ISS is ~4685km from Miami, Miami-Dade County, Florida, USA
INFO:root:ISS is ~15205km from Beirut, Beirut Governorate, Lebanon
INFO:root:ISS is ~5919km from Seattle, King County, Washington, USA
INFO:root:ISS is ~6279km from Autonomous City of Buenos Aires, Comuna 6, Autonomous City␣
↪of Buenos Aires, Argentina
INFO:root:ISS is ~12625km from Berlin, 10117, Germany
INFO:root:ISS is ~13137km from Cape Town, City of Cape Town, Western Cape, 8001, South␣
↪Africa
INFO:root:ISS is ~16194km from Singapore
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
INFO:root:ISS is ~16298km from Nairobi, Kenya
INFO:root:ISS is ~13905km from Beijing, Dongcheng District, Beijing, 100010, China
INFO:root:ISS is ~8405km from Wellington, Wellington City, Wellington, 6011, New Zealand
```

[16]:
```
('Miami, Miami-Dade County, Florida, USA', 4685.887400314564)
```

KAPITEL 5

Daten visualisieren

Wir haben die Visualisierung von Daten in ein eigenes Tutorial ausgelagert: PyViz Tutorial.

KAPITEL 6

Performance

Mit Python lässt sich zwar schnell Code schreiben und testen, da es eine interpretierte Sprache ist, die dynamisch typisiert. Dies sind jedoch auch die Gründe, die sie bei der wiederholten Ausführung von einfachen Aufgaben, z.B. in Schleifen, langsam macht.

Bei der Entwicklung von Code kann es häufig zu Kompromissen zwischen verschiedenen Implementierungen kommen. Zu Beginn der Entwicklung eines Algorithmus ist es jedoch meist kontraproduktiv, sich um die Effizienz des Codes zu kümmern.

> »Wir sollten kleine Effizienzsteigerungen in sagen wir etwa 97 % der Zeit, vergessen: Vorzeitige Optimierung ist die Wurzel allen Übels. Dennoch sollten wir unsere Chancen in diesen kritischen 3 % nicht verpassen.«[1]

6.1 k-Means-Beispiel

Im Folgenden zeige ich Beispiele für den k-Means-Algorithmus, um aus einer Menge von Objekten eine vorher bekannte Anzahl von Gruppen zu bilden. Dies lässt sich in den folgenden drei Schritten ereichen:

1. Wähle die ersten k Elemente als Clusterzentren
2. Weise jedes neue Element dem Cluster zu, bei dem sich die Varianz am wenigsten erhöht
3. Aktualisiere das Clusterzentrum

Die Schritte 2 und 3 werden dabei so lange wiederholt, bis sich die Zuordnungen nicht mehr ändern.

Eine mögliche Implementierung mit reinem Python könnte so aussehen:

Quellcode 1: py_kmeans.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause
```

(Fortsetzung auf der nächsten Seite)

[1] Donald Knuth, Begründer des Literate programming, in Computer Programming as an Art (1974)

(Fortsetzung der vorherigen Seite)

```
def dist(x, y):
    """Calculate the distance"""
    return sum((xi - yi) ** 2 for xi, yi in zip(x, y))


def find_labels(points, centers):
    """Assign points to a cluster."""
    labels = []
    for point in points:
        distances = [dist(point, center) for center in centers]
        labels.append(distances.index(min(distances)))
    return labels


def compute_centers(points, labels):
    """Calculate the cluster centres."""
    n_centers = len(set(labels))
    n_dims = len(points[0])

    centers = [[0 for i in range(n_dims)] for j in range(n_centers)]
    counts = [0 for j in range(n_centers)]

    for label, point in zip(labels, points):
        counts[label] += 1
        centers[label] = [a + b for a, b in zip(centers[label], point)]

    return [
        [x / count for x in center] for center, count in zip(centers, counts)
    ]


def kmeans(points, n_clusters):
    """Calculates the cluster centres repeatedly until nothing changes."""
    centers = points[-n_clusters:].tolist()
    while True:
        old_centers = centers
        labels = find_labels(points, centers)
        centers = compute_centers(points, labels)
        if centers == old_centers:
            break
    return labels
```

Beispieldaten können wir uns erstellen mit:

```
from sklearn.datasets import make_blobs


points, labels_true = make_blobs(
    n_samples=1000, centers=3, random_state=0, cluster_std=0.60
)
```

Und schließlich können wir die Berechnung ausführen mit:

```
kmeans(points, 10)
```

6.2 Performance-Messungen

Wenn ihr erst einmal mit eurem Code gearbeitet habt, kann es nützlich sein, die Effizienz genauer zu untersuchen. Hierfür kann der *iPython Profiler* oder *scalene* genutzt werden.

Siehe auch:

- airspeed velocity (asv) ist ein Werkzeug zum Benchmarking von Python-Paketen während ihrer Lebensdauer. Laufzeit, Speicherverbrauch und sogar benutzerdefinierte Werte können aufgezeichnet und in einem interaktiven Web-Frontend angezeigt werden.
- Python Speed Center
- Tracing the Python GIL

6.2.1 iPython Profiler

IPython bietet Zugriff auf eine breite Palette von Funktionen um die Zeiten zu messen und Profile zu erstellen. Hier werden die folgenden magischen IPython-Befehle erläutert:

Befehl	Beschreibung
`%time`	Zeit für die Ausführung einer einzelnen Anweisung
`%timeit`	Durchschnittliche Zeit für die wiederholte Ausführung einer einzelnen Anweisung
`%prun`	Code mit dem Profiler ausführen
`%lprun`	Code mit dem zeilenweisen Profiler ausführen
`%memit`	Messen der Speichernutzung einer einzelnen Anweisung
`%mprun`	Führt den Code mit dem zeilenweisen Memory-Profiler aus

Die letzten vier Befehle sind nicht in IPython selbst, sondern in den Modulen line_profiler und memory_profiler enthalten.

Siehe auch

- Penn Machine Learning Benchmarks

`%timeit` und `%time`

Wir haben die `%timeit`Zeilen- und `%%timeit`-Zellmagie bereits in der Einführung der magischen Funktionen in IPython Magic Commands gesehen. Sie können für Zeitmessungen bei der wiederholten Ausführung von Code-Schnipseln verwendet werden:

```
[1]: %timeit sum(range(100))
```

```
1.42 µs ± 222 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

Beachtet, dass `%timeit` die Ausführung mehrfach in einer Schleife (`loops`) ausführt. Wenn mit `-n` nicht die Anzahl der Schleifen festgelegt wird, passt `%timeit` die Anzahl automatisch so an, dass ein ausreichende Messgenauigkeit erreicht wird:

[2]:
```
%%timeit

total = 0

for i in range(1000):
    for j in range(1000):
        total += i * (-1) ** j
```

```
503 ms ± 83.6 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

Manchmal ist das Wiederholen einer Operation nicht die beste Option, z.B. wenn wir eine Liste haben, die wir sortieren möchten. Hier werden wir möglicherweise durch eine wiederholte Operation in die Irre geführt. Das Sortieren einer vorsortierten Liste ist viel schneller als das Sortieren einer unsortierten Liste, sodass die Wiederholung das Ergebnis verzerrt:

[3]:
```
import random


L = [random.random() for i in range(100000)]

%timeit L.sort()
```

```
478 µs ± 19.1 µs per loop (mean ± std. dev. of 7 runs, 1000 loops each)
```

Dann ist die `%time`-Funktion möglicherweise die bessere Wahl. Auch bei länger laufenden Befehlen, wenn kurze systembedingte Verzögerungen das Ergebnis wahrscheinlich kaum beeinflussen, dürfte `%time` die bessere Wahl sein:

[4]:
```
import random


L = [random.random() for i in range(100000)]

%time L.sort()
```

```
CPU times: user 18.8 ms, sys: 0 ns, total: 18.8 ms
Wall time: 18.7 ms
```

Sortieren einer bereits sortierten Liste:

[5]:
```
%time L.sort ()
```

```
CPU times: user 1.13 ms, sys: 0 ns, total: 1.13 ms
Wall time: 1.14 ms
```

Beachtet, wie viel schneller die vorsortierte Liste zu sortieren ist, aber beachtet auch, wie viel länger das Timing mit `%time` gegenüber `%timeit` dauert, sogar für die vorsortierte Liste. Dies ist auf die Tatsache zurückzuführen, dass `%timeit` einige clevere Dinge unternimmt, um zu verhindern, dass Systemaufrufe die Zeitmessung stören. So wird beispielsweise die *Garbage Collection* nicht mehr verwendeter Python-Objekte verhindert, die sich andernfalls auf die Zeitmessung auswirken könnten. Aus diesem Grund sind die `%timeit`-Ergebnisse normalerweise merklich schneller als die `%time`-Ergebnisse.

Profilerstellung für Skripte: %prun

Ein Programm besteht aus vielen einzelnen Anweisungen, und manchmal ist es wichtiger, diese Anweisungen im Kontext zu messen, als sie selbst zu messen. Python enthält einen integrierten Code-Profiler. IPython bietet jedoch eine wesentlich bequemere Möglichkeit, diesen Profiler in Form der Magic-Funktion zu verwenden: %prun.

Als Beispiel definieren wir eine einfache Funktion, die einige Berechnungen durchführt:

[6]:
```
def sum_of_lists(N):
    total = 0
    for i in range(5):
        L = [j ^ (j >> i) for j in range(N)]
        total += sum(L)
    return total
```

[7]:
```
%prun sum_of_lists(1000000)
```

Im Notebook sieht die Ausgabe ungefähr so aus:

```
14 function calls in 9.597 seconds

   Ordered by: internal time

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        5    8.121    1.624    8.121    1.624 <ipython-input-15-f105717832a2>:4(
↪<listcomp>)
        5    0.747    0.149    0.747    0.149 {built-in method builtins.sum}
        1    0.665    0.665    9.533    9.533 <ipython-input-15-f105717832a2>:1(sum_of_
↪lists)
        1    0.065    0.065    9.597    9.597 <string>:1(<module>)
        1    0.000    0.000    9.597    9.597 {built-in method builtins.exec}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler'␣
↪objects}
```

Das Ergebnis ist eine Tabelle, die sortiert nach Gesamtzeit für jeden Funktionsaufruf die Ausführungsdauer angibt. In diesem Fall wird die meiste Zeit mit *List Comprehension* innerhalb von `sum_of_lists` verbraucht. Dies gibt uns Anhaltspunkte, an welcher Stelle wir die Effizienz des Algorithmus verbessern könnten.

Zeilenweise Profilerstellung: %lprun

Die Profilerstellung von %prun ist nützlich, aber manchmal ist ein zeilenweiser Profilreport aufschlussreicher. Dies ist nicht in Python oder IPython integriert, aber mit line_profiler steht ein Paket zur Verfügung, das dies ermöglicht. Diese kann in eurem Kernel bereitgestellt werden mit

```
$ spack env activate python-311
$ spack install py-line-profiler
```

Alternativ könnt ihr `line-profiler` auch mit anderen Paketmanagern installieren, z.B.

```
$ pipenv install line_profiler
```

Falls ihr Python 3.7.x verwendet und die Fehlermeldung bekommt `error: command 'clang' failed with exit status 1`, bleibt aktuell nur, `Cython` zusammen mit den Ressourcen aus dem Git-Repository zu installieren:

```
$ pipenv install Cython git+https://github.com/rkern/line_profiler.git#egg=line_profiler
```

Nun könnt ihr IPython mit der `line_profiler`-Erweiterung laden:

```
[8]: %load_ext line_profiler
```

Der `%lprun`-Befehl führt eine zeilenweise Profilerstellung für jede Funktion durch. In diesem Fall muss explizit angegeben werden, welche Funktionen für die Profilerstellung interessant sind:

```
[9]: %lprun -f sum_of_lists sum_of_lists(5000)
```

Das Ergebnis sieht ungefähr so aus:

```
Timer unit: 1e-06 s

Total time: 0.015145 s
File: <ipython-input-6-f105717832a2>
Function: sum_of_lists at line 1

Line #      Hits         Time  Per Hit   % Time  Line Contents
==============================================================
     1                                           def sum_of_lists(N):
     2         1          1.0      1.0      0.0      total = 0
     3         6         11.0      1.8      0.1      for i in range(5):
     4         5      14804.0   2960.8     97.7          L = [j ^ (j >> i) for j in␣
↪range(N)]
     5         5        329.0     65.8      2.2          total += sum(L)
     6         1          0.0      0.0      0.0      return total
```

Die Zeit wird in Mikrosekunden angegeben und wir können sehen, in welcher Zeile die Funktion die meiste Zeit verbringt. Eventuell können wir das Skript dann so ändern, dass die Effizienz der Funktion gesteigert werden kann.

Weitere Informationen zu `%lprun` sowie die verfügbaren Optionen findet ihr in der IPython-Hilfefunktion `%lprun?`.

Speicherprofil erstellen: `%memit` und `%mprun`

Ein weiterer Aspekt der Profilerstellung ist die Speichermenge, die eine Operation verwendet. Dies kann mit einer anderen IPython-Erweiterung ausgewertet werden, dem `memory_profiler`. Diese kann in eurem Kernel bereitgestellt werden mit

```
$ spack env activate python-374
$ spack install py-memory-profiler ^python@3.7.4%gcc@9.1.0
```

Alternativ könnt ihr `memory-profiler` auch mit anderen Paketmanagern installieren, z.B.

```
$ pipenv install memory_profiler
```

```
[10]: %load_ext memory_profiler
```

```
[11]: %memit sum_of_lists(1000000)
```

```
peak memory: 136.37 MiB, increment: 69.42 MiB
```

Wir sehen, dass diese Funktion ungefähr 100 MB Speicher belegt.

Für eine zeilenweise Beschreibung der Speichernutzung können wir die `%mprun`-Magie verwenden. Leider funktioniert diese Magie nur für Funktionen, die in separaten Modulen definiert sind, und nicht für das Notebook selbst. Daher erstellen wir zunächst mit der `%%file`-Magie ein einfaches Modul mit dem Namen `mprun_demo.py`, das unsere `sum_of_lists`-Funktion enthält.

[12]:
```
%%file mprun_demo.py
from memory_profiler import profile


@profile
def my_func():
    a = [1] * (10**6)
    b = [2] * (2 * 10**7)
    del b
    return a
```

```
Writing mprun_demo.py
```

[13]:
```
from mprun_demo import my_func


%mprun -f my_func my_func()
```

```
Filename: /srv/jupyter/jupyter-tutorial-de/docs/performance/mprun_demo.py

Line #    Mem usage    Increment   Line Contents
================================================
     3     67.3 MiB     67.3 MiB   @profile
     4                             def my_func():
     5     74.8 MiB      7.5 MiB       a = [1] * (10 ** 6)
     6    227.4 MiB    152.6 MiB       b = [2] * (2 * 10 ** 7)
     7     74.9 MiB      0.0 MiB       del b
     8     74.9 MiB      0.0 MiB       return a
```

Hier zeigt die `Increment`-Spalte, wie stark sich jede Zeile auf den gesamten Speicherverbrauch auswirkt: Beachtet, dass wir beim Berechnen von `b` etwa 160 MB Speicher zusätzlich benötigen; dieser wird aber durch das Löschen von `b` nicht wieder freigegeben.

Weitere Informationen zu `%memit` und `%mprun` sowie deren Optionen findet ihr in der IPython-Hilfe mit `%memit?`.

pyheatmagic

pyheatmagic ist eine Erweiterung, die den IPython-Magic-Befehl `%%heat` zum Anzeigen von Python-Code als Heatmap mit Py-Heat erlaubt.

Sie lässt sich einfach im Kernel installieren mit

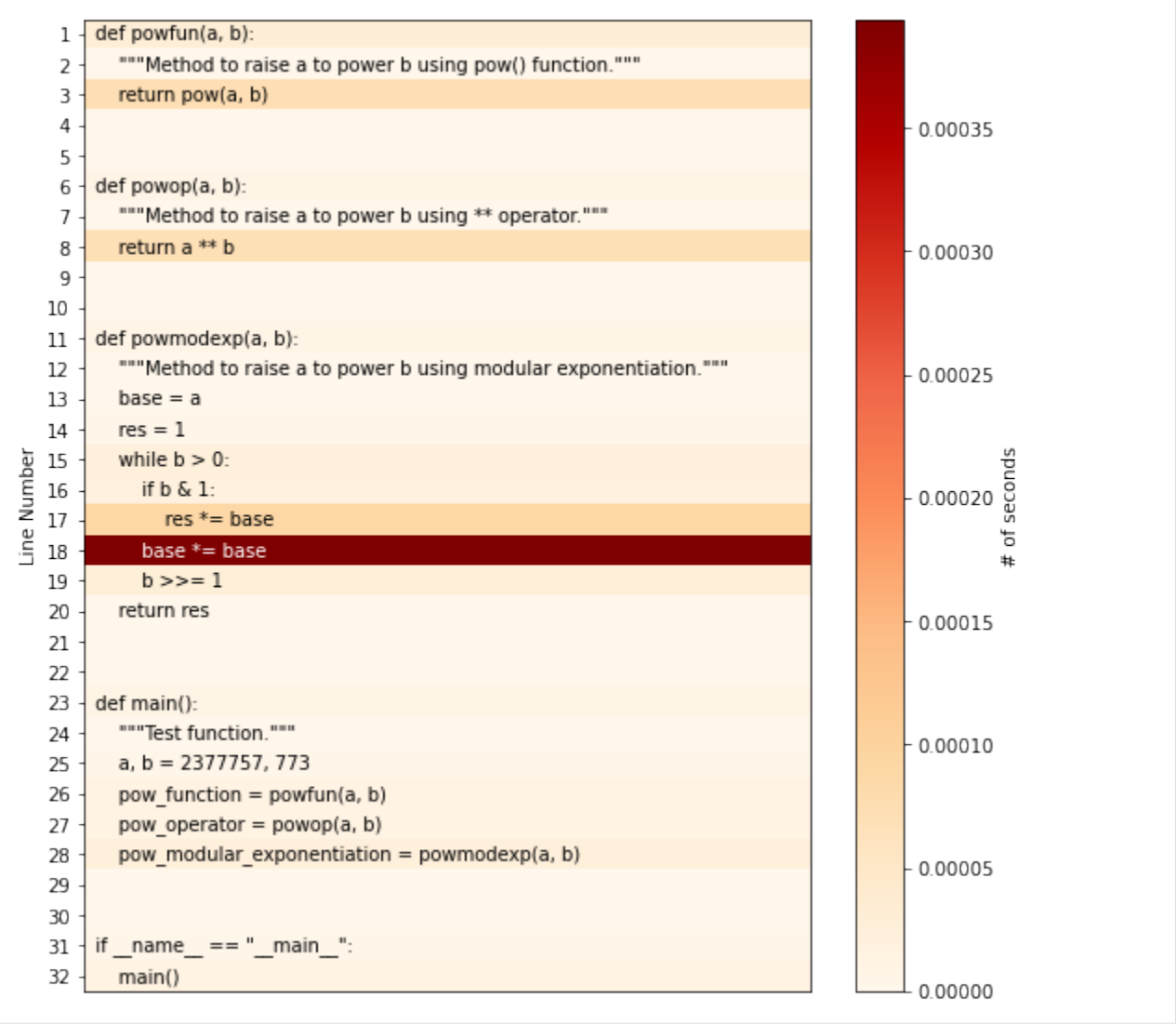
```
$ pipenv install py-heat-magic Installing py-heat-magic... ...
```

Laden der Extension in IPython

[14]:
```
%load_ext heat
```

Anzeigen der Heatmap

[15]:
```
%%heat
def powfun(a, b):
    """Method to raise a to power b using pow() function."""
    return pow(a, b)


def powop(a, b):
    """Method to raise a to power b using ** operator."""
    return a**b


def powmodexp(a, b):
    """Method to raise a to power b using modular exponentiation."""
    base = a
    res = 1
    while b > 0:
        if b & 1:
            res *= base
        base *= base
        b >>= 1
    return res


def main():
    """Test function."""
    a, b = 2377757, 773
    pow_function = powfun(a, b)
    pow_operator = powop(a, b)
    pow_modular_exponentiation = powmodexp(a, b)


if __name__ == "__main__":
    main()
```




```
1  def powfun(a, b):
2      """Method to raise a to power b using pow() function."""
3      return pow(a, b)
4
5
6  def powop(a, b):
7      """Method to raise a to power b using ** operator."""
8      return a ** b
9
10
11 def powmodexp(a, b):
12     """Method to raise a to power b using modular exponentiation."""
13     base = a
14     res = 1
15     while b > 0:
16         if b & 1:
17             res *= base
18         base *= base
19         b >>= 1
20     return res
21
22
23 def main():
24     """Test function."""
25     a, b = 2377757, 773
26     pow_function = powfun(a, b)
27     pow_operator = powop(a, b)
28     pow_modular_exponentiation = powmodexp(a, b)
29
30
31 if __name__ == "__main__":
32     main()
```

Alternativ kann die Heatmap auch als Datei gespeichert werden, z.B. mit

```
%%heat -o pow-heatmap.png
```

6.2.2 scalene

scalene erstellt sehr schnell Profile für CPU und Arbeitsspeicher. Dabei ist der Overhead üblicherweise mit 10–20% sehr gering.

Siehe auch

- GitHub
- PyPI
- scalene-paper.pdf

Installation

Linux, MacOS und WSL:

```
$ pipenv install scalene
```

Verwendung

1. Ein Beispielprogramm zum Profilieren

[1]:
```
import numpy as np


def profile_me():
    for i in range(6):
        x = np.array(range(10**7))
        y = np.array(np.random.uniform(0, 100, size=(10**8)))
```

2. Scalene laden

[2]:
```
%load_ext scalene
```

```
Scalene extension successfully loaded. Note: Scalene currently only
supports CPU+GPU profiling inside Jupyter notebooks. For full Scalene
profiling, use the command line version.

NOTE: in Jupyter notebook on MacOS, Scalene cannot profile child
processes. Do not run to try Scalene with multiprocessing in Jupyter
Notebook.
```

3. Profil mit nur einer Codezeile

[3]:
```
%scrun profile_me()
import numpy as np


def profile_me():
    for i in range(6):
        x = np.array(range(10**7))
        y = np.array(np.random.uniform(0, 100, size=(10**8)))
```

```
<IPython.lib.display.IFrame at 0x125931350>
```

Erzeugen Sie ein reduziertes Profil (nur Zeilen mit Zählungen ungleich Null)

[4]:
```
%scrun --reduced-profile profile_me()
import numpy as np


def profile_me():
    for i in range(6):
        x = np.array(range(10**7))
        y = np.array(np.random.uniform(0, 100, size=(10**8)))
```

```
<IPython.lib.display.IFrame at 0x1259531d0>
```

Eine vollständige Liste der Optionen erhalter ihr mit:

[5]:
```
%scrun --help
```

```
usage: scalene [-h] [--version] [--column-width COLUMN_WIDTH]
               [--outfile OUTFILE] [--html] [--json] [--cli] [--stacks]
               [--web] [--viewer] [--reduced-profile]
               [--profile-interval PROFILE_INTERVAL] [--cpu] [--cpu-only]
               [--gpu] [--memory] [--profile-all]
               [--profile-only PROFILE_ONLY]
               [--profile-exclude PROFILE_EXCLUDE] [--use-virtual-time]
               [--cpu-percent-threshold CPU_PERCENT_THRESHOLD]
               [--cpu-sampling-rate CPU_SAMPLING_RATE]
               [--allocation-sampling-window ALLOCATION_SAMPLING_WINDOW]
               [--malloc-threshold MALLOC_THRESHOLD]
               [--program-path PROGRAM_PATH] [--memory-leak-detector]
               [--on | --off]

Scalene: a high-precision CPU and memory profiler, version 1.5.24 (2023.08.09)
]8;id=423762;https://github.com/plasma-umass/scalene\
↪https://github.com/plasma-umass/scalene]8;;\

command-line:
  % scalene [options] your_program.py [--- --your_program_args]
or
  % python3 -m scalene [options] your_program.py [--- --your_program_args]

in Jupyter, line mode:
  %scrun [options] statement

in Jupyter, cell mode:
  %%scalene [options]
   your code here


options:
  -h, --help            show this help message and exit
  --version             prints the version number for this release of Scalene and exits
  --column-width COLUMN_WIDTH
                        Column width for profile output (default: 132)
  --outfile OUTFILE     file to hold profiler output (default: stdout)
  --html                output as HTML (default: web)
  --json                output as JSON (default: web)
  --cli                 forces use of the command-line
  --stacks              collect stack traces
  --web                 opens a web tab to view the profile (saved as 'profile.html')
  --viewer              only opens the web UI (https://plasma-umass.org/scalene-gui/)
  --reduced-profile     generate a reduced profile, with non-zero lines only (default:␣
↪False)
  --profile-interval PROFILE_INTERVAL
                        output profiles every so many seconds (default: inf)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  --cpu                 profile CPU time (default:  True )
  --cpu-only            profile CPU time (deprecated: use --cpu )
  --gpu                 profile GPU time and memory (default: False )
  --memory              profile memory (default: True )
  --profile-all         profile all executed code, not just the target program (default:␣
↪only the target program)
  --profile-only PROFILE_ONLY
                        profile only code in filenames that contain the given strings,␣
↪separated by commas
(default: no restrictions)
  --profile-exclude PROFILE_EXCLUDE
                        do not profile code in filenames that contain the given strings,␣
↪separated by commas
(default: no restrictions)
  --use-virtual-time    measure only CPU time, not time spent in I/O or blocking␣
↪(default: False)
  --cpu-percent-threshold CPU_PERCENT_THRESHOLD
                        only report profiles with at least this percent of CPU time␣
↪(default: 1%)
  --cpu-sampling-rate CPU_SAMPLING_RATE
                        CPU sampling rate (default: every 0.01s)
  --allocation-sampling-window ALLOCATION_SAMPLING_WINDOW
                        Allocation sampling window size, in bytes (default: 10485767␣
↪bytes)
  --malloc-threshold MALLOC_THRESHOLD
                        only report profiles with at least this many allocations␣
↪(default: 100)
  --program-path PROGRAM_PATH
                        The directory containing the code to profile (default: the path␣
↪to the profiled program)
  --memory-leak-detector
                        EXPERIMENTAL: report likely memory leaks (default: True)
  --on                  start with profiling on (default)
  --off                 start with profiling off

When running Scalene in the background, you can suspend/resume profiling
for the process ID that Scalene reports. For example:

   % python3 -m scalene  yourprogram.py &
 Scalene now profiling process 12345
   to suspend profiling: python3 -m scalene.profile --off --pid 12345
   to resume profiling:  python3 -m scalene.profile --on  --pid 12345
```

Profil mit mehr als einer Codezeile in einer Zelle

[6]:
```
%%scalene --reduced-profile

x = 0

for i in range(1000):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    for j in range(1000):
        x += 1
```

```
<IPython.lib.display.IFrame at 0x1259694d0>
```

6.3 Suche nach bestehenden Implementierungen

Ihr solltet nicht das Rad neu erfinden wollen: Wenn es bestehende Implementierungen gibt, solltet ihr diese auch verwenden. für den k-Means-Algorithmus gibt es sogar gleich zwei Implementierungen:

- sklearn.cluster.KMeans

```
from sklearn.cluster import KMeans


KMeans(10).fit_predict(points)
```

- dask_ml.cluster.KMeans

```
from dask_ml.cluster import KMeans


KMeans(10).fit(points).predict(points)
```

Gegen diese bestehenden Lösungen könnte bestenfalls sprechen, dass sie einen erheblichen Overhead in eurem Projekt erzeugen könnten wenn ihr nicht schon an anderen Stellen scikit-learn oder Dask-ML einsetzt. Im Folgenden werde ich daher weitere Möglichkeiten zeigen, euren eigenen Code zu optimieren.

6.4 Anti-Patterns finden

Anschließend könnt ihr mit *perflint* euren Code durchsuchen nach den häufigsten Performance-Anti-Patterns in Python.

6.4.1 perflint

perflint ist eine Erweiterung für pylint für Performance-Anti-Patterns, U.A.:

W8101: `unnecessary-list-cast`
: Unnötige Verwendung von `list()` bei einem bereits iterierbarem Typ.

W8102: `incorrect-dictionary-iterator`
: Falsche Iterator-Methode für `dict`: Python-Dictionaries speichern Schlüssel und Werte in zwei separaten Tabellen. Sie können einzeln iteriert werden. Die Verwendung von `.items()` und das Verwerfen entweder des Schlüssels oder des Wertes mit _ ist ineffizient, wenn stattdessen `.keys()` oder `.values()` verwendet werden können.

W8201: `loop-invariant-statement`
: Die Schleife wird untersucht, um Anweisungen oder Ausdrücke zu ermitteln, deren Ergebnis bei jeder Iteration einer Schleife konstant ist, da sie auf benannten Variablen basieren, die während der Iteration nicht verändert werden.

W8202: `loop-global-usage`
Globale Namensverwendung in einer Schleife: Das Laden globaler Variablen ist langsamer als das Laden lokaler Variablen. Der Unterschied ist marginal, aber wenn er in einer Schleife weitergegeben wird, kann es zu einer spürbaren Geschwindigkeitsverbesserung kommen.

R8203: `loop-try-except-usage`
Bis Python 3.10 sind `try`...`except`-Blöcke im Vergleich zu `if`-Anweisungen sehr rechenintensiv.

Vermeidet es, sie in einer Schleife zu verwenden, da sie erhebliche Overheads verursachen können. Refaktoriert euren Code so, dass keine iterationsspezifischen Details erforderlich sind und legt die gesamte Schleife in den `try`-Block.

W8204: `memoryview-over-bytes`
Das Slicing von Byte-Objekten in Schleifen ist ineffizient, da eine Kopie der Daten erstellt wird. Verwendet stattdessen `memoryview()`.

Siehe auch:

- Zero-copy interactions
- Memoryview Benchmarks
- Memoryview Benchmarks 2

W8205: `dotted-import-in-loop`
Der direkte Import des Namens `%s` ist in einer Schleife effizienter. In Python könnt ihr ein Modul importieren und dann auf Untermodule als Attribute zugreifen. Ihr könnt auch auf Funktionen als Attribute dieses Moduls zugreifen. Dadurch werden die Importanweisungen minimal gehalten. Wenn ihr diese Methode jedoch in einer Schleife verwendet, ist sie ineffizient, da bei jedem Schleifendurchlauf erst global, dann das Attribut und dann die Methode geladen wird.

W8301: `use-tuple-over-list`
Verwendet ein Tupel anstelle einer Liste für eine unveränderliche Sequenz: Sowohl die Konstruktion als aoch die Indizierung eines Tupels ist schneller als die einer Liste.

W8401: `use-list-comprehension`
Verwendet List Comprehensions mit oder ohne `if`-Anweisung anstelle einer `for`-Schleife.

W8402: `use-list-copy`
Verwendet eine Listenkopie mit `list.copy()` anstelle einer `for`-Schleife.

W8403: `use-dict-comprehension`
Verwendet ein Dictionary Comprehensions anstelle einer einfachen `for`-Schleife.

Siehe auch:

- Effective Python

6.5 Vektorisierungen mit NumPy

NumPy verlagert sich wiederholte Operationen in eine statisch typisierte kompilierte Schicht, um so die schnelle Entwicklungszeit von Python mit der schnellen Ausführungszeit von C zu kombinieren. Evtl. (Eventuell) könnt ihr *Universelle Funktionen (ufunc)*, *Vektorisierung* und *Indizierung und Slicing* in allen Kombinationen nutzen um sich wiederholende Operationen in kompilierten Code zu verschieben und damit langsame Schleifen zu vermeiden.

Mit NumPy können wir auf einige Schleifen verzichten:

Quellcode 2: np_kmeans.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import numpy as np


def find_labels(points, centers):
```

Die Vorteile von NumPy sind, dass der Python-Overhead nur je Array und nicht je Array-Element auftritt. Da NumPy eine spezifische Sprache für Array-Operationen verwendet, erfordert es jedoch auch eine andere Denkweise beim Schreiben von Code. Schließlich können die Batch-Operationen auch zu übermäßigem Speicherverbrauch führen.

6.6 Spezielle Datenstrukturen

pandas
: für SQL-ähnliche *Gruppenoperationen* und *Aggregation*.

  So könnt ihr auch die Schleife in der Methode `compute_centers` umgehen:

Quellcode 3: pd_kmeans.py

```
#
# SPDX-License-Identifier: BSD-3-Clause

    diff = points[:, None, :] - centers
    distances = (diff**2).sum(-1)
    return distances.argmin(1)
```

scipy.spatial
: für räumliche Abfragen wie Entfernungen, nächstgelegene Nachbarn, k-Means usw. (und so weiter)

  Unsere `find_labels`-Methode kann dann noch spezifischer geschrieben werden:

Quellcode 4: sp_kmeans.py

```
import numpy as np
import pandas as pd

from scipy.spatial import cKDTree
```

scipy.sparse
: dünnbesetzte Matrizen für 2-dimensionale strukturierte Daten.

Sparse
: für N-diemensional strukturierte Daten.

scipy.sparse.csgraph
: für graphenähnliche Probleme, z.B. die Suche nach kürzesten Wegen.

Xarray
: für die Gruppierung über mehrere Dimensionen hinweg.

6.6.1 pandas parallelisieren

In Enhancing performance werden einige Möglichkeiten beschrieben, wie die Performance von Pandas verbessert werden kann. Es gibt jedoch auch spezielle Bibliotheken, die die Verarbeitung von Dataframes parallelisieren können.

cuDF

cuDF ist eine GPU-DataFrame-Bibliothek, die eine Pandas-ähnliche API implementiert.

Siehe auch:

- Docs
- GitHub
- PyPI
- Beispiel Notebooks

Modin

Modin parallelisiert fast die gesamte Pandas-API. Dabei muss der bestehende Pandas-Code meist nur um folgenden Import erweitert werden:

```
import modin.pandas as pd
```

Die Einschränkungen beziehen sich auf `pd.read_json`, das nur für `lines=True` implementiert ist.

Siehe auch:

- Docs
- GitHub

Dask

Dask DataFrame ist ein großer paralleler DataFrame aus mehreren Pandas DataFrames. Dabei ist die `dask.dataframe`-API eine Teilmenge der Pandas-API, wobei es jedoch geringfügige Änderungen gibt.

Siehe auch:

- Home
- API docs
- Example notebook
- Tutorial

6.7 Compiler wählen

6.7.1 Faster Cpython

Auf der PyCon US im Mai 2021 stellte Guido van Rossum mit Faster CPython ein Projekt vor, das die Geschwindigkeit von Python 3.11 verdoppeln soll. Die Zusammenarbeit mit den anderen Python-Kernentwicklern ist in **PEP 659 – Specializing Adaptive Interpreter** geregelt. Zudem gibt es einen offenen Issue Tracker und diverse Werkzeuge zum Sammeln von Bytecode-Statistiken. Von den Änderungen profitieren dürfte vor allem CPU-intensiver Python-Code; bereits in C geschriebener Code, I/O-lastige Prozesse und Multithreading-Code dürften hingegen kaum profitieren.

Siehe auch:

- Faster CPython

Wenn ihr mit eurem Projekt nicht bis zur Veröffentlichung von Python 3.11 in der finalen Version voraussichtlich am 24. Oktober 2022 warten wollt, könnt ihr euch auch die folgenden Python-Interpreter anschauen:

6.7.2 Cython

Bei intensiven numerischen Operationen kann Python sehr langsam sein, auch wenn ihr alle Anti-Patterns vermieden und Vektorisierungen mit NumPy genutzt habt. Dann kann das Übersetzen von Code in Cython hilfreich sein. Leider muss der Code jedoch häufig umstrukturiert werden und nimmt dadurch an Komplexität zu. Auch werden explizite Type Annotations und das Bereitstellen des Codes umständlicher.

Unser Beispiel könnte dann so aussehen:

Quellcode 5: cy_kmeans.pyx

```
# SPDX-FileCopyrightText: 2023 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

cimport numpy as np
import numpy as np


cdef double dist(double[:] x, double[:] y):
    """Calculate the distance"""
    cdef double dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist

def find_labels(double[:, :] points, double[:, :] centers):
    """Assign points to a cluster."""
    cdef int n_points = points.shape[0]
    cdef int n_centers = centers.shape[0]
    cdef double[:] labels = np.zeros(n_points)
    cdef double distance, nearest_distance
    cdef int nearest_index

    for i in range(n_points):
        nearest_distance = np.inf
        for j in range(n_centers):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            distance = dist(points[i], centers[j])
            if distance < nearest_distance:
```

Siehe auch:

- Cython Tutorials

6.7.3 Numba

Numba ist ein JIT-Compiler, der vor allem wissenschaftlichen Python- und NumPy-Code in schnellen Maschinencode übersetzt, z.B.:

Quellcode 6: nb_kmeans.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import numba


@numba.jit(nopython=True)
def dist(x, y):
    """Calculate the distance"""
    dist = 0
    for i in range(len(x)):
        dist += (x[i] - y[i]) ** 2
    return dist


@numba.jit(nopython=True)
def find_labels(points, centers):
    """Assign points to a cluster."""
    labels = []
    min_dist = np.inf
    min_label = 0
    for i in range(len(points)):
        for j in range(len(centers)):
            distance = dist(points[i], centers[j])
```

Numba benötigt allerdings LLVM und einige Python-Konstrukte werden nicht unterstützt.

6.8 Aufgabenplaner

ipyparallel, *Dask* und Ray können Aufgaben in einem Cluster verteilen. Dabei haben sie unterschiedliche Schwerpunkte:

- `ipyparallel` integriert sich einfach in ein JupyterHub.
- *Dask* imitiert pandas, NumPy, Iteratoren, Toolz und PySpark bei der Verteilung ihrer Aufgaben.
- Ray bietet eine einfache, universelle API für den Aufbau verteilter Anwendungen.

- RLlib skaliert insbesondere reinforcement Learning.
- Ein Backend für Joblib unterstützt verteilte scikit-learn-Programme.
- XGBoost-Ray ist ein Backend für verteiltes XGBoost.
- LightGBM-Ray ist ein Backend für verteiltes LightGBM
- Collective Communication Lib bietet eine Reihe von nativen Collective-Primitiven für Gloo und die NVIDIA Collective Communication Library (NCCL).

Unser Beispiel könnte mit Dask so aussehen:

Quellcode 7: ds_kmeans.py

```
# SPDX-FileCopyrightText: 2021 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

import numpy as np

from dask import array as da
from dask import dataframe as dd


def find_labels(points, centers):
    """Assign points to a cluster."""
    diff = points[:, None, :] - centers
    distances = (diff**2).sum(-1)
    return distances.argmin(1)


def compute_centers(points, labels):
    """Calculate the cluster centres."""
    points_df = dd.from_dask_array(points)
    labels_df = dd.from_dask_array(labels)
    return points_df.groupby(labels_df).mean()


def kmeans(points, n_clusters):
    """Calculates the cluster centres repeatedly until nothing changes."""
    centers = points[-n_clusters:]
    points = da.from_array(points, 1000)
    while True:
        old_centers = centers
        labels = find_labels(points, da.from_array(centers, 5))
        centers = compute_centers(points, labels)
        centers = centers.compute().values
        if np.all(centers == old_centers):
            break
    return labels.compute()
```

6.8.1 Dask

Dask erfüllt zwei verschiedene Aufgaben:

1. die dynamische Aufgabenplanung wird optimiert, ähnlich wie bei Airflow, Luigi oder Celery
2. Arrays, Dataframes und Lists werden parallel mit dynamischem Task Scheduling ausgeführt.

Skalierung von Laptops bis hin zu Clustern

Dask kann mit pipenv einfach auf einem Laptop installiert werden und erweitert die Größe der Datensätze von *passt in den Arbeitsspeicher* zu *passt auf die Festplatte*. Dask kann jedoch auch auf einen Cluster mit Hunderten von Rechnern skaliert werden. Dask ist robust, flexibel, Data Local und hat eine geringe Latenzzeit. Weitere Informationen findet ihr in der Dokumentation zum Distributed Scheduler. Dieser einfache Übergang zwischen einer einzelnen Maschine und einem Cluster ermöglicht einen einfachen Start und ein Wachstum nach Bedarf.

Dask installieren

Ihr könnt alles installieren, was für die meisten gängigen Anwendungen von Dask erforderlich ist (Arrays, Dataframes, ...). Dies installiert sowohl Dask als auch Abhängigkeiten wie NumPy, Pandas, usw., die für verschiedene Arbeiten benötigt werden:

```
$ pipenv install "dask[complete]"
```

Es können aber auch nur einzelne Subsets installiert werden:

```
$ pipenv install "dask[array]"
$ pipenv install "dask[dataframe]"
$ pipenv install "dask[diagnostics]"
$ pipenv install "dask[distributed]"
```

Testen der Installation

```
[1]: !pytest   /Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
     ↪packages/dask/tests  /Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/
     ↪python3.11/site-packages/dask/array/tests
```

```
============================= test session starts ==============================
platform darwin -- Python 3.7.4, pytest-6.2.5, py-1.10.0, pluggy-1.0.0
Users/veit
collected 4218 items / 16 skipped / 4202 selected
...
plugins: anyio-3.3.1
=============================== warnings summary ===============================
...
-- Docs: https://docs.pytest.org/en/stable/warnings.html
==== 4013 passed, 207 skipped, 14 xfailed, 22 warnings in 357.82s (0:05:57) ====
```

Vertraute Bedienung

Dask DataFrame

… imitiert Pandas

[2]:
```
import pandas as pd


df = pd.read_csv("2021-09-01.csv")
df.groupby(df.user_id).value.mean()
```

[3]:
```
import dask.dataframe as dd


dd = pd.read_csv("2021-09-01.csv")
dd.groupby(dd.user_id).value.mean()
```

Siehe auch

- Dask DataFrame Docs
- Dask DataFrame Best Practices

Dask Array

… imitiert NumPy

[4]:
```
import numpy as np


f = h5py.File("mydata.h5")
x = np.array(f["."])
```

[5]:
```
import dask.array as da


f = h5py.File("mydata.h5")
x = da.array(f["."])
```

Siehe auch

- Dask Array Docs
- Dask Array Best Practices

Dask Bag

... imitiert iterators, Toolz und PySpark.

[6]:
```
import json

import dask.bag as db


b = db.read_text("2021-09-01.csv").map(json.loads)
b.pluck("user_id").frequencies().topk(10, lambda pair: pair[1]).compute()
```

Siehe auch

- Dask Bag Docs

Dask Delayed

... imitiert loops und umschließt benutzerdefinierten Code.

[7]:
```
from dask import delayed


L = []
for fn in "2021-*-*.csv":  # Use for loops to build up computation
    data = delayed(load)(fn)  # Delay execution of function
    L.append(delayed(process)(data))  # Build connections between variables

result = delayed(summarize)(L)
result.compute()
```

Siehe auch

- Dask Delayed Docs
- Dask Delayed Best Practices
- *Dask-Pipeline-Beispiel: Tracking der Internationalen Raumstation mit Dask*

Das `concurrent.futures`-Interface ermöglicht die Übermittlung von selbstdefinierten Aufgaben.

Bemerkung

Für das folgende Beispiel muss Dask mit der `distributed`-Option installiert werden, z.B.

```
$ pipenv install "dask[distributed]"
```

[8]:
```
from dask.distributed import Client


client = Client("scheduler:port")

futures = []
for fn in filenames:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    future = client.submit(load, fn)
    futures.append(future)

summary = client.submit(summarize, futures)
summary.result()
```

Siehe auch

- Dask Futures Docs
- Dask Futures Quickstart
- Dask Futures Examples

6.9 Multithreading, Multiprocessing und Async

Nach einem kurzen *Überblick* werden anhand von drei Beispielen zu *Threading*, *Multiprocessing* und *Async* die Regeln und Best Practices veranschaulicht.

6.9.1 Einführung in Multithreading, Multiprocessing und async

Martelli's Modell der Skalierbarkeit

Anzahl Kerne	Beschreibung
1	Einzelner Thread und einzelner Prozess
2–8	Mehrere Threads und mehrere Prozesse
>8	Verteilte Verarbeitung

Martelli's Beobachtung war, dass im Laufe der Zeit die zweite Kategorie immer unbedeutender wird, da einzelne Kerne werden immer leistungsfähiger und große Datensätze immer größer werden.

Global Interpreter Lock (GIL)

CPython verfügt über eine Sperre für seinen intern geteilten globalen Status. Dies hat zur Folge, dass nicht mehr als ein Thread gleichzeitig laufen kann.

Für I/O-lastige Anwendungen ist das GIL kein großes Problem; bei CPU-lastigen Anwendungen führt die Verwendung von Threading jedoch zu einer Verlangsamung. Dementsprechend ist Multi-Processing für uns spannend um mehr CPU-Zyklen zu erhalten.

Literate programming und *Martelli's Modell der Skalierbarkeit* bestimmten die Design-Entscheidungen zur Performance von Python über lange Zeit. An dieser Einschätzung hat sich bis heute wenig geändert: Entgegen der intuitiven Erwartungen führen mehr CPUs und Threads in Python zunächst zu weniger effizienten Anwendungen. Dennoch wünschen sich laut der Umfrage von 2020 zu den gewünschten Python-Features 20% Performance-Verbesserungen und 15% bessere Nebenläufigkeit und Parallelisierung. Das Gilectomy-Projekt, das das GIL ersetzen sollte, stieß jedoch auch noch auf ein weiteres Problem: Die Python C-API legt sehr viele Implementierungsdetails offen. Damit würden Leistungsverbesserungen jedoch schnell zu inkompatiblen Änderungen führen, die dann vor allem bei einer so beliebten Sprache wie Python inakzeptabel erscheinen.

Überblick

Kriterium	Multithreading	Multiprocessing	asyncio
Trennung	Threads teilen sich einen Status. Das Teilen eines Status kann jedoch zu *Race Conditions* führen, d.h. die Ergebnisse einer Operation können vom zeitlichen Verhalten bestimmter Einzeloperationen abhängen.	Die Prozesse sind unabhängig voneinander. Sollen sie dennoch miteinander kommunizieren, wird Interprocess communication (IPC), object pickling und anderer Overhead nötig.	Mit `run_coroutine_threadsafe()` können `asyncio`-Objekte auch von anderen Threads verwendet werden. Fast alle `asyncio`-Objekte sind nicht threadsicher.
Wechsel	Threads wechseln präemptiv, d.h., es muss kein expliziter Code hinzugefügt werdenm um einen Wechsel der Tasks zu verursachen. Ein solcher Wechsel ist jedoch jederzeit möglich; dementsprechend müssen kritische Bereiche mit `lock` geschützt werden.	Sobald ihr den Prozess erhaltet, sollten deutliche Fortschritte gemacht werden. Ihr solltet also nicht zu viele Roundtrips hin und her machen.	asyncio wechselt kooperativ, d.h., es muss explizit yield oder await angegeben werden um einen Wechsel herbeizuführen. Ihr könnt daher die Aufwände für diese Wechsel sehr gering halten.
Tooling	Threads erfordern sehr wenig Tooling: Lock und Queue. Locks sind in nicht-trivialen Beispielen schwer zu verstehen. Bei komplexen Anwendungen sollten daher besser atomare Message Queues oder asyncio verwendet werden.	Einfaches Tooling u.a. mit map und imap_unordered um einzelne Prozesse in einem einzelnen Thread zu testen, bevor zu Multiprocessing gewechselt wird. Wird IPC oder object pickling verwendet, wird das Tooling jedoch aufwändiger.	Zumindest bei komplexen Systemen führt `asyncio` einfacher zum Ziel als Multithreading Locks. `asyncio` benötigt jedoch eine große Menge von Werkzeugen: futures, Event Loops und nicht blockierende Versionen von fast allem.
Performance	Multithreading führt bei IO-lastigen Aufgaben zu den besten Ergebnissen. Die Leistungsgrenze für Threads ist eine CPU abzüglich der Kosten für Task-Switches und Aufwänden für die Synchronisation.	Die Prozesse können auf mehrere CPUs verteilt werden und sollten daher für CPU-lastige Aufgaben verwendet werden. Für die Kommunikation und die Synchronisierung der Prozesse entstehen jedoch ggf. zusätzliche Aufwände.	Der Aufruf einer reinen Python-Funktion erzeugt mehr Overhead als die erneute Anfrage eines `generator` oder `awaitable` – d.h., `asyncio` kann die CPU effizient auslasten. Für CPU-intensive Aufgaben ist jedoch Multiprocessing besser geeignet.

Resümee

Es gibt nicht die eine ideale Implementierung von Nebenläufigkeit – jeder der im folgenden vorgestellten Ansätze hat spezifische Vor- und Nachteile. Bevor ihr euch also entscheidet, welchen Ansatz ihr verfolgen wollt, solltet ihr die Performance-Probleme genau analysieren und anschließend die jeweils passende Läsung wählen. In unseren Projekten verwenden wir dabei häufig mehrere Ansätze, je nachdem, für welchen Teil die Performance optimiert werden soll.

6.9.2 Threading-Beispiel

Aktualisieren und Ausgeben eines Counters:

[1]:
```
counter = 0

print("Starting up")
for i in range(10):
    counter += 1
    print("The count is %d" % counter)
print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
```

Beginnt mit Code, der klar und einfach ist und von oben nach unten ausgeführt wird. Er ist einfach zu entwickeln und inkrementell zu testen.

Hinweis

Testet und debuggt eure Anwendung bevor ihr mit Threading beginnt. Threading macht das Debugggen niemals einfacher.

Umwandeln in Funktionen

Im nächsten Schritt wird dann wiederverwendbarer Code als Funktion erstellt:

[2]:
```
counter = 0


def worker():
    "My job is to increment the counter and print the current count"
    global counter

    counter += 1
    print("The count is %d" % counter)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
print("Starting up")
for i in range(10):
    worker()
print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
```

Multi-Threading

Jetzt können einige Worker-Threads gestartet werden:

[3]:
```
import threading


counter = 0


def worker():
    "My job is to increment the counter and print the current count"
    global counter

    counter += 1
    print("The count is %d" % counter)


print("Starting up")
for i in range(10):
    threading.Thread(target=worker).start()
print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
The count is 9
The count is 10
Finishing up
```

Test

Ein einfacher Testlauf gleicht perfekt der ursprünglichen Ausgabe.

Erkennen von Race Conditions

Hinweis

Tests können nicht die Abwesenheit von Fehlern beweisen. Viele interessante Race Conditions zeigen sich nicht in Testumgebungen.

Fuzzing

Fuzzing ist eine Technik um das Erkennen von Race Conditions zu verbessern:

[4]:
```
import random
import threading
import time


FUZZ = True


def fuzz():
    if FUZZ:
        time.sleep(random.random())


counter = 0


def worker():
    "My job is to increment the counter and print the current count"
    global counter

    fuzz()
    oldcnt = counter
    fuzz()
    counter = oldcnt + 1
    fuzz()
    print("The count is %d" % counter, end="")
    fuzz()


print("Starting up")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
fuzz()
for i in range(10):
    threading.Thread(target=worker).start()
    fuzz()
print("Finishing up")
fuzz()
```

```
Starting up
The count is 1The count is 2The count is 2The count is 2The count is 4The count is 4The␣
↪count is 5The count is 5Finishing up
The count is 6
```

Diese Technik ist auf relativ kleine Blöcke von Code beschränkt und ist insofern unvollkommen, als sie weiterhin nicht die Abwesenheit von Fehlern beweisen kann. Dennoch können Fuzzed Tests ggf. Race Conditions aufdecken.

Sorgfältiges Threading mit Queues

Dabei sind folgende Regeln zu beachten:

1. Alle gemeinsamen Ressourcen sollten in genau einem Thread ausgeführt werden. Alle Kommunikation mit diesem Thread sollte mit nur einer atomaren Message Queue erfolgen – in der Regel mit dem Queue-Modul, E-Mail oder Message Queues wie RabbitMQ oder ZeroMQ.

   Ressourcen, die diese Technik benötigen sind z.B. globale Variablen, Benutzereingaben, Ausgabegeräte, Dateien, Sockets usw.

2. Eine Kategorie von Sequenzierungsproblemen besteht im Sicherzustellen, dass Schritt A vor Schritt B ausgeführt wird. Die Lösung besteht darin, beide im selben Thread auszuführen, in dem alle Aktionen nacheinander ablaufen.

3. Um eine *Barriere* zu implementieren, die darauf wartet, dass alle parallelen Threads abgeschlossen sind, müssen nur alle Threads mit join() verbunden werden.

4. Ihr könnt nicht darauf warten, dass Daemon-Threads abgeschlossen werden (sie sind Endlosschleifen); stattdessen solltet ihr join() auf der Queue selbst ausführen, so dass die Aufgaben erst zusammengefügt werden wenn alle Aufgaben in der Queue erledigt sind.

5. Ihr könnt globale Variablen verwenden um zwischen Funktionen zu kommunizieren, allerdings nur innerhalb eines single-threaded Programm. In einem multi-thread-Programm könnt ihr globale Variablen jedoch nicht verwenden da sie mutable sind. Dann ist die bessere Lösung threading.local(), da sie in einem Thread zwar global ist, jedoch nicht darüber hinaus.

6. Versucht niemals, einen Thread von außen zu beenden: ihr wisst nie, ob dieser Thread ein Lock einthält. Daher bietet Python auch keinen direkten Mechanismus zum beenden von Threads. Falls ihr dies jedoch mit ctypes versucht, ist dies ein Rezept für Deadlocks.

Wenn wir nun diese Regeln anwenden, sieht unser Code so aus:

[5]:
```
import queue
import threading


counter = 0

counter_queue = queue.Queue()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
def counter_manager():
    "I have EXCLUSIVE rights to update the counter variable"
    global counter

    while True:
        increment = counter_queue.get()
        counter += increment
        print_queue.put(
            [
                "The count is %d" % counter,
            ]
        )
        counter_queue.task_done()


t = threading.Thread(target=counter_manager)
t.daemon = True
t.start()
del t

print_queue = queue.Queue()


def print_manager():
    while True:
        job = print_queue.get()
        for line in job:
            print(line)
        print_queue.task_done()


t = threading.Thread(target=print_manager)
t.daemon = True
t.start()
del t


def worker():
    "My job is to increment the counter and print the current count"
    counter_queue.put(1)


print_queue.put(["Starting up"])
worker_threads = []
for i in range(10):
    t = threading.Thread(target=worker)
    worker_threads.append(t)
    t.start()
for t in worker_threads:
    t.join()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
counter_queue.join()
print_queue.put(["Finishing up"])
print_queue.join()
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
```

Sorgfältiges Threading mit Locks

Wenn wir Threading mit Locks statt mit Queues machen, wirkt der Code noch aufgeräumter:

[6]:
```
import random
import threading
import time


counter_lock = threading.Lock()
printer_lock = threading.Lock()

counter = 0


def worker():
    global counter
    with counter_lock:
        counter += 1
        with printer_lock:
            print("The count is %d" % counter)


with printer_lock:
    print("Starting up")

worker_threads = []
for i in range(10):
    t = threading.Thread(target=worker)
    worker_threads.append(t)
    t.start()
for t in worker_threads:
    t.join()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
with printer_lock:
    print("Finishing up")
```

```
Starting up
The count is 1
The count is 2
The count is 3
The count is 4
The count is 5
The count is 6
The count is 7
The count is 8
The count is 9
The count is 10
Finishing up
The count is 10
```

Schließlich noch einige Hinweise zu Locks:

1. Locks sind nur sog. *Flags*, sie verhindern nicht wirklich zuverlässig.
2. Im Allgemeinen sollten Locks als primitives Hilfsmittel betrachtet werden, das in nicht-trivialen Beispielen schwierig zu verstehen ist. Bei komplexeren Anwendungen sollten also besser atomare Message Queues verwendet werden.
3. Je mehr Locks gleichzeitig gesetzt sind, desto geringer werden die Vorteile gleichzeitiger Verarbeitung.

6.9.3 Multi-Processing-Beispiel

Wir beginnen hier mit Code, der klar und einfach ist und von oben nach unten ausgeführt wird. Er ist einfach zu entwickeln und inkrementell zu testen:

[1]:
```
from multiprocessing.pool import ThreadPool as Pool

import requests


sites = [
    "https://github.com/veit/jupyter-tutorial/",
    "https://jupyter-tutorial.readthedocs.io/en/latest/",
    "https://github.com/veit/pyviz-tutorial/",
    "https://pyviz-tutorial.readthedocs.io/de/latest/",
    "https://cusy.io/en",
]


def sitesize(url):
    with requests.get(url) as u:
        return url, len(u.content)


pool = Pool(4)
for result in pool.imap_unordered(sitesize, sites):
    print(result)
```

```
('https://pyviz-tutorial.readthedocs.io/de/latest/', 32803)
('https://jupyter-tutorial.readthedocs.io/en/latest/', 40884)
('https://github.com/veit/jupyter-tutorial/', 237670)
('https://github.com/veit/pyviz-tutorial/', 213932)
('https://cusy.io/en', 33545)
```

Hinweis

Eine gute Entwicklungsstrategie ist die Verwendung von map, um den Code in einem einzelnen Prozess und einem einzelnen Thread zu testen, bevor zu Multi-Processing gewechselt wird.

Hinweis

Um besser einschätzen zu können, wann `ThreadPool` und wann `Pool` verwendet werden sollte, hier einige Faustregeln:

- Für CPU-lastige Jobs sollte `multiprocessing.pool.Pool` verwendet werden. Üblicherweise beginnen wir hier mit der doppelten Anzahl von CPU-Kernen für die Pool-Größe, mindestens jedoch mit 4.
- Für I/O-lastige Jobs sollte `multiprocessing.pool.ThreadPool` verwendet werden. Üblicherweise beginnen wir hier mit der fünffachen Anzahl von CPU-Kernen für die Pool-Größe.
- Verwenden wir Python 3 und benötigen kein mit `Pool` identisches Interface, nutzen wir concurrent.future.Executor statt `multiprocessing.pool.ThreadPool`; er hat ein einfacheres Interface und wurde von Anfang an für Threads konzipiert. Da er Instanzen von `concurrent.futures.Future` zurückgibt, ist er kompatibel zu vielen anderen Bibliotheken, einschließlich `asyncio`.
- Für CPU- und I/O-lastige Jobs bevorzugen wir `multiprocessing.Pool`, da hierdurch eine bessere Prozess-Isolierung erreicht wird.

[2]:
```
from multiprocessing.pool import ThreadPool as Pool

import requests


sites = [
    "https://github.com/veit/jupyter-tutorial/",
    "https://jupyter-tutorial.readthedocs.io/en/latest/",
    "https://github.com/veit/pyviz-tutorial/",
    "https://pyviz-tutorial.readthedocs.io/de/latest/",
    "https://cusy.io/en",
]


def sitesize(url):
    with requests.get(url) as u:
        return url, len(u.content)


for result in map(sitesize, sites):
    print(result)
```

```
('https://github.com/veit/jupyter-tutorial/', 237669)
('https://jupyter-tutorial.readthedocs.io/en/latest/', 40884)
('https://github.com/veit/pyviz-tutorial/', 213932)
('https://pyviz-tutorial.readthedocs.io/de/latest/', 32803)
('https://cusy.io/en', 33545)
```

Was ist parallelisierbar?

Amdahlsche Gesetz

> Der Geschwindigkeitszuwachs vor allem durch den sequentiellen Anteil des Problems beschränkt, da sich dessen Ausführungszeit durch Parallelisierung nicht verringern lässt. Zudem entstehen durch Parallelisierung zusätzliche Kosten wie etwa für die Kommunikation und die Synchronisierung der Prozesse.

In unserem Beispiel können die folgenden Aufgaben nur seriell abgearbeitet werden:

- UDP DNS request für die URL
- UDP DNS response
- Socket vom OS
- TCP-Connection
- Senden des HTTP Request für die Root-Ressource
- Warten auf die TCP Response
- Zählen der Zeichen auf der Website

[3]:
```
from multiprocessing.pool import ThreadPool as Pool

import requests


sites = [
    "https://github.com/veit/jupyter-tutorial/",
    "https://jupyter-tutorial.readthedocs.io/en/latest/",
    "https://github.com/veit/pyviz-tutorial/",
    "https://pyviz-tutorial.readthedocs.io/de/latest/",
    "https://cusy.io/en",
]


def sitesize(url):
    """Determine the size of a website"""
    with requests.get(url, stream=True) as u:
        return url, len(u.content)


pool = Pool(4)
for result in pool.imap_unordered(sitesize, sites):
    print(result)
```

```
('https://github.com/veit/pyviz-tutorial/', 213932)
('https://cusy.io/en', 33545)
('https://jupyter-tutorial.readthedocs.io/en/latest/', 40884)
('https://github.com/veit/jupyter-tutorial/', 237670)
('https://pyviz-tutorial.readthedocs.io/de/latest/', 32803)
```

Hinweis

imap_unordered wird verwendet, um die Reaktionsfähigkeit zu verbessern. Dies ist jedoch nur möglich, da die Funktion das Argument und das Ergebnis als Tuple zurückgibt.

Tipps

- Macht nicht zu viele Trips hin und her

  Erhaltet ihr zu viele iterierbare Ergebnisse, ist dies ein guter Indikator für zu viele Trips, wie z.B. in

```
>>>     def sitesize(url, start):
...         req = urllib.request.Request()
...         req.add_header('Range:%d-%d' % (start, start+1000))
...         u = urllib.request.urlopen(url, req)
...         block = u.read()
...         return url, len(block)
```

- Macht auf jedem Trip relevante Fortschritte

  Sobald ihr den Prozess erhaltet, solltet ihr deutliche Fortschritte erzielen und euch nicht verzetteln. Das folgende Beispiel verdeutlicht zu kleine Zwischenschritte:

```
>>> def sitesize(url, results):
...     with requests.get(url, stream=True) as u:
...         while True:
...             line = u.iter_lines()
...             results.put((url, len(line)))
```

- Sendet und empfangt nicht zu viele Daten

  Das folgende Beispiel erhöht unnötig die Datenmenge:

```
>>> def sitesize(url):
...     with requests.get(url) as u:
...         return url, u.content
```

6.9.4 Threading und Forking kombinieren

Das Mischen von Multiprocessing und Threading ist generell problematisch und ein Rezept für Deadlocks.

Der folgende Code wurde 2016 unter https://bugs.python.org/issue27422 im Python Bugtracker eingetragen:

[1]:
```
import multiprocessing
import subprocess
import sys

from concurrent.futures import ThreadPoolExecutor


def run(arg):
    print("starting %s" % arg)
    p = multiprocessing.Process(target=print, args=("running", arg))
    p.start()
    p.join()
    print("finished %s" % arg)


if __name__ == "__main__":
    n = 16
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    tests = range(n)
    with ThreadPoolExecutor(n) as pool:
        for r in pool.map(run, tests):
            pass
```

```
starting 0starting 1

starting 2
starting 3
starting 4
starting 5
starting 6
starting 7
starting 8
starting 9
starting 10
starting 11
starting 12
starting 13
starting 14
starting 15
running 5
finished 5
running 4
running 3
finished 3
finished 4
running 1
running 2
running 7
finished 2
finished 1
running 6
finished 7
finished 6
running 12
running 0
running 8
running 14
finished 12
finished 0
finished 8
finished 14
running 10
running 9
finished 10
finished 9
running 13
running 11
finished 13
finished 11
running 15
finished 15
```

Üblicherweise empfiehlt sich Threading nach dem Fork, nicht vorher. Andernfalls werden die beim Ausführen der Threads verwendeten Locks über die Prozesse dupliziert. Stirbt einer dieser Prozesse mit einem Lock, so geraten alle anderen Prozesse mit diesem Lock in einen Deadlock.

6.9.5 `asyncio` Beispiel

Ab IPython7.0 könnt ihr `asyncio` direkt in Jupyter Notebooks verwenden; seht auch IPython 7.0, Async REPL.

Wenn ihr die Fehlermeldung `RuntimeError: This event loop is already running` erhaltet, hilft euch vielleicht [nest-asyncio] weiter.

Ihr könnt das Paket in eurer Jupyter- oder JupyterHub-Umgebung installieren mit

```
$ pipenv install nest-asyncio
```

Ihr könnt es dann in euer Notebook importieren und verwenden mit:

[1]:
```
import nest_asyncio


nest_asyncio.apply()
```

Zum Weiterlesen

- Lynn Root: asyncio: We Did It Wrong
- Mike Driscoll: An Intro to asyncio
- Yeray Diaz: Asyncio Coroutine Patterns: Beyond await

Einfaches *Hello world*-Beispiel

[2]:
```
import asyncio


async def hello():
    print("Hello")
    await asyncio.sleep(1)
    print("world")


await hello()
```

```
Hello
world
```

Ein bisschen näher an einem realen Beispiel

[3]:
```
import asyncio
import random


async def produce(queue, n):
    for x in range(1, n + 1):
        # produce an item
        print("producing {}/{}".format(x, n))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())
        item = str(x)
        # put the item in the queue
        await queue.put(item)

    # indicate the producer is done
    await queue.put(None)


async def consume(queue):
    while True:
        # wait for an item from the producer
        item = await queue.get()
        if item is None:
            # the producer emits None to indicate that it is done
            break

        # process the item
        print("consuming {}".format(item))
        # simulate i/o operation using sleep
        await asyncio.sleep(random.random())


loop = asyncio.get_event_loop()
queue = asyncio.Queue()
asyncio.ensure_future(produce(queue, 10), loop=loop)
loop.run_until_complete(consume(queue))
```

```
producing 1/10
producing 2/10
consuming 1
producing 3/10
producing 4/10
producing 5/10
consuming 2
producing 6/10
consuming 3
producing 7/10
consuming 4
producing 8/10
consuming 5
producing 9/10
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
consuming 6
consuming 7
consuming 8
producing 10/10
consuming 9
consuming 10
```

Ausnahmebehandlung

Siehe auch:

- set_exception_handler

[4]:
```
def main():
    loop = asyncio.get_event_loop()
    # May want to catch other signals too
    signals = (signal.SIGHUP, signal.SIGTERM, signal.SIGINT)
    for s in signals:
        loop.add_signal_handler(
            s, lambda s=s: asyncio.create_task(shutdown(loop, signal=s))
        )
    loop.set_exception_handler(handle_exception)
    queue = asyncio.Queue()
```

Testen mit `pytest`

Beispiel:

[5]:
```
import pytest


@pytest.mark.asyncio
async def test_consume(mock_get, mock_queue, message, create_mock_coro):
    mock_get.side_effect = [message, Exception("break while loop")]

    with pytest.raises(Exception, match="break while loop"):
        await consume(mock_queue)
```

Bibliotheken von Drittanbietern

- pytest-asyncio hat hilfreiche Dinge wie Test-Fixtures für `event_loop`, `unused_tcp_port`, und `unused_tcp_port_factory`; und die Möglichkeit zum Erstellen eurer eigenen asynchronen Fixtures.
- asynctest verfügt über hilfreiche Werkzeuge, einschließlich Coroutine-Mocks und exhaust_callbacks so dass wir `await task` nicht manuell erstellen müssen.
- aiohttp hat ein paar wirklich nette eingebaute Test-Utilities.

Debugging

`asyncio` hat bereits einen debug mode in der Standardbibliothek. Ihr könnt ihn einfach mit der Umgebungsvariablen `PYTHONASYNCIODEBUG` oder im Code mit `loop.set_debug(True)` aktivieren.

Verwendet den Debug-Modus zum Identifizieren langsamer asynchroner Aufrufe

Der Debug-Modus von `asyncio` hat einen kleinen eingebauten Profiler. Wenn der Debug-Modus aktiviert ist, protokolliert `asyncio` alle asynchronen Aufrufe, die länger als 100 Millisekunden dauern.

Debugging im Produktivbetrieb mit `aiodebug`

aiodebug ist eine kleine Bibliothek zum Überwachen und Testen von Asyncio-Programmen.

Beispiel

[6]:
```
from aiodebug import log_slow_callbacks


def main():
    loop = asyncio.get_event_loop()
    log_slow_callbacks.enable(0.05)
```

Logging

aiologger ermöglicht eine nicht-blockierendes Logging.

Asynchrone Widgets

Siehe auch

- Asynchronous Widgets

[7]:
```
def wait_for_change(widget, value):
    future = asyncio.Future()

    def getvalue(change):
        # make the new value available
        future.set_result(change.new)
        widget.unobserve(getvalue, value)

    widget.observe(getvalue, value)
    return future
```

[8]:
```
from ipywidgets import IntSlider


slider = IntSlider()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
async def f():
    for i in range(10):
        print("did work %s" % i)
        x = await wait_for_change(slider, "value")
        print("async function continued with value %s" % x)


asyncio.ensure_future(f())

slider
```

```
IntSlider(value=0)
```

```
did work 0
```

KAPITEL 7

Produkt erstellen

> »Nicht reproduzierbare Einzelereignisse sind für die Wissenschaft ohne Bedeutung.«[1]

Damit auch andere euren Code nutzen können, sollte er einige Bedingungen erfüllen:

- Ihr solltet Euch nicht stillschweigend auf bestimmte Ressourcen und Umgebungen verlassen
- Erforderliche Software-Pakete und Hardware sollten in den Anforderungen angegeben werden
- Pfadangaben werden in einem anderen Kontext nur innerhalb eures Pakets oder in vorher generierten Verzeichnissen und Dateien funktionieren
- Teilt keine Geheimnisse wie Zugangsdaten oder interne IP-Nummern in eurem veröffentlichten Produkt

Es gibt diverse Werkzeuge, die Euch beim Erstellen von gemeinsam nutzbaren Produkten unterstützen. Dies können Werkzeuge einerseits für die Versionierung des *Quellcodes* und der *Trainingsdaten* sowie für die Reproduzierbarkeit der *Ausführungsumgebungen*, andererseits für das *Testen*, *Logging*, *Dokumentieren* und Erstellen von Paketen sein.

Siehe auch:

- Dustin Boswell, Trevor Foucher: The Art of Readable Code
- TIB workshop «FAIR Data and Software»
  - GitHub Page
  - GitHub Repository
  - Slides
- Dryad: Best practices for creating reusable data publications

[1] Karl Popper in *Logik der Forschung*, 1935

7.1 Code verwalten mit Git

Um eine bessere Kontrolle über euren Quellcode zu erhalten, wird dieser üblicherweise mit Git verwaltet. Git ist ein ausgereiftes und sehr aktiv gepflegtes Open-Source-Projekt, das 2005 ursprünglich von Linus Torvalds, dem Initiator des Linux-Betriebssystem-Kernels, entwickelt wurde. Git lässt sich gut mit vielen Betriebssystemen und IDEs (Integrierte Entwicklungsumgebungen) kombinieren.

Mit seiner verteilten Architektur ist Git ein Beispiel für ein DVCS (Distributed Version Control System) – einem verteilten Versionskontrollsystem. Somit muss sich nicht mehr die gesamte Versionshistorie an einem einzigen Ort befinden, wie dies bei früher beliebten Versionskontrollsystemen wie CVS oder Subversion (SVN) üblich war. In Git kann jedes lokale Repository spezifische Änderungen enthalten.

Git kann jedoch nicht nur verteilt genutzt werden, sondern ist auch performant, sicher und flexibel.

7.1.1 Performance

Git ist im Vergleich zu vielen anderen Versionsverwaltungssystemen sehr schnell bei Commits von Änderungen, beim Verzweigen und Zusammenführen und dem Vergleich mit früheren Versionen. Dies ist auch erforderlich, wenn wir uns das Linux-Kernel-Repository mit über einer Millionen Commits anschauen. Dabei orientiert sich Git nicht an Dateinamen, sondern konzentriert sich auf inhaltliche Änderungen, sodass Dateien effizient umbenannt, aufgeteilt und neu angeordnet werden können. Dies erreicht Git durch die Speicherung von Deltas für die inhaltlichen Unterschiede, Metadaten der Dateien und Komprimierung.

Das verteilte Versionsverwaltungssystem sorgt darüberhinaus dafür, dass z.B. für die Implementierung einer neuen Funktion **kein** Netzwerkzugriff auf einen entfernten Server erforderlich ist und daraus folgende Verzögerungen ausbleiben. Auch könnt ihr lokal an einer früheren Version eine Fehlerkorrektur durchführen. Später können mit einem einzigen Befehl beide Änderungen an einen zentralen Server übermittelt werden.

7.1.2 Sicherheit

Die Integrität des verwalteten Quellcodes hatte hohe Priorität bei der Konzipierung von Git. So werden die Beziehungen zwischen Dateien und Commits durch einen Hashing-Algorithmus (SHA1) geschützt, sodass versehentliche oder vorsätzliche Änderungen erschwert und der tatsächliche Verlauf sichergestellt werden.

7.1.3 Flexibilität

Git erlaubt nicht nur sehr flexible *Arbeitsabläufe* sondern ist auch für große wie kleine Projekte auf verschiedenen Plattformen geeignet.

7.1.4 Kritikpunkte

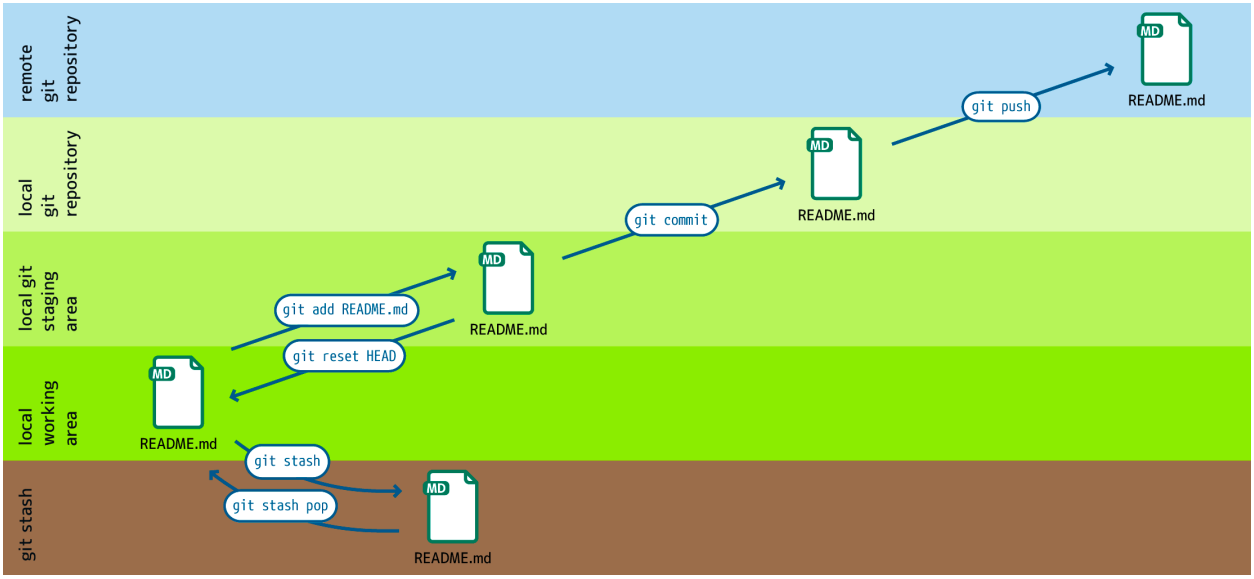
Eine häufige Kritik an Git ist, dass es schwer erlernbar sei: entweder sind große Teile der Git-Terminologie neu oder in anderen Systemen haben Bezeichnungen eine andere Bedeutung, wie z.B. `revert` in SVN oder CVS. Zudem bietet Git viel Funktionalität, die zum Erlernen jedoch einige Zeit benötigt.

7.1.5 Zum Weiterlesen

Siehe auch:

- `Git Cheat Sheet (PDF)`
- Interactive Git Cheatsheet
- Software Carpentry Version Control with Git
- Flight rules for Git
- First Aid git
- git-tips
- Pro Git book
- Git reference

Arbeitsbereiche



Git verwaltet mehrere Speicherorte oder **Workspaces**, in denen Dateien gespeichert werden:

local working copy
: enthält Dateien und Verzeichnisse, die normal bearbeitet werden.

staging area
: enthält Änderungen an Dateien, die in die Versionsgeschichte geschrieben werden sollen.

local repository
: enthält die gesamte Historie aller Dateien im Projekt.

remote repository
: enthält ebenfalls die gesamte Historie, ist aber auf einem entfernten Server gespeichert.

stash
: enthält Änderungen, die vorübergehend an einem anderen Ort gespeichert werden, um sie aus dem aus dem Weg zu schaffen.

Basic Git commands

The following basic Git commands move changes between these workspaces.

`git add`
: fügt Dateien aus dem Arbeitsverzeichnis dem Bühnenbereich (ENGL. (englisch): *staging area*) hinzu.

`git reset HEAD`
: stellt eine Datei im Arbeitsbereich aus dem Bühnenbereich wieder her.

`git stash`
: verschiebt Dateien aus dem Arbeitsbereich in ein Versteck (ENGL.: *stash*).

`git stash pop`
: holt Dateien aus dem Versteck in den Arbeitsbereich.

`git commit`
: schreibt Änderungen im Bühnenbereich in das lokale Repository.

`git pull`
: kopiert Änderungen aus dem entfernten in das lokale Repository und aktualisiert das Arbeitsverzeichnis.

`git push`
: kopiert Änderungen aus dem lokalen Repository in das entfernte (ENGL.: *remote*) Repository.

  `git push -u` ***UPSTREAM BRANCHNAME***

  `-u` (Langform `--set-upstream`)
  : erlaubt die Angabe eines entfernten Repository und des darin enthaltenen Zweiges.

  UPSTREAM
  : ist der Name des entfernten Repository, üblicherweise `origin`.

  BRANCHNAME
  : ist der Name des Zweiges im entfernten Repository, üblicherweise derselbe Name wie im lokalen Repository.

Git-Installation und -Konfiguration

Installation

Für iX-Distributionen sollte Git im Standard-Repository vorhanden sein.

Das Paket git-all stellt euch eine vollständige Umgebung für Git bereit. Dieses installiert ihr wie folgt:

```
$ sudo apt install git-all
```

Geht es ausschliesslich um Git, genügt das Paket namens git:

```
$ sudo apt install git
```

Mit der Bash-Autovervollständigung lässt sich Git auf der Kommandozeile einfacher bedienen. Das entsprechende Paket dazu heisst bash-completion und ihr installiert es wie folgt:

```
$ sudo apt install bash-completion
```

Es gibt verschiedene Möglichkeiten, Git auf einem Mac zu installieren. Am einfachsten ist es vermutlich, die Xcode Command Line Tools zu installieren. Hierfür müsst ihr nur `git` das erste Mal vom Terminal aufrufen:

```
$ git --version
```

`git-completion` könnt ihr mit Homebrew installieren:

Anschließend müsst ihr folgende Zeile in der Datei `~/.bash_profile` hinzufügen:

```
[[ -r "$(brew --prefix)/etc/profile.d/bash_completion.sh" ]] && . "$(brew --prefix)/etc/
↪profile.d/bash_completion.sh"
```

Ruft dazu https://git-scm.com/download/win auf und startet den passenden Download dazu.

Siehe auch:

- git for windows

Konfiguration

Für jede Änderung muss ersichtlich sein, wer diese verursacht hat. Dazu hinterlegt ihr euren Namen und eure E-Mail-Adresse wie folgt:

$ git config --global user.name "*NAME*"
: legt den Namen *NAME* fest, der mit euren Commit-Transaktionen verknüpft wird.

$ git config --global user.email "*EMAIL-ADDRESS*"
: legt die E-Mail-Adresse *EMAIL-ADDRESS* fest, die mit euren Commit-Transaktionen verknüpft wird.

Für eine bessere Lesbarkeit der Ausgabe sorgt die Kolorierung der Befehlszeilenausgabe. Diese schaltet ihr mit Hilfe dieses Aufrufs ein:

$ git config --global color.ui auto
: aktiviert die Kolorierung der Befehlszeilenausgabe.

Die `~/.gitconfig`-Datei

Mit den oben angegebenen Befehlen wird dir folgende `~/.gitconfig`-Datei erstellt:

```
[user]
    name = veit
    email = veit@cusy.io

[color]
    ui = auto
```

In der `~/.gitconfig`-Datei können jedoch auch Aliase festgelegt werden:

```
[alias]
    st = status
    ci = commit
    br = branch
    co = checkout
    df = diff
    dfs = diff --staged
```

Siehe auch:

Shell-Konfiguration:

- oh-my-zsh
  - Git plugin aliases
  - zsh-you-should-use
- Starship
  - git_branch-Modul
  - git_commit-Modul
  - git_state
  - git_status-Modul

Auch der Editor lässt sich angeben, z.B. mit:

```
[core]
    editor = vim
```

oder für Visual Studio Code mit:

```
[core]
    editor = code --wait
```

Bemerkung: Auf macOS müsst ihr zunächst Visual Studio Code starten, dann die Befehlspalette mit `+-p` öffnen und schließlich den Befehl `Install 'code' command in PATH` ausführen.

Auch die Hervorhebung von Leerzeichenfehlern in `git diff` lässt sich konfigurieren:

```
[core]
    # Highlight whitespace errors in git diff:
    whitespace = tabwidth=4,tab-in-indent,cr-at-eol,trailing-space
```

Bemerkung: Neben `~/.gitconfig` sucht Git seit Version 1.17.12 auch in `~/.config/git/config` nach einer globalen Konfigurationsdatei.

Unter Linux kann `~/.config` manchmal ein anderer Pfad sein, der durch die Umgebungsvariable `XDG_CONFIG_HOME` festgelegt wird. Dieses Verhalten ist Teil der Spezifikation der X Desktop Group (XDG). Den anderen Pfad erhaltet ihr mit:

```
$ echo $XDG_CONFIG_HOME
```

Siehe auch:

- git config files

Da ihr Optionen an mehreren Ebenen festlegen könnt, möchtet ihr vielleicht nachvollziehen, woher Git einen bestimmten Wert liest. Mit `git config --list`[1] könnt ihr alle überschriebenen Optionen und Werte auflisten. Dies könnt ihr kombinieren mit `--show-scope`[2] um zu sehen, woher Git den Wert bezieht:

```
$ git config --list --show-scope
system  credential.helper=osxkeychain
global  user.name=veit
global  user.email=veit@cusy.io
...
```

Ihr könnt auch `--show-origin`[3] verwenden, um die Namen der Konfigurationsdateien aufzulisten:

```
$ git config --list --show-origin
file:/opt/homebrew/etc/gitconfig        credential.helper=osxkeychain
file:/Users/veit/.config/git/config     user.name=veit
file:/Users/veit/.config/git/config     user.email=veit@cusy.io
...
```

[1] git config –list
[2] git config –show-scope
[3] git config –show-origin

Alternative Konfigurationsdatei

Ihr könnt für bestimmte Arbeitsverzeichnisse andere Konfigurationsdateien verwenden, z.B. um zwischen privaten und beruflichen Projekten zu unterscheiden. Dazu könnt ihr eine lokale Konfiguration in eurem Repository verwenden oder aber Conditional Includes am Ende eurer globalen Konfiguration:

```
...
[includeIf "gitdir:~/private"]
path = ~/.config/git/config-private
```

Dieses Konstrukt sorgt dafür, dass Git zusätzliche Konfigurationen einbezieht oder bestehende überschreibt, wenn ihr in `~/private` arbeitet.

Erstellt dazu nun die Datei `~/.config/git/config-private` und legt dort eure alternative Konfiguration fest, z.B.:

```
[user]
    email = kontakt@veit-schiele.de

[core]
    sshCommand = ssh -i ~/.ssh/private_id_rsa
```

Siehe auch:

- core.sshCommand

Anmeldedaten verwalten

Seit der Git-Version 1.7.9 lassen sich die Zugangsdaten zu git-Repositories mit gitcredentials verwalten. Um diese zu nutzen, könnt ihr z.B. folgendes angeben:

```
$ git config --global credential.helper Cache
```

Hiermit wird euer Passwort für 15 Minuten im Cache-Speicher gehalten. Der Timeout kann GGF. (gegebenenfalls) erhöht werden, z.B. mit:

```
$ git config --global credential.helper 'cache --timeout=3600'
```

Unter macOS lässt sich mit `osxkeychain` die Schlüsselbundverwaltung (*Keychain*) nutzen um die Zugangsdaten zu speichern. `osxkeychain` setzt Git in der Version 1.7.10 oder neuer voraus und kann im selben Verzeichnis wie Git installiert werden mit:

```
$ git credential-osxkeychain
git: 'credential-osxkeychain' is not a git command. See 'git --help'.
$ curl -s -O http://github-media-downloads.s3.amazonaws.com/osx/git-credential-
↪osxkeychain
$ chmod u+x git-credential-osxkeychain
$ sudo mv git-credential-osxkeychain /usr/bin/
Password:
git config --global credential.helper osxkeychain
```

Dies trägt folgendes in die `~/.gitconfig`-Datei ein:

```
[credential]
    helper = osxkeychain
```

Für Windows steht der Git Credential Manager (GCM) zur Verfügung. Er ist integriert in Git for Windows und wird standardmäßig mitinstalliert. Zusätzlich besteht jedoch auch ein eigenständiges Installationsprogramm in Releases.

GCM wird mit dem nachfolgenden Aufruf konfiguriert:

```
$ git credential-manager configure
Configuring component 'Git Credential Manager'...
Configuring component 'Azure Repos provider'...
```

Dies trägt den `[credential]`-Abschnitt in eure `~/.gitconfig`-Datei ein:

```
[credential]
    helper =
    helper = C:/Program\\ Files/Git/mingw64/bin/git-credential-manager.exe
```

Nun öffnet sich beim Clonen eines Repository ein Fenster des GCM und fordert euch zur Eingabe eurer Zugangsdaten auf.

Zudem wird die `~/.gitconfig`-Datei ergänzt, z.B. um die folgenden beiden Zeilen:

```
[credential "https://ce.cusy.io"]
    provider = generic
```

Bemerkung: Ein umfangreiches Beispiel einer Konfigurationsdatei findet ihr in meinem dotfiles-Repository: .gitconfig.

Siehe auch:

- Git Credential Manager: authentication for everyone

Die `.gitignore`-Datei

In der `.gitignore`-Datei eines Repository könnt ihr Dateien von der Versionsverwaltung ausschließen. Eine typische `.gitignore`-Datei kann z.B. so aussehen:

```
/logs/*
!logs/.gitkeep
/tmp
*.swp
```

Dabei verwendet Git Globbing-Muster, U.A.:

Muster	Beispiel	Erläuterung
`**/logs`	`logs/instance.log`, `logs/instance/error.log`, `prod/logs/instance.log`	Ihr könnt zwei Sternchen voranstellen um Verzeichnisse an einer beliebigen Stelle im Verzeichnisbaum zu finden.
`**/logs/instance.log`	`logs/instance.log`, `prod/logs/instance.log` aber nicht `logs/prod/instance.log`	Ihr könnt zwei Sternchen voranstellen um Dateien anhand ihres Namens in einem übergeordneten Verzeichnis zu finden.
`*.log`	`instance.log`, `error.log`, `logs/instance.log`	Ein Sternchen ist ein Platzhalter für null oder mehr Zeichen.
`/logs` `!/logs/.gitkeep`	`/logs/instance.log`, `/logs/error.log`, nicht jedoch `/logs/.gitkeep` oder `/instance.log`	Ein vor ein Muster gestelltes Anführungszeichen ignoriert dieses. Wenn eine Datei mit einem Muster übereinstimmt, aber auch mit einem negierenden, das später definiert ist, wird sie nicht ignoriert.
`/instance.log`	`/instance.log`, nicht jedoch `logs/instance.log`	Mit dem vorangestellten Schrägstrich passt das Muster nur zu Dateien im Stammverzeichnis des Repository.
`instance.log`	`instance.log`, `logs/instance.log`	Üblicherweise passen die Muster zu Dateien in jedem Verzeichnis.
`instance?.log`	`instance0.log`, `instance1.log`, aber nicht `instance.log` oder `instance10.log`	Ein Fragezeichen passt genau zu einem Zeichen.
`instance[0-9].log`	`instance0.log`, `instance1.log`, aber nicht `instance.log` oder `instance10.log`	Eckige Klammern können verwendet werden um ein einzelnes Zeichen aus einem bestimmten Bereich zu finden.
`instance[01].log`	`instance0.log`, `instance1.log`, aber nicht `instance2.log` oder `instance01.log`	Eckige Klammern passen auf ein einzelnes Zeichen aus einer bestimmten Menge.
`instance[!01].log`	`instance2.log`, aber nicht `instance0.log`, `instance1.log` oder `instance01.log`	Ein Ausrufezeichen kann verwendet werden um ein beliebiges Zeichen aus einer angegebenen Menge zu finden.
`logs`	`logs` `logs/instance.log` `prod/logs/instance.log`	Wenn kein Schrägstrich anhängt, passt das Muster sowohl auf Dateien als auch auf den Inhalt von Verzeichnissen mit diesem Namen.
`logs/`	`logs/instance.log`, `logs/prod/instance.log`, `prod/logs/instance.log`	Das Anhängen eines Schrägstrichs zeigt an, dass das Muster ein Verzeichnis ist. Der gesamte Inhalt jedes Verzeichnisses im Repository, das diesem Namen entspricht – einschließlich all seiner Dateien und Unterverzeichnisse – wird ignoriert.
`var/**/instance.log`	`var/instance.log`, `var/logs/instance.log`, nicht jedoch `var/logs/instance/error.log`	Zwei Sternchen passen zu null oder mehr Verzeichnissen.
`logs/instance*/error.log`	`logs/instance/error.log`, `logs/instance1/error.log`	Wildcards können auch in Verzeichnisnamen verwendet werden.
	`logs/instance.log`, nicht jedoch	Muster, die eine Datei in einem be-

Git-commit eines leeren Verzeichnisses

In obigem Beispiel seht ihr, dass mit `/logs/*` keine Inhalte des `logs`-Verzeichnisses mit Git versioniert werden sollen, in der Folgezeile jedoch eine Ausnahme definiert wird:

```
!logs/.gitkeep
```

Diese Angabe erlaubt, dass die Datei `.gitkeep` mit Git verwaltet werden darf. Damit wird dann auch das `logs`-Verzeichnis in das Git-Repository übernommen. Diese Hilfskonstruktion ist erforderlich, da leere Verzeichnisse nicht mit Git verwaltet werden können.

Eine andere Möglichkeit besteht darin, in einem leeren Verzeichnis eine `.gitignore`-Datei mit folgendem Inhalt zu erstellen:

```
# ignore everything except .gitignore
*
!.gitignore
```

Dateien zentral mit `excludesfile` ausschließen

Ihr könnt jedoch auch zentral für alle Git-Repositories Dateien ausschließen. Hierfür wird üblicherweise in der `~/.gitconfig`-Datei folgendes angegeben:

```
[core]

    # Use custom `.gitignore`
    excludesfile = ~/.gitignore
    ...
```

Bemerkung: Hilfreiche Vorlagen findet ihr in meinem dotfiles-Repository oder auf der Website gitignore.io.

Ignorieren einer Datei aus dem Repository

Wenn ihr eine Datei ignorieren wollt, die in der Vergangenheit bereits dem Repository hinzugefügt wurde, müsst ihr die Datei aus eurem Repository löschen und dann eine `.gitignore`-Regel für sie hinzufügen. Die Verwendung der Option `--cached` bei `git rm` bedeutet, dass die Datei aus dem Repository gelöscht wird, aber als ignorierte Datei in eurem Arbeitsverzeichnis verbleibt.

```
$ echo *.log >> .gitignore
$ git rm --cached *.log
rm 'instance.log'
$ git commit -m "Remove log files"
```

Bemerkung: Ihr könnt die Option `--cached` weglassen, wenn ihr die Datei sowohl aus dem Repository als auch aus eurem lokalen Dateisystem löschen wollt.

Commit einer ignorierten Datei

Es ist möglich, den Commit einer ignorierten Datei an das Repository mit der Option `-f` (oder `--force`) bei `git add` zu erzwingen:

```
$ cat data/.gitignore
*
$ git add -f data/iris.csv
$ git commit -m "Force add iris.csv"
```

Ihr könnt dies in Erwägung ziehen, wenn ihr ein allgemeines Muster (wie *) definiert habt, aber eine bestimmte Datei übertragen wollt. Eine bessere Lösung ist meist jedoch, eine Ausnahme von der allgemeinen Regel zu definieren:

```
$ echo '!iris.csv' >> data/.gitignore
$ cat data/.gitignore
*
!iris.csv
$ git add data/iris.csv
$ git commit -m "Add iris.csv"
```

Dieser Ansatz dürfte für euer Team offensichtlicher und weniger verwirrend sein.

Fehlersuche in `.gitignore`-Dateien

Bei komplizierten `.gitignore`-Mustern oder bei Mustern, die über mehrere `.gitignore`-Dateien verteilt sind, kann es schwierig sein, herauszufinden, ob oder warum eine bestimmte Datei ignoriert wird.

Mit dem Aufruf `git status --ignored=matching`[4] wird der Ausgabe ein Abschnitt *Ignorierte Dateien* hinzugefügt, der zusätzlich alle von Git ignorierten Dateien und Verzeichnisse beinhaltet:

```
$ git status --ignored=matching
Auf Branch main
Ignorierte Dateien:
  (benutzen Sie "git add -f <Datei>...", um die Änderungen zum Commit vorzumerken)
    .DS_Store
    docs/.DS_Store
    docs/_build/doctrees/
    docs/_build/html/
    docs/clean-prep/.ipynb_checkpoints/
    ...
    nichts zu committen, Arbeitsverzeichnis unverändert
```

Ihr könnt den Befehl `git check-ignore`[5] mit der Option `-v` (Langform: `--verbose`) verwenden, um festzustellen, welches Muster die Ursache für das Ignorieren einer bestimmten Datei ist:

```
$ git check-ignore -v data/iris.csv
data/.gitignore:2:!iris.csv data/iris.csv
```

Obige Ausgabe besteht aus vier Feldern (Trennzeichen sind drei Doppelpunkte und ein Leerzeichen) und beinhaltet:

FILE_CONTAINING_THE_PATTERN
: den Namen der Datei, die das Muster enthält.

[4] git status –ignored
[5] git check-ignore

LINE_NUMBER_OF_THE_PATTERN
: die Zeilennummer, in der in der Datei *FILE_CONTAINING_THE_PATTERN* das Muster gefunden wurde.

PATTERN
: das gefundene Muster.

FILE_NAME
: den Namen der Datei inklusive Pfad, die Git ignoriert.

Ihr könnt mehrere Dateinamen an `git check-ignore` übergeben, wenn ihr möchtet, und die Namen selbst müssen nicht einmal den Dateien entsprechen, die in eurem Repository existieren.

Eine vollständige Liste aller ignorierten Dateien erhaltet ihr mit `git ls-files --ignored --exclude-standard --others`[6]. Mit `--exclude-standard` werden die Standard-`ignore`-Dateien gelesen und mit `--others` werden die nicht-versionierten Dateien statt der versionierten angezeigt:

```
$ git ls-files --ignored --exclude-standard --others
.DS_Store
_build/doctrees/clean-prep/bulwark.doctree
_build/doctrees/clean-prep/dask-pipeline.doctree
_build/doctrees/clean-prep/deduplicate.doctree
...
```

Gelegentlich möchtet ihr vielleicht die globale `~/.gitignore`-Datei umgehen um zu sehen, welche Dateien Git unabhängig von eurer Konfiguration immer ignoriert. Ihr könnt dies tun, indem ihr zu einer anderen `exclude`-Option wechselt, `--exclude-per-directory`, die nur die `.gitignore`-Dateien des Repositorys verwendet:

```
$ git ls-files --ignored --exclude-per-directory=.gitignore --others
docs/_build/doctrees/clean-prep/bulwark.doctree
docs/_build/doctrees/clean-prep/dask-pipeline.doctree
docs/_build/doctrees/clean-prep/deduplicate.doctree
...
```

Beachtet, dass die Datei `.DS_Store` nicht mehr als ignoriert aufgeführt wird.

Wenn ihr `--others` durch `--cached` ersetzt, listet `git ls-files` Dateien auf, die ignoriert werden würden, es sei denn, sie wurden bereits übertragen:

```
$ git ls-files --ignored --exclude-per-directory=.gitignore --cached
data/iris.csv
```

Möglicherweise habt ihr solche Dateien, weil jemand sie vor den relevanten Mustern in einer `.gitignore`-Datei hinzugefügt hat, oder weil jemand sie mit `git add --force` hinzugefügt hat. So oder so, wenn ihr die Datei nicht mehr mit Git verwalten wollt, könnt ihr sie mit dem folgenden Einzeiler aus der Git-Verwaltung nehmen, sie aber nicht löschen:

```
$ git ls-files --ignored --exclude-per-directory=.gitignore --cached | xargs -r git rm --
↪cached
rm 'data/iris.csv'
```

---

[6] git ls-files –ignored

Mit Git arbeiten

Die Arbeit an einem Projekt beginnen

Ein eigenes Projekt starten

`$ git init` *`MY_PROJECT`*
: erstellt ein neues, lokales Git-Repository.

  `MY_PROJECT`
  : wenn der Projektname angegeben wird, erzeugt Git ein neues Verzeichnis und initialisiert es.

    Wird kein Projektname angegeben, wird das aktuelle Verzeichnis initialisiert.

An einem Projekt mitarbeiten

`$ git clone` *`PROJECT_URL`*
: lädt ein Projekt mit allen Zweigen (engl.: *branches*) und der gesamten Historie vom entfernten Repository herunter.

  `--depth`
  : gibt die Anzahl der Commits an, die heruntergeladen werden sollen.

  `-b` **(Langform: `--branch`)**
  : gibt den Namen des entfernten Zweigs an, der heruntergeladen werden soll.

An einem Projekt arbeiten

`$ git status`
: zeigt den Status des aktuellen Zweiges im Arbeitsverzeichnisses an mit neuen, geänderten und bereits zum Commit vorgemerkten Dateien.

  `-v`
  : zeigt die Änderungen im Bühnenbereich als Diff an.

  `-vv`
  : zeigt auch die Änderungen im Arbeitsverzeichnis als zweites Diff an.

  Siehe auch:

  git status -v

`$ git add` *`FILE`*
: fügt eine Datei dem Bühnenbereich hinzu.

  `-p`
  : fügt Teile einer Datei dem Bühnenbereich hinzu.

  `-e`
  : die zu übernehmenden Änderungen können im Standardeditor bearbeitet werden.

`$ git diff` *`FILE`*
: zeigt Unterschiede zwischen Arbeits- und Bühnenbereich, z.B.:

```
$ git diff docs/productive/git/work.rst
diff --git a/docs/productive/git/work.rst b/docs/productive/git/work.rst
index e2a5ea6..fd84434 100644
--- a/docs/productive/git/work.rst
+++ b/docs/productive/git/work.rst
@@ -46,7 +46,7 @@

 :samp:`$ git diff {FILE}`
-    zeigt Unterschiede zwischen Arbeits- und Bühnenbereich.
+    zeigt Unterschiede zwischen Arbeits- und Bühnenbereich, :abbr:`z.B. (zum␣
↪Beispiel)`.
```

`index e2a5ea6..fd84434 100644` zeigt einige interne Git-Metadaten an, die ihr vermutlich nie benötigen werdet. Die Zahlen entsprechen den Hash-Kennungen der Git-Objektversionen.

Die übrige Ausgabe ist eine Liste von SOG. (sogenannten) *diff chunks*, deren Header von `@@`-Symbolen eingeschlossen ist. Jeder *diff chunk* zeigt in einer Datei vorgenommene Änderungen. In unserem Beispiel wurden 7 Zeilen ab Zeile 46 extrahiert und 7 Zeilen ab Zeile 46 hinzugefügt.

Standardmäßig führt `git diff` den Vergleich gegen `HEAD` aus. Wenn ihr im obigen Beispiel `git diff HEAD docs/productive/git/work.rst` verwendet, hat das denselben Effekt.

Mit `git diff` können Git-Referenzen auf Commits an `diff` übergeben werden. Neben `HEAD` sind einige weitere Beispiele für Referenzen Tags und Zweignamen, z.B. `git diff` *`MAIN..FEATURE_BRANCH`*. Der Punkt-Operator in diesem Beispiel zeigt an, dass die `diff`-Eingabe die Spitzen der beiden Zweige sind. Der gleiche Effekt tritt ein, wenn die Punkte weggelassen werden und ein Leerzeichen zwischen den Zweigen verwendet wird. Zusätzlich gibt es einen Operator mit drei Punkten: `git diff` *`MAIN...FEATURE_BRANCH`*, der ein Diff initiiert, bei dem der erste Eingabeparammeter *`MAIN`* so geändert wird, dass die Referenz der gemeinsame Vorfahre von *`MAIN`* und *`FEATURE`* ist.

Jeder Commit in Git hat eine Commit-ID, die ihr mittels `git log` erhaltet. Anschließend könnt ihr diese Commit-ID auch an `git diff` übergeben:

```
$ git log --pretty=oneline
af1a395a08221ffa83b46f562b6823cf044a108c (HEAD -> main, origin/main, origin/HEAD) :
↪memo: Add some git diff examples
d650de52306b63b93e92bba4f15be95eddfea425 :memo: Add „Debug .gitignore files“ to git␣
↪docs
...
$ git diff af1a395a08221ffa83b46f562b6823cf044a108c␣
↪d650de52306b63b93e92bba4f15be95eddfea425
```

`--staged`, `--cached`
: zeigt Unterschiede zwischen Bühnenbereich und Repository an.

`--word-diff`
: zeigt die geänderten Wörter an.

`$ git restore` *`FILE`*
: ändert Dateien im Arbeitsverzeichnis in einen Zustand, der Git zuvor bekannt war. Standardmäßig checkt Git `HEAD`, den letzten Commit des aktuellen Zweigs, aus.

Bemerkung: In Git < 2.23 steht euch `git restore` noch nicht zur Verfügung. In diesem Fall müsst ihr noch `git checkout` verwenden:

```
$ git checkout FILE
```

$ git commit
einen neuen Commit mit den hinzugefügten Änderungen machen.

-m '*COMMIT_MESSAGE*'
direkt in der Kommandozeile eine Commit-Message schreiben.

--dry-run --short
zeigt, was committet werden würde mit dem Status im Kurzformat.

$ git reset [--hard|--soft] [target-reference]
setzt die Historie auf einen früheren Commit zurück.

$ git rm *FILE*
entfernt eine Datei namens *FILE* aus dem Arbeits- und Bühnenbereich.

$ git stash
verschiebt die aktuellen Änderungen aus dem Arbeitsbereich in das Versteck (ENGL.: *stash*).

Um eure versteckten Änderungen möglichst gut unterscheiden zu können, empfehlen sich die folgenden beiden Optionen:

-p (Langform --patch)
erlaubt euch, Änderungen partiell zu verstecken, z.B.:

```
$ git stash -p
diff --git a/docs/productive/git/work.rst b/docs/productive/git/work.rst
index cff338e..1988ab2 100644
--- a/docs/productive/git/work.rst
+++ b/docs/productive/git/work.rst
@@ -83,7 +83,16 @@ An einem Projekt arbeiten
     ``list``
        listet die versteckten Änderungen auf.
     ``show``
-       zeigt die Änderungen in den versteckten Dateien an.
+       zeigt die Änderungen in den versteckten Dateien an, :abbr:`z.B. (zum
+       Beispiel)`
...
(1/1) Stash this hunk [y,n,q,a,d,e,?]? y
```

Mit ? erhaltet ihr eine vollständige Liste der Optionen. Die gebräuchlichsten sind:

Befehl	Beschreibung
y	Diese Änderung verstecken
n	Diese Änderung nicht in das Versteck übernehmen
q	Nur die bereits ausgewählten Änderungen werden in das Versteck übernommen
a	Diese und alle folgenden Änderungen übernehmen
e	Diese Änderung manuell editieren
?	Hilfe

branch
erstellt aus versteckten Dateien einen Zweig, z.B.:

```
$ git stash branch stash-example stash@{0}
Auf Branch stash-example
Zum Commit vorgemerkte Änderungen:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  (benutzen Sie "git restore --staged <Datei>..." zum Entfernen aus der Staging-
↪Area)
    neue Datei:     docs/productive/git/work.rst

Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
  (benutzen Sie "git restore <Datei>...", um die Änderungen im␣
↪Arbeitsverzeichnis zu verwerfen)
    geändert:       docs/productive/git/index.rst

stash@{0} (6565fdd1cc7dff9e0e6a575e3e20402e3881a82e) gelöscht
```

save ***MESSAGE***
: fügt den Änderungen eine Nachricht hinzu.

-u ***UNTRACKED_FILE***
: versteckt unversionierte Dateien.

list
: listet die versteckten Änderungen auf.

show
: zeigt die Änderungen in den versteckten Dateien an.

pop
: übernimmt Änderungen aus dem Versteck in den Arbeitsbereich und leert das Versteck, z.B.:

```
$ git stash pop stash@{2}
```

drop
: leert ein spezifisches Versteck, z.B.:

```
$ git stash drop stash@{0}
stash@{0} (defcf56541b74a1ccfc59bc0a821adf0b39eaaba) gelöscht
```

clear
: löscht alle eure Verstecke.

log und reflog

log

Siehe auch:

- Git's Database Internals III: File History Queries

Filtern und sortieren

git log [-n *COUNT*]
auflisten der Commit-Historie des aktuellen Zweiges.

-n
beschränkt die Anzahl der Commits auf die angegebene Zahl.

git log [--after="*YYYY-MM-DD*"] [--before="*YYYY-MM-DD*"]
Commit-Historie gefiltert nach Datum.

Auch relative Angaben wie `1 week ago` oder `yesterday` sind zulässig.

git log --author="*VEIT*"
filtert die Commit-Historie nach Autor*innen.

Es kann auch nach mehreren Autor*innen gleichzeitig gesucht werden, z.B.:

`git log --author="VEIT|VSC"`

git log --grep="*TERM*" [-i]
filtert die Commit-Historie nach regulären Ausdrücken in der Commit-Nachricht.

-i
ignoriert Groß- und Kleinschreibung.

git log -S"*FOO*" [-i]
filtert Commits nach bestimmten Zeilen im Quellcode.

-i
ignoriert Groß- und Kleinschreibung.

git log -G"*BA"**
filtert Commits nach regulären Ausdrücken im Quellcode.

git log -- *PATH/TO/FOO.PY*
filtert die Commit-Historie nach bestimmten Dateien.

git log *MAIN..FEATURE*
filtert nach unterschiedlichen Commits in verschiedenen Zweigen (Branches), in unserem Fall zwischen den Branches `MAIN` und `FEATURE`.

Dies ist jedoch nicht dasselbe wie `git log FEATURE..MAIN`. Nehmen wir folgendes Beispiel.

```
A - B main
 \
  C - D feature
```

git log *MAIN..FEATURE*
zeigt Änderungen in *FEATURE* an, die nicht in *MAIN* enthalten sind, also die Commits `C` und `D`.

git log *FEATURE..MAIN*
zeigt Änderungen in *MAIN* an, die nicht in *FEATURE* enthalten sind, also den Commit `B`.

git log *MAIN...FEATURE*
zeigt die Änderungen auf beiden Seiten an, also die Commits `B`, `C` und `D`.

$git log --follow *PATH/TO/FOO.PY*
Dies sorgt dafür, dass das Log Änderungen an einer einzelnen Datei anzeigt, auch wenn diese umbenannt oder verschoben wurde.

Ihr könnt `--follow` für einzelne Dateiaufrufe standardmäßig aktivieren, indem ihr die Option `log.follow` in eurer globalen Konfiguration aktiviert:

```
git config --global log.follow true
```

Dann müsst ihr nicht mehr `--follow` angeben, sondern nur noch den Dateinamen.

git log -L *LINE_START_INT|LINE_START_REGEX,LINE_END_INT|LINE_END_REGEX:PATH/TO/FOO.PY*
git log -L :*FUNCNAME_REGEX:PATH/TO/FOO.PY*
Mit der Option -L könnt ihr eine verfeinerte Suche durchführen, indem ihr das Log nur eines Teils einer Datei überprüft. Mit dieser Funktion könnt ihr die Historie einer einzelnen Funktion, einer Klasse oder eines anderen Code-Blocks gründlich durchforsten. Sie ist ideal, um herauszufinden, wann etwas erstellt und wie es geändert wurde, so dass ihr es getrost korrigieren, refaktorisieren oder löschen könnt.

Für umfassendere Untersuchungen könnt ihr auch mehrere Blöcke verfolgen. Hierfür könnt ihr mehrere `-L`-Optionen auf einmal verwenden.

git log --reverse
Üblicherweise zeigt das Protokoll den neuesten Commit zuerst an. Ihr könnt dies mit `--reverse` umkehren. Dies ist besonders nützlich, wenn ihr mit den bereits erwähnten Optionen `-S` und `-G` untersucht. Indem ihr die Reihenfolge der Commits umkehrt, könnt ihr schnell den ersten Commit finden, der eine bestimmte Zeichenfolge zur Codebasis hinzugefügt hat.

Ansicht

git log --stat --patch|-p

--stat
Den üblichen Metadaten wird noch eine eine Zusammenfassung der Anzahl der geänderten Zeilen pro Datei hinzugefügt.

--patch|-p
ergänzt die Ausgabe um den vollständigen Commit-Diff.

git log --oneline --decorate --graph --all|*FEATURE*
anzeigen des Verlaufsdiagramms mit Referenzen, ein Commit pro Zeile.

--oneline
Ein Commit pro Zeile.

--decorate
Die Präfixe `refs/heads/`, `refs/tags/` und `refs/remotes/` werden nicht ausgegeben.

--graph
Üblicherweise *glättet* das Log historische Zweige und zeigt Commits nacheinander an. Damit wird die parallele Struktur der Historie beim Zusammenführen von Zweigen verborgen. `--graph` stellt den Verlauf der Zweige in ASCII-Art dar.

--all|*FEATURE*
`--all` zeigt das Log für alle Zweige; *FEATURE* zeigt nur die Commits dieses Zweiges an.

reflog

Mit git reflog wird euer Git-Repository nicht ein zweites Mal überprüft. Stattdessen zeigt es das Reference-Log an, eine Aufzeichnung aller vorgenommenen Commits. Das Reflog verfolgt nicht nur Änderungen an einem Zweig, es zeichnet auch Änderungen am *aktuellen* Commit, den Wechsel des Zweiges, Rebasing, ETC. (et cetera) auf. Ihr könnt es benutzen, um alle unerreichbaren Commits zu finden, sogar solche auf gelöschten Zweigen. Damit könnt ihr viele, ansonsten destruktive Aktionen wieder rückgängig machen.

Schauen wir uns die Grundlagen der Verwendung von Reflog und einige typische Anwendungsfälle an.

Warnung: Das Reflog ist nur Teil eures lokalen Repository. Wenn ihr ein Projektarchiv löscht und neu klont, wird der neue Klon ein frisches, leeres Reflog haben.

Das Reflog für `HEAD` anzeigen

`git reflog`
: Wenn keine Optionen angegeben sind, zeigt der Befehl standardmäßig das Reflog für `HEAD` an. Es ist die Abkürzung für `git reflog show HEAD`. git reflog hat weitere Unterbefehle zur Verwaltung des Logs, aber `show` ist der Standardbefehl, wenn kein Unterbefehl übergeben wird.

```
$ git reflog
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{0}: merge my-feature-branch: Fast-forward
900844a HEAD@{1}: checkout: moving from my-feature-branch to main
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{2}: commit (amend): Add my feature and␣
↪more
982d93a HEAD@{3}: commit: Add my feature
900844a HEAD@{4}: checkout: moving from main to my-feature-branch
900844a HEAD@{5}: commit (initial): Initial commit
```

- Die Ausgabe ist ziemlich dicht.
- Jede Zeile ist ein Reflog-Eintrag, der neueste zuerst.
- Die Zeilen beginnen mit dem abgekürzten SHA des entsprechenden Commits, z.B. `12bc4d4`.
- Der erste Eintrag ist das, worauf `HEAD` derzeit verweist: `(HEAD -> main, my-feature)`.
- Die Namen `HEAD@\{N}` sind alternative Referenzen für die angegebenen Commits. `N` ist die Anzahl der zurückgehenden reflog-Einträge.
- Der restliche Text beschreibt die Änderung. Oben könnt ihr mehrere Arten von Einträgen sehen:
  - `commit: `*`MESSAGE`* für Commits
  - `commit (amend): `*`MESSAGE`* für eine Commit-Änderung
  - `checkout: moving from `*`SRC`*` TO `*`DST`* für einen Zweigwechsel

Es gibt viele weitere mögliche Arten von Einträgen. Der Text sollte so beschreibend sein, dass ihr den Vorgang auch ohne Nachschlagen in der Dokumentation nachvollziehen könnt. In den meisten Fällen werdet ihr solche Reflog-Einträge durchsehen wollen, um den entsprechenden Commit SHA zu finden.

Das Reflog für einen Zweig anzeigen

Ihr könnt euch auf Einträge für einen einzelnen Zweig fokussieren, indem ihr den expliziten Unterbefehl `show` und dem Zweignamen verwendet:

```
$ git reflog show my-feature-branch
12bc4d4 (HEAD -> main, my-feature-branch) my-feature-branch@{0}: commit (amend): Add my␣
↪feature and more
982d93a my-feature-branch@{1}: commit: Add my feature
900844a my-feature-branch@{2}: branch: Created from HEAD
```

Zeitstempel der Einträge anzeigen

Wenn ihr zwischen ähnlich betitelten Änderungen unterscheiden müsst, können die Zeitstempel helfen. Für relative Zeitstempel könnt ihr `--date=relative` verwenden:

```
$ git reflog --date=relative
12bc4d4 (HEAD -> main, my-feature) HEAD@{vor 37 Minuten}: merge my-feature-branch: Fast-
↪forward
900844a HEAD@{vor 37 Minuten}: checkout: moving from my-feature-branch to main
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{vor 37 Minuten}: commit (amend): Add my␣
↪feature and more
982d93a HEAD@{vor 38 Minuten}: commit: Add my feature
900844a HEAD@{vor 39 Minuten}: checkout: moving from main to my-feature-branch
900844a HEAD@{vor 40 Minuten}: commit (initial): Initial commit
```

Und für absolute Zeitstempel könnt ihr auch `--date=iso` verwenden:

```
$ git reflog --date=iso
12bc4d4 (HEAD -> main, my-feature) HEAD@{2024-01-11 15:26:53 +0100}: merge my-feature-
↪branch: Fast-forward
900844a HEAD@{2024-01-11 15:26:47 +0100}: checkout: moving from my-feature-branch to main
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{2024-01-11 15:26:11 +0100}: commit␣
↪(amend): Add my feature and more
982d93a HEAD@{2024-01-11 15:25:38 +0100}: commit: Add my feature
900844a HEAD@{2024-01-11 15:24:37 +0100}: checkout: moving from main to my-feature-branch
900844a HEAD@{2024-01-11 15:23:56 +0100}: commit (initial): Initial commit
```

Übergebt alle Optionen, die `git log` unterstützt

`git reflog show` hat die gleichen Optionen wie `git log`. So könnt ihr beispielsweise mit `--grep` nach Commit-Meldungen suchen, in denen *`my feature`* erwähnt wird, ohne die Groß- und Kleinschreibung zu berücksichtigen:

```
$ git reflog -i --grep 'my feature'
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{0}: merge my-feature: Fast-forward
12bc4d4 (HEAD -> main, my-feature-branch) HEAD@{2}: commit (amend): Add my feature and␣
↪more
982d93a HEAD@{3}: commit: Add my feature
```

Beachtet den Verfall von Einträgen

Reflog-Einträge verfallen nach einer gewissen Zeit, wenn Git den automatischen GC (engl.: Garbage Collection)-Prozess für euer Repository ausführt. Diese Verfallszeit wird durch zwei `gc.*`-Optionen gesteuert:

`gc.reflogExpire`
: Die allgemeine Verfallszeit, die standardmäßig auf 90 Tage eingestellt ist.

`gc.reflogExpireUnreachable`
: Die Verfallszeit für Einträge, die sich auf nicht mehr erreichbare Commits beziehen, ist standardmäßig auf 30 Tage eingestellt.

Ihr könnt diese Optionen auf einen längeren Zeitrahmen erhöhen, was allerdings nur selten sinnvoll sein dürfte.

Git-Tags

Git-Tags sind Referenzen, die auf bestimmte Commits in der Git-Historie verweisen. So können bestimmte Punkte in der Historie für eine bestimmte Version markiert werden, z.B. `v3.9.16`. Tags sind wie *Git-Verzweigungen*, die sich nicht ändern, also keine weitere Historie von Commits haben.

`git tag *TAGNAME*`
: erstellt einen Tag, wobei *`TAGNAME`* eine semantische Bezeichnung für den aktuellen Zustand des Git-Repositories ist. Dabei unterscheidet Git zwei verschiedene Arten von Tags: annotierte und leichtgewichtige Tags. Sie unterscheiden sich in der Menge der zugehörigen Metadaten.

  Annotierte Tags
  : Sie speichern nicht nur den *`TAGNAME`*, sondern auch zusätzliche Metadaten wie Namen und E-Mail-Adresse derjenigen Person, die den Tag erstellt hat sowie das Datum. Zudem haben annotierte Tags, ähnlich wie Commits Nachrichten. Ihr könnt solche Tags erstellen, z.B. mit `git tag -a *v3.9.16* -m '*Python 3.9.16*'`. Anshließend könnt ihr euch diese zusätzlichen Metadaten z.B. anzeigen lassen mit `git show *v3.9.16*`.

  Leichtgewichtige Tags
  : Leichtgewichtige Tags können z.B. mit `git tag *v3.9.16*` ohne die Optionen `-a`, `-s` oder `-m` erstellt werden. Sie erstellen eine Tag-Prüfsumme, die im `.git/`-Verzeichnis eures Repos gespeichert werden.

`git tag`
: listet die Tags eures Repos auf, z.B.:

```
v0.9.9
v1.0.1
v1.0.2
v1.1
...
```

  `git tag -l '*REGEX*'`
  : listet nur Tags auf, die zu einem regulären Ausdruck passen.

`git tag -a *TAGNAME COMMIT-SHA*`
: erstellt einen Tag für einen früheren Commit.

  Die vorangegangenen Beispiele erstellen Tags für implizite Commits, die auf `HEAD` verweisen. Alternativ kann `git tag` auch die Referenz auf einen bestimmten Commit übergeben werden, die ihr mit *log und reflog* erhaltet.

  Wenn ihr jedoch versucht, ein Tag mit dem gleichen Bezeichner wie ein bestehendes Tag zu erstellen, gibt Git eine Fehlermeldung aus, z.B. `Schwerwiegend: Tag '*v3.9.16*' existiert bereits`. Wenn ihr versucht, einen älteren Commit mit einem bestehenden Tag zu markieren, gibt Git denselben Fehler aus.

  Für den Fall, dass ihr einen bestehendes Tag aktualisieren müsst, könnt ihr die Option `-f` verwenden, z.B.:

```
$ git tag -af v3.9.16 595f9ccb0c059f2fb5bf13643bfc0cdd5b55a422 -m 'Python 3.9.16'
Tag 'v3.9.16' aktualisiert (war 4f5c5473ea)
```

`git push origin *TAGNAME*`

Das Teilen von Tags ist ähnlich wie der Push von Zweigen: standardmäßig werden mit `git push` keine Tags freigegeben, sondern sie müssen explizit an `git push` übergeben werden z.B.:

```
$ git tag -af v3.9.16 -m 'Python 3.9.16'
$ git push origin v3.9.16
Counting objects: 1, done.
Writing objects: 100% (1/1), 161 bytes, done.
Total 1 (delta 0), reused 0 (delta 0)
To git@github.com:python/cpython.git
 * [new tag]         v3.9.16 -> v3.9.16
```

Um mehrere Tags gleichzeitig zu pushen, übergebt die Option `--tags` an den Befehl `git push`. Andere erhalten die Tags bei `git clone` oder `git pull` des Repos.

Mit `git push --follow-tags` könnt ihr mit einem Commit auch gleichzeitig die zugehörigen annotierten Tags teilen.

Bemerkung: `--follow-tags` funktioniert nur für annotierte Tags, nicht für die leichtgewichtigen Tags.

Wenn ihr für alle zukünftigen Pushes `--follow-tags` verwenden wollt, könnt ihr dies konfigurieren mit

```
$ git config --global push.followTags true
```

Siehe auch:

- git push –follow-tags
- git config push.followTags

`git checkout *TAGNAME*`

wechselt in den Zustand des Repository mit diesem Tag und trennt `HEAD` ab. D.h. (Das heißt), dass alle Änderungen, die nun vorgenommen werden, das Tag nicht aktualisieren, sondern in einem losgelösten Commit landen, der nicht Teil eines Zweiges sein kann und nur direkt über den SHA-Hash des Commits erreichbar sein wird. Daher wird meist ein neuer Zweig erstellt, wenn solche Änderungen vorgenommen werden sollen, z.B. mit `git checkout -b v3.9.17 v3.9.16`.

`git tag -d *TAGNAME*`

löscht einen Tag, z.B.:

```
$ git tag -d v3.9.16
$ git push origin --delete v3.9.16
```

Git-Verzweigungen

Verzweigen ist eine Funktion, die in den meisten modernen Versionskontrollsystemen verfügbar ist. In anderen VCS-Systemen kann das Verzweigen eine teure Operation sein, die sowohl Zeit als auch Speicherplatz kostet; in Git sind Verzweigungen jedoch Verweise auf einen Schnappschuss eurer Änderungen. Wenn ihr eine neue Funktion hinzufügen oder einen Fehler beheben wollt, legt ihr einen neuen Zweig an, um eure Änderungen darin zu kapseln. Dadurch könnt ihr euch auf diese Aufgabe konzentrieren ohne zunächst gleichzeitige Änderungen im Hauptzweig berücksichtigen zu müssen. Umgekehrt hält es auch den Hauptzweig frei von fragwürdigem Code. Git-Zweige wurden daher ein fester Bestandteil des täglichen Arbeitsablaufs.

Ihr könnt euch Verzweigungen auch als ein neues Arbeitsverzeichnis mit neuen Staging-Bereich und Projektverlauf vorstellen wobei eure Commits zunächst in der Historie für den aktuellen Zweig aufgezeichnet werden.

Siehe auch:

- Git Branching - Branches auf einen Blick
- *Merge-Strategien: Merge vs. Squash vs. Rebase*

Gebräuchliche Befehle

`$ git branch [-a] [-l "`*`GLOB_PATTERN`*`"]`
zeigt alle lokalen Verzweigungen in einem Repository an.

`-a`
zeigt auch alle entfernten Verzweigungen an.

`-l`
beschränkt die Zweige auf diejenigen, die einem bestimmten Muster entsprechen.

`$ git branch --sort=-committerdate`
sortiert die Zweige nach dem Commit-Datum.

Mit `git config --global branch.sort -committerdate` könnt ihr diese Einstellung auch zu eurer Standardeinstellung machen.

`$ git branch [`*`BRANCH_NAME`*`]`
erstellt auf Basis des aktuellen `HEAD` einen neuen Zweig.

`$ git switch [-c] [`*`BRANCH_NAME`*`]`
wechselt zwischen Zweigen.

`-c`
erstellt einen neuen Zweig.

Bemerkung: In Git < 2.23 steht euch `git switch` noch nicht zur Verfügung. In diesem Fall müsst ihr noch `git checkout` verwenden:

`$ git checkout [-b] [`*`BRANCH_NAME`*`]`
ändert das Arbeitsverzeichnis in den angegebenen Zweig.

`-b`
erstellt den angegebenen Zweig, wenn dieser nicht schon besteht.

`$ git merge [`*`FROM_BRANCH_NAME`*`]`
verbindet den angegebenen mit dem aktuellen Zweig, in dem ihr euch gerade befindet, z.B.:

```
$ git switch main
$ git merge hotfix
Updating f42c576..3a0874c
Fast forward
 setup.py |    1 -
 1 files changed, 0 insertions(+), 1 deletions(-)
```

Fast forward
: besagt, dass der neue Commit direkt auf den ursprünglichen Commit folgte und somit der Zeiger (*branch pointer*) nur weitergeführt werden musste.

In anderen Fällen kann die Ausgabe z.B. so aussehen:

```
$ git switch main
$ git merge 'my-feature'
Merge made by recursive.
 setup.py |    1 +
 1 files changed, 1 insertions(+), 0 deletions(-)
```

recursive
: ist eine Merge-Strategie, die verwendet wird, sofern die Zusammenführung nur zu `HEAD` erfolgt.

Merge-Konflikte

Gelegentlich stößt Git beim Zusammenführen jedoch auf Probleme, z.B.:

```
$ git merge 'my-feature'
automatischer Merge von setup.py
KONFLIKT (Inhalt): Merge-Konflikt in setup.py
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie dann das␣
↪Ergebnis.
```

Die Historie kann dann z.B. so aussehen:

```
*   49770a2 (HEAD -> main) Fix merge conflict with my-feature
|\
| * 9412467 (my-feature) My feature
* | 46ab1a2 Hotfix directly in main
|/
* 0c65f04 Initial commit
```

Siehe auch:

- Git Branching - Einfaches Branching und Merging
- Git Tools - Fortgeschrittenes Merging

rerere, um aufgezeichnete Konfliktlösungen wiederzuverwenden

RERERE (engl.: reuse recorded resolutions) erleichtert euch, immer wieder dieselben Merge-Konflikte lösen zu müssen. Dies kann z.B. passieren, wenn ihr einen Commit in mehrere Zweige zusammenführen oder wenn ihr einen Zweig wiederholt rebasen müsst. Das Beheben von Merge-Konflikten erfordert Konzentration und Energie, und es ist Verschwendung, denselben Konflikt immer wieder neu zu lösen. git rerere wird jedoch nur selten direkt aufgerufen, sondern meist global aktiviert. Dann wird er automatisch von `git merge`, `git rebase` und `git commit` verwendet. Seine wichtigste Auswirkung besteht darin, dass er der Ausgabe dieser Befehle einige Meldungen hinzufügt. Ihr könnt ihn aktivieren mit:

```
$ git config --global rerere.enabled true
```

Schauen wir uns ein Beispiel für `git rerere` in Aktion an. Angenommen, ihr versucht eine Zusammenführung und stoßt auf Konflikte:

```
% git merge rerere-example
automatischer Merge von README.md
KONFLIKT (Inhalt): Merge-Konflikt in README.md
Preimage für 'README.md' aufgezeichnet.
Automatischer Merge fehlgeschlagen; beheben Sie die Konflikte und committen Sie dann das␣
↪Ergebnis.
```

`git rerere` schrieb die dritte Zeile, `Preimage für 'README.md' aufgezeichnet.`, D.H. (das bedeutet), dass der Konflikt aufgezeichnet wurde, bevor wir ihn beheben. Wenn wir den Konflikt nun beheben, können wir mit der Zusammenführung fortfahren, in unserem Beispiel mit:

```
$ git add README.md
$ git merge --continue
Konfliktauflösung für 'README.md' aufgezeichnet.
[main 5935d00] Merge branch 'rerere-example'
```

`git rerere` meldet nun `Konfliktauflösung für 'README.md' aufgezeichnet.`, D.H., dass es gespeichert hat, wie wir die Konflikte in dieser Datei aufgelöst haben.

Angenommen, ihr macht diese Zusammenführung rückgängig, weil ihr feststellgestellt habt, dass sie nicht fertig war:

```
$ git reset --keep @~
```

Später wiederholt ihr die Zusammenführung:

```
$ git merge rerere-example
Auto-merging README.md
CONFLICT (content): Merge conflict in README.md
Resolved 'README.md' using previous resolution.
Automatic merge failed; fix conflicts and then commit the result.
When finished, apply stashed changes with `git stash pop`
```

`git rerere` löste den Konflikt unter Verwendung der früheren Lösung, D.H., es hat eure vorherige Zusammenführung wiederverwendet. Prüft nun, ob die Datei korrekt ist, und fahrt dann fort:

```
$ git add README.md
$ git merge --continue
[main c922b21] Merge branch 'rerere-example'
```

`git rerere` speichert seine Daten innerhalb des `.git`-Verzeichnisses eures Git-Repositorys in einem `rr-cache`-Verzeichnis. Dabei solltet ihr zweierlei beachten:

1. Der Rerere-Cache ist lokal. Er wird nicht geteilt, wenn ihr `git push` durchführt, so dass eure Teamkollegen die von euch durchgeführten Merges nicht wiederverwenden können.
2. Git's automatische Garbage-Collection löscht Einträge aus dem `rr-cache`. Sie wird durch zwei Konfigurationsoptionen gesteuert:

   gc.rerereResolved
   bestimmt, wie lange Einträge für gelöste Konflikte aufbewahrt werden. Der Standardwert ist 60 Tage. Und mit `git config gc.rerereResolved` könnt ihr die Standardwerte für euer Projekt ändern.

   gc.rerereUnresolved
   bestimmt, wie lange Einträge für ungelöste Konflikte aufbewahrt werden. Der Standardwert ist 15 Tage.

Zweige löschen

`$ git branch -d [`*`BRANCH_NAME`*`]`
löscht den ausgewählten Zweig, wenn er bereits in einen anderen überführt wurde.

`-D` statt `-d` erzwingt die Löschung.

Entfernte Zweige

Bisher haben diese Beispiele alle lokalen Verzweigungen gezeigt. Der Befehl `git branch` funktioniert jedoch auch mit entfernten Zweigen. Um mit entfernten Zweigen arbeiten zu können, muss zunächst ein entferntes Repository konfiguriert und zur lokalen Repository-Konfiguration hinzugefügt werden:

`$ git remote add origin https://ce.cusy.io/veit/`*`NEWREPO`*`.git`

Entfernte Zweige hinzufügen

Nun kann der Zweig auch im entfernten Repository hinzugefügt werden:

`$ git push --set-upstream origin [`*`BRANCH_NAME`*`]`

Wollt ihr alle Zweige eines lokalen Repositories dem entfernten Repo hinzufügen, könnt ihr dies mit:

```
$ git push --set-upstream origin --all
```

Damit dies für Zweige ohne Tracking-Upstream automatisch geschieht, könnt ihr folgendes konfigurieren:

```
$ git config --global push.autoSetupRemote true
```

Entfernte Zweige löschen

Mit `git branch -d` löscht ihr die Zweige nur lokal. Um sie auch auf dem entfernten Server zu löschen, könnt ihr folgendes eingeben:

`$ git push origin --delete [`*`BRANCH_NAME`*`]`

Um entfernte Zweige auch bei euch lokal zu entfernen, könnt ihr `git fetch` mit der Option `--prune` oder `-p` ausführen. Ihr könnt dieses Verhalten auch zur Standardeinstellung machen, indem ihr `fetch.prune` aktiviert:

```
$ git config --global fetch.prune true
```

Siehe auch:

PRUNING

Zweige umbenennen

Ihr könnt Zweige umbenennen, z.B. mit

```
$ git branch --move master main
```

Dies ändert euren lokalen `master`-Zweig in `main`. Damit andere den neuen Zweig sehen können, müsst ihr ihn auf den entfernten Server pushen. Dadurch wird der `main`-Zweig auch auf dem entfernten Server verfügbar:

```
$ git push origin main
```

Der aktuelle Zustand eures Repository kann nun z.B. so aussehen:

```
$ git branch -a
* main
  remotes/origin/HEAD -> origin/master
  remotes/origin/main
  remotes/origin/master
```

- Euer lokaler `master`-Zweig ist verschwunden, da er durch den `main`-Zweig ersetzt wurde.
- Der `main`-Zweig ist auch auf dem entfernten Rechner vorhanden.
- Auch der `master`-Zweig ist jedoch auch noch auf dem entfernten Server vorhanden. Vermutlich werden also andere weiterhin den `master`-Zweig für ihre Arbeit verwenden, bis ihr die folgenden Änderungen vorgenommen habt:
  - Für alle Projekte, die von diesem Projekt abhängen, muss der Code und/oder die Konfiguration aktualisiert werden.
  - Die Konfigurationsdateien des test-runner müssen GGF. aktualisiert werden.
  - Build- und Release-Skripte müssen angepasst werden.
  - Die Einstellungen auf eurem Repository-Server wie der Standardzweig des Repository, Zusammenführungsregeln und anderes müssen angepasst werden.
  - Verweise auf den alten Zweig in der Dokumentation müssen aktualisiert werden.
  - Alle Pull- oder Merge-Requests, die auf den `master`-Zweig abzielen, sollten geschlossen werden.

Nachdem ihr all diese Aufgaben erledigt habt und sicher seid, dass der `main`-Zweig genauso funktioniert wie der `master`-Zweig, könnt ihr den `master`-Zweig löschen:

```
$ git push origin --delete master
```

Team-Mitgliedeer können ihre lokal noch vorhandenen Referenzen auf den `master`-Zweig löschen mit

```
$ git fetch origin --prune
```

Git rebase

Die Befehle `git rebase` und `git merge` ermöglichen das Zusammenführen von *Git-Verzweigungen*. Während `git merge` immer ein sich vorwärtsbewegender Änderungsansatz ist, verfügt `git rebase` über leistungsfähige Funktionen zum Umschreiben der Historie. Hier wollen wir die Konfiguration, Anwendungsfälle und Fallstricke genauer betrachten.

Dabei verschiebt `git rebase` eine Folge von Commits zu einem neuen Basis-Commit und kann so für *Feature-Branches*-Workflows nützlich sein. Intern erreicht Git dies, indem es neue Commits erstellt und diese auf die angegebene Basis anwendet; die gleichaussehenden Commits von Zweigen sind also völlig neue Commits.

Der Hauptgrund für `git rebase` ist, einen linearen Projektverlauf aufrechtzuerhalten. Wenn sich der Hauptzweig weiterentwickelt hat, seit ihr mit der Arbeit an einem Funktionszweig begonnen habt, wollt ihr vielleicht die letzten Aktualisierungen des Hauptzweigs in eurem Funktionszweig erhalten, aber die Historie eures Zweigs sauber halten. Dies hätte den Vorteil, dass ihr später ein sauberes `git merge` eures Funktionszweiges in den Hauptzweig durchführen könntet. Diese *saubere Historie* erleichtert euch auch, eine Regression mit *Regressionen finden mit git bisect* zu finden. Ein realistischeres Szenario wäre folgendes:

1. Im Hauptzweig wird ein Fehler in einer Funktion festgestellt, die früher einmal fehlerfrei funktionierte.
2. Durch die *saubere Historie* des Hauptzweigs sollte *log und reflog* schnell Rückschlüsse ermöglichen.
3. Sollte *log und reflog* nicht zum gewünschten Ergebnis führen, hilft vermutlich *git bisect* weiter. Dabei hilft `git bisect` die saubere Git-Historie bei der Suche nach der Regression.

Warnung: Die veröffentlichte Historie sollte nur in sehr seltenen Ausnahmefällen geändert werden, da die alten Commits durch neue ersetzt und es so aussehen würde, als wäre dieser Teil der Projektgeschichte plötzlich verschwunden.

Siehe auch:

git rebase: what can go wrong?

Bemerkung: `git rebase` wird auch kurz in *Jupyter Notebooks unter Git* und *Feature-Branches* behandelt

---

Rebase abhängiger Zweige mit `-update-refs`

Wenn ihr an einem großen Feature arbeitet, ist es oft hilfreich, die Arbeit auf mehrere Zweige zu verteilen, die aufeinander aufbauen.

Diese Zweige können jedoch umständlich zu verwalten sein, wenn ihr den Verlauf in einem früheren Zweig überschreiben müsst. Da jeder Zweig von den vorherigen Zweigen abhängt, führt das Umschreiben von Commits in einem Zweig dazu, dass die nachfolgenden Zweige nicht mehr mit der Historie verbunden sind.

Git 2.38 wird mit einer neuen `--update-refs`-Option für `git rebase` ausgeliefert, die solche Aktualisierungen für euch durchführt , ohne dass ihr jeden einzelnen Zweig manuell aktualisieren müsst und ohne dass die nachfolgenden Zweige ihre Historie verlieren.

Wenn ihr diese Option bei jedem Rebase verwenden möchtet, könnt ihr `git config --global rebase.updateRefs true` ausführen, damit sich Git so verhält, als ob die Option `--update-refs` immer angegeben ist.

Siehe auch:

rebase: add –update-refs option

Commits mit `git rebase` löschen

```
$ git rebase -i SHA origin/main
```

`-i` interaktiver Modus, in dem euer Standardeditor geöffnet wird und eine Liste aller Commits nach dem zu entfernenden Commit mit dem Hash-Wert *SHA* angezeigt wird, z.B.:

```
pick d82199e Update readme
pick 410266e Change import for the interface
...
```

Wenn ihr nun eine Zeile entfernt, so wird dieser Commit nach dem Speichern und Schließen des Editors gelöscht. Anschließend kann das entfernte Repository aktualisiert werden mit:

```
$ git push origin HEAD:main -f
```

Ändern einer Commit-Nachricht mit `git rebase`

Dies lässt sich ebenfalls einfach mit `git rebase` realisieren wobei ihr in eurem Editor nicht die Zeile löschen sondern in der Zeile `pick` durch `r` (*reword*) ersetzen müsst.

`rebase` als Standard-`git pull`-Strategie

Normalerweise holt und führt `git pull` neue Remote-Commits ohne Probleme zusammen. Meistens werden nur neue Commits aus dem entfernten Zweig hinzugefügt, ein sog. Fast-Forward-Merge. Wenn aber sowohl der lokale als auch der entfernte Zweig neue Commits haben, weichen die Zweige voneinander ab. Ihr müsst dann die verschiedenen Historien irgendwie in Einklang bringen. Standardmäßig führt ab Git 2.33.1 jede Abweichung dazu, dass `git pull` anhält und die folgende Meldung ausgibt:

```
$ git pull
Hinweis: Sie haben abweichende Branches und müssen angeben, wie mit diesen
Hinweis: umgegangen werden soll.
Hinweis: Sie können dies tun, indem Sie einen der folgenden Befehle vor dem
Hinweis: nächsten Pull ausführen:
Hinweis:
Hinweis:   git config pull.rebase false  # Merge
Hinweis:   git config pull.rebase true   # Rebase
Hinweis:   git config pull.ff only       # ausschließlich Vorspulen
Hinweis:
Hinweis: Sie können statt "git config" auch "git config --global" nutzen, um
Hinweis: einen Standard für alle Repositories festzulegen. Sie können auch die
Hinweis: Option --rebase, --no-rebase oder --ff-only auf der Kommandozeile nutzen,
Hinweis: um das konfigurierte Standardverhalten pro Aufruf zu überschreiben.
Schwerwiegend: Es muss angegeben werden, wie mit abweichenden Branches umgegangen werden␣
↪sollen.
```

Die Hinweise erlauben drei Optionen:

`git config pull.rebase false`
: führt die lokalen und entfernten Commits zusammen. Vor Git 2.33.1 verwendete Git immer diese Zusammenführung.

`git config pull.rebase true`
: Die lokalen Commits werden auf die Remote-Commits übernommen.

`git config pull.ff only`
: führt bei divergierenden Zweigen immer zu einen Fehler. Ihr könnt dann von Fall zu Fall mit `--no-rebase` (was `merge` bedeutet) oder `--rebase` entscheiden, ob ihr mergen oder rebasen wollt.

Tipp: Ich empfehle `git config pull.rebase true`, da Merging verwirrend sein kann. Das Rebasen der lokalen Commits auf die entfernten macht die Geschichte linear, was verständlicher ist.

Macht `rebase` zu eurer Standardstrategie mit:

```
$ git config --global pull.rebase interactive
```

Wenn `git pull` dann auf abweichende lokale und entfernte Zweige stößt, wird es ein `rebase` durchgeführt:

```
$ git pull
automatischer Merge von README.md
KONFLIKT (Inhalt): Merge-Konflikt in README.md
Fehler: Konnte e50dfe5... nicht anwenden
Hinweis: Resolve all conflicts manually, mark them as resolved with
Hinweis: "git add/rm <conflicted_files>", then run "git rebase --continue".
Hinweis: You can instead skip this commit: run "git rebase --skip".
Hinweis: To abort and get back to the state before "git rebase", run "git rebase --abort
↪".
Konnte e50dfe5... nicht anwenden
```

Änderungen zurücknehmen

Mit Git 2.23 kam `git restore` zum Rückgängigmachen von Datei-Änderungen hinzu. Vorher übernahm `git reset` diese Aufgabe, das aber auch noch andere Aufgaben hat:

`$ git restore`
: ändert Dateien im Arbeitsverzeichnis in einen Zustand, der Git zuvor bekannt war. Standardmäßig checkt Git `HEAD` den letzten Commit des aktuellen Zweigs aus.

Bemerkung:

In Git < 2.23 steht euch `git restore` noch nicht zur Verfügung. In diesem Fall müsst ihr noch `git checkout` verwenden:

`$ git checkout` *`FILE`*

`git restore [-S|--staged]` ***`FILE`***
: nimmt das Hinzufügen von Dateien zurück. Die Änderungen bleiben in eurem Arbeitsbereich erhalten, so dass ihr sie bei Bedarf ändern und wieder hinzufügen könnt.

  Der Befehl entspricht `git reset` *`PATH`*.

`git restore [-SW]` ***`FILE`***
: nimmt das Hinzufügen und Änderungen im Arbeitsbereich zurück.

`git restore [-s|--source]` ***`BRANCH FILE`***
: setzt eine Änderung auf die Version im Zweig *`BRANCH`* zurück.

git restore [-s|--source] @~ *FILE*
: setzt eine Änderung auf den vorherigen Commit zurück.

git restore [-p|--patch]
: lässt euch die rückgängig zu machenden Änderungen einzeln auswählen.

$ git reset [--hard | --mixed | --soft | --keep] *TARGET_REFERENCE*
: setzt die Historie auf einen früheren Commit zurück.

> **Warnung:** Das Risiko bei `reset` ist, dass Arbeit verloren gehen kann. Zwar werden Commits nicht unmittelbar gelöscht, allerdings können sie verwaisen, so dass es keinen direkten Pfad mehr zu ihnen gibt. Sie müssen dann zeitnah mit *reflog* gefunden und wiederhergestellt werden da Git üblicherweise alle verwaisten Commits nach 30 Tagen löscht.

```
$ git reset @~
```

@~
: nimmt den letzten Commit zurück wobei dessen Änderungen nun wieder in den Bühnenbereich übernommen werden.

Sind Änderungen im Bühnenbereich vorhanden, so werden diese in den Arbeitsbereich verschoben, z.B.:

```
$ echo 'My first repo' > README.rst
$ git add README.rst
$ git status
Auf Branch main
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-Area)
    neue Datei:     README.rst
$ git reset
$ git status
Auf Branch main
Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
    README.rst
```

@~3
: nimmt den letzten drei Commits zurück.

'@{u}'
: nimmt die entfernte Version (*Upstream*) des aktuellen Zweigs.

--hard
: verwirft die Änderungen auch im Staging- und Arbeitsbereich.

```
$ git status
Auf Branch main
Zum Commit vorgemerkte Änderungen:
  (benutzen Sie "git rm --cached <Datei>..." zum Entfernen aus der Staging-Area)
    neue Datei:     README.rst
$ git reset --hard
$ git status
Auf Branch main
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
nichts zu committen (erstellen/kopieren Sie Dateien und benutzen
Sie "git add" zum Versionieren)
```

--mixed
: setzt den Bühnen-, aber nicht den Arbeitsbereich zurück, D.H., die geänderten Dateien bleiben erhalten, werden aber nicht für den Commit markiert.

Tipp: Ich bevorzuge meist `--soft` gegenüber `--mixed`: es hält die rückgängig gemachten Änderungen getrennt, so dass alle zusätzlichen Änderungen explizit sind. Dies ist Besonders nützlich, wenn ihr im Bühnen- und Arbeitsbereich Änderungen an der gleichen Datei habt.

--soft
: nimmt den oder die Commits zurück, lässt jedoch Bühnen- und Arbeitsbereich unverändert.

--keep
: setzt den Bühnenbereich zurück und aktualisiert die Dateien im Arbeitsbereich, die sich zwischen `COMMIT` und `HEAD` unterscheiden, behält aber diejenigen bei, die sich zwischen Bühnen- und Arbeitsbereich unterscheiden, D.H., die Änderungen haben, aber noch nicht hinzugefügt wurden. Wenn eine Datei, die sich zwischen `COMMIT` und Bühnenbereich unterscheidet, nicht hinzugefügte Änderungen aufweist, wird `reset` abgebrochen.

  Ihr könnt euch dann mit euren nicht im Commit enthaltenen Änderungen befassen, sie vielleicht mit `git restore` rückgängig macht oder mit `git stash` versteckt, bevor ihr es erneut versucht.

Tipp: Viele andere Anleitungen empfehlen `--hard` für diese Aufgabe, wahrscheinlich weil es diesen Modus schon länger gibt. Dieser Modus ist jedoch riskanter, da er die nicht im Commit enthaltenen Änderungen unwiderruflich verwirft ohne Fragen zu stellen. Ich verwende jedoch `--keep` und wenn ich alle nicht zum Commit vorgesehenen Änderungen vor dem `reset` verwerfen will, verwende ich `git restore -SW`.

$ git revert *COMMIT SHA*
: erstellt einen neuen Commit und nimmt die Änderungen des angegebenen Commits zurück, sodass die Änderungen invertiert werden.

$ git fetch --prune *REMOTE*
: Remote-Refs werden entfernt wenn sie im Remote-Repository entfernt wurden.

$ git commit --amend
: aktualisiert und ersetzt den letzten Commit durch einen neuen Commit, der alle bereitgestellten Änderungen mit dem Inhalt des vorherigen Commits kombiniert. Wenn nichts bereitgestellt ist, wird nur die vorherige Commit-Nachricht neu geschrieben.

Referenz für häufige Befehle zum Zurücksetzen

Alle lokalen Änderungen an einem Zweig rückgängig machen

```
$ git reset --keep '@{u}'
```

Alle Commits im aktuellen Zweig rückgängig machen

git merge-base wählt den Commit aus, bei dem sich zwei Zweige getrennt haben. Übergebt `@` und `main`, um den Commit auszuwählen, bei dem der aktuelle Zweig von `main` abgezweigt ist. Setzt ihn zurück, um alle Commits auf dem lokalen Zweig rückgängig zu machen mit:

```
$ git reset --soft $(git merge-base @ main)
```

Alle Änderungen im aktuellen Zweig rückgängig machen

```
$ git reset --keep main
```

Commit im falschen Zweig zurücknehmen

Wenn ihr versehentlich einen Commit in einem bestehenden Zweig gemacht habt, anstatt zunächst einen neuen Zweig zu erstellen, könnt ihr das in den folgenden drei Schritten ändern:

1. Erstellt einen neuen Zweig mit `$ git branch` *NEW_BRANCH*
2. Nehmt den letzten Commit in eurem aktiven Branch zurück mit `$ git reset --keep @~`
3. Übernehmt die Änderungen in den neuen Zweig mit `$ git switch` *NEW_BRANCH*

Wiederherstellen eines gelöschten Zweigs

Angenommen, ihr habt versehentlich einen nicht zusammengefügten Zweig gelöscht, so könnt ihr den Zweig mit dem zugehörigen SHA neu erstellen:

```
$ git branch -D new-feature
Branch new-feature entfernt (war d53e431).
```

Die Ausgabe enthält den SHA-Commit, auf den der Zweig zeigte. Ihr könnt den Zweig mit diesem SHA neu erstellen:

```
$ git branch new-feature d53e431
```

Was aber, wenn ihr die Verzweigung gelöscht haben und der entsprechende Terminalverlauf verloren gegangen ist? Um die SHA wiederzufinden, könnt ihr die *reflog*-Ausgabe an `grep` übergeben:

```
$ git reflog | grep -A 1 new-feature
12bc4d4 HEAD@{0}: checkout: moving from new-feature to main
d53e431 HEAD@{1}: commit: Add new feature
12bc4d4 HEAD@{2}: checkout: moving from main to new-feature
12bc4d4 HEAD@{3}: merge my-feature: Fast-forward
```

`-A 1` zeigt nach nach jedem Treffer eine zusätzliche Zeile an. Die Ausgabe zeigt mehrere *reflog*-Einträge, die sich auf den Zweig beziehen. Der erste Eintrag zeigt einen Wechsel von `new-feature` zu `main`, mit dem Commit-SHA auf `main`. Der Eintrag davor ist die letzte Änderung an `new-feature` mit dem SHA zum Wiederherstellen:

```
$ git branch triceratops-enclosure 43f66f9
```

Standardmäßig könnt ihr einen solchen Zweig innerhalb von 30 Tagen nach dem Löschen des Zweigs speichern, da `gc.reflogExpireUnreachable` üblicherweise so eingestellt ist.

Rückgängigmachen einer Commit-Änderung

Kehren wir zum einleitenden Beispiel zurück. Stellt euch vor, ihr hättet einen Commit gemacht und ihn später geändert. Dann stellt ihr fest, dass die Änderung rückgängig gemacht werden sollte. Wie könnt ihr vorgehen? Wenn ihr die ursprüngliche Git-Commit-Ausgabe noch in Ihrem Terminalverlauf seht, könnt ihr die SHA von dort abrufen und die Änderung rückgängig machen. Aber wenn das nicht mehr möglich ist, ist es Zeit für das Reflog. Prüft das Reflog für den Zweig:

```
$ git reflog my-feature-branch
12bc4d4 (HEAD -> main, my-feature-branch) my-feature-branch@{0}: commit (amend): Add my␣
↪feature and more
982d93a my-feature-branch@{1}: commit: Add my feature
900844a my-feature-branch@{2}: branch: Created from HEAD
```

Der erste Eintrag, `commit (amend)`, zeigt die Erstellung des geänderten Commits an. Der zweite Eintrag zeigt den ursprünglichen Commit, zu der wir nun wiederum mit einem Hard Reset zurückkehren wollen:

```
$ git reset --hard 982d93a
```

Vielleicht möchtet ihr dann den Inhalt des geänderten Commits wiederherstellen, um ihn zu korrigieren und erneut zu ändern. Tut dies mit git `restore` aus dem geänderten Commit SHA, der oben in der vorherigen *reflog*-Ausgabe steht:

```
$ git restore -s 12bc4d4
```

Rückgängig machen eines fehlerhaften Rebase

Stellt euch vor, ihr arbeitet an einem Zweig `new-feature` mit drei Commits, wovon ihr den mittleren mit *Git rebase* rückgängig machen wollt:

```
$ git rebase -i main
```

```
 pick d53e431 Add new feature
-pick 329271a More performant implementation for the new feature
-pick 1d6c477 Add API docs
```

Versehentlich habt ihr jetzt jedoch auch den letzten Commit gelöscht. Falls ihr den SHA-Wert nicht mehr im Terminalverlauf sehen könnt, könnt ihr wieder die *reflog*-Ausgabe an `grep` übergeben:

```
$ git reflog| grep 'API docs'
1d6c477 HEAD@{2}: commit: Add API docs
```

Mit dieser SHA kann der Commit nun mit *Git Cherry-Pick* wiederhergestellt werden:

```
$ git cherry-pick 1d6c477
```

Entfernen einer Datei aus der Historie

Eine Datei kann vollständig aus Git-Historie des aktuellen Branches entfernt werden. Das ist nötig, wenn ihr beispielsweise aus Versehen Passwörter oder eine sehr große Datei zum Repository hinzugefügt habt.

```
$ git filter-repo --invert-paths --path path/somefile
$ git push --no-verify --mirror
```

Bemerkung: Informiert die Team-Mitglieder, dass sie erneut einen Klon des Repository erstellen sollten.

Entfernen einer Zeichenkette aus der Historie

Das Entfernen funktioniert auch mit einzelnen Wörtern oder Zeichenketten:

```
$ git filter-repo --message-callback 'return re.sub(b"^git-svn-id:.*\n", b"", message,␣
↪flags=re.MULTILINE)'
```

Siehe auch:

- git-filter-repo — Man Page
- git-reflog
- git-gc

Git Best Practices

Macht früh Commits

Macht euren ersten Commit nachdem ihr die initiale Installation fertiggestellt habt und noch bevor ihr erste Änderungen vornehmt.

Verwendet ihr ein Cookiecutter-Template, committet unmittelbar nach den folgenden Schritten:

```
$ pipenv run cookiecutter https://github.com/veit/cookiecutter-namespace-template.git
full_name [Veit Schiele]:
email [veit@cusy.io]:
github_username [veit]:
project_name [cusy.example]:
...
```

Anschließend könnt ihr die initialen Änderungen in ein neu erstelltes, leeres Repository auf GitHub einchecken:

```
$ cd cusy.example
$ git init
$ git add *
$ git commit -m 'Initial commit'
$ git remote add origin ssh://git@github.com:veit/cusy.example.git
$ git push -u origin main
```

Schließt unerwünschte Dateien aus

Temporäre Dateien, Jupyter Checkpoints und builds haben in einem Repository nichts zu suchen. Passwörter erst recht nicht. Die `.gitignore`-Datei enthält eine Liste von Dateipfaden, die git nicht hinzufügt, außer ihr wünscht das explizit.

Eine Vorlage mit `.gitignore`-Einträgen für Python-Projekte findet ihr im Repository dotfiles. Auf der Website gitignore.io könnt ihr euch `.gitignore`-Vorlagen für euer Projekt generieren lassen. Auch die `.gitignore`-Datei gehört in das Repository eingecheckt:

```
$ git add .gitignore
$ git commit -m 'Add .gitignore file'
```

Falls ihr versehentlich schon entsprechende Dateien in euer Git-Repository eingecheckt habt, beispielsweise einen `.ipynb_checkpoints`-Ordner, könnt ihr diese wieder entfernen mit:

```
$ git rm -r .ipynb_checkpoints/
```

Schreibt eine `README`-Datei

Auch eine `README`-Datei sollte in jedem Repository vorhanden sein, in der das Deployment und der grundsätzliche Aufbau des Codes beschrieben wird.

Macht oft Commits

Nach jeder abgeschlossenen Aufgabe und Teilaufgabe sollte ein Commit erfolgen. Auch nicht abgeschlossene Aufgaben können auf git gesichert werden. Als Faustregel gilt: Committe mindestens einmal pro Tag, nämlich kurz vor Feierabend. In Stoßzeiten kann es auch vorkommen, dass ihr alle 10 Minuten committet.

Häufige Commits erleichtern euch:

- das Eingrenzen von Fehlern
- das Verständnis für den Code
- die zukünftige Wartung und Pflege.

Falls ihr doch einmal mehrere Änderungen an einer Datei durchgeführt habt, könnt ihr diese auch später noch in mehrere Commits aufteilen mit:

```
$ git add -p my-changed-file.py
```

Ändert die veröffentlichte Historie nicht

Auch wenn ihr zu einem späteren Zeitpunkt herausfindet, dass ein Commit, der mit `git push` bereits veröffentlicht wurde, einen oder mehrere Fehler enthält, so solltet ihr dennoch niemals versuchen, diesen Commit ungeschehen zu machen. Vielmehr solltet ihr durch weitere Commits den oder die aufgetretenen Fehler zu beheben.

> **Warnung:** Die große Ausnahme zu dieser Regel sind Workflows mit `git-rebase` wie in *Feature-Branches*.

Wählt einen Git-Workflow

Wählt einen Workflow, der am besten zu eurem Projekt passt. Projekte sind keineswegs identisch und ein Workflow, der zu einem Projekt passt, muss nicht zwingend auch in einem anderen Projekt passen. Auch kann sich initial ein anderer Workflow empfehlen als im weiteren Fortschritt des Projekts.

Schreibt aussagekräftige Commit-Nachrichten

Aufschlussreiche und beschreibende Commit-Nachrichten erleichtern euch die Arbeit im Team ungemein. Sie ermöglichen anderen und euch selbst, eure Änderungen zu verstehen. Auch sind sie zu einem späteren Zeitpunkt hilfreich um nachvollziehen zu können, welches Ziel mit dem Code erreicht werden sollte.

Üblicherweise sollten kurze, 50–72 Zeichen lange Nachrichten angegeben werden, die in einer Zeile ausgegeben werden, *z.B. (zum Beispiel)* mit `git log --oneline`.

Mit `git blame` könnt ihr euch auch später noch für jede Zeile angeben lassen, in welcher Revision und von welchem Autor sie kam. Weitere Informationen hierzu findet ihr in der Git-Dokumentation: git-blame.

Bemerkung:

- A Note About Git Commit Messages

Gitmojis

Wenn ihr Gitmojis in euren Commit-Nachrichten verwendet, könnt ihr später leicht die Absicht des Commits erkennen.

Bemerkung:

- gitmoji.dev
- github.com/carloscuesta/gitmoji
- github.com/carloscuesta/gitmoji-cli
- Visual Studio Code Extension

GitLab

GitLab interpretiert bestimmte Commit-Nachrichten auch als Links, z.B.:

```
$ git commit -m "Expand section on meaningful commit messages (#21: Add
multi-line commit messages and close group/project#22)"
```

- zu Issues: *#NUMBER*
  - auch in anderen Projekten: *GROUP/PROJECT#NUMBER*
- zu Merge Requests: *!NUMBER*
- zu Snippets: *$NUMBER*

Dabei sollte es zu jedem Commit mindestens ein Ticket geben, das ausführlichere Hinweise zu den Änderungen geben sollte. Alternativ könnt ihr auch mehrzeilige Commit-Nachrichten schreiben, die diese Informationen enthalten, z.B. + (zum Beispiel) mit:

```
$ git commit -m 'Expand section on meaningful commit messages' -m 'Fix the serious␣
↪problem'
```

Oder, wenn ihr nur `git commit` eingebt, öffnet sich euer Editor, z.B. mit folgendem Text:

```
# Bitte geben Sie eine Commit-Beschreibung für Ihre Änderungen ein. Zeilen,
# die mit '#' beginnen, werden ignoriert, und eine leere Beschreibung
# bricht den Commit ab.
#
# Auf Branch main
```

Git erwartet, dass ihr eure Commit-Nachricht am Anfang der Datei einfügt. Nachdem ihr die Bearbeitung der Datei abgeschlossen habt, liest Git ihren Inhalt und fährt fort. Es *bereinigt* die Datei, indem es mit # kommentierte Zeilen und nachfolgende Leerzeilen entfernt. Wenn die Nachricht nach dem Aufräumen leer ist, bricht Git den Commit ab – das ist praktisch, wenn ihr merkt, dass ihr etwas vergessen habt. Andernfalls wird der Commit mit dem verbleibenden Inhalt erstellt. GitLab verwendet # jedoch als Präfix für die Nummer eines Items. Diese doppelte Bedeutung von # kann zu einer Verwechslung führen, wenn ihr eine Commit-Nachricht schreibt, die sich auf ein Item bezieht:

```
Expand section on meaningful commit messages

#21: Add multi-line commit messages

# Bitte geben Sie eine Commit-Beschreibung für Ihre Änderungen ein. Zeilen,
# die mit '#' beginnen, werden ignoriert, und eine leere Beschreibung
# bricht den Commit ab.
#
# Auf Branch main
# Ihr Branch ist auf demselben Stand wie 'origin/main'.
#
# Zum Commit vorgemerkte Änderungen:
#       geändert:       productive/git/best-practices.rst
#
```

Üblicherweise entfernt Git die Zeile, die mit #21 beginnt, so dass die Nachricht wie folgt aussieht:

```
Expand section on meaningful commit messages
```

Vermeidet dieses Missgeschick, indem ihr einen alternativen Bereinigungsmodus namens *Scissors* verwendet. Ihr könnt ihn global aktivieren mit:

```
$ git config --global commit.cleanup scissors
```

Dann beginnt Git jede neue Commit-Nachricht mit der *Scissors*-Zeile:

```
# ------------------------ >8 ------------------------
# Ändern oder entfernen Sie nicht die obige Zeile.
# Alles unterhalb von ihr wird ignoriert.
#
# Auf Branch main
# Ihr Branch ist auf demselben Stand wie 'origin/main'.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
#
# ...
#
```

Co-Autoren angeben

Wenn ihr mit einem Teammitglied an einem Commit arbeiten, ist es gut, dessen Beitrag mit dem `co-authored-by`-Trailer anzuerkennen. Trailer sind zusätzliche Metadaten am Ende der Commit-Nachricht, die eine *`KEY: VALUE`*-Syntax verwenden und wiederholt werden kann, um mehrere Werte aufzulisten:

```
Expand section on meaningful commit messages

#21: Add multi-line commit messages

co-authored-by: Kristian Rother <kristian.rother@cusy.io>
co-authored-by: Frank Hofmann <frank.hofmann@cusy.io>
```

GitLab analysiert die `co-authored-by`-Zeilen, um alle Avatare des Commits anzuzeigen und auch die Profilstatistiken der Co-Autoren zu aktualisieren usw (und so weiter).

Wartet euer Repository regelmäßig

Folgende Wartungsarbeiten solltet ihr regelmäßig durchführen:

Validiert das Repo

Der Befehl `git fsck` prüft, ob alle Objekte in der internen Datenstruktur von Git noch miteinander verknüpft sind.

Komprimiert das Repo

Spart Speicherplatz mit den Befehlen `git gc` bzw. `git gc --aggressive`.

Siehe auch:

- git gc
- Git Interna - Wartung und Datenwiederherstellung

Bereinigt Remote Tracking Branches

Nicht genutzte Zweige auf einem entfernten Server lassen sich mit `git remote update --prune` löschen. Noch besser ist, wenn ihr die Standardeinstellung so ändert, dass entfernt gelöschte Zweige auch bei `git fetch` und `git pull` bei euch lokal gelöscht werden. Dies erreicht ihr mit:

```
$ git config --global fetch.prune true
```

Überprüft vergessene Arbeiten

Mit `git stash list` seht ihr eine List von gespeicherten stashes. Diese könnt ihr mit `git stash drop` entfernen.

Überprüft eure Repositories auf unerwünschte Dateien

Mit Gitleaks könnt ihr eure Repositories regelmäßig auf ungewollt gespeicherte Zugangsdaten überprüfen.

Ihr könnt Gitleaks auch automatisch als GitLab-Action ausführen. Hierzu müsst ihr die Secret-Detection.gitlab-ci.yml-Vorlage z.B. in eine Stufe namens `secrets-detection` in eurer `.gitlab-ci.yml`-Datei einbinden:

```
include:
- template: Security/Secret-Detection.gitlab-ci.yml
```

Die Vorlage erstellt *Secret Detection*-Aufträge in eurer CI/CD-Pipeline und durchsucht den Quellcode eures Projekts nach *Secrets*. Die Ergebnisse werden als Secret Detection Report Artefakt gespeichert, den ihr später herunterladen und analysieren könnt.

Siehe auch:

- GitLab Secret Detection

Mit *Entfernen einer Datei aus der Historie* könnt ihr unerwünschte Dateien oder Zugangsdaten aus eurer Git-Historie entfernen.

Git-Workflows

Als Git-Workflow wird hier eine Empfehlung zur Verwendung von Git bezeichnet, die eine konsistente und effiziente Arbeitsweise ermöglicht. Da Git das Verzweigen und Zusammenführen im Vergleich zu älteren Versionierungssystemen wie SVN deutlich vereinfacht, ist eine Vielzahl von unterschiedlichen Workflows möglich. Es gibt nicht den einen idealen Prozess für die beste Interaktion mit Git.

Alle vorgestellten Workflows erwarten jedoch, dass alle im Team für Änderungen denselben Workflow verwenden. Daher sollte ein Team sich zu Beginn gemeinsam auf einen bestimmten Git-Workflow einigen, der ihnen für dieses Projekt am geeignetsten erscheint. Dabei spielen die Größe und Teamkultur eine Rolle, um die Komplexität des Workflows und die Fehlerzahl so gering wie möglich zu halten.

Im Folgenden behandeln wir einige dieser Git-Workflows.

Feature-Branches

Die Grundidee hinter Workflows mit Feature-Branches ist, dass die Entwicklung einzelner Features jeweils in einem dedizierten Branch und nicht im `main`-Branch stattfindet. Diese Kapselung erleichtert in einem Entwicklungsteam die Arbeit, da Veränderungen im `main`-Branch nicht stören und zunächst vernachlässigt werden können. Umgekehrt sollte so vermieden werden, dass der `main`-Branch durch unfertigen Code verunreinigt wird. Dieses zweite Argument erleichtert dann auch die kontinuierliche Integration mit anderen Komponenten.

Siehe auch:

- Feature Driven Development
- Martin Fowler: Feature Branch

Merge- oder Pull-Requests

Die Kapselung der Entwicklung einzelner Features in einem Branch ermöglicht zudem die Verwendung von sog. Merge- oder Pull-Requests um Änderungen mit anderen im Team diskutieren zu können und ihnen die Möglichkeit zu geben, ein Feature freizugeben, bevor es in das offizielle Projekt integriert wird. Wenn ihr in eurer Feature-Entwicklung auf Probleme stoßt, könnt ihr Merge- oder Pull-Requests jedoch auch nutzen um mit anderen im Team Lösungsmöglichkeiten zu erörtern.

Merge- oder Pull-Requests werden von Web-basierten Diensten wie GitHub, GitLab und Atlassian zum Review und Kommentieren der Änderungen bereitgestellt. Mit `@ID` in euren Kommentaren könnt ihr auch bestimmte Personen aus dem Projektteam direkt um Feedback bitten. Sofern ihr automatisiert testet, könnt ihr hier auch die Testergebnisse sehen; evtl. (eventuell) entspricht ja der Coding Style nicht euren Projektrichtlinien, oder die Testabdeckung ist ungenügend. In den Merge- oder Pull-Requests werden solche Diskussionen gefördert und dokumentiert ohne dass sie als unmittelbar als Commits im Repository erscheinen.

Warnung: Merge- oder Pull-Requests sind kein Bestandteil von Git selbst, sondern des jeweiligen Web-basierten Dienstes. Sie sind auch nicht standardisiert, sodass sie beim Wechsel auf einen anderen Dienst nur mühsam übernommen werden können.

Siehe auch:

- About pull requests
- Making a Pull Request
- Merge requests

GitHub Flow

GitHub Flow ist ein einfacher Workflow, bei dem es neben dem `main`-Branch nur verschiedene Feature-Branches geben sollte. Der Lebenszyklus eines Feature-Branches könnte dann so aussehen:

1. Alle Feature-Branches starten auf Basis des aktuellen `main`-Branches.

   Hierzu wechseln wir zunächst in den `main`-Branch, holen uns die neuesten Änderungen vom Server und aktualisieren unsere lokale Kopie des Repositories:

   ```
   $ git switch main
   $ git fetch origin
   $ git reset --hard origin/main
   ```

2. Erstellen des Feature-Branches.

   Wir erstellen einen Feature-Branch mit `git switch -c` und der Nummer des Tickets in der Aufgabenverwaltung, das dieses Feature beschreibt.

   ```
   $ git switch -c 17-some-feature
   ```

3. Hinzufügen und Committen von Änderungen.

   ```
   $ git add SOMEFILE
   $ git commit
   ```

4. Pushen des Feature-Branches mit den Änderungen.

Durch das Pushen des Feature-Branches mit Deinen Änderungen erstellt Ihr nicht nur eine Sicherungskopie eurer Änderungen, sondern ihr ermöglicht auch anderen im Team, sich die Änderungen anzuschauen.

```
$ git push -u origin 17-some-feature
```

Der `-u`-Parameter fügt den `17-some-feature`-Branch dem Upstream-Git-Server (`origin`) als Remote-Branch hinzu. Zukünftig könnt ihr dann in diesen Branch pushen ohne weitere Parameter angeben zu müssen.

5. Merge- oder Pull-Request stellen

   Sobald ihr ein Feature fertiggestellt habt, wird dieses nicht sofort in den `main`-Branch gemergt, sondern ein Merge- oder Pull-Request erstellt, durch den andere aus dem Entwicklungsteam die Gelegenheit erhalten, eure Änderungen zu überprüfen. Alle Änderungen an diesem Branch werden nun ebenfalls in diesem Merge- oder Pull-Request angezeigt.

6. Zusammenführen

   Sobald euer Merge- oder Pull-Request akzeptiert wird, müsst ihr zunächst sicherstellen, dass euer lokaler `main`-Branch mit dem Upstream-`main`-Branch synchronisiert ist; erst dann könnt ihr den Feature-Branch in den `main`-Branch mergen und schließlich den aktualisierten `main`-Branch zurück in den Upstream-`main`-Branch pushen. Dies wird jedoch nicht selten zu einem Merge-Commit führen. Dennoch hat dieser Workflow den Vorteil, dass klar zwischen der Feature-Entwicklung und dem Zusammenführen unterschieden werden kann.

Simple-Git-Workflow

Auch Atlassian empfiehlt eine ähnliche Strategie, wobei sie jedoch ein *rebase* der Feature-Branches empfehlen. Hiermit erhaltet ihr einen linearen Verlauf, indem die Änderungen im Feature-Branch vor dem Zusammenführen mit einem Fast-Forward-Merge an die Spitze des `main`-Branch verschoben werden.

1. Verwendet `rebase`, um euren Feature-Branch auf dem neuesten Stand von `main` zu halten:

   ```
   $ git fetch origin
   $ git rebase -i origin/main
   ```

   In dem selteneren Fall, dass andere aus dem Team auch im selben Feature-Zweig arbeiten, solltet ihr auch deren Änderungen übernehmen:

   ```
   $ git rebase -i origin/17-some-feature
   ```

   Löst zu diesem Zeitpunkt alle Konflikte, die sich aus `rebase` ergeben. Dies sollte am Ende der Feature-Entwicklung zu einer Reihe von sauberen Merges geführt haben. Außerdem bleibt die Historie eurer Feature-Zweige sauber und fokussiert, ohne störendes Rauschen.

2. Wenn ihr bereit für Feedback seid, pusht euren Zweig:

   ```
   $ git push -u origin 17-some-feature
   ```

   Anschließend könnt ihr einen Merge- oder Pull-Request stellen.

   Nach diesem Push könnt ihr als Reaktion auf Feedback den entfernten Zweig immer wieder aktualisieren.

3. Nachdem die Überprüfung abgeschlossen solltet ihr eine letzte Bereinigung der Commit-Historie des Feature-Zweiges vornehmen, um unnötige Commits zu entfernen, die keine relevanten Informationen liefern.

4. Wenn die Entwicklung abgeschlossen ist, führt die beiden Zweige mit `-no-ff` zusammen. Dadurch bleibt der Kontext der Arbeit erhalten und es wird einfach sein, das gesamte Feature bei Bedarf zurückzunehmen:

```
$ git switch main
$ git pull origin main
$ git merge --no-ff 17-some-feature
```

Der Simple-Git-Workflow über `rebase` schafft eine strikt lineare Versionshistorie. In der linearen Historie ist es tendenziell leichter, Änderungen nachzuvollziehen und Fehler zu finden, beispielsweise über *bisect*.

Zusammenfassung

Die Vorteile von Feature-Branches-Workflows sind vor allem

- Features werden in einzelnen Branches isoliert, sodass jedes Teammitglied unabhängig arbeiten kann.
- Gleichzeitig wird die Zusammenarbeit im Team enger über Merge- oder Pull-Requests.
- Das zu verwaltende Code-Inventar bleibt relativ klein da die Feature-Branches üblicherweise schnell in den `main` übernommen werden können.
- Die Workflows entsprechen den üblichen Methoden kontinuierlicher Integration.

Sie können jedoch nicht beantworten, wie Deployments in unterschiedliche Umgebungen oder die Aufteilung in verschiedene Releases erfolgen sollen. Mögliche Antworten hierfür werden in *Deployment und Release Branches* beschrieben.

Siehe auch:

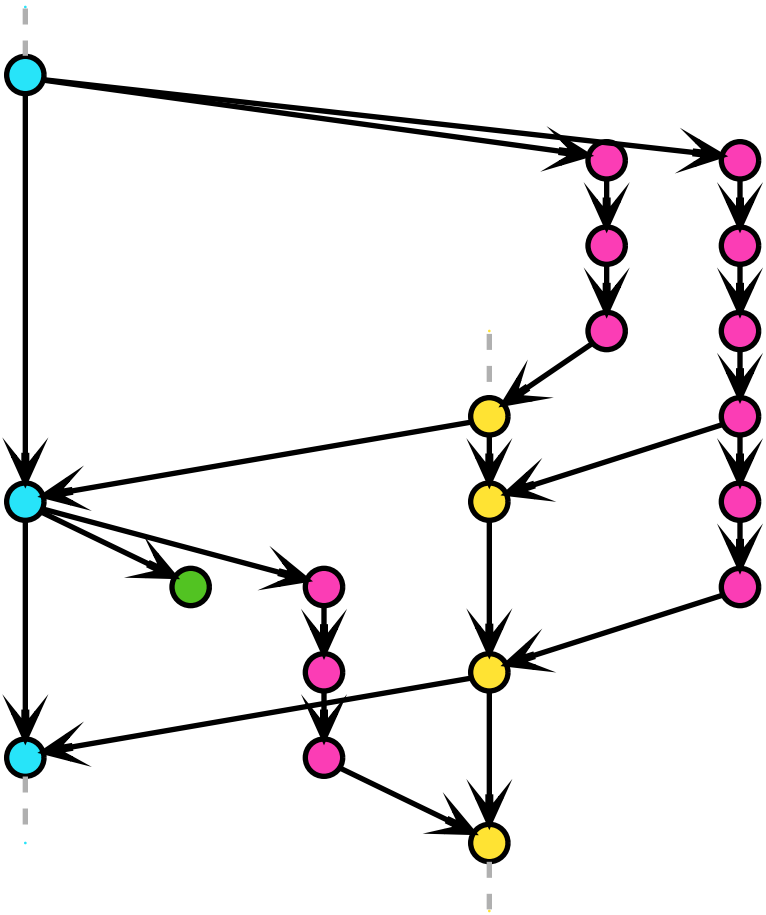
Beide Varianten mit Feature Branches sind stark vereinfachte Alternativen des deutlich komplexeren *Git Flow*.

Deployment und Release Branches

Deployment-Branches

Ein oder mehrere Deployment-Branches empfehlen sich, wenn ihr z.B. den Release-Zeitpunkt nicht selbst bestimmen könnt, wie bei einer iOS-Anwendung, die die App-Store-Validierung bestehen muss, oder wenn euch nur ein bestimmtes Zeitfenster für die Bereitstellung zur Verfügung steht. In diesen Fällen empfiehlt sich ein Deployment-Branch, der den bereitgestellten Code widerspiegelt. Ein solcher Arbeitsablauf verhindert dann zusätzliche Arbeitsaufwände bei *Git rebase* und *Git-Tags*.

Angenommen, ihr verfügt über eine `development`-, `staging`- und `production`-Umgebung, dann wird zunächst ein Merge- oder Pull-Request für eine Feature-Entwicklung beim `staging`-Branch gestellt. Sofern die Qualitätsprüfung dort bestanden wurde und der Code produktionsreif ist, können die Änderungen in den `main`-Branch übernommen werden. Dieser Vorgang kann sich für neue Features mehrfach wiederholen, bis z.B. der Zeitpunkt für das *Going Live* dieser Änderungen gekommen ist und ein Deployment-Branch erstellt werden kann.



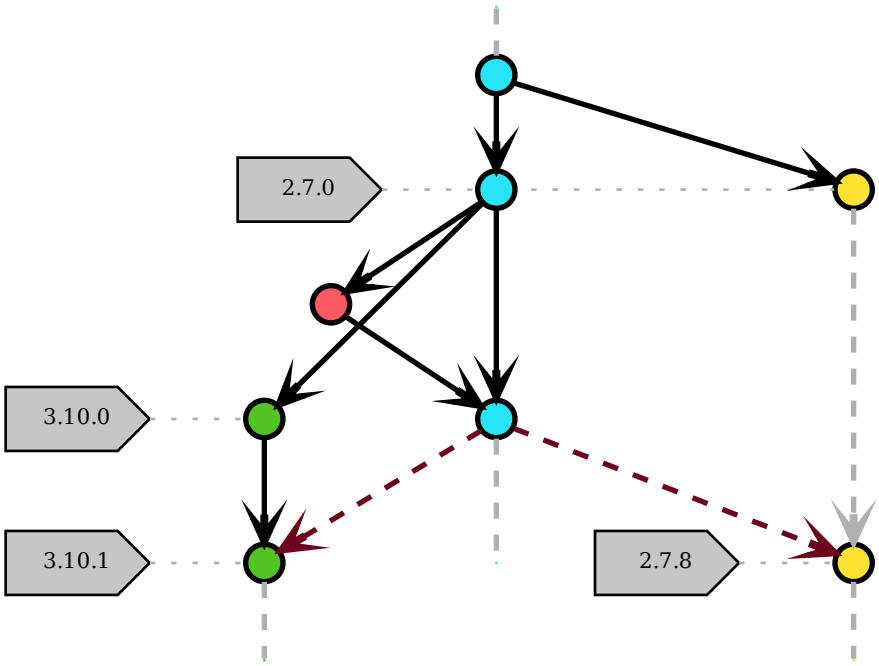
Release-Branches

Wenn Software an Kunden geliefert werden soll, empfehlen sich SOG. Release-Branches. In diesen Fällen sollte jeder Branch eine *Minor Version* erhalten, also z.B. `2.7` oder `3.10`. Üblicherweise werden diese Branches so spät wie möglich aus dem `main`-Branch erzeugt um bei Bugfixes die Anzahl der Merges, die auf mehrere Branches verteilt werden müssen, zu reduzieren. Nachdem ein neuer Release-Branch erstellt wurde, erhält dieser nur noch Bugfixes. Meist werden diese zunächst in den `main`-Branch übernommen und kommen anschließend von dort mit *Git Cherry-Pick* in den Release-Branch, z.B.:

```
$ git switch 3.10
$ git cherry-pick 61de025
[3.10 b600967] Fix bug #17
 Date: Thu Sep 15 11:17:35 2022 +0200
 1 file changed, 9 insertions(+)
```

Dieser *upstream first*-Ansatz wird U.A. von Google und Red Hat verwendet. Jedes Mal, wenn ein Bugfix in einen Release-Branch übernommen wurde, wird das Release mit einem *Git-Tags* um eine Patch-Version angehoben, S.A.

(siehe auch) Semantic Versioning.



Trunk-Based Development

Trunk Based Development empfiehlt kurzlebige Themenzweige, die zu einem einzigen `main`-Zweig zusammengeführt werden. TBD (Trunk-Based Development) führt zu einem leicht zu verwaltenden linearen Verlauf.

Trunk Based Development eignet sich hervorragend für ein-Personen-Projekte. Verzweigungen sind nicht notwendig, das Verwenden einer Versionskontrolle zahlt sich aber auch für einen selbst schnell aus.

In kleineren Entwicklungsteams übertragen wir kleine Commits direkt in den Trunk (oder den `main`-Branch), wobei vor der Integration zunächst der Build erfolgreich ausgeführt sein muss.

Trunk Based Development in großem Maßstab wird am besten mit kurzlebigen Feature-Zweigen durchgeführt, wobei eine Person maximal über ein paar Tage entwickelt und die Änderungen anschließend mit Pull- oder Merge-Requests, Code-Review und Build-Automatisierung in den Trunk (oder `main`) integriert werden.

Git Flow

Git Flow war einer der ersten Vorschläge zur Verwendung von Git-Branches. Es empfahl einen `main`-Branch und einen separaten `develop`-Branch sowie diverse weitere Branches für Features, Releases und Hotfixes. Die verschiedenen Entwicklungen sollten im `develop`-Branch zusammengeführt werden, anschließend in den `release`-Branch überführt werden und schließlich im `main`-Branch landen.

Nachteile von Git Flow

Git Flow ist zwar ein wohldefinierter, aber komplexer Standard, der in der Praxis folgende zwei Probleme erzeugt:

- Die meisten Entwickler und Werkzeuge gehen von der Annahme aus, dass der `main`-Branch der Hauptzweig ist von dem aus `branch` und `merge` ausgeführt wird. Bei Git Flow entsteht nun zusätzlicher Aufwand da immer zunächst in den `develop`-Branch gewechselt werden muss.
- Auch die `hotfixes`- und `release`-Branches bringen eine zusätzliche Komplexität, die nur in den seltensten Fällen Vorteile bringen dürfte.

Als Reaktion auf die Probleme von Git Flow entwickelten GitHub und Atlassian einfachere Alternativen, die sich meist auf sog. *Feature-Branches* beschränken.

Siehe auch:

Vincent Driessen: A successful Git branching model

Erste Schritte

Git-flow ist nur eine abstrakte Vorstellung eines Git-Workflows, bei dem die Zweige und die Zusammenführung vorgegeben werden. Es gibt mit git-flow auch Software, die bei diesem Workflow unterstützen soll.

Installation

```
$ wget -q -O - --no-check-certificate https://github.com/nvie/gitflow/raw/develop/
↪contrib/gitflow-installer.sh | bash
```

```
$ sudo apt install git-flow
```

```
$ brew install git-flow
```

Initialisieren

`git-flow` ist ein sog. Wrapper für Git. Dabei initiiert der Befehl `git flow init` nicht nur ein Verzeichnis, sondern erstellt auch Verzweigungen für Dich:

```
$ git flow init
Leeres Git-Repository in /home/veit/my_repo/.git/ initialisiert
No branches exist yet. Base branches must be created now.
Branch name for production releases: [master] main
Branch name for "next release" development: [develop]
```

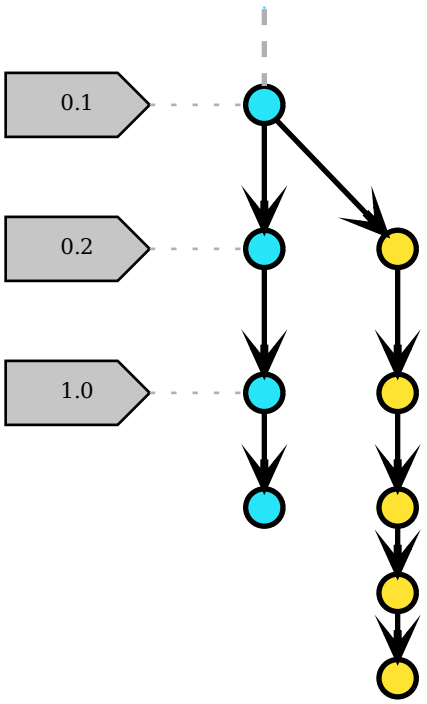
(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
How to name your supporting branch prefixes?
Feature branches? [feature/]
Bugfix branches? [bugfix/]
Release branches? [release/]
Hotfix branches? [hotfix/]
Support branches? [support/]
Version tag prefix? []
Hooks and filters directory? [.git/hooks]
```

Alternativ hättet ihr auch folgendes eingeben können:

```
$ git branch develop
$ git push -u origin develop
```



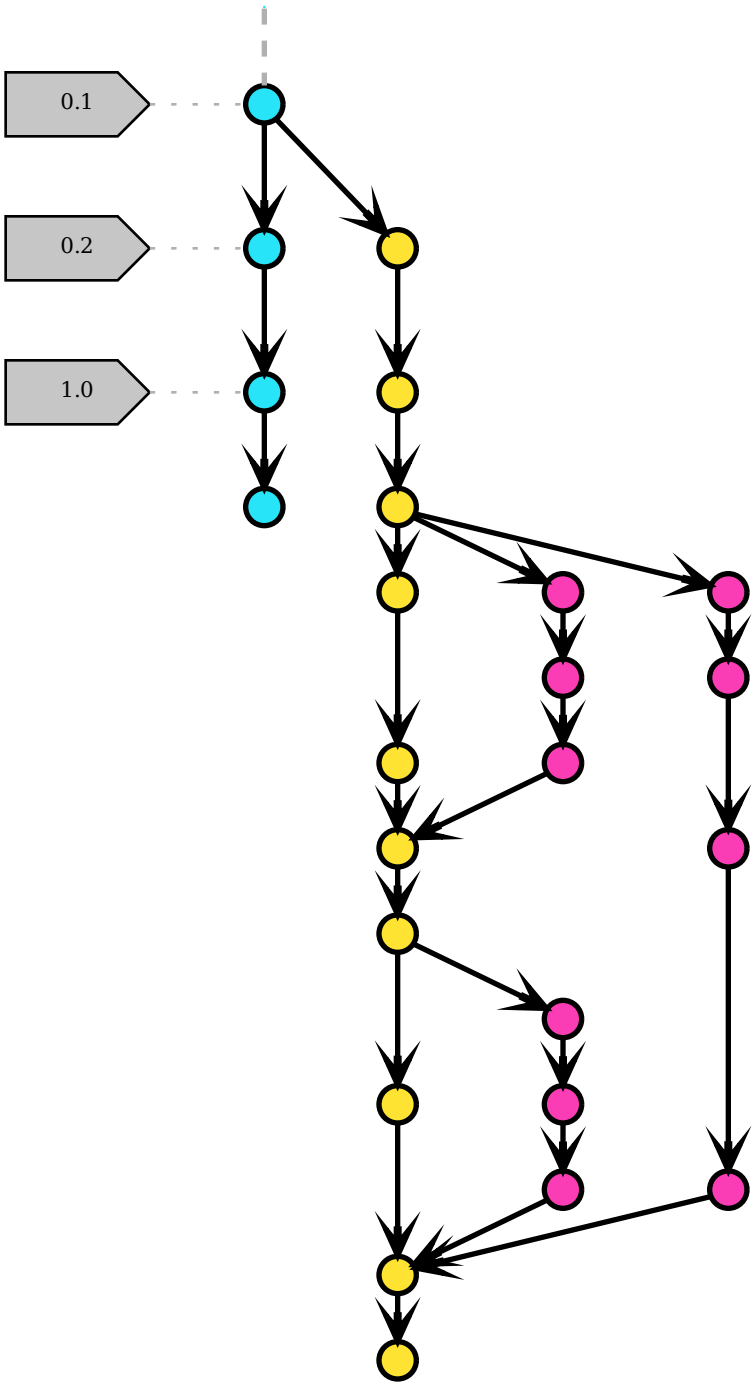
Dieser Workflow sieht zwei Zweige vor, um die Historie des Projekts aufzuzeichnen:

main
: enthält den offiziellen Release-Verlauf, wobei alle Commits in diesem Zweig mit einer Versionsnummer getaggt sein sollten.

develop
: integriert die Features.

Feature-Branches

Jedes neue Feature sollte in einem eigenen Branch erstellt werden, der jederzeit zum entfernten Repository gepusht werden kann. Dabei wird ein Feature-Branch jedoch nicht aus dem `main`-Branch erstellt sondern aus dem `develop`-Branch; und wenn ein Feature fertig ist, wird es auch zurück in den `develop`-Branch gemergt.



Ihr könnt solche Feature-Branches erstellen mit `git flow`

```
$ git flow feature start 17-some-feature
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Zu neuem Branch 'feature/17-some-feature' gewechselt

Summary of actions:
- A new branch 'feature/17-some-feature' was created, based on 'develop'
- You are now on branch 'feature/17-some-feature'
...
```

… oder mit

```
$ git switch -c feature/17-some-feature
Zu neuem Branch 'feature/17-some-feature' gewechselt
```

Umgekehrt könnt ihr euren Feature-Branch abschließen mit

```
$ git flow feature finish 17-some-feature
Zu Zweig »develop« gewechselt
Bereits aktuell.
Branch feature/17-some-feature entfernt (war 653e88a).
...
```

… oder mit

```
$ git switch develop
$ git merge feature/17-some-feature
$ git branch -d feature/17-some-feature
Branch feature/17-some-feature entfernt (war 11a9417).
```

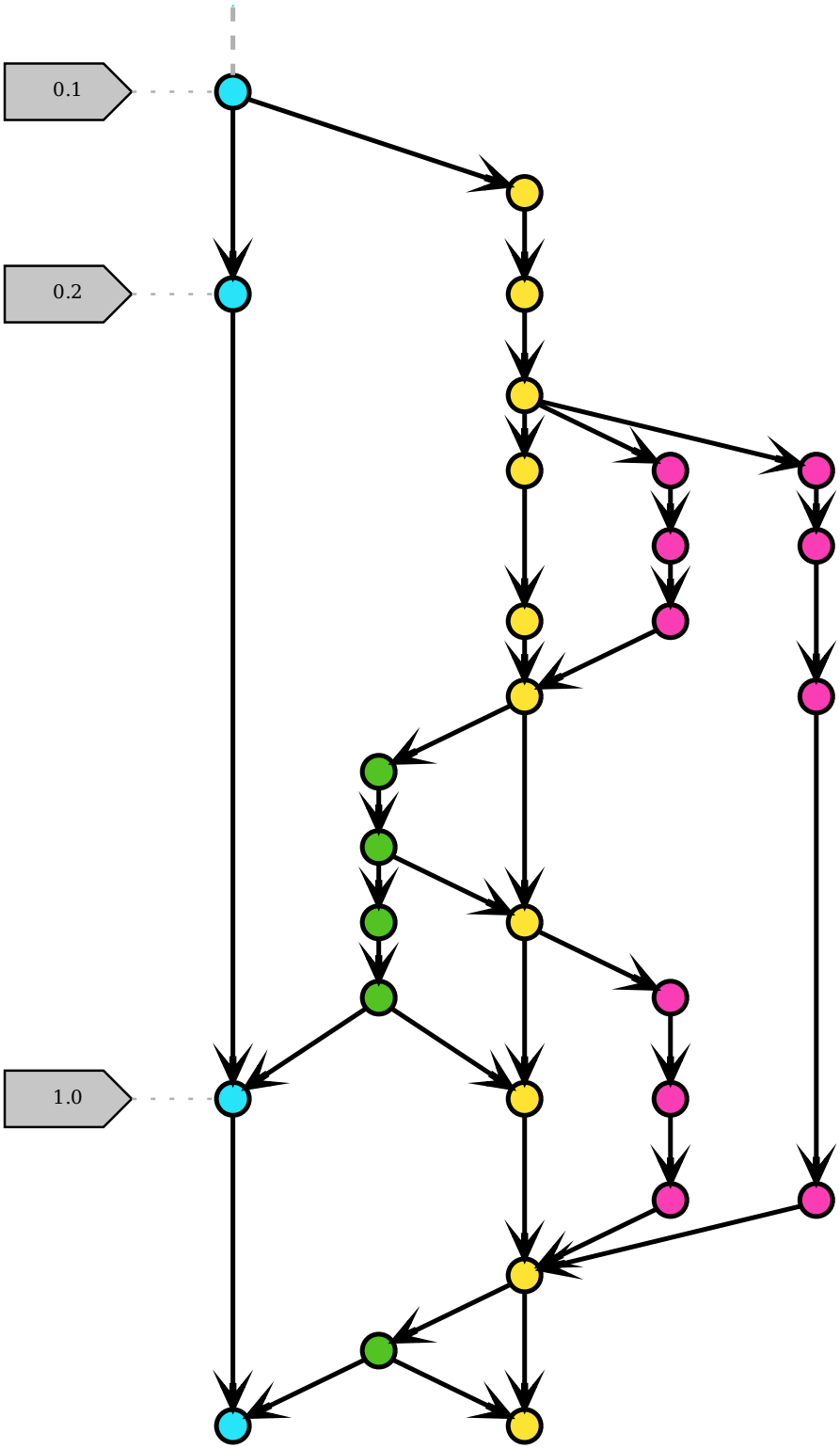
Release-Branches

Wenn der `develop`-Branch genügend Features für ein Release enthält oder ein festgelegter Release-Termin ansteht, wird vom `develop`-Branch ein `release`-Branch erstellt, zu dem ab diesem Zeitpunkt keine neuen Features mehr hinzukommen sollten, sondern nur noch Bugfixes und auf dieses Release bezogene Änderungen. Kann das Release ausgeliefert werden, wird der `release`-Branch einerseits in den `main`-Branch gemergt und mit einer Versionsnummer getaggt, andererseits zurück in den `develop`-Branch gemergt, der sich seit der Erstellung des `release`-Branch weiterentwickelt haben dürfte.

```
$ git flow release start 0.1.0
Zu neuem Branch 'release/0.1.0' gewechselt
...
$ git flow release finish '0.1.0'
Zu Zweig »main« gewechselt
...
Branch release/0.1.0 entfernt (war 11a9417).

Summary of actions:
- Release branch 'release/0.1.0' has been merged into 'main'
- The release was tagged '0.1.0'
- Release tag '0.1.0' has been back-merged into 'develop'
- Release branch 'release/0.1.0' has been locally deleted
- You are now on branch 'develop'
```

… oder

```
$ git switch develop
$ git branch develop/0.1.0
...
$ git switch main
$ git merge release/0.1.0
$ git tag -a 0.1.0
$ git switch develop
$ git merge release/0.1.0
$ git branch -d release/0.1.0
```


0.1
0.2
1.0

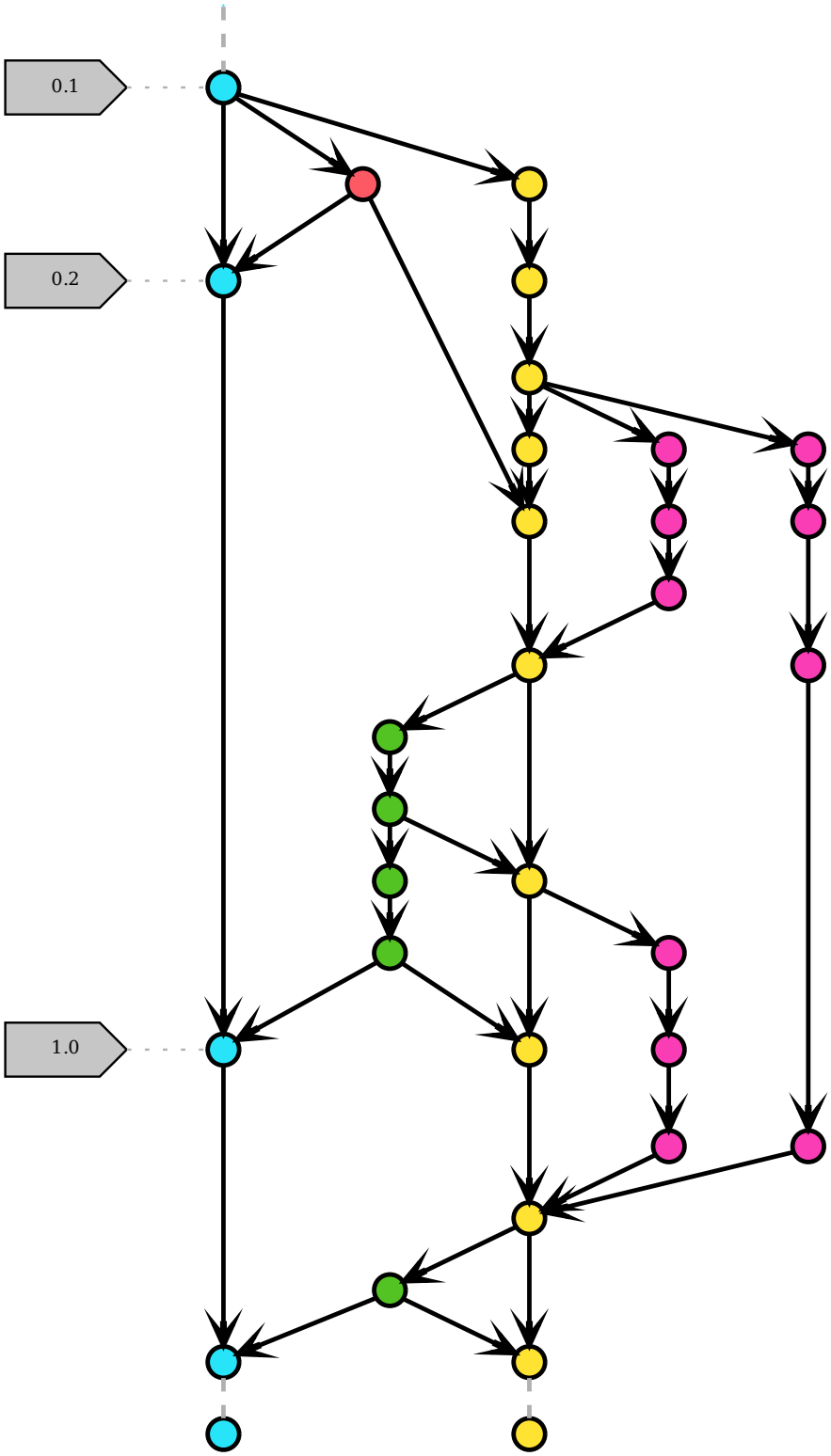
Hotfix-Branches

Hotfix-Branches eignen sich für schnelle Patches von Produktion-Versionen. Sie sind Release-Branches und Feature-Branches ähnlich, basieren jedoch auf dem `main`- statt auf dem `develop`-Branch. Damit ist er der einzige Branch, der direkt vom `main`-Branch geforkt werden sollte. Sobald der Hotfix abgeschlossen wurde, sollte er sowohl in den `main`- als auch in den `develop`-Branch und GGF. in den aktuellen `release`-Branch gemergt werden. Der `main`-Branch sollte außerdem mit einer neuen Versionsnummer getaggt werden.

```
$ git flow hotfix finish 37-some-bug
Zu Zweig »develop« gewechselt
Merge made by the 'recursive' strategy.
 ...
Branch hotfix/37-some-bug entfernt (war ca0814e).

Summary of actions:
- Hotfix branch 'hotfix/37-sombe-bug' has been merged into 'main'
- The hotfix was tagged '0.2.0'
- Hotfix tag '0.2.0' has been back-merged into 'develop'
- Hotfix branch 'hotfix/37-some-bug' has been locally deleted
- You are now on branch 'develop'
```

... oder

```
$ git switch main
Zu Zweig »main« gewechselt
...
$ git merge hotfix/37-some-bug
$ git tag -a 0.2.0
$ git switch develop
$ git merge hotfix/37-some-bug
$ git branch -d hotfix/37-some-bug
```


0.1
0.2
1.0

Merge-Strategien: Merge vs. Squash vs. Rebase

Ich verwende `git merge`, `git merge squash` und `git rebase` situationsabhängig. Sie haben alle ihre Vorzüge, aber ihre Verwendung hängt sehr stark vom Kontext ab.

`git merge`
: fügt einen neuen Commit hinzu, wenn die Zweige zusammengeführt werden.

  Dies hat den Vorteil, dass es die wahre Geschichte am besten darstellt. Ihr könnt den Merge und alle WIP (work in progress)-Commits sehen, die beim Entwickeln durchlaufen wurden. GGF. (Gegebenenfalls) könnt ihr den Merge einfach rückgängig machen mit `git revert -m|--mainline 1|2 MERGE-COMMIT_SHA`.

  `-m 1`
  : führt zurück zu dem Verhalten des Elternelements aus dem Zweig, in den ihr die Änderungen übernommen habt.

  `-m 2`
  : führt zurück zu dem Verhalten des Elternelements aus dem Zweig, aus dem ihr die Änderungen übernommen habt.

  Tipp: Mehr Commits machen auch *git bisect* besser, solange für jeden Commit ein Build erstellt werden kann. Bei hundert oder maximal tausend Zeilen, die sich geändert haben, habe ich noch eine Chance, den Bug in angemessener Zeit zu finden.

  Siehe auch:

  - Advanced Merging

`git merge --squash`
: ermöglicht euch, alle Änderungen aus einem Zweig in einen einzigen Commit über dem aktuellen Zweig zusammenzufassen.

  Dies ist sinnvoll, wenn ihr viele kleine WIP-Commits habt, die wirklich alle auf ein Feature abzielen. Dabei achte ich beim Squash darauf, die Commit-Nachricht so umzuschreiben, dass sie möglichst aussagekräftig ist. Die übliche Squash-Commit-Nachricht, die von Git, *GitLab* ETC. erstellt wird, ist meist nicht hinreichend und fügt einfach alle Squash-Commit-Nachrichten zusammen, GGF. eine Reihe von WIP-Commit-Nachrichten.

`git rebase`
: verschiebt eine Folge von Commits zu einem neuen Basis-Commit. Damit bleibt der Vorteil erhalten, mittels *git bisect* schnell einen Bug finden zu können. Darüberhinaus wird es nun jedoch einfacher, den Kontext, in dem der Bug entstanden ist, zu erkennen.

  Tipp: Bei einem großen Diff und vielen WIP-Commits kann `git rebase -i` interaktiv angewendet werden, um selektiv Commits für die `squash`-Option auszuwählen und die Commits neu anzuordnen. Macht jedoch immer nur eine Sache:

  - Commits mit der `squash`-Option zusammenfassen oder
  - die Reihenfolge der Commits ändern oder
  - die Commits bearbeiten.

  Versucht nicht, alle Änderungen auf einmal zu machen.

  Tipp: Wenn ihr euch bei `git rebase` nicht sicher fühlt, dann lasst es einfach sein! Ihr könnt stattdessen `git`

`merge` oder `git commit --amend` verwenden.

Siehe auch:

- *Git rebase*
- Rewriting History: Squashing Commits
- Rewriting History: Reordering Commits

Commits für ein sauberes Log ändern

Mit `git commit --fixup` und `git rebase --autosquash` könnt ihr relativ einfach eine Reihe von Commits korrigieren. Um dies an einem Beispiel zu demonstrieren, stelle ich euch folgendes Szenario vor:

1. Wir haben in unserem `my-feature`-Branch zwei Commits: den einen für die eigentliche Funktion, den anderen für die zugehörigen Tests:

```
$ git log --oneline my-feature ^origin/main
a4587fa (my-feature) Add test for my new feature
56e34e9 Add new feature
```

2. Beim *Merge-* oder *Pull-Request* erhalten wir sowohl zu unserer Funktion wie auch zu unseren Tests Feedback, das wir gerne in unsere bestehenden Commits integrieren würden. Hierfür erstellen wir nun zunächst temporäre Commits:

```
$ git commit -m "Feedback on the tests from my function"
$ git commit -m "Feedback on my function"
$ git log --oneline my-feature ^origin/main
556c1e8 (my-feature) Feedback on my function
8780db6 Feedback on the tests from my function
a4587fa Add test for my new feature
56e34e9 Add new feature
```

... mit `git rebase`

3. Mit `git rebase -i` können wir interaktiv die `pick`-Zeilen neu anordnen:

```
$ git rebase -i origin/main
```

Dies öffnet unseren Editor:

```
pick 56e34e9 Add new feature
pick a4587fa Add test for my new feature
pick 8780db6 Feedback on the tests from my function
pick 556c1e8 Feedback on my function
```

Anschließend können wir die Zeilen umändern, z.B. in:

```
pick 56e34e9 Add new feature
squash 556c1e8 Feedback on my function
pick a4587fa Add test for my new feature
squash 8780db6 Feedback on the tests from my function
```

Nun haben wir erneut zwei Commits:

```
$ git log --oneline my-feature ^origin/main
31a140a (my-feature) Add test for my new feature
132ae9b Add new feature
```

4. Die Änderungen können nun mit `git push -f` an unseren entfernten Zweig gesendet werden.

...mit `git commit --fixup` und `git rebase --autosquash`

In Git gibt es jedoch noch einen einfacheren Weg, einen vorherigen Commit zu korrigieren: mit `git commit--fixup` und `git rebase --autosquash`.

5. Wir erstellen erneut zwei temporäre Commits, diesmal jedoch mit `git commit--fixup`:

```
# Further changes to the tests
$ git commit --fixup=31a140a
[my-feature dd0c0d1] fixup! Add test for my new feature
 1 file changed, 1 insertion(+)
# Further changes to my function
$ git commit --fixup=132ae9b
[my-function bc2298a] fixup! Add new feature
 1 file changed, 1 insertion(+)
$ git log --oneline my-feature ^origin/main
bc2298a (my-feature) fixup! Add new feature
dd0c0d1 fixup! Add test for my new feature
31a140a Add test for my new feature
132ae9b Add new feature
```

Bei Commits mit der Option `--fixup=`*`SHA`* schreibt Git eine speziell formatierte Commit-Nachricht, die als *dieser Commit korrigiert jenen* gelesen werden kann.

6. Anstatt nun mit `git rebase -i` die `Pick`/`Squash`-Zeilen manuell festzulegen, können wir nun einfach `git rebase --autosquash` ausführen:

```
$ git rebase --autosquash origin/main
Successfully rebased and updated refs/heads/my-feature.
$ git log --oneline my-feature ^origin/main
694cb48 (my-feature) Add test for my new feature
55cbe9b Add new feature
```

`git rebase --autosquash` automatisiert das, was wir gerade manuell mit `git rebase -i` getan haben – es öffnete sich jedoch kein Editor, in dem wir die Commits manuell verschieben mussten.

Tipp: Die Option `--fixup` enthält auch die Optionen `amend` und `reword`, um die Commit-Nachricht umzuformulieren, z.B. `git commit --fixup:amend=`*`SHA`*.

Weitere Optionen findet ihr in der Git Commit-Dokumentation.

Monorepos und große Repositories

In einem großen Projekt kann es sinnvoll sein, einzelne Komponenten in separaten Repositories zu pflegen. Manchmal schafft dies jedoch unnötige Komplexität, z.B. welche Versionen der Repositories miteinander kompatibel sind. In diesen Fällen kann es sinnvoll sein, alle Teile eines Projekts in einem monolithischen Repository oder *Monorepo* zu halten.

Definition

- In einem Monorepo enthält das Repository mehr als ein logisches Projekt (z.B. einen iOS-Client und eine Webanwendung).
- Diese Projekte können unabhängig voneinander gebaut, getestet oder deployt werden.
- Diese Projekte sind meist nur lose miteinander verbunden oder können auf andere Weise miteinander verbunden werden, z.B. über Tools zur Verwaltung von Abhängigkeiten.
- Das Repository enthält viele Commits, Zweige und/oder Tags. Oder es enthält viele und/oder große Dateien.

Mit Tausenden von Commits von hunderten Autoren in tausenden von Dateien pro Monat ist das Linux-Kernel-Repository riesig.

Vor- und Nachteile

Ein Vorteil von Monorepos kann sein, dass die Aufwände um zu bestimmen, welche Versionen des einen Projekts mit welchen Versionen des anderen Projekts kompatibel sind, deutlich verringert sein könnten. Dies ist zumindest immer dann der Fall, wenn alle Projekte eines Repository von nur einem Entwicklerteam bearbeitet werden. Dann empfiehlt sich, mit jedem *Merge* wieder eine lauffähige Version zu erhalten auch wenn die API zwischen den beiden Projekten geändert wurde.

Als Nachteil können sich jedoch Performance-Einbußen erweisen. Diese können z.B. entstehen durch:

eine große Anzahl an Commits
: Da Git DAGs (*directed acyclic graphs*) verwendet, um die Historie eines Projekts darzustellen, werden alle Operationen, die diesen Graphen durchlaufen, also z.B. `git log` oder `git blame`, langsam werden.

eine große Anzahl von Git-Referenzen
: Eine große Anzahl von Branches und Tags verlangsamen Git ebenfalls. Mit `git ls-remote` könnt ihr euch die Referenzen eines Repository anzeigen lassen und mit `git gc` werden lose Referenzen in einer einzigen Datei zusammengefasst.

  Jede Operation, die den Commit-Verlauf eines Repositories durchlaufen und die einzelnen Referenzen berücksichtigen muss, wie z.B. bei `git branch --contains <commit>`, werden bei einem Repo mit vielen Referenzen langsam.

eine große Anzahl an versionierten Dateien
: Der Index des Directory Cache (`.git/index`) wird von Git verwendet um zu ermitteln, ob die Datei verändert wurde. Dabei verlangsamen sich mit zunehmender Anzahl an Dateien viele Vorgänge, wie z.B. `git status` und `git commit`.

große Dateien
: Große Dateien in einem Teilbaum oder einem Projekt verringern die Leistung des gesamten Repository.

Strategien für große Repositories

Die Designziele von Git, die es so erfolgreich und beliebt gemacht haben, stehen manchmal im Widerspruch zu dem Wunsch, es auf eine Weise zu verwenden, für die es nicht konzipiert wurde. Dennoch gibt es eine Reihe von Strategien, die bei der Arbeit mit großen Repositories hilfreich sein können:

`git clone --depth`

Auch wenn die Schwelle, ab der eine Historie als *riesig* eingestuft wird, ziemlich hoch ist, kann es immer noch mühsam sein, sie zu klonen. Dennoch können wir lange Historien nicht immer vermeiden, wenn sie aus rechtlichen oder regulatorischen Gründen beibehalten werden müssen.

Die Lösung für einen schnellen Clone eines solchen Repositories besteht darin, nur die jüngsten Revisionen zu kopieren. Mit der *Shallow*-Option von `git clone` könnt ihr nur die letzten *N* Commits der Historie abrufen, z.B. `git clone --depth N REMOTE-URL`.

Tipp: Auch Build-Systeme, die mit eurem Git-Repository verbunden sind, profitieren von solchen Shallow Clones!

Shallow Clones waren in Git bisher eher selten, da einige Operationen Anfangs kaum unterstützt wurden. Seit einiger Zeit (in den Versionen 1.9 und höher) könnt ihr jetzt sogar von einem Shallow Clone aus Pull- und Push-Vorgänge in Repositories durchführen.

`git filter-branch`

Für große Repositories, in denen viele Binärdateien versehentlich übertragen wurden, oder alte Assets, die nicht mehr benötigt werden, ist `git filter-branch` eine gute Lösung um die gesamte Historie durchzugehen und Dateien nach vordefinierten Mustern herauszufiltern, zu ändern oder zu überspringen.

Es ist ein sehr leistungsfähiges Werkzeug, sobald ihr herausgefunden habt, wo euer Projektarchiv *schwer* ist. Es gibt auch Hilfsskripte, um große Objekte zu identifizieren: `git filter-branch --tree-filter 'rm -rf /PATH/TO/BIG/ASSETS'`.

Warnung: `git filter-branch` schreibt allerdings die gesamte Historie eures Projekts um, D.H., dass sich einerseits alle Commit-Hashes ändern und andererseits, dass jedes Teammitglied das aktualisierte Repository neu klonen muss.

Siehe auch:

- How to tear apart a repository: the Git way

`git clone --branch`

Ihr könnt den Umfang der geklonten Historie auch begrenzen, indem ihr einen einzelnen Zweig klont, etwa mit `git clone REMOTE-URL --branch BRANCH-NAME --single-branch FOLDER`.

Dies kann nützlich sein, wenn ihr mit langlaufenden und abweichenden Zweigen arbeitet, oder wenn ihr viele Zweige habt und nur mit einigen davon arbeiten müsst. Wenn ihr jedoch nur eine wenige Zweige mit wenigen Unterschieden habt, werdet ihr damit jedoch wahrscheinlich keinen großen Unterschied feststellen.

Git LFS

Git LFS ist eine Erweiterung, die Pointer auf große Dateien in eurem Repository speichert, anstatt die Dateien selbst; diese werden auf einem entfernten Server gespeichert, wodurch die Zeit für das Klonen eures Projektarchivs drastisch verkürzt wird. Git LFS greift dabei auf die nativen Push-, Pull-, Checkout- und Fetch-Operationen von Git zu, um die Objekte zu übertragen und zu ersetzen, D.H., dass ihr mit großen Dateien in eurem Repository wie gewohnt arbeiten könnt.

Ihr könnt Git LFS installieren mit

```
$ sudo apt install git-lfs
```

```
$ brew install git-lfs
```

Git LFS lässt sich mit git for windows mitinstallieren.

Anschließend könnt ihr Git LFS in eurem Repository installieren mit

```
$ git lfs install
Updated Git hooks.
Git LFS initialized.
```

Um nun Git LFS auf bestimmte Dateitypen anzuwenden, könnt ihr z.B. folgendes angeben:

```
$ git lfs track "*.pdf"
Tracking "*.pdf"
```

Dies erstellt in eurer `.gitattributes`-Datei folgende Zeile:

```
*.pdf filter=lfs diff=lfs merge=lfs -text
```

Schließlich sollter ihr die `.gitattributes`-Datei mit Git verwalten:

```
$ git add .gitattributes
```

git-sizer

git-sizer berechnet verschiedene Metriken für ein lokales Git-Repository und kennzeichnet diejenigen, die euch Probleme oder Unannehmlichkeiten bereiten könnten, z.B.:

```
$ git-sizer
Processing blobs: 1903
Processing trees: 4126
Processing commits: 1055
Matching commits to trees: 1055
Processing annotated tags: 2
Processing references: 5
| Name                         | Value     | Level of concern               |
| ---------------------------- | --------- | ------------------------------ |
| Biggest objects              |           |                                |
| * Blobs                      |           |                                |
|   * Maximum size         [1] |  35.8 MiB | ***                            |
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[1]  9fe7b8048891965e476aac0410e08e050fd21354 (refs/heads/main:docs/workspace/pandas/
↪descriptive-statistics.ipynb)
```

Installation

1. Ruft die Releases-Seite auf und ladet die ZIP-Datei herunter, die eurer Plattform entspricht.
2. Entpackt die Datei.
3. Verschiebt die ausführbare Datei (`git-sizer` oder `git-sizer.exe`) in euren `PATH`.

$ brew install git-sizer

Git file system monitor (FSMonitor)

`git status` und `git add` sind langsam, weil sie den gesamten Arbeitsbaum nach Änderungen durchsuchen müssen. Mit der Funktion `git fsmonitor--daemon`, die ab Git-Version 2.36 zur Verfügung steht, werden diese Befehle beschleunigt, indem der Umfang der Suche reduziert wird:

```
$ time git status
Auf Branch master
Ihr Branch ist auf demselben Stand wie 'origin/master'.
real    0m1,969s
user    0m0,237s
sys     0m1,257s
$ git config core.fsmonitor true
$ git config core.untrackedcache true
$ time git status
Auf Branch master
Ihr Branch ist auf demselben Stand wie 'origin/master'.
real    0m0,415s
user    0m0,171s
sys     0m0,675s
$ git fsmonitor--daemon status
fsmonitor-daemon beobachtet '/srv/jupyter/linux'
```

Siehe auch:

- Improve Git monorepo performance with a file system monitor
- Scaling monorepo maintenance

Scalar

`scalar`, ein Repository-Management-Tool für große Repositories von Microsoft, ist seit Version 2.38 Teil der Git-Kerninstallation. Um es zu verwenden, könnt ihr entweder ein neues Repository mit `scalar clone` *`/path/to/repo`* klonen oder `scalar` auf einen bestehenden Klon mit `scalar register` *`/path/to/repo`* anwenden.

Weitere Optionen von `scalar clone` sind:

`-b, --branch` ***`BRANCH`***
: Branch, der nach dem Klonen ausgecheckt werden soll.

`--full-clone`
: Vollständiges Arbeitsverzeichnis beim Klonen erstellen.

`--single-branch`
: Lade nur Metadaten des Branches herunter, der ausgecheckt wird.

Mit `scalar list` könnt ihr sehen, welche Repositories derzeit von Scalar verfolgt werden und mit `scalar unregister` */path/to/repo* wird das Repository aus dieser Liste entfernt.

Standardmäßig ist die Sparse-Checkout-Funktion aktiviert und es werden nur die Dateien im Stammverzeichnis des Git-Repositorys angezeigt. Verwendet `git sparse-checkout set`, um die Menge der Verzeichnisse zu erweitern, die ihr sehen möchtet, oder `git sparse-checkout disable`, um alle Dateien anzuzeigen. Wenn ihr nicht wisst, welche Verzeichnisse im Repository verfügbar sind, könnt ihr `git ls-tree -d --name-only HEAD` ausführen, um die Verzeichnisse im Stammverzeichnis zu ermitteln, oder `git ls-tree -d --name-only HEAD` */path/to/repo*, um die Verzeichnisse in */path/to/repo* zu ermitteln.

Siehe auch:

git ls-tree

Um Sparse-Checkout nachträglich zu aktivieren, führt `git sparse-checkout init --cone` aus. Dadurch werden eure Sparse-Checkout-Patterns so initialisiert, dass sie nur mit den Dateien im Stammverzeichnis übereinstimmen.

Aktuell sind neben `sparse-checkout` noch die folgende Funktionen für `scalar` verfügbar:

- *FSMonitor*
- multi-pack-index (MIDX)
- commit-graph
- Git maintenance
- Partielles Klonen mit *git clone --depth* und *git filter-branch*

Die Konfiguration von `scalar` wird aktualisiert, wenn neue Funktionen in Git eingeführt werden. Um sicherzustellen, dass ihr immer die neueste Konfiguration verwendet, solltet ihr `scalar reconfigure` */PATH/TO/REPO* nach einer neuen Git-Version ausführen, um die Konfiguration eures Repositorys zu aktualisieren oder `scalar reconfigure -a`, um alle eure mit Scalar registrierten Repositories auf einmal zu aktualisieren.

Siehe auch:

- Git - scalar Documentation

Repos aufteilen

Häufig ist es sinnvoll, ein großes Git-Repository in mehrere kleinere aufzuteilen. Das kann in einem Projekt nötig sein, das stark angewachsen ist, oder wenn wir ein Teilprojekt als eigenständiges Repo auslagern möchten. Natürlich könnten wir einfach ein neues Repository erstellen und die Dateien für das Teilprojekt hineinkopieren. Allerdings würden wir dabei die gesamte Versionsgeschichte verlieren.

Hier beschreibe ich, wie ihr ein Git-Repository aufteilen könnt, ohne die jeweils zugehörige Historie zu verlieren.

Szenario und Ziele

Wir wollen aus dem Jupyter-Tutorial-Repository denjenigen Teil herauslösen, der sich mit der Visualisierung der Daten befasst: `docs/viz/`. Die Herausforderung besteht darin, dass die Historie für das `docs/viz/`-Verzeichnis mit anderen Änderungen vermischt ist. Daher klonen wir zunächst zweimal dasselbe Repository:

```
$ git clone git@github.com:veit/jupyter-tutorial.git
Klone nach 'jupyter-tutorial'...
$ git clone git@github.com:veit/jupyter-tutorial.git pyviz-tutorial
Klone nach 'pyviz-tutorial' ...
```

Der nächste Schritt besteht darin, die unerwünschten Historien aus jedem der beiden Repos herauszufiltern. Um die Historie umzuschreiben und nur diejenigen Commits zu behalten, die tatsächlich dein Inhalt eines bestimmten Unterordners betreffen, verwenden wir git-filter-repo:

```
$ curl https://raw.githubusercontent.com/newren/git-filter-repo/main/git-filter-repo -o␣
↪git-filter-repo
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  161k  100  161k    0     0   578k      0 --:--:-- --:--:-- --:--:--  584k

$ cd pyviz-tutorial
$ python3 ../git-filter-repo --path docs/viz
```

Das Einzige, was jetzt noch zu tun ist, ist die Anpassung der Remote-URL:

```
$ git remote add origin git@github.com:veit/pyviz-tutorial.git
$ git push -u origin main
```

Für unser Jupyter-Tutorial-Repository invertieren wir jetzt den ausgewählten Pfad:

```
$ cd jupyter-tutorial
$ python3 ../git-filter-repo --invert-paths --path docs/viz
$ git remote add origin git@github.com:veit/jupyter-tutorial.git
$ git push -f -u origin main
```

CI-freundliche Git-Repos

Im Folgenden möchte ich einige Tipps geben, wie Git-Repositories und Kontinuierliche Integration mit GitLab CI/CD oder GitHub Actions gut zusammenspielen können.

Speichert große Dateien außerhalb eures Repositories

Jedes Mal, wenn ein neuer *Build* erstellt wird, muss das Arbeitsverzeichnis geklont werden. Wenn euer Repository jedoch mit großen Artefakten aufgebläht ist, verlangsamt sich dieser und ihr müsst länger auf die Ergebnisse warten.

Wenn euer Build jedoch von Binaries aus anderen Projekten oder großen Artefakten abhängt, kann ein externes Speichersystem sinnvoll sein, das diejenigen Dateien, die ihr zu Beginn eures Builds im Build-Verzeichnis benötigt, zum Download bereitstellt.

Verwendet Shallow-Clones

Jedes Mal, wenn ein Build ausgeführt wird, klont euer Build-Server euer Projektarchiv in das aktuelle Arbeitsverzeichnis. Dabei klont Git üblicherweise die gesamte Historie des Repos, wodurch dieser Vorgang mit der Zeit immer länger dauert. Es sei denn, ihr verwendet SOG. Shallow-Clones, bei denen mit *git clone --depth* nur der aktuelle Snapshot des Repos und mit *git clone --branch* nur der relevante Zweig heruntergezogen wird. Das verkürzt die Build-Zeit vor allem bei Repositories mit einer langen Geschichte und vielen Zweigen.

Dabei kann Git seit Version 1.9 einfache Änderungen an Dateien, wie z.B. das Aktualisieren einer Versionsnummer, vornehmen, ohne dass die gesamte Historie gepusht wurde.

Warnung: In einem shallow clone kann `git fetch` dazu führen, dass ein fast vollständiger Commit-Verlauf heruntergeladen wird. Auch andere Git-Operationen können zu unerwarteten Ergebnissen führen und die vermeintlichen Vorteile von Shallow-Clones zunichte machen, sodass wir empfehlen, Shallow-Clones nur für Builds zu verwenden und das Repository sofort danach wieder zu löschen.

Wollt ihr die Repositories jedoch weiterverwenden, kann der folgende Tipp hilfreich sein.

Cache des Repos auf Build-Servern

Dies beschleunigt auch das Klonen, da die Repos nur aktualisiert werden müssen.

Bemerkung: Der Cache von Repos ist nur dann von Vorteil, wenn die Build-Umgebung von Build zu Build bestehen bleibt. Wenn euer Build-Agent, z.B. Amazon EC2, den Build wieder abbaut, könnt ihr mit Caching jedoch nichts gewinnen.

Wählt die Trigger mit Bedacht

Es versteht sich fast von selbst, dass es eine gute Idee ist, CI auf all euren aktiven Zweigen laufen zu lassen. Aber es ist meist keine gute Idee, alle Builds auf allen Zweigen gegen alle Commits laufen zu lassen.

Üblicherweise geben wir allen im Entwicklungsteam die Möglichkeit, ihre Zweig-Builds auf Knopfdruck zu erstellen, anstatt sie automatisch auszulösen. Dies scheint uns ein guter Weg, um ein Gleichgewicht zwischen regelmäßigen Tests und der Einsparung von Ressourcen zu schaffen. In kritischen Zweigen wie `main` oder `stable` werden Builds jedoch automatisch ausgelöst. Zudem erhalten wir automatisiert auch bei jedem Merge- oder Pull-Request zeitnahe Testergebnisse.

Fortgeschrittenes Git

git cherry-pick
: ermöglicht euch, beliebige Git-Commits anhand ihres Hash-Wertes dem aktuellen `HEAD` anzuhängen.

git bisect
: ermöglicht euch, einen Git-Commit, der eine Regression eingeführt hat, schnell zu finden.

git notes
: fügt Textnotizen zu Commits, Tags und anderen Objekten hinzu.

Git Hooks
: sind Skripte, die bei bestimmten Ereignissen in einem Git-Repository automatisch ausgeführt werden.

Jupyter Notebooks
: können zu Problemen führen bei der Verwaltung mit Git.

Binärdateien
: können in Git so konfiguriert werden, dass sinnvolle Diffs angezeigt werden.

Visual Studio Code
: kann eine bereits vorhandene Git-Installation nutzen um die entsprechenden Funktionalitäten zur Verfügung zu stellen.

GitLab
: ist eine Webanwendung zur Versionsverwaltung auf Basis von Git.

git-big-picture
: visualisiert Git-Repositories als DAGs (gerichteter azyklischer Graph, engl.: directed acyclic graph).

etckeeper
: ist eine Sammlung von Werkzeugen, mit denen das `/etc`-Verzeichnis in einem Git-Repository verwaltet werden kann.

Git's Datenbank-Interna
: verweist auf Artikel zu Git's Datenbank-Interna.

Git Cherry-Pick

`git cherry-pick` ermöglicht euch, beliebige Git-Commits anhand ihres Hash-Wertes dem aktuellen `HEAD` anzuhängen. Beim Cherry-Picking wird ein Commit aus einem Branch ausgewählt und auf einen anderen angewendet, z.B.:

```
$ git checkout 3.10
$ git cherry-pick 61de025
[3.10 b600967] Fix bug #17
 Date: Thu Sep 15 11:17:35 2022 +0200
 1 file changed, 9 insertions(+)
```

Dabei kann `git cherry-pick` mit verschiedenen Optionen eingesetzt werden:

`--edit`, `-e`
: übernimmt nicht die bestehende Commit-Nachricht sondern ermöglicht euch, eine eigene Commit-Nachricht für diesen Cherry-Pick zu erstellen.

`--no-commit`, `-n`
: erstellt keinen neuen Commit sondern verschiebt die Inhalte des Commits in das Arbeitsverzeichnis.

`--signoff`, `-s`
: fügt am Ende der Commit-Nachricht eine Signaturzeile mit `Signed-off-by` hinzu.

`git cherry-pick` akzeptiert auch Optionen zum Beheben von Merge-Konflikten, darunter `--abort`, `--continue` und `--quit`.

`git cherry-pick` kann hilfreich sein, um Änderungen rückgängig zu machen, wenn beispielsweise ein Commit versehentlich für den falschen Branch durchgeführt wurde, könnt ihr zu dem Branch wechseln, in dem die Änderung eigentlich vorgenommen weerden sollte, und den Commit dann per Cherry-Pick auf diesen Branch übertragen.

Beim Cherry-Picking entstehen jedoch üblicherweise doppelte Commits, und in vielen Fällen bevorzugen wir daher eher Git Merges. Dennoch kann sich `git cherry-pick` für einige Szenarien sehr gut eignen, z.B. für *Release-Branches*-Workflows.

git range-diff

git range-diff zeigt die Differenz zwischen zwei Commit-Bereichen an, d.h., welche Commits zwischen diesen Bereichen gleich sind oder sich geändert haben. Dieser Befehl kann z.B. beim Überprüfen helfen, welche Commits mit `git cherry-pick` auf welche Zweige verteilt wurden.

Regressionen finden mit `git bisect`

`git bisect` ermöglicht euch, einen Git-Commit, der eine Regression eingeführt hat, schnell zu finden. Der Name *bisect* stammt von der binären Suche, den der Befehl anwendet. Dabei wird die Liste der Commits wiederholt halbiert, bis der zuständige Commit gefunden ist. So müssen nur $\log_2(n+1)$ Commits getestet werden.

1. Hierzu beginnt ihr die Suche mit `git bisect start`. Anschließend könnt ihr den Bereich mit `git bisect new [COMMIT]` und `git bisect old [COMMIT]` eingrenzen, in dem ein Fehler eingeführt wurde. Alternativ kann auch die Kurzform `git bisect start [BAD COMMIT] [GOOD COMMIT]` verwendet werden. `git bisect` checkt dann einen Commit in der Mitte aus und fordert euch auf diesen zu testen, z.B.:

```
$ git bisect start v2.6.27 v2.6.25
Bisecting: 10928 revisions left to test after this (roughly 14 steps)
[2ec65f8b89ea003c27ff7723525a2ee335a2b393] x86: clean up using max_low_pfn on 32-bit
```

2. Die Suche kann nun manuell oder automatisch mit einem Skript fortgesetzt werden. Manuell könnt ihr mit `git bisect new` und `git bisect old` den Bereich immer weiter eingrenzen, in dem ein Fehler eingeführt wurde. Wird dieser Commit gefunden, kann die Ausgabe z.B. folgendermaßen aussehen:

```
$ git bisect new
2ddcca36c8bcfa251724fe342c8327451988be0d is the first bad commit
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Author: Linus Torvalds <torvalds@linux-foundation.org>
Date:   Sat May 3 11:59:44 2008 -0700

    Linux 2.6.26-rc1

:100644 100644 5cf82581... 4492984e... M      Makefile
```

3. Anschließend überprüfen wir mit `git show HEAD`, welche Änderungen in diesem Commit vorgenommen wurden:

```
$ git show HEAD
commit 2ddcca36c8bcfa251724fe342c8327451988be0d
Autor: Linus Torvalds <torvalds@linux-foundation.org>
Datum: Sa 3. Mai 11:59:44 2008 -0700

    Linux 2.6.26-rc1

diff --git a / Makefile b / Makefile
index 5cf8258 ..4492984 100644
--- a / Makefile
+++ b / Makefile
@@ -1,7 +1,7 @@
 VERSION = 2
 PATCHLEVEL = 6
-SUBLEVEL = 25
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
-EXTRAVERSION =
+ SUBLEVEL = 26
+ EXTRAVERSION = -rc1
 NAME = Funky Weasel ist Jiggy wit it

 # * DOKUMENTATION *
```

4. Schließlich könnt ihr mit `git bisect reset` in den Branch zurückkehren, in dem ihr euch vor der Bisect-Suche befunden habt:

```
$ git bisect reset
Checking out files: 100% (21549/21549), done.
Previous HEAD position was 2ddcca3... Linux 2.6.26-rc1
Switched to branch 'master'
```

Siehe auch:

- Fighting regressions with git bisect
- Docs

Nicht-testbare Commits markieren mit `git bisect skip`

Manchmal landet man mit `git bisect` bei einem Commit, den man nicht testen kann, weil es ein anderes Problem gibt. Normalerweise ist dies auf einen Fehler zurückzuführen, der verhindert, dass ihr euren Code ausführen oder das Testergebnis sehen könnt, z.B. ein Syntaxfehler. In diesem Fall solltet ihr den Commit nicht als `old` oder `new` markieren, da ihr aufgrund des Fehlers nicht feststellen könnt, welches Verhalten vorliegt. Stattdessen solltet ihr den Commit mit git bisect skip überspringen. `git bisect` checkt stattdessen einen benachbarten Commit zum Testen aus. Wenn dieser funktioniert, macht wie gewohnt mit dem Testen und Ausführen von `new` oder `old` weiter. Wenn nicht, führt `git bisect skip` erneut aus. Wenn ihr wisst, dass es einen Bereich von nicht testbaren Commits gibt, weist `git bisect` an, diesen gesamten Bereich zu überspringen mit `git bisect skip` *`COMMIT1..COMMIT2`*.

Siehe auch:

- Avoiding testing a commit

Automatisches Testen mit `git bisect run`

Oft ist es möglich, den Test, ob ein Commit altes oder neues Verhalten zeigt, zu automatisieren. Dadurch wird die Verwendung von `git bisect` massiv beschleunigt, da ihr nicht mehr bei jedem Schritt eine Eingabe machen müsst. Außerdem wird der Prozess dadurch weniger fehleranfällig, da ihr nicht aus Versehen den falschen Unterbefehl `old` und `new` ausführt. Automatisierte Tests sind auch dann von Vorteil, wenn euer Testprozess eine Weile dauert, z.B. wenn ihr einen langen Kompilierungsschritt habt. Die Suche wird nicht unterbrochen, um auf eure Eingabe zu warten, und ihr könnt in der Zwischenzeit an etwas anderem arbeiten.

Um automatische Tests zu starten, verwendet git bisect run mit eurem Testbefehl und optionalen Argumenten. Möglicherweise müsst ihr ein kurzes Testskript erstellen, das den betroffenen Teil eures Codes ausführt und prüft, welches Verhalten vorhanden ist. `git bisect` führt den angegebenen Befehl bei jedem Schritt der Binärsuchschleife aus und verwendet seine Ergebnisse, um je nach Bedarf `old`, `new` oder `skip` aufzurufen.

Ein Beispiel hierfür findet ihr im Issue fetch_california_housing fails in CI on master von scikit-learn:

```
$ git bisect run pytest sklearn/utils/tests/test_multiclass.py -k test_unique_labels_non_
↪specific
```

Automatisiertes Testen von Performance-Regressionen

Mit ein wenig Mehraufwand könnt ihr mit automatisierten Tests nach komplizierteren Verhaltensänderungen suchen. Für Performance-Tests benötigen wir hierfür ein Testprogramm, das mehrere Durchläufe durchführen und die minimale Zeit ermitteln kann, wobei mögliches Rauschen eliminiert werden soll:

```
from subprocess import run
from time import perf_counter


times = []
for _ in range(10):
    start = perf_counter()
    run(
        [./perftest, PARAM],
        check=True,
        capture_output=True,
    )
    elapsed = perf_counter() - start
    times.append(elapsed)
if min(times) > X.0:
    print("Too slow")
    raise SystemExit(1)
else:
    print("Fast enough")
    raise SystemExit(0)
```

Das Programm führt `python perftest.py` *`PARAM`* zehnmal aus und misst bei jeder Ausführung die Zeit. Anschließend vergleicht es die minimale Ausführungszeit mit einem Grenzwert von `X` Sekunden. Liegt die Mindestzeit über dem Grenzwert, gibt es *Too slow* aus und beendet sich mit dem Exit-Code 1, andernfalls gibt es *Fast enough* aus und beendet sich mit dem Exit-Code `0`:

```
$ python perftest.py PARAM
Fast enough
$ echo $? 0
```

Reproduzieren der Binärsuche mit `git bisect log` und `git bisect replay`

Das scikit-learn-Issue zeigt auch, wie ihr anderen die Ergebnisse eurer Bisect-Suche mit `git bisect log` nachvollziehbar mitteilen könnt:

```
$ git bisect log
81f2d3a0e *   massich/multiclass_type_of_target Merge branch 'master' into multiclass_
↪type_of_target
        |\
15f24f25d | * bad DOC Cleaning for what's new
fbb2c7c70 | * good-fbb2c7c7007dc373c462e39ab273a183a8823d58 @ ENH Adds _
↪MultimetricScorer for Optimized Scoring  (#14593)
...
```

Mit `$ git bisect log > bisect_log.txt` könnt ihr eure Suche auch für andere reproduzierbar abspeichern:

```
$ git bisect replay bisect_log.txt
```

Git Notes

Git Notes fügen Textnotizen zu Commits, Tags und anderen Objekten hinzu. Solche Notizen können alle Arten von Metadaten enthalten, z.B. Kommentare zur Codeüberprüfung, Links zu Fehlerberichten usw:

1. Hinzufügen einer Git-Notiz:

   ```
   $ git notes add -m 'Example note'
   ```

2. Anzeigen einer Git-Notiz:

   ```
   $ git log
   commit 859de540cda23f510f4ecbe0f38d07666e933f08 (HEAD -> main)
   Author: Veit Schiele <veit@cusy.io>
   Date:   Sun Mar 24 11:17:56 2024 +0100

       A commit message

   Notes:
       Example note
   ```

3. Ändern einer Git-Notiz:

   ```
   $ git notes edit
   ```

Git Notes werden jedoch nicht standardmäßig mit `git push` oder `git pull` an das entfernte Repository übermittelt; sie müssen mit `git push origin 'refs/notes/*'` und `git fetch origin 'refs/notes/*:refs/notes/*'` synchronisiert werden.

Warnung: Verwendet nicht `git pull` anstelle von `git fetch`: ihr könnt `refs/notes/commits` nicht mit eurem aktuellen Zweig zusammenführen.

Bemerkung: Git Notes werden nicht in die Git-Commit-Historie aufgenommen, sodass sie nicht sind nicht für Regulatorisches verwendet werden können, bei dem die Herkunft, Nichtabstreitbarkeit oder Manipulationssicherheit nachgewiesen werden muss. Sie könnxen jedoch z.B. für Build-Tags und ähnliches nützlich sein.

Siehe auch:

- Git Notes: Git's Coolest, Most Unloved Feature
- git-appraise
- github-issues-git-notes

Git Hooks

Git-Hooks sind Skripte, die bei bestimmten Ereignissen in einem Git-Repository automatisch ausgeführt werden, U.A.:

Befehl	Hook
`comit`	`comit-msg`, `pre-commit`
`merge`	`pre-merge`, `comit-msg`
`rebase`	`pre-rebase`
`pull`	`pre-merge`, `comit-msg`
`push`	`pre-push`

Sie können sich entweder in lokalen oder serverseitigen Repositories befinden. So können Git-Repositories individuell angepasst und benutzerdefinierte Aktionen ausgelöst werden.

Git Hooks befinden sich im Verzeichnis `.git/hooks/`. Beim Anlegen eines Repository werden dort auch bereits einige Beispielskripte angelegt:

```
.git/hooks/
├── applypatch-msg.sample
├── commit-msg.sample
├── fsmonitor-watchman.sample
├── post-update.sample
├── pre-applypatch.sample
├── pre-commit.sample
├── pre-merge-commit.sample
├── prepare-commit-msg.sample
├── pre-push.sample
├── pre-rebase.sample
├── pre-receive.sample
└── update.sample
```

Damit die Skripte ausgeführt werden, muss lediglich der Suffix `.sample` entfernt werden und GGF. die Dateiberechtigung ausführbar sein, z.B. mit `chmod +x .git/PREPARE-COMMIT-MSG`.

Die integrierten Skripte sind Shell- und Perl-Skripte, es können jedoch beliebige Skriptsprachen verwenden werden. Dabei bestimmt die Shebang-Zeile (`#!/bin/sh`), wie die Datei interpretiert werden soll.

Die Skripte können jedoch nicht in das serverseitige Repository kopiert werden.

pre-commit-Framework

pre-commit ist ein Framework zum Verwalten und Pflegen mehrsprachiger Commit-Hooks.

Eine wesentliche Aufgabe ist es, dem gesamten Entwicklungsteam dieselben Skripte zur Verfügung zu stellen. pre-commit von yelp verwaltet solche Hooks und verteilt sie auf verschiedene Projekte und Entwickler.

Git Hooks werden meist verwendet um vor Code Reviews automatisch auf Probleme im Code hinzuweisen, z.B. um die Formatierung zu überprüfen oder Debug-Anweisungen zu finden. pre-commit vereinfacht das projektübergreifende Teilen vom Hooks. Dabei ist auch die Sprache, in der z.B. ein Linter geschrieben wurde, wegabstrahiert – so ist `scss-lint` in Ruby geschrieben, ihr könnt ihn jedoch mit pre-commit verwenden ohne eurem Projekt ein Gemfile hinzufügen zu müssen.

Installation

Bevor ihr die Hooks ausführen könnt, muss das pre-commit Framework installiert werden:

Bevor das pre-commit Framework mit Pipenv installiert werden kann, müssen zunächst noch die Microsoft Build Tools für C++ heruntergeladen und ausgeführt werden damit anschließend die *Desktopentwicklung mit C++* ausgewählt und mit den Standardoptionen installiert werden kann.

Erst dann kann das pre-commit Framework installiert werden mit:

```
$ pipenv install pre-commit
```

```
$ apt install pre-commit
```

```
$ brew install pre-commit
```

```
$ pipenv install pre-commit
```

Überprüfen der Installation z.B. mit

```
$ pipenv run pre-commit -V
pre-commit 2.21.0
```

Konfiguration

Nachdem Pre-Commit installiert ist, können mit der `.pre-commit-config.yaml`-Datei im Root-Verzeichnis eures Projekts Plugins für dieses Projekt konfiguriert werden.

```
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v3.2.0
    hooks:
    -   id: trailing-whitespace
    -   id: end-of-file-fixer
    -   id: check-yaml
    -   id: check-added-large-files
```

Ihr könnt euch eine solche initiale `.pre-commit-config.yaml`-Datei auch generieren lassen mit

```
$ pipenv run pre-commit sample-config > .pre-commit-config.yaml
```

Wenn ihr `check-json` auf eure Jupyter Notebooks anwenden möchtet, müsst ihr zunächst konfigurieren, dass die Überprüfung auch für den Datei-Suffix `.ipynb` verwendet werden soll:

```
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v3.2.0
    hooks:
    ...
    - id: check-json
      types: [file]
      files: \.(json|ipynb)$
```

Siehe auch:

Eine vollständige Liste der Konfigurationsoptionen erhaltet ihr in Adding pre-commit plugins to your project.

Ihr könnt auch eigene Hooks schreiben, siehe Creating new hooks.

Installieren der Git-Hook-Skripte

Damit Pre-Commit auch vor jedem Commit zuverlässig ausgeführt wird, wird das Skript in unserem Projekt installiert:

```
$ pre-commit install
pre-commit installed at .git/hooks/pre-commit
```

Wollt ihr die Git-Hook-Skripte wieder deinstallieren, könnt ihr dies mit `pre-commit uninstall`.

Ausführen

`pre-commit run --all-files`
: führt alle pre-commit-Hooks unabhängig von `git commit` aus:

```
$ pipenv run pre-commit run --all-files
Trim Trailing Whitespace.....................................................Passed
Fix End of Files.............................................................Passed
Check Yaml...................................................................Passed
Check for added large files..................................................Passed
```

`pre-commit run` *`HOOK`*
: führt einzelne pre-commit-Hooks aus, z.B. `pre-commit run trailing-whitespace`

Bemerkung: Beim ersten Aufruf eines pre-commit-Hooks wird dieser zunächst heruntergeladen und anschließend installiert. Dies kann einige Zeit benötigen, z.B. wenn eine Kopie von `node` erstellt werden muss.

`pre-commit autoupdate`
: aktualisiert die Hooks automatisch:

  Siehe auch:

  - pre-commit autoupdate [options].

Die vom pre-commit-Framework verwalteten Hooks jedoch nicht darauf beschränkt, vor Commits ausgeführt zu werden; sie können auch für andere Git-Hooks verwendet werden, siehe *Weitere pre-commit-Hooks*.

pre-commit-Skripte

pre-commit-hooks

Das pre-commit-Framework bringt bereits eine ganze Reihe von Skripten mit, U.A.:

`check-added-large-files`
: verhindert, dass große Dateien übertragen werden

`check-case-conflict`
: sucht nach Dateien, die in Dateisystemen, die Groß- und Kleinschreibung nicht berücksichtigen, in Konflikt geraten würden

`check-executables-have-shebangs`
: stellt sicher, dass (nicht-binäre) ausführbare Dateien eine Shebang-Zeile haben

`check-shebang-scripts-are-executable`
: stellt sicher, dass (nicht-binäre) Dateien mit einer Shebang-Zeile ausführbar sind

`check-merge-conflict`
: sucht nach Dateien, die Merge-Conflict-Strings enthalten

`check-symlinks`
: prüft auf Symlinks, die auf nichts verweisen

`destroyed-symlinks`
: erkennt Symlinks, die in reguläre Dateien mit dem Inhalt des Pfades, auf den der Symlink verweist, geändert wurden.

`no-commit-to-branch`
: schützt Zweige vor dem Commit

pygrep-hooks
: stellt reguläre Ausdrücke für Python und reStructuredText bereit, U.A.:

`python-no-log-warn`
: such nach der veralteten `.warn()`-Methode von Python-Loggern

`python-use-type-annotations`
: erzwingt, dass Type-Annotations anstelle von Type-Comments verwendet werden

`rst-backticks`
: erkennt die Verwendung einzelner Backticks beim Schreiben von reStructuredText

`rst-directive-colons`
: erkennt, dass reStructuredText-Direktiven nicht mit einem Doppelpunkt oder einem Leerzeichen vor dem Doppelpunkt enden

`rst-inline-touching-normal`
: erkennt, dass Inline-Code in normalem Text in reStructuredText verwendet wird

`text-unicode-replacement-char`
: verhindert Dateien, die UTF-8-Unicode-Replacement-Character enthalten

Linter und Formatierer
: Sie werden in eigenen Repositories bereitgestellt, U.A.:

autopep8
: stellt autopep8 für das pre-commit-Framework bereit

mypy
: stellt mypy bereit

validate-pyproject
: überprüft `pyproject.toml`-Dateien

sp-repo-review
: evaluiert bestehende Repos anhand der Scientific Python-Richtlinien.

clang-format
: stellt clang-format-wheel bereit

csslint
: stellt csslint bereit

scss-lint
: stellt scss-lint bereit

eslint
stellt eslint bereit

fixmyjs
stellt fixmyjs bereit

prettier
stellt prettier bereit

black
für die Formatierung von Python-Code

`black`
Python-Code-Formattierer

`black-jupyter`
Python-Code-Formattierer für Jupyter-Notebooks

Python Code Quality Authority
Codequalitätswerkzeuge (und Plugins) für die Programmiersprache Python:

flake8
fördert die Durchsetzung eines konsistenten Python-Stils

autoflake
entfernt unbenutzte Importe und unbenutzte Variablen aus Python-Code

bandit
Werkzeug zum Auffinden von Sicherheitslücken in Python-Code

pydocstyle
statisches Analysewerkzeug zur Überprüfung der Einhaltung von Python-Docstring-Konventionen.

docformatter
formatiert docstrings gemäß **PEP 257**

pylint
Python-Linter

doc8
führt doc8 zum Linting von Dokumenten aus

prospector
analysiert Python-Code mit Prospector

isort
sortiert Python-Importe

nbQA
führt isort, pyupgrade, mypy, pylint, flake8 und mehr auf Jupyter Notebooks aus:

`nbqa`
führt jedes Standard-Python-Codequalitätswerkzeug auf einem Jupyter-Notebook aus

`nbqa-black`
führt `black` auf einem Jupyter-Notebook aus

`nbqa-check-ast`
führt `check-ast` auf einem Jupyter-Notebook aus

`nbqa-flake8`
führt `flake8` auf einem Jupyter-Notebook aus

nbqa-isort
: führt `isort` auf einem Jupyter-Notebook aus

nbqa-mypy
: führt `mypy` auf einem Jupyter-Notebook aus

nbqa-pylint
: führt `pylint` auf einem Jupyter-Notebook aus

nbqa-pyupgrade
: führt `ppyupgrade` auf einem Jupyter-Notebook aus

nbqa-yapf
: führt `yapf` auf einem Jupyter-Notebook aus

nbqa-autopep8
: führt `autopep8` auf einem Jupyter-Notebook aus

nbqa-pydocstyle
: führt `pydocstyle` auf einem Jupyter-Notebook aus

nbqa-ruff
: führt `ruff` auf einem Jupyter-Notebook aus

blacken-docs
: wendet `black` auf Python-Codeblöcke in Dokumentationsdateien an

Misc

pyupgrade
: aktualisiert automatisch die Syntax für neuere Versionen

reorder-python-imports
: ordnet Importe in Python-Dateien neu an

dead
: erkkent toten Python-Code

python-safety-dependencies-check
: analysiert Python-Requirements auf bekannte Sicherheitsschwachstellen

gitlint
: Git commit message Linter

nbstripout
: entfernt die Ausgabe von Jupyter Notebooks

ripsecrets
: verhindert, dass geheime Schlüssel in euren Quellcode aufgenommen werden

detect-secrets
: erkennt Zeichenfolgen mit hoher Entropie, bei denen es sich wahrscheinlich um Passwörter handelt

pip-compile
: kompiliert automatisch Anforderungen

kontrolilo
: Werkzeug zur Kontrolle der Lizenzen für OSS-Abhängigkeiten

Siehe auch:

- Supported hooks

Weitere pre-commit-Hooks

Die vom pre-commit-Framework verwalteten Hooks beschränken sich nicht darauf, vor Commits ausgeführt zu werden; sie können auch für andere Git-Hooks verwendet werden:

post-commit
Ab Version 2.4.0 kann das Framework auch post-commit-Hooks ausführen mit:

```
$ pipenv run pre-commit install --hook-type post-commit
pre-commit installed at .git/hooks/post-commit
```

Da `post-commit` jedoch nicht auf Dateien wirkt, müssen all diese Hooks `always_run` setzen:

```
- repo: local
  hooks:
  - id: post-commit-local
    name: post commit
    always_run: true
    stages: [post-commit]
    # ...
```

pre-merge
Ab Git 2.24 gibt es den pre-merge-commit-Hook, der ausgelöst wird, ausgelöst, nachdem eine Zusammenführung erfolgreich war, aber bevor der Merge-Commit erstellt wird. Ihr könnt ihn mit dem pre-commit-Framework nutzen mit:

```
$ pre-commit install --hook-type pre-merge-commit
pre-commit installed at .git/hooks/pre-merge-commit
```

post-merge
Ab Version 2.11.0 kann das Framework auch Skripte für den post-merge-Hook ausführen:

```
$ pipenv run pre-commit install --hook-type post-merge
pre-commit installed at .git/hooks/post-merge
```

Mit `$PRE_COMMIT_IS_SQUASH_MERGE` könnt ihr herausfinden, ob es sich um einen Squash-Merge handelte.

pre-push
Um den pre-push-Hook mit dem pre-commit-Framework verwenden zu können, gebt folgendes ein:

```
$ pre-commit install --hook-type pre-push
pre-commit installed at .git/hooks/pre-push
```

Hierfür werden die folgenden Umgebungsvariablen bereitgestellt:

$PRE_COMMIT_FROM_REF
Die entfernte Revision, zu der gepusht wurde

$PRE_COMMIT_TO_REF
Die lokale Revision, die an die entfernte Revision gepusht wurde

$PRE_COMMIT_REMOTE_NAME
Die lokale Revision, die an die entfernte Revision gepusht wurde, z.B. `origin`

$PRE_COMMIT_REMOTE_URL
Die URL des entfernten Repository, zu dem gepusht wurde, z.B. `git@github.com:veit/python4datascience`

$PRE_COMMIT_REMOTE_BRANCH
Der Name des entfernten Zweigs, zu dem gepusht wurde, z.B. `refs/heads/`*TARGET-BRANCH*

$PRE_COMMIT_LOCAL_BRANCH
Der Name des lokalen Zweigs, der in den entfernten Zweig verschoben wurde, z.B. *HEAD*

commit-msg
commit-msg kann verwendet werden mit:

```
$ pre-commit install --hook-type commit-msg
pre-commit installed at .git/hooks/commit-msg
```

Der `commit-msg`-Hook kann mit `stages: [commit-msg]` konfiguriert werden, wobei der Name einer Datei übergeben wird, die den aktuellen Inhalt der Commit-Nachricht enthält, der überprüft werden kann.

prepare-commit-msg
prepare-commit-msg kann mit pre-commit verwendet werden mit:

```
$ pre-commit install --hook-type prepare-commit-msg
pre-commit installed at .git/hooks/prepare-commit-msg
```

Der `prepare-commit-msg`-Hook wird mit `stages: [prepare-commit-msg]` konfiguriert, wobei der Name einer Datei übergeben wird, die die anfängliche Commit-Nachricht enthält, z.B. von `git commit -m "`*COMMIT-MESSAGE*`"` um daraus eine dynamische Vorlage zu erstellen, die im Editor angezeigt wird. Schließlich sollte der Hook noch überprüfen, ob kein Editor gestartet wird mit `GIT_EDITOR=:`.

post-checkout
Der post-checkout-Hook wird aufgerufen, wenn `git checkout` oder `git switch` ausgeführt wird.

Der `post-checkout`-Hook kann z.B. verwendet werden für

- die Überprüfung von Repositories
- die Ansicht der Unterschiede zum vorherigen `HEAD`
- das Ändern der Metadaten des Arbeitsverzeichnisses.

In pre-commit kann kann er verwendet werden mit:

```
$ pre-commit install --hook-type post-checkout
pre-commit installed at .git/hooks/post-checkout
```

Da `post-checkout` nicht auf Dateien wirkt, muss für alle `post-checkout`-Skripte `always_run` gesetzt werden, z.B.:

```
- repo: local
  hooks:
  - id: post-checkout-local
    name: Post checkout
    always_run: true
    stages: [post-checkout]
    # ...
```

Dabei gibt es drei Umgebungsvariablen, die den drei Arguementen von `post-checkout` entsprechen:

$PRE_COMMIT_FROM_REF
gibt die Referenz des vorherigen `HEAD` aus

$PRE_COMMIT_TO_REF
gibt die Referenz des neuen `HEAD` aus, der sich geändert haben kann oder auch nicht

`$PRE_COMMIT_CHECKOUT_TYPE`
: gibt `Flag=1` aus, wenn es ein Branch-Checkout war und `Flag=0`, wenn es ein File-Checkout war.

`post-rewrite`
: post-rewrite wird aufgerufen, wenn Commits umgeschrieben werden, also von `git commit --amend` oder von `git rebase`.

```
$ pre-commit install --hook-type post-rewrite
pre-commit installed at .git/hooks/post-rewrite
```

Da `post-rewrite` nicht auf Dateien wirkt, muss `always_run: true` gesetzt werden.

Git teilt dem `post-rewrite`-Hook mit, welcher Befehl das Rewrite ausgelöst hat. pre-commit gibt dies als `$PRE_COMMIT_REWRITE_COMMAND` aus.

pre-commit in CI-Pipelines

Pre-Commit kann auch für kontinuierliche Integration (CI (Continuous Integration)) verwendet werden.

Beispiele für GitHub Actions

pre-commit ci

Service, der eurem GitHub-Repository die *pre-commit ci*-App in eurem Repository unter `https://github.com/PROFILE/REPOSITORY/installations` hinzufügt.

Neben der automatischen Änderung von Pull-Requests führt die App auch autoupdate aus, um eure Konfiguration aktuell zu halten.

Weitere Installationen könnt ihr hinzufügen unter Install pre-commit ci.

`.github/workflows/pre-commit.yml`
: Alternative Konfiguration als GitHub-Workflow, z.B.:

```yaml
name: pre-commit

on:
  pull_request:
  push:
    branches: [main]

jobs:
  pre-commit:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - uses: actions/setup-python@v3
    - uses: actions/cache@v3
      with:
        path: ~/.cache/pre-commit
        key: pre-commit|${{ env.pythonLocation }}|${{ hashFiles('.pre-commit-config.
↪yaml') }}
    - uses: pre-commit/action@v3.0.0
```

Siehe auch:

- pre-commit/action

Beispiel für GitLab Actions

```
stages:
  - validate

pre-commit:
  stage: validate
  image:
    name: python:3.10
  variables:
    PRE_COMMIT_HOME: ${CI_PROJECT_DIR}/.cache/pre-commit
  only:
    refs:
      - merge_requests
      - tags
      - main
  cache:
    paths:
      - ${PRE_COMMIT_HOME}
  before_script:
    - pip install pre-commit
  script:
    - pre-commit run --all-files
```

Siehe auch:

Weitere Informationen zur Feinabstimmung des Caching findet ihr in Good caching practices.

Hooks überspringen

Die meisten *Git Hooks* lassen sich mit der Option `--no-verify` umgehen. So könnt ihr z.B. den pre-Commit-Hook überspringen mit:

```
$ git commit --no-verify -m "Quick and dirty"
```

Wenn ihr nur bestimmte *pre-commit-Skripte* überspringen möchtet, könnt ihr hierfür die Umgebungsvariable `SKIP` mit einer kommagetrennten Liste von Hook-IDs verwenden, z.B.:

```
$ SKIP=check-added-large-file,no-commit-to-branch git commit -m "Hotfix"
```

Vorlage für Git-Repositories

`pre-commit init-templatedir` kann verwendet werden, um eine Vorlage für die init.templateDir-Option von Git einzurichten, womit jedes neu geklonte Repository automatisch die pre-commit-Hooks erhält, ohne dass `pre-commit install` ausgeführt werden muss, z.B.:

```
$ git config --global init.templateDir ~/.config/git/template
$ pre-commit init-templatedir ~/.config/git/template
pre-commit installed at /Users/veit/.config/git/template/hooks/pre-commit
```

Jupyter Notebooks unter Git

Probleme bei der Versionsverwaltung von Jupyter Notebooks

Es gibt mehrere Probleme, um Jupyter Notebooks mit Git zu verwalten:

- Die Metadaten von Zellen der Jupyter Notebooks ändern sich auch, wenn keine inhaltlichen Änderungen an den Zellen vorgenommen wurden. Damit werden Git-Diffs unnötig kompliziert.
- Die Zeilen, die Git bei *Merge-Konflikten* in die `*.ipynb`-Dateien schreibt, führen dazu, dass die Notebooks nicht mehr gültiges JSON sind und von Jupyter deswegen nicht geöffnet werden kann: ihr erhaltet dann beim Öffnen die Fehlermeldung *Error loading notebook*.

  Konflikte treten besonders häufig in Notebooks auf, da Jupyter bei jeder Ausführung eines Notizbuchs Folgendes ändert:

  - Jede Zelle enthält eine Nummer, die angibt, in welcher Reihenfolge sie ausgeführt wurde. Wenn Team-Mitglieder die Zellen in unterschiedlicher Reihenfolge ausführen, hat jede einzelne Zelle einen Konflikt! Dies manuell zu beheben, würde sehr lange dauern.
  - Für jede Abbildung, z.B. einen Plot, nimmt Jupyter nicht nur das Bild selbst in das Notizbuch auf, sondern auch eine einfache Textbeschreibung, die die ID des Objekts enthält, z.B. *<matplotlib.axes._subplots.AxesSubplot at 0x7fbc113dbe90>*. Dies ändert sich jedes Mal, wenn ihr ein Notizbuch ausführt, und führt daher jedes Mal zu einem Konflikt, wenn zwei Personen diese Zelle ausführen.
  - Einige Ausgaben können nicht-deterministisch sein, z.B. ein Notebook, das Zufallszahlen verwendet oder mit einem Dienst interagiert, der im Laufe der Zeit unterschiedliche Ausgaben liefert.
  - Jupyter fügt dem Notizbuch Metadaten hinzu, die die Umgebung beschreiben, in der es zuletzt ausgeführt wurde, wie z.B. den Namen des Kernels. Dies variiert oft zwischen verschiedenen Installationen, und daher werden zwei Personen, die ein Notizbuch speichern (auch ohne andere Änderungen), oft einen Konflikt in den Metadaten haben.

nbdev2

nbdev2 bietet eine Reihe von Git-Hooks, die saubere Git-Diffs bereitstellen, die die meisten Git-Konflikte automatisch lösen und sicherstellen, dass alle verbleibenden Konflikte vollständig innerhalb der Standard-Jupyter-Notebook-Umgebung aufgelöst werden können:

- Ein neuer `git merge`-Treiber bietet *notebook-native* Konfliktmarkierungen, die dazu führen, dass Notebooks direkt in Jupyter geöffnet werden können, auch wenn es Git-Konflikte gibt. Lokale und entfernte Änderung werden jeweils als separate Zellen im Notizbuch angezeigt, so dass ihr die Version, die ihr nicht behalten möchtet, einfach löschen oder die beiden Zellen nach Bedarf kombinieren könnt.

Siehe auch:

nbdev.merge docs

- Git-Merges lokal zu lösen ist äußerst hilfreich, aber wir müssen sie auch Remote lösen. Wenn z.B. ein *Merge-Request* eingereicht wird und dann jemand anderes dasselbe Notebook überträgt, bevor der Merge-Request zusammengeführt wird, könnte dieser einen Konflikt hervorrufen:

```
   "outputs": [
    {
<<<<<< HEAD
     "execution_count": 8,
======
     "execution_count": 5,
>>>>>> 83e94d58314ea43ccd136e6d53b8989ccf9aab1b
     "metadata": {},
```

  Der *save hook* von nbdev2 entfernt automatisch alle unnötigen Metadaten (einschließlich `execution_count`) und nicht-deterministischen Zellausgaben; D.H., dass es keine sinnlosen Konflikte wie den obigen gibt, da diese Informationen gar nicht erst in den Commits gespeichert werden.

Um loszulegen, folgt den Anweisungen in Git-Friendly Jupyter.

jq

Im Notebook-Dateiformat nbformat können auch die Ergebnisse der Berechnungen gespeichert werden. Dies können auch Base-64-codierte Blobs für Bilder und andere Binärdaten sein, die üblicherweise nicht in eine Versionsverwaltung übernommen werden sollen. Diese können zwar manuell entfernt werden mit *Cell* → *All Output* → *Clear*, ihr müsst diese Schritte jedoch vor jedem `git add` ausführen, und es löst auch eine zweite Ursache für das Rauschen in `git diff` nicht, nämlich dasjenige in den Metadaten.

Um nun systematisch vergleichbare Versionen von Notebooks in der Versionsverwaltung zu erhalten, können wir jq verwenden, einen leichtgewichtigen JSON-Prozessor. Zwar benötigt man einige Zeit um `jq` einzurichten da es eine eigene Abfrage-/Filtersprache mitbringt, aber meist sind schon die Standardeinstellungen gut gewählt.

Installation

`jq` kann installiert werden mit:

```
$ sudo apt install jq
```

```
$ brew install jq
```

Beispiel

Ein typischer Aufruf ist:

```
jq --indent 1  \
  '(.cells [] | select (has ("output")) | .outputs) = []
  | (.cells [] | select (has ("execution_count")) | .execution_count) = null
  | .metadata = {"language_info": {"name": "python", "pygments_lexer": "ipython3"}}
  | .Cells []. Metadaten = {}
  '  example.ipynb
```

Jede Zeile innerhalb der einfachen Anführungszeichen definiert einen Filter – die erste wählt alle Einträge aus der Liste *cells* aus und löscht die Ausgaben. Der nächste Eintrag setzt alle Ausgaben zurück. Der dritte Schritt löscht die Metadaten des Notebooks und ersetzt sie durch ein Minimum an erforderlichen Informationen, damit das Notebook noch ohne Beanstandungen ausgeführt werden kann, folgendes eingeben:wenn es mit nbsphinx formatiert sind. Die vierte Filterzeile, `.cells []. metadata = {}`, löscht alle Metainformationen. Falls ihr bestimmte Metainformationen beibehalten wollt, könnt ihr dies hier angeben.

Einrichten

1. Um euch die Arbeit zu erleichtern, könnt ihr einen Alias in der `~/.bashrc`-Datei anlegen:

```
alias nbstrip_jq="jq --indent 1 \
    '(.cells[] | select(has(\"outputs\")) | .outputs) = []  \
    | (.cells[] | select(has(\"execution_count\")) | .execution_count) = null  \
    | .metadata = {\"language_info\": {\"name\": \"python\", \"pygments_lexer\": \
↪"ipython3\"}} \
    | .cells[].metadata = {} \
    '"
```

2. Anschließend könnt ihr bequem im Terminal folgendes eingeben:

```
$ nbstrip_jq example.ipynb > stripped.ipynb
```

3. Wenn ihr von einem bereits vorhandenen Notebook ausgeht, solltet ihr zunächst einen `filter`-Commit hinzufügen, indem ihr einfach die neu gefilterte Version eures Notebooks ohne die unerwünschten Metadaten einlest. Nachdem ihr mit `git add` das Notebook hinzugefügt habt, könnt ihr mit `git diff --cached` schauen, ob der Filter auch wirklich gewirkt hat bevor ihr dann `git commit -m 'filter'` angebt.
4. Wenn ihr diesen Filter für alle Git-Repositories verwenden wollt, könnt ihr euer Git auch global konfigurieren:
    1. Zunächst fügt ihr in `~/.gitconfig` folgendes hinzu:

```
[core]
attributesfile = ~/.gitattributes

[filter "nbstrip_jq"]
clean = "jq --indent 1 \
        '(.cells[] | select(has(\"outputs\")) | .outputs) = []  \
        | (.cells[] | select(has(\"execution_count\")) | .execution_count) =␣
↪null  \
        | .metadata = {\"language_info\": {\"name\": \"python\", \"pygments_
↪lexer\": \"ipython3\"}} \
        | .cells[].metadata = {} \
        '"
smudge = cat
required = true
```

    clean
    : wird beim Hinzufügen von Änderungen in den Bühnenbereich angewendet.

    smudge
    : wird beim Zurücksetzen des Arbeitsbereichs durch Änderungen aus dem Bühnenbereich angewendet.

    2. Anschließend müsst ihr in `~/.gitattributes` nur noch folgendes angeben:

```
*.ipynb filter=nbstrip_jq
```

5. Wenn ihr anschließend mit `git add` euer Notebook in den Bühnenbereich übernehmt, wird der `nbstrip_jq`-Filter angewendet.

Bemerkung: `git diff` zeigt euch jedoch keine Änderungen zwischen Arbeits- und Bühnenbereich an. Lediglich mit `git diff --staged` könnt ihr erkennen, dass nur die gefilterten Änderungen übernommen wurden.

> **Warnung:** `clean` und `smudge`-Filter spielen oft nicht gut mit `git rebase` über solche gefilterten Commits hinweg zusammen. Dann solltet ihr vor dem Rebase diese Filter deaktivieren.

6. Und es gibt noch ein weiteres Problem: Wenn ein solches Notebook erneut ausgeführt wird, zeigt zwar `git diff` keine Änderungen an, `git status` jedoch schon. Daher sollte in der `~/.bashrc`-Datei folgendes eingetragen sein um schnell das jeweilige Arbeitsverzeichnis reinigen zu können:

```
function nbstrip_all_cwd {
    for nbfile in *.ipynb; do
        echo "$( nbstrip_jq $nbfile )" > $nbfile
    done
    unset nbfile
}
```

ReviewNB

ReviewNB löst das Problem, *Merge-Requests* mit Notebooks durchzuführen. Die Code-Review-GUI von GitLab funktioniert nur bei zeilenbasierten Dateiformaten, wie z.B. Python-Skripten. Meistens bevorzuge ich jedoch, die Quelltext-Notebooks zu prüfen, weil:

- ich die Dokumentation und die Tests überprüfen möchte, nicht nur die Implementierung
- ich die Änderungen an den Zellausgaben sehen möchte, wie Diagrammen und Tabellen, nicht nur den Code.

Für diesen Zweck ist ReviewNB perfekt.

nbdime

nbdime ist ein GUI für nbformat-Diffs und ersetzt nbdiff. Es versucht lokal *Content-Aware*-Diffing sowie das Merging von Notebooks, beschränkt sich nicht nur auf die Darstellung von Diffs, sondern verhindert auch, dass unnötige Änderungen eingecheckt werden. Es ist jedoch nicht kompatibel mit `nbdev2`.

nbstripout

nbstripout automatisiert *Clear all outputs*. Es nutzt auch nbformat und ein paar Automagien um `git config` einzurichten. Meines Erachtens hat es jedoch zwei Nachteile:

- es beschränkt sich auf den problematischen Metadaten-Abschnitt
- es ist langsam.

Git für Binärdateien

`git diff` lässt sich konfigurieren, sodass es auch bei Binärdateien sinnvolle Diffs anzeigen kann.

... für Excel-Dateien

Hierfür benötigen wir openpyxl und pandas:

```
$ pipenv install openpyxl pandas
```

Anschließend können wir in `exceltocsv.py` pandas.DataFrame.to_csv zum Konvertieren der Excel-Dateien verwenden:

Quellcode 1: exceltocsv.py

```
import sys

from io import StringIO

import pandas as pd


for sheet_name in pd.ExcelFile(sys.argv[1]).sheet_names:
    output = StringIO()
    print("Sheet: %s" % sheet_name)
    pd.read_excel(sys.argv[1], sheet_name=sheet_name).to_csv(
        output, header=True, index=False
    )
    print(output.getvalue())
```

Nun fügt ihr noch in der globalen Git-Konfiguration in der Datei `~/.gitconfig` den folgenden Abschnitt hinzu:

```
[diff "excel"]
    textconv=python3 /PATH/TO/exceltocsv.py
    binary=true
```

Schließlich wird in der globalen `~/.gitattributes`-Datei unser `excel`-Konverter mit `*.xlsx`-Dateien verknüpft:

```
*.xlsx diff=excel
```

... für PDF-Dateien

Hierfür wird zusätzlich `pdftohtml` benötigt. Ihr installiert es mit

```
$ sudo apt install poppler-utils
```

```
$ brew install pdftohtml
```

Anschließend fügt ihr den folgenden Abschnitt der globalen Git-Konfiguration in der Datei `~/.gitconfig` hinzu:

```
[diff "pdf"]
    textconv=pdftohtml -stdout
```

Schließlich wird in der globalen `~/.gitattributes`-Datei unser `pdf`-Konverter mit `*.pdf`-Dateien verknüpft:

```
*.pdf diff=pdf
```

Nun wird beim Aufruf von `git diff` die PDF-Datei zunächst konvertiert und dann ein Diff über den Ausgaben des Konverters durchgeführt.

... für Word-Dokumente

Auch Unterschiede in Word-Dokumenten lassen sich anzeigen. Hierfür kann Pandoc verwendet werden, das einfach installiert werden kann mit

```
$ sudo apt install pandoc
```

```
$ brew install pandoc
```

Herunterladen und Installieren der aktuellen `.msi`-Datei von GitHub.

Anschließend wird der globalen Git-Konfiguration `~/.gitconfig` folgender Abschnitt hinzugefügt:

```
[diff "word"]
        textconv=pandoc --to=markdown
        binary=true
        prompt=false
```

Schließlich wird in der globalen `~/.gitattributes`-Datei unser `word`-Konverter mit `*.docx`-Dateien verknüpft:

```
*.docx diff=word
```

Die gleiche Vorgehensweise kann auch angewandt werden, um nützliche Diffs von anderen Binärdateien zu erhalten, z.B. `*.zip`, `*.jar` und andere Archive mit `unzip` oder für Änderungen in den Metainformationen von Bildern mit `exiv2`. Zudem gibt es Konvertierungswerkzeuge für die Umwandlung von `*.odt`, `*.doc` und anderen Dokumentenformaten in einfachen Text. Für Binärdateien, für die es keinen Konverter gibt, reichen oft auch Strings aus.

Visual Studio Code

Visual Studio Code kann eine bereits vorhandene *Git-Installation* nutzen um die entsprechenden Funktionalitäten zur Verfügung stellen zu können.

Klonen

Abb. 1: Source-Control-Icon

Wenn ihr noch kein Repository geöffnet habt, habt ihr in der *Source Code*-Ansicht die Möglichkeit, *Open Folder* oder *Clone Repository* auszuwählen. Bei *Clone Repository* werdet ihr nach der URL des Repository gefragt.

Wenn ihr ein Git-Repository öffnet und beginnt, Änderungen vorzunehmen, fügt VS-Code nützliche Anmerkungen hinzu:

- ein rotes Dreieck zeigt an, wo Zeilen gelöscht worden sind
- ein grüner Balken zeigt neu hinzugefügte Zeilen an
- ein blauer Balken zeigt geänderte Zeilen an

Commit

`git add` und `git reset` können entweder im Kontextmenü einer Datei ausgewählt werden oder per drag & drop. Nach einem `git commit` könnt ihr eine Commit-Nachricht eingeben und mit `Ctrl` oder bestätigen. Wenn es bereits Änderungen im Bühnenbereich gibt, werden nur diese übernommen; andernfalls werdet ihr aufgefordert, Änderungen auszuwählen. Ggf. erhaltet ihr spezifischere Commit-Aktionen in *Views and More Actions...*.

Bemerkung: Wenn ihr euren Commit versehentlich im falschen Branch erstellt habt, könnt ihr diesen Commit zurücknehmen mit *Git: Undo Last Commit* in der *Command Palette* (`P`).

Das Source-Control-Icon in der Aktivitätsleiste auf der linken Seite zeigt euch an, wieviele Änderungen ihr in eurem Repository gemacht habt. Wenn ihr das Icon auswählt, erhaltet ihr einen detaillierteren Überblick über eure Änderungen. Bei der Auswahl einer einzelnen Datei werden euch dann die zeilenweisen Textänderungen angezeigt. Ihr könnt auch den Editor auf der rechten Seite nutzen um weitere Änderungen vorzunehmen.

Zweige und Tags

Ihr könnt Zweige erstellen und in diese wechseln mit *Git: Create Branch* und *Git: Checkout to* aus der *Command Palette* (`P`). Wenn ihr *Git: Checkout to* aufruft, erscheint anschließend eine Dropdown-Liste mit allen Zweigen und Tags des Repository. Ihr könnt hier auch einen neuen Zweig erstellen.

Git-Statuszeile

Abb. 2: Statuszeile

In der unteren linken Ecke seht ihr die Statusanzeige mit weiteren Indikatoren über den Zustand eures Repository:

- den aktuellen Zweig mit der Möglichkeit, in einen anderen Zweig zu wechseln
- ein- und ausgehenden Commits
- die *Synchronize Changes*-Aktion, die zunächst `git pull` und dann `git push` ausführt.

Erweiterungen

- Git Blame
- Git History
- Git Lens
- *GitLab VS Code Extension*

GitLab VS Code Extension

GitLab VS Code Extension integriert GitLab 13.0 in Visual Studio Code:

Anzeigen von GitLab-Issues und Merge-Requests
: Issues, Kommentare, Merge Requests und geänderte Dateien werden in der Seitenleiste oder in einer benutzerdefinierten Suche angezeigt.

Merge-Requests erstellen und überprüfen
: Issues können direkt in VS Code kommentiert werden, und dabei werden auch GitLab Slash Commands unterstützt. In der Ansicht des Diff eines Merge-Requests könnt ihr Kommentare erstellen, bearbeiten und löschen.

GitLab CI/CD konfigurieren und validieren
: Ihr könnt Die `gitlab-ci.yml`-Datei editieren, wbei die Variablen automatisch vervollständigt werden. Zudem könnt ihr die Datei lokal validieren lassen.

Repository ohne Klonen durchsuchen
: Ihr könnt lesend auf Repositories zugreifen, sofern ein Zugriffstoken für die zugehörige GitLab-Instanz registriert ist.

GitLab

GitLab ist eine Webanwendung zur Versionsverwaltung auf Basis von Git. Später kamen weitere Funktionen hinzu wie ein Issue-Tracking-System mit Kanban-Board, ein System für Continuous Integration und Continuous Delivery (CI/CD) sowie ein Wiki und Snippets. Die GitLab Community Edition (CE) wird als Open-Source-Software unter der MIT-Lizenz entwickelt und kann On-Premises, also in den eigenen Räumlichkeiten, installiert werden.

Die GitLab CI Tools ermöglichen automatisierte Builds und Deployments ohne dass externe Integrationen erforderlich wären. Wenn bereits eine PaaS-Lösung wie Kubernetes verwendet wird, können mit GitLab-CI/CD Apps automatisch bereitgestellt, getestet und skaliert werden. Zudem kann Code automatisch auf potenzielle Sicherheitsrisiken gescannt werden.

GitLab ist eine komplett paketierte Plattform, während GitHub mit Apps aus dem Marketplace erweitert werden kann. Das bedeutet aber nicht, dass GitLab nicht integriert werden kann, z.B. mit Asana, Jira, Microsoft Teams, Slack etc.

Siehe auch:

Visual Studio Code: GitLab Workflow

Rollen, Gruppen und Berechtigungen

Je nach der Rolle, die eine Person in einer bestimmten Gruppe oder einem Projekt innehat, verfügt sie über unterschiedliche Berechtigungen. Wenn sie sowohl in einer Projektgruppe als auch im Projekt ist, wird die höchste Rolle verwendet.

Siehe auch:

- Permissions and roles

Mitglieder eines Projekts

Mitglieder sind die Personen und Gruppen, die Zugang zu eurem Projekt haben. Jedes Mitglied erhält eine Rolle, die bestimmt, was es im Projekt tun kann. Projektmitglieder können:

- direkte Mitglieder des Projekts sein
- die Mitgliedschaft im Projekt von der Projektgruppe erben
- Mitglied einer Gruppe sein, die mit dem Projekt geteilt wurde
- Mitglied einer Gruppe sein, die mit der Gruppe des Projekts geteilt wurde

Berechtigungen in GitLab

Gäste
: sind keine aktiven Mitwirkenden in privaten Projekten; sie können nur sehen und Kommentare und Issues hinterlassen.

Reporter*innen
: nehmen lesend teil. sie können nicht in das Repository schreiben, aber sie können an Issues mitarbeiten.

Entwickler*innen
: wirken direkt mit und haben Zugang zu allem, um von der Idee bis zur Produktion, es sei denn, etwas wurde ausdrücklich eingeschränkt, z.B. durch den Schutz von Zweigen.

Maintainer
: können zu `main` pushen und den Code in die `production`-Umgebung überführen.

Eigentümer*innen
: administrieren im Wesentlichen die Gruppen und Workflows. Sie können Zugang zu Gruppen gewähren und dürfen löschen.

Geschützte Zweige

In GitLab werden Berechtigungen grundsätzlich so definiert, dass Lese- oder Schreibrechte für das Repository und die Zweige vergeben werden. Um bestimmten Zweigen weitere Einschränkungen aufzuerlegen, können sie geschützt werden. Der Standardzweig für euer Projektarchiv ist standardmäßig geschützt. Wenn ein Zweig geschützt ist, werden standardmäßig üblicherweise die folgenden Einschränkungen für den Zweig erzwungen:

Aktion	Rolle
Einen Zweig schützen	Maintainer
Push in diesen Zweig	GitLab-Admins und alle, denen dies explizit erlaubt wurde.
Force push in diesen Zweig	Niemand
Löschen der Verzweigung	Mit einem Git-Kommando niemand; mit GitLab-UI oder API zumindest Maintainer

Siehe auch:

- Protected branches
- Pipeline security on protected branches

Geschützte Zweige konfigurieren

Voraussetzung ist, dass ihr mindestens die *Maintainer*-Rolle habt.

1. Wählt in der oberen Leiste *Menü* → *Projekte* und sucht euer Projekt.
2. Wählt in der linken Seitenleiste *Einstellungen* → *Repository*.
3. Erweitert * Protected branches*.
4. Wählt in der Dropdown-Liste *Protect branch…* den Zweig aus, den ihr schützen möchtet. Alternativ könnt ihr auch Wildcards verwenden:

Wildcard	Beispiele
`*-stage`	`#17-some-feature-stage`, `#42-other-feature-stage`
`production/*`	`production/app-server`, `production/load-balancer`
`*app-server*`	`app-server`, `production/app-server`

5. Wählt in der Dropdown-Liste *Allowed to merge:* eine Rolle aus, die in diesen Zweig zusammenführen darf.
6. Wählt in der Dropdown-Liste *Allowed to push:* eine Rolle aus, der in diesen Zweig pushen darf.
7. Wählt *Schützen*.
8. Der geschützte Zweig wird nun in der Liste der geschützten Zweige angezeigt.

Merge-Requests

Mit Merge-Requests könnt ihr Quellcodeänderungen in einen Zweig einchecken. Wenn ihr eine Zusammenführungsanforderung öffnet, könnt ihr die Codeänderungen vor der Zusammenführung visualisieren und gemeinsam daran arbeiten. Zusammenführungsanfragen enthalten:

- eine Beschreibung der Anfrage
- Codeänderungen und Codeüberprüfungen
- Informationen über *CI/CD-Pipelines*
- Diskussionsbeiträge
- die Liste der Commits

Siehe auch:

- Merge requests

Merge-Request-Arbeitsabläufe

1. Ihr checkt einen neuen Zweig aus und übermittelt eure Änderungen durch eine Zusammenführungsanforderung.
2. Ihr holt Feedback von eurem Team ein.
3. Ihr arbeitet an der Implementierung und optimiert den Code mit Codequalitätsberichten.
4. Ihr verifiziert eure Änderungen mit Berichten von Unit-Tests in *GitLab CI/CD*.
5. Ihr vermeidet die Verwendung von Abhängigkeiten, deren Lizenz nicht mit eurem Projekt kompatibel ist, mit *Berichten zur Lizenzkonformität*.
6. Ihr beantragt die Genehmigung eurer Änderungen.
7. Wenn die Zusammenführungsanforderung genehmigt wurde, wird die *GitLab CI/CD* die Änderungen in der `production`-Umgebung bereitstellen.

GitLab CI/CD

GitLab CI/CD kann bei iterativen Code-Änderungen eure Anwendungen automatisch erstellen, testen, bereitstellen und überwachen. Dies verringert die Gefahr, dass ihr neuen Code auf der Grundlage fehlerhafter Vorgängerversionen entwickelt. Dabei sollen von der Entwicklung bis zu seiner Bereitstellung von Code-Änderungen wenige oder gar keine menschlichen Eingriffe erforderlich sein.

Die drei wichtigsten Ansätze für diese kontinuierliche Entwicklung sind:

Continuous Integration
: führt eine Reihe von Skripten sequentiell oder parallel aus, die eure Anwendung automatisch erstellt und testet, z.B. nach jedem `git pull` in einem *Feature-Branch*. Damit soll es weniger wahrscheinlich werden, dass ihr Fehler in eure Anwendung einbringt.

  Wenn die Überprüfungen wie erwartet funktionieren, könnt ihr einen *Merge Request* stellen; schlagen die Überprüfungen fehl, könnt ihr die Änderungen GGF. zurücknehmen.

  Siehe auch:

  - Kontinuierliche Integration

Continuous Delivery
geht einen Schritt weiter als Kontinuierliche Integration und stellt die Anwendung auch kontinuierlich bereit. Dies erfordert jedoch noch einen manuellen Eingriff, um die Änderungen manuell in einem *Deployment Branch* bereitzustellen.

Siehe auch:

- Continuous Delivery
- Continuous Delivery

Continuous Deployment
führt auch die Bereitstellung der Software auf die Produktiv-Infrastruktur automatisch durch.

Aktivieren von CI/CD in einem Projekt

1. Wählt in der oberen Leiste *Menü* → *Projekte* und sucht euer Projekt.
2. Wählt in der linken Seitenleiste *Einstellungen* → *Allgemein*.
3. Erweitert *Visibility, project features, permissions*.
4. Aktiviert im Abschnitt *Repository* die Option *CI/CD*.
5. Wählt *Änderungen speichern*.

CI/CD-Pipelines

Pipelines sind die wichtigste Komponente der Continuous Integration, Delivery und Deployment.

Pipelines bestehen aus:

Jobs
legen fest, was zu tun ist, z.B. Kompilieren von Code oder Testen.

Siehe auch:

Jobs

Stages
legen fest, wann die Jobs ausgeführt werden sollen, z.B. die Phase `test`, die nach der Phase `build` ausgeführt werden soll.

Siehe auch:

Stages

Jobs werden von sog. Runners ausgeführt. Mehrere *Jobs* in einem *Stage* werden parallel ausgeführt, sofernes genügend gleichzeitige Runner zur Verfügung stehen.

Wenn alle *Jobs* in einem *Stage* erfolgreich sind, fährt die Pipeline mit dem nächsten *Stage* fort.

Schlägt ein *Job* in einem *Stage* fehl, wird der nächste *Stage* normalerweise nicht ausgeführt, und die Pipeline wird vorzeitig beendet.

Im Allgemeinen werden Pipelines automatisch ausgeführt und erfordern nach ihrer Erstellung keinen Eingriff. Es gibt jedoch auch Fälle, in denen ihr manuell in eine Pipeline eingreifen könnt.

Eine typische Pipeline kann aus vier *Stages* bestehen, die in der folgenden Reihenfolge ausgeführt werden:

1. Eine `build`-*Stage* mit einem *Job* namens `compile`.
2. Eine `test`-*Stage* mit zwei parallelen *Jobs* namens `unit-test` und `lint`.

3. Eine `staging`-*Stage* mit einem *Job* namens `deploy-to-stage`.
4. Eine `production`-*Stage* mit einem *Job* namens `deploy-to-prod`.

Die zugehörige `.gitlab-ci.yml`-Datei könnte dann so aussehen:

```
image: "docker.io/ubuntu"

stages:
  - build
  - test
  - staging
  - production

compile:
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."

unit-test:
  stage: test
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 60
    - echo "Code coverage is 0%"

lint:
  stage: test
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."

deploy-to-stage:
  stage: stage
  script:
    - echo "Deploying application in staging environment..."
    - echo "Application successfully deployed to staging."

deploy-to-production:
  stage: production
  script:
    - echo "Deploying application in production environment..."
    - echo "Application successfully deployed to production."
```

Pipelines anzeigen

Ihr findet die aktuellen und historischen Pipeline-Runs auf der Seite *CI/CD → Pipelines* eures Projekts. Ihr könnt auch auf Pipelines für einen *Merge-Request* zugreifen, indem ihr zu deren Registerkarte *Pipelines* navigiert. Wählt eine Pipeline aus, um die Seite *Pipeline-Details* zu öffnen und die *Jobs* anzuzeigen, die für diese Pipeline ausgeführt wurde. Von hier aus könnt ihr eine laufende Pipeline abbrechen, *Jobs* in einer fehlgeschlagenen Pipeline erneut versuchen oder eine Pipeline löschen.

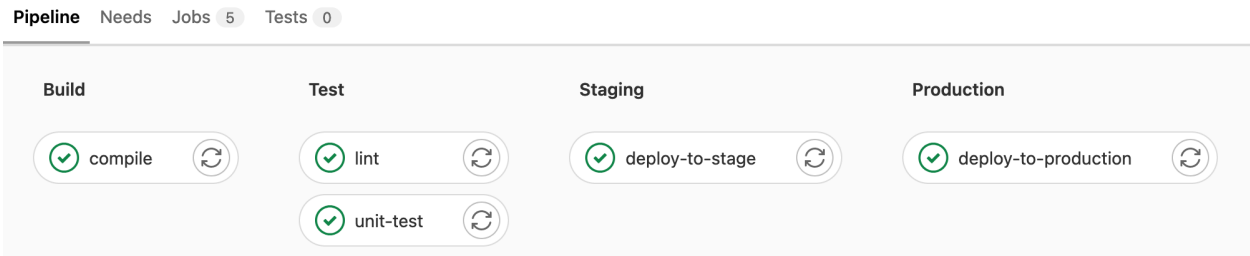


Abb. 3: GitLab-CI/CD-Pipeline

Siehe auch:

- Customize pipeline configuration
- Scheduled pipelines
- GitLab CI/CD variables
- Predefined variables reference

Migration von GitHub-Aktionen

GitLab CI/CD und GitHub Actions weisen einige Ähnlichkeiten in der Konfiguration auf, wodurch die Migration zu GitLab CI/CD relativ einfach ist:

- Workflow-Konfigurationsdateien sind in *YAML* geschrieben und werden im Repository zusammen mit dem Code gespeichert.
- Workflows enthalten einen oder mehrere Jobs.
- Jobs umfassen einen oder mehrere Schritte oder einzelne Befehle.
- Jobs können entweder auf verwalteten oder selbst gehosteten Rechnern laufen.

Es gibt jedoch auch einige Unterschiede, und dieser Leitfaden wird euch die wichtigsten Unterschiede aufzeigen, damit ihr euren Workflow auf GitLab CI/CD migrieren könnt.

Jobs

Jobs in GitHub Actions sind den Jobs in GitLab CI/CD sehr ähnlich. Beide haben folgende Merkmale:

- Jobs enthalten eine Reihe von Schritten oder Skripten, die nacheinander ausgeführt werden.
- Jobs können auf separaten Rechnern oder in separaten Containern ausgeführt werden.
- Jobs werden standardmäßig parallel ausgeführt, können aber auch für sequentielle Ausführung konfiguriert werden.

- Jobs können ein Skript oder einen Shell-Befehl ausführen, wobei in GitHub-Aktionen alle Skripte mit dem Schlüssel `run` angegeben werden. In GitLab CI/CD werden die Skript-Schritte hingegen mit dem `script`-Schlüssel angegeben.

Im Folgenden findet ihr ein Beispiel für die Syntax der beiden Systeme.

GitHub Actions-Syntax für Jobs

```
jobs:
  my_job:
    steps:
      - uses: actions/checkout@v3
      - run: echo "Run my script here"
```

GitLab CI/CD-Syntax für Jobs

```
my_job:
  variables:
    GIT_CHECKOUT: "true"
  script:
    - echo "Run my script here"
```

Runners

Runner sind Maschinen, auf denen die Jobs ausgeführt werden. Sowohl GitHub Actions als auch GitLab CI/CD bieten verwaltete und selbst gehostete Varianten von Runners. In GitHub Actions wird der `runs-on`-Schlüssel verwendet, um Jobs auf verschiedenen Plattformen auszuführen, während dies in GitLab CI/CD mit `tags` erfolgt.

GitHub Actions-Syntax für Runner

```
my_job:
  runs-on: ubuntu-latest
  steps:
    - run: echo "Hello Pythonistas!"
```

GitLab CI/CD-Syntax für Runner

```
my_job:
  tags:
    - linux
  script:
    - echo "Hello Pythonistas!"
```

Docker-Images

GitHub Actions-Syntax für Docker-Images

```
jobs:
  my_job:
    container: python:3.10
```

GitLab CI/CD-Syntax für Docker-Images

```
my_job:
  image: python:3.10
```

Siehe auch:

- Run your CI/CD jobs in Docker containers

Syntax für Bedingungen und Ausdrücke

GitHub Actions verwendet das `if`-Schlüsselwort, um zu verhindern, dass ein Job ausgeführt wird, wenn eine Bedingung nicht erfüllt ist. GitLab CI/CD verwendet `rules`, um zu bestimmen, ob ein Job unter einer bestimmten Bedingung ausgeführt wird.

Im Folgenden findet ihr ein Beispiel für die Syntax der beiden Systeme.

GitHub-Syntax für Bedingungen und Ausdrücke

```
jobs:
  deploy:
    if: contains( github.ref, 'main')
    runs-on: ubuntu-latest
    steps:
      - run: echo "Deploy to production server"
```

GitLab-Syntax für Bedingungen und Ausdrücke

```
deploy:
  stage: deploy
  script:
    - echo "Deploy to production server"
  rules:
    - if: '$CI_COMMIT_BRANCH == "main"'
```

Neben `if` bietet GitLab auch noch weitere Regeln wie `changes`, `exists`, `allow_failure`, `variables` und `when`.

Siehe auch:

- rules
- Complex rules

Abhängigkeiten zwischen Jobs

Sowohl GitHub Actions als auch GitLab CI/CD ermöglichen euch, Abhängigkeiten für einen Job festzulegen. In beiden Systemen werden Jobs standardmäßig parallel ausgeführt, aber GitLab CI/CD verfügt über ein `stages`-Konzept, bei dem Jobs einer Stufe gleichzeitig ausgeführt werden, die nächste Stufe aber erst dann beginnt, wenn alle Aufträge der vorherigen Stufe abgeschlossen sind. In GitHub Actions können Abhängigkeiten zwischen Jobs explizit mit dem `needs`-Schlüssel nachgebildet werden.

Nachfolgend findet ihr ein Beispiel für die Syntax für jedes System. Die Workflows beginnen mit zwei parallel laufenden Jobs mit den Namen `unit-test` und `lint`. Wenn diese Jobs abgeschlossen sind, wird ein weiterer Job mit dem Namen `deploy-to-stage` ausgeführt. Wenn `deploy-to-stage` abgeschlossen ist, wird schließlich der Auftrag `deploy-to-prod` ausgeführt.

GitHub Actions-Syntax für Abhängigkeiten zwischen Jobs

```yaml
jobs:
  unit-test:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Running unit tests... This will take about 60 seconds."
      - run: sleep 60
      - run: echo "Code coverage is 0%"

  lint:
    runs-on: ubuntu-latest
    steps:
      - run: echo "Linting code... This will take about 10 seconds."
      - run: sleep 10
      - run: echo "No lint issues found."

  deploy-to-stage:
    runs-on: ubuntu-latest
    needs: [unit-test,lint]
    steps:
      - run: echo "Deploying application in staging environment..."
      - run: echo "Application successfully deployed to staging."

  deploy-to-prod:
    runs-on: ubuntu-latest
    needs: [deploy-to-stage]
    steps:
      - run: echo "Deploying application in production environment..."
      - run: echo "Application successfully deployed to production."
```

GitLab CI/CD-Syntax für Abhängigkeiten zwischen Jobs

```
stages:
  - test
  - stage
  - prod

unit-test:
  stage: test
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 60
    - echo "Code coverage is 0%"

lint:
  stage: test
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."

deploy-to-stage:
  stage: stage
  script:
    - echo "Deploying application in staging environment..."
    - echo "Application successfully deployed to staging."

deploy-to-prod:
  stage: prod
  script:
    - echo "Deploying application in production environment..."
    - echo "Application successfully deployed to production."
```

Artefakte

Sowohl GitHub Actions als auch GitLab CI/CD können Dateien und Verzeichnisse, die von einem Job erstellt wurden, als Artefakte hochladen. Diese Artefakte können verwendet werden, um Daten über mehrere Jobs hinweg zu erhalten.

Im Folgenden findet ihr ein Beispiel für die Syntax der beiden Systeme.

GitHub Actions Syntax für Artefakte

```
- name: Archive code coverage results
  uses: actions/upload-artifact@v3
  with:
    name: code-coverage-report
    path: output/test/code-coverage.html
```

GitLab CI/CD-Syntax für Artefakte

```
script:
artifacts:
  paths:
    - output/test/code-coverage.html
```

Datenbanken und Service-Container

Beide Systeme ermöglichen euch, zusätzliche Container für Datenbanken, Caching oder andere Abhängigkeiten einzubinden.

GitHub Actions verwendet den `container`-Schlüssel, während in GitLab CI/CD ein Container für den Job mit dem `image`-Schlüssel angegeben wird. In beiden Systemen werden zusätzliche Service-Container mit dem `services`-Schlüssel angegeben.

Im Folgenden findet ihr ein Beispiel für die Syntax der beiden Systeme.

GitHub Actions-Syntax für Datenbanken und Service-Container

```
jobs:
  test:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres
        env:
          POSTGRES_USER: postgres
          POSTGRES_PASSWORD: postgres
          POSTGRES_DB: postgres
        options: >-
          --health-cmd pg_isready
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - name: Python
        uses: actions/checkout@v3
        uses: actions/setup-python@v4
        with:
          python-version: '3.10'

      - name: Test with pytest
        run: python -m pytest
        env:
          DATABASE_URL: 'postgres://postgres:postgres@localhost:${{ job.services.
↪postgres.ports[5432] }}/postgres'
```

GitLab CI/CD-Syntax für Datenbank- und -Service-Container

```
test:
  variables:
    POSTGRES_PASSWORD: postgres
    POSTGRES_HOST: postgres
    POSTGRES_PORT: 5432
  image: python:latest
  services:
    - postgres
  script:
    - python -m pytest
```

Zuordnungen

GitHub	GitLab
Konzepte	
actions/upload-artifact@v2	artifacts
actions/cache@v2	cache
actions/download-artifact@v2	dependencies
environment	environment
container	image
actions/deploy-pages@main	pages
actions/create-release@v1	release
run	script, after_scri
hashicorp/vault-action@v2.5.0	secrets
services	services
runs-on	tags
timeout-minutes	timeout
Umgebungsvariablen	
${{ github.api_url }}	CI_API_V4_URL
${{ github.workspace }}	CI_BUILDS_DIR
${{ github.ref }}	CI_COMMIT_BRANCH,
${{ github.sha }}	CI_COMMIT_SHA, CI_
${{ github.job }}	CI_JOB_ID, CI_JOB_
${{ github.event_name == 'workflow_dispatch' }}	CI_JOB_MANUAL
${{ job.status }}	CI_JOB_STATUS
${{ github.server_url }}/${{ github.repository }}	CI_MERGE_REQUEST_
${{ github.token }}	CI_NODE_INDEX
${{ strategy.job-total }}	CI_NODE_TOTAL
${{ github.repository}}/${{ github.workflow }}	CI_PIPELINE_ID
${{ github.workflow }}	CI_PIPELINE_IID
${{ github.event_name }}	CI_PIPELINE_SOURC
${{ github.actions }}	CI_PIPELINE_TRIGG
${{ github.server_url }}/${{ github.repository }}/actions/runs/${{ github.run_id }}	CI_PIPELINE_URL
${{ github.workspace }}	CI_PROJECT_DIR
${{ github.repository }}	CI_PROJECT_ID CI_I
${{ github.event.repository.name }}	CI_PROJECT_NAME
${{ github.repository_owner }}	CI_PROJECT_NAMESP

Tab. 1 – Fortsetzung der vo

GitHub	GitLab
${{ github.event.repository.full_name }}	CI_PROJECT_TITLE
${{ github.server_url }}/${{ github.repository }}	CI_PROJECT_URL
${{ github.event.repository.clone_url }}	CI_REPOSITORY_URL
${{ runner.os }}	CI_RUNNER_EXECUTA
${{ github.server_url }}	CI_SERVER_HOST, CI
${{ github.actions }}	CI_SERVER, GITLAB_
${{ github.actor }}	GITLAB_USER_EMAIL
${{ github.event_path }}	TRIGGER_PAYLOAD
${{ github.event.pull_request.assignees }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.number }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.labels }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.milestone }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.head.ref }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.head.sha }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.head.repo.full_name }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.head.repo.url }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.base.ref }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.base.sha }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.title }}	CI_MERGE_REQUEST_
${{ github.event.pull_request.number }}	CI_EXTERNAL_PULL_
${{ github.event.pull_request.head.repo.full_name }}	CI_EXTERNAL_PULL_
${{ github.event.pull_request.base.repo.full_name }}	RCI_EXTERNAL_PULL

GitLab Package-Registry

Ihr könnt eure Verteilungspakete auch in der Paketregistrierung eures GitLab-Projekts veröffentlichen und sowohl mit Pip als auch mit twine nutzen.

Siehe auch:

GitLab Package Registry

git-big-picture

`git-big-picture` visualisiert Git-Repositories als DAGs. Das Tool kommt mit einigen Filtern um sich nur die interessanten Bereiche anzeigen zu lassen, z.B. nur die Hierarchie von Tags und Branches.

Beispiele

```
git big-picture -o git-big-picture.svg
```

```
$ git big-picture -ao git-big-picture_all.svg
```

Installation

Ihr könnt `git-big-picture` einfach installieren mit:

```
$ pipenv install git-big-picture
Installing git-big-picture...
Adding git-big-picture to Pipfile's [packages]...
✓ Installation Succeeded
...
```

Git-Integration

Ihr könnt das Tool einfach in Git integrieren indem ihr das Skript `git-big-picture` einfach `$PATH` hinzufügt. Anschließend könnt ihr es verwenden z.B. mit:

```
$ git big-picture -h
Usage: git-big-picture OPTIONS [<repo-directory>]

Options:
  --version             show program's version number and exit
  -h, --help            show this help message and exit
  --pstats=FILE         run cProfile profiler writing pstats output to FILE
  -d, --debug           activate debug output

  Output Options:
    Options to control output and format

    -f FMT, --format=FMT
                        set output format [svg, png, ps, pdf, ...]
    -g, --graphviz      output lines suitable as input for dot/graphviz
    -G, --no-graphviz   disable dot/graphviz output
    -p, --processed     output the dot processed, binary data
    -P, --no-processed  disable binary output
    -v CMD, --viewer=CMD
                        write image to tempfile and start specified viewer
    -V, --no-viewer     disable starting viewer
    -o FILE, --outfile=FILE
                        write image to specified file
    -O, --no-outfile    disable writing image to file

  Filter Options:
    Options to control commit/ref selection

    -a, --all           include all commits
    -b, --branches      show commits pointed to by branches
    -B, --no-branches   do not show commits pointed to by branches
    -t, --tags          show commits pointed to by tags
    -T, --no-tags       do not show commits pointed to by tags
    -r, --roots         show root commits
    -R, --no-roots      do not show root commits
    -m, --merges        include merge commits
    -M, --no-merges     do not include merge commits
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    -i, --bifurcations  include bifurcation commits
    -I, --no-bifurcations
                        do not include bifurcation commits
```

Konfiguration

Die Standard-`git config`-Infrastruktur kann verwendet werden um auch `git-big-picture` zu konfigurieren. Die meisten Kommandozeilen-Argumente können konfiguriert werden im `[big-picture]`-Abschnitt, z.B. um Firefox als Viewer zu konfigurieren

```
$ git config --global big-picture.viewer firefox
```

erstellt den folgenden Abschnitt in eurer `~/.gitconfig`-Datei:

```
[big-picture]
    viewer = firefox
```

Bemerkung: Bitte beachtet jedoch, dass ihr dann keine anderen Optionen mehr auswählen könnt. So könnt ihr nun den Graph nicht mehr als Graphviz ausgeben lassen:

```
$ git-big-picture -g
fatal: Options '-g | --graphviz' and '-p | --processed' are incompatible with other␣
↪output options.
```

In diesem Fall müsst ihr dann die `-V` oder `--no-viewer`-Option wählen:

```
$ git-big-picture -g -V
digraph {
    "c509669a01b156900eed9f1c9f927b6d2f7bb95b"[label="origin/pyup-scheduled-update-2020-
↪11-16", color="/pastel13/2", style=filled];
...
```

---

etckeeper

etckeeper ist eine Sammlung von Werkzeugen, mit denen das `/etc`-Verzeichnis in einem Git-Repository verwaltet werden kann. So können Änderungen überprüft und GGF. rückgängig gemacht werden. Zudem verbindet es sich mit Paketmanagern wie apt, um Änderungen, die während eines Paket-Upgrades an `/etc` vorgenommen werden, automatisch zu übertragen. Schließlich werden auch Metadaten von Dateien berücksichtigt, die Git normalerweise nicht verwaltet, die aber für `/etc` wichtig sind, wie z.B. die Berechtigungen von` `/etc/shadow`.

Installation

etckeeper kann einfach installiert werden mit

```
$ sudo apt install git etckeeper
```

Konfiguration

1. Die Konfiguration von etckeeper erfolgt in der `etckeeper.conf`-Datei:

   ```
   $ sudo vi /etc/etckeeper/etckeeper.conf
   # The VCS to use.
   #VCS="hg"
   VCS="git"
   #VCS="bzr"
   #VCS="darcs"
   ...
   ```

2. Außerdem sollten die folgenden beiden automatischen Commits vermieden werden:

   ```
   # Uncomment to avoid etckeeper committing existing changes
   # to /etc automatically once per day.
   AVOID_DAILY_AUTOCOMMITS=1
   ...
   # Uncomment to avoid etckeeper committing existing changes to
   # /etc before installation. It will cancel the installation,
   # so you can commit the changes by hand.
   AVOID_COMMIT_BEFORE_INSTALL=1
   ```

3. Nun sollte noch git selbst konfiguriert werden, s. (siehe) *Konfiguration*.

4. Schließlich kann das `/etc`-Verzeichnis unter die Git-Versionsverwaltung genommen werden mit:

   ```
   $ cd /etc/
   $ sudo etckeeper init
   Initialized empty Git repository in /etc/.git/
   $ sudo etckeeper commit "Initial commit"
   ```

Verwendung

Wird nun eine Konfigurationsdatei editiert, so können die Änderungen nun einfach mit Git protokolliert werden.

Metadaten verwalten

Da Git an sich keine vollständigen Metadaten aufzeichnet, wurde von etckeeper ein *pre-commit Hook* in `/etc/.git/hooks/pre-commit` eingerichtet. Dieser protokolliert in der Datei `/etc/.etckeeper` die `chmod`- und `chgrp`-Angaben für alle Dateien die nicht den Standardrechten entsprechen:

```
maybe chmod 0755 '.'
maybe chmod 0700 './.etckeeper'
maybe chmod 0644 './.gitignore'
...
. gitignore
```

Dateien, die nicht mit Git im `/etc`-Verzeichnis versioniert werden sollen, können in der Datei `/etc/.gitignore` hinzugefügt werden. Diese Datei wird beim Initiieren von etckeeper erzeugt und kann GGF. ergänzt werden nach dem Kommentar

```
# end section managed by etckeeper
```

Git's Datenbank-Interna

Siehe auch:

- Commits are snapshots, not diffs
- Git's database internals
  - Part I: packed object store
  - Part II: commit history queries
  - Part III: file history queries
  - Part IV: distributed synchronization
  - Part V: scalability

Git-Glossar

Branch
Zweig
: Ein Zweig ist eine Entwicklungslinie. Der letzte Commit auf einem Zweig wird als Spitze des Zweiges bezeichnet, der durch einen `head` referenziert wird und der sich weiterbewegt, wenn weitere Entwicklungen auf dem Zweig vorgenommen werden. Ein einzelnes Git-Repository kann eine beliebige Anzahl von Zweigen haben, aber ihr *Working Tree* ist nur mit einem von ihnen verbunden – dem *aktuellen* oder *ausgecheckten* Zweig – und *HEAD* zeigt auf diesen Zweig.

Cache
: Veraltet für *Index*.

Clone
Klon
: Lokale Version eines Repository einschließlich aller Commits und Branches.

Commit
Ein Snapshot des gesamten Git-Repository, komprimiert in einem SHA.

Fork
Kopie eines Repository auf GitLab, die einem anderen User oder einer anderen Gruppe gehört.

Git
Git ist eine verteilte Versionsverwaltung.

GitLab
Web-Anwendung zur Versionsverwaltung auf Basis von *git*. Später kamen Gitlab CI, ein System zur kontinuierlichen Integration, GitLab Runner, Container-Registry und vieles andere hinzu.

`HEAD`
Der `HEAD`-Zeiger repräsentiert euer aktuelles Arbeitsverzeichnis und kann mit `git switch` in verschiedene Zweige, Tags oder Commits verschoben werden.

Index
Eine Sammlung von Dateien mit Statusinformationen, deren Inhalt als Objekte gespeichert wird. Der Index ist eine gespeicherte Version eures *Working Tree*.

Merge request
Ort zum Vergleichen und Diskutieren der in einem Branch eingeführten Änderungen mit Bewertungen, Kommentaren, Tests etc.; siehe auch Merge requests.

`origin`
Das übliche Upstream-Repository. Die meisten Projekte haben mindestens ein Upstream-Projekt, das sie verfolgen. Standardmäßig wird `origin` für diesen Zweck verwendet. Neue Upstream-Aktualisierungen werden in Zweige mit dem Namen `origin/`*`NAME_OF_UPSTREAM_BRANCH`* geholt, die ihr mit `git branch -r` sehen könnt.

Remote Repository
Ein Repository, das zum Nachverfolgen eines gemeinsamen Projekt verwendet wird, sich aber an einem anderen Ort befindet.

Trunk-Based Development
TBD
Git-Workflow mit kurzlebigen Themenzweigen, die schnell zum einem einzigen `main`-Zweig zusammengeführt werden.

Siehe auch:

- Trunk Based Development

Working Tree
Der Baum der tatsächlich ausgecheckten Dateien. Der Arbeitsbaum enthält normalerweise den Inhalt des *HEAD*-Commit-Baums sowie alle lokalen Änderungen, die ihr vorgenommen, aber noch nicht übertragen habt.

7.2 Daten verwalten mit `DVC`

Für Datenanalysen und vor allem bei Machine Learning ist es äußerst wertvoll, verschiedene Versionen von Analysen, die mit verschiedenen Datensätzen und Parametern durchgeführt wurden, reproduzieren zu können. Um jedoch reproduzierbare Analysen zu erhalten, müssen sowohl die Daten als auch das Modell (einschließlich der Algorithmen, Parameter. etc.) versioniert werden. Die Versionierung von Daten für reproduzierbare Analysen ist aufgrund der Datengröße ein größeres Problem als die Versionierung von Modellen. Werkzeuge wie DVC helfen bei der Verwaltung von Daten, indem Nutzer diese mit einem *Git*-artigen Workflow an einen entfernten Datenspeicher übertragen können. Hierdurch vereinfacht sich der Abruf bestimmter Versionen von Daten um eine Analyse zu reproduzieren.

DVC wurde entwickelt, um ML-Modelle und Datensätze gemeinsam nutzen zu können und nachvollziehbar zu verwalten. Es arbeitet zwar mit verschiedenen Versionsverwaltungen zusammen, benötigt diese jedoch nicht. Im Gegensatz z.B. zu DataLad/git-annex ist es auch nicht auf Git als Versionsverwaltung beschränkt, sondern kann z.B auch zusammen mit Mercurial verwendet werden, siehe github.com/crobarcro/dvc/dvc/scm.py. Zudem nutzt es ein eigenes System zum Speichern der Dateien mit Unterstützung u.a. für SSH und HDFS.

DataLad konzentriert sich hingegen mehr auf die Entdeckung und Verwendung von Datasets, die dann einfach mit Git verwaltet werden. DVC hingegen speichert jeden Schritt der Pipeline in einer separaten `.dvc`-Datei, die dann durch Git verwaltet werden kann.

Diese `.dvc`-Dateien erlauben jedoch praktische Tools zur Manipulation und Visualisierung von DAGs, siehe z.B. die *Visualisierung der DAGs*.

Schließlich lassen sich mit *dvc remote* auch externe Abhängigkeiten angeben.

Siehe auch:

- Tutorial
- Documentation
- Git Repository

7.2.1 Installation

DVC lässt sich mit Pipenv installieren. Beachtet dabei jedoch bitte, dass ihr hierbei die Extras explizit angeben müsst. Dies können `[ssh]`, `[s3]`, `[gs]`, `[azure]`, und `[oss]` oder `[all]` sein. Für `ssh` sieht das Kommando dann so aus:

```
$ pipenv install dvc[ssh]
```

Alternativ kann DVC auch über andere Paketmanager installiert werden:

```
$ sudo wget https://dvc.org/deb/dvc.list -O /etc/apt/sources.list.d/dvc.list
$ sudo apt update
$ sudo apt install dvc
```

```
$ brew install iterative/homebrew-dvc/dvc
```

Bemerkung: Bitte beachtet, dass das folgende Beispiel mit einer aktuellen DVC-Version erstellt wurde (1.0.0a9), die z.T. (zum Teil) eine andere Syntax als frühere Versionen verwendet. Dies könnt ihr aktuell (8. Juni 2020) nur mit pip installieren:

```
$ pipenv install dvc[all]==1.0.0a9
```

Projekt erstellen

DVC lässt sich einfach initialisieren mit:

```
$ mkdir -p dvc-example/data
$ cd dvc-example
$ git init
$ dvc init
$ git add .dvc
$ git commit -m "Initialize DVC"
```

dvc init
: erstellt ein Verzeichnis `.dvc/` mit `config`, `.gitignore` und `cache`-Verzeichnis.

git commit
: stellt `.dvc/config` und `.dvc/.gitignore` unter Git-Versionskontrolle.

Konfigurieren

Bevor DVC verwendet wird, sollte noch ein entfernter Speicherplatz (*remote storage*) eingerichtet werden. Dieser sollte für alle zugänglich sein, die auf die Daten oder das Modell zugreifen sollen. Es ähnelt der Verwendung eines Git-Server. Häufig ist das jedoch auch ein NFS-Mount, der z.B. folgendermaßen eingebunden werden kann:

```
$ sudo mkdir -p /var/dvc-storage
$ dvc remote add -d local /var/dvc-storage
Setting 'local' as a default remote.
$ git commit .dvc/config -m "Configure local remote"
[master efaeb84] Configure local remote
 1 file changed, 4 insertions(+)
```

-d, --default
: Standardwert für den entfernten Speicherplatz

local
: Name des entfernten Speicherplatzes

/var/dvc-storage
: URL des entfernten Speicherplatzes

  Daneben werden noch weitere Protokolle unterstützt, die dem Pfad vorangestellt werden, u.a. `ssh:`, `hdfs:`, `https:`.

Es kann also einfach noch ein weiterer entfernter Datenspeicher hinzugefügt werden, z.B. mit:

```
$ dvc remote add webserver https://dvc.example.org/myproject
```

Die zugehörige Konfigurationsdatei `.dvc/config` sieht dann so aus:

```
['remote "local"']
url = /var/dvc-storage
[core]
remote = local
['remote "webserver"']
url = https://dvc.example.org/myproject
```

Daten und Verzeichnisse hinzufügen

Mit DVC könnt ihr Dateien, ML-Modelle, Verzeichnisse und Zwischenergebnisse mit Git speichern und versionieren, ohne dass der Dateiinhalt in Git eingecheckt werden muss:

```
$ dvc get https://github.com/iterative/dataset-registry get-started/data.xml \
    -o data/data.xml
$ dvc add data/data.xml
```

Dies fügt die Datei `data/data.xml` in `data/.gitignore` hinzu und schreibt die Meta-Angaben in `data/data.xml.dvc`. Weitere Informationen zum Dateiformat der `*.dvc`-Datei erhaltet ihr unter DVC-File Format.

Um nun verschiedene Versionen eurer Projektdaten mit Git verwalten zu können, müsst ihr jedoch nur die DVC-Datei hinzufügen:

```
$ git add data/.gitignore data/fortune500.csv.dvc
$ git commit -m "Add raw data to project"
```

Daten speichern und abrufen

Die Daten können vom Arbeitsverzeichnis eures Git-Repository auf den entfernten Speicherplatz kopiert werden mit

```
$ dvc push
```

Falls ihr aktuellere Daten abrufen wollt, könnt ihr dies mit

```
$ dvc pull
```

Importieren und Aktualisieren

Ihr könnt auch Daten und Modelle eines anderen Projekts importieren mit dem `dvc import`-Befehl, z.B.:

```
$ dvc import https://github.com/iterative/dataset-registry  get-started/data.xml
Importing 'get-started/data.xml (https://github.com/iterative/dataset-registry)' ->
↪'data.xml'
```

Dies lädt die Datei aus der dataset-registry in das aktuelle Arbeitsverzeichnis, fügt sie `.gitignore` hinzu und erstellt `data.xml.dvc`.

Mit `dvc update` können wir diese Datenquellen aktualisieren, bevor wir eine Pipeline reproduzieren, die von diesen Datenquellen abhängt, z.B.:

```
$ dvc update data.xml.dvc
Stage 'data.xml.dvc' didn't change.
Saving information to 'data.xml.dvc'.
```

Pipelines

Code und Daten verbinden

Befehle wie `dvc add`, `dvc push` und `dvc pull` können unabhängig von Änderungen im Git-Repository vorgenommen werden und bieten daher nur die Basis, um große Datenmengen und Modelle zu verwalten. Um tatsächlich reproduzierbare Ergebnisse zu erzielen, müssen Code und Daten miteinander verbunden werden.

Mit `dvc run` könnt ihr einzelne Verarbeitungsstufen erstellen, wobei jede Stufe durch eine, mit Git verwaltete, Quellcode-Datei sowie weiteren Abhängigkeiten und Ausgabedaten beschrieben wird. Alle Stufen zusammen bilden dann die DVC-Pipeline.

In unserem Beispiel dvc-example soll die erste Stufe die Daten in Trainings- und Testdaten aufteilen:

```
$ dvc run -n split -d src/split.py -d data/data.xml -o data/splitted \
    python src/split.py data/data.xml
```

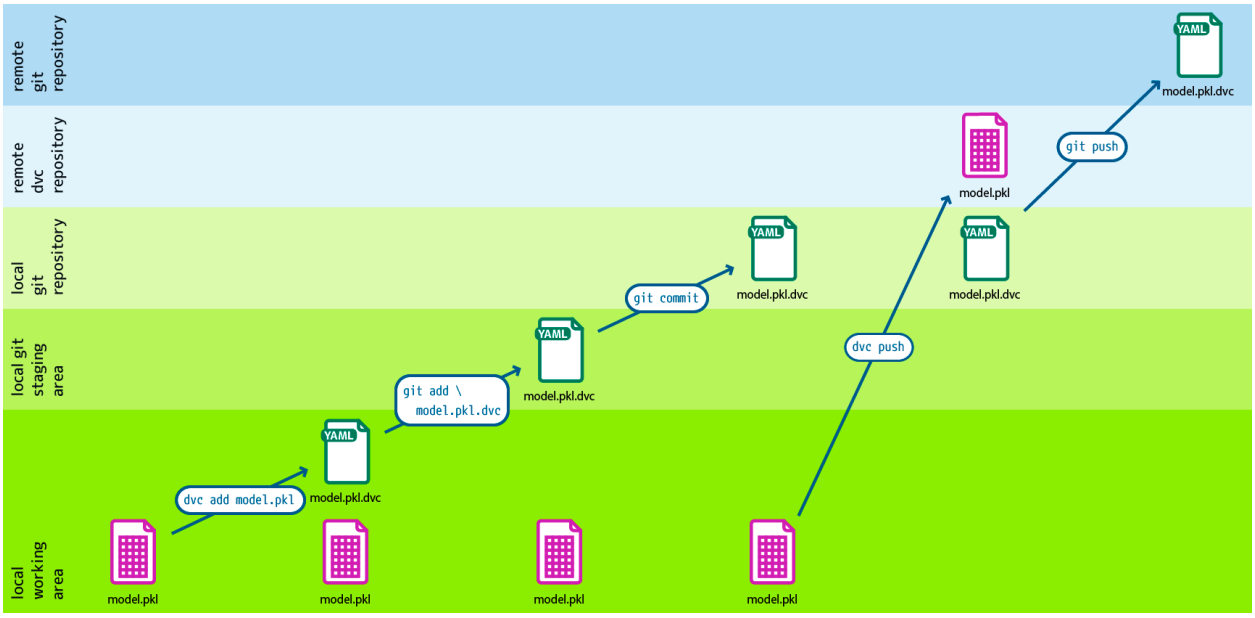


Abb. 4: Design: André Henze, Berlin

-n

gibt den Namen der Verarbeitungsstufe an.

-d

gibt Abhängigkeiten (*dependencies*) für das reproduzierbare Kommando an.

Wenn zum Reproduzieren der Ergebnisse beim nächsten Mal `dvc repo` aufgerufen wird, überprüft DVC diese Abhängigkeiten und entscheidet, ob diese auf dem aktuellen Stand sind oder erneut ausgeführt werden müssen, um aktuellere Ergebnisse zu erhalten.

-o

gibt die Ausgabedatei oder das Ausgabeverzeichnis an.

In unserem Fall sollte sich der Arbeitsbereich geändert haben in:

```
  .
  ├── data
  │   ├── data.xml
  │   ├── data.xml.dvc
+ │   └── splitted
+ │       ├── test.tsv
+ │       └── train.tsv
+ ├── dvc.lock
+ ├── dvc.yaml
  ├── requirements.txt
  └── src
      └── split.py
```

Die generierte `dvc.yaml`-Datei sieht dann z.B. folgendermaßen aus:

```
stages:
  split:
    cmd: pipenv run python src/split.py data/data.xml
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    deps:
    - data/data.xml
    - src/split.py
    outs:
    - data/splitted
```

Da die Daten im Ausgabeverzeichnis nie mit Git versioniert werden sollten, hat `dvc run` dies auch bereits die `data/.gitignore`-Datei geschrieben:

```
  /data.xml
+ /splitted
```

Anschließend müssen die geänderten Daten nur noch in Git bzw. DVC übernommen werden:

```
$ git add data/.gitignore dvc.yaml
$ git commit -m "Create split stage"
$ dvc push
```

Werden nun mehrere Phasen mit `dvc run` erstellt, wobei die Ausgabe eines Kommandos als Abhängigkeit eines anderen angegeben wird, entsteht eine DVC Pipeline.

Parametrisierung

In der nächsten Phase unseres Beispiels parametrisieren wir die Verarbeitung und erstellen hierfür die Datei `params.yaml` mit folgendem Inhalt:

```
max_features: 6000
ngram_range:
  lo: 1
  hi: 2
```

Damit die Parameter gelesen werden, wird dem `dvc run`-Befehl noch `-p <filename>:<params_list>` hinzugefügt, also in unserem Beispiel:

```
$ dvc run -n featurize -d src/featurization.py -d data/splitted \
    -p params.yaml:max_features,ngram_range.lo,ngram_range.hi -o data/features \
    python src/featurization.py data/splitted data/features
```

Dies ergänzt die `dvc.yaml`-Datei um:

```
featurize:
  cmd: python src/featurization.py data/splitted data/features
  deps:
  - data/splitted
  - src/featurization.py
  params:
  - max_features
  - ngram_range.lo
  - ngram_range.hi
  outs:
  - data/features
```

Damit diese Phase wiederholt werden kann, werden die MD5-Hashwerte und Parameterwerte in der `dvc.lock`-Datei gespeichert:

```
featurize:
  cmd: python src/featurization.py data/splitted data/features
  deps:
  - path: data/splitted
    md5: 1ce9051bf386e57c03fe779d476d93e7.dir
  - path: src/featurization.py
    md5: a56570e715e39134adb4fdc779296373
  params:
    params.yaml:
      max_features: 1000
      ngram_range.hi: 2
      ngram_range.lo: 1
```

Schließlich müssen noch `dvc.lock`, `dvc.yaml` und `data/.gitignore` im Git-Repository aktualisiert werden:

```
$ git add dvc.lock dvc.yaml data/.gitignore
```

Siehe auch:

- dvc params

Versuchsmetriken

Mit dem dvc metrics-Kommando ist DVC auch ein Framework zum Erfassen und Vergleichen der Performance von Experimenten.

evaluate.py berechnet den AUC (**A** rea **U** nder the **C** urve, deutsch Fläche unter der Kurve)-Wert. Dabei verwendet es den Testdatensatz, ließt die Features aus `features/test.pkl` und erstellt die Metrikdatei `auc.metric`. Sie kann DVC als Metrik kenntlich gemacht werden mit der `-M`-Option von dvc run, in unserem Beispiel also mit:

```
$ dvc run -n evaluate -d src/evaluate.py -d model.pkl -d data/features \
    -M auc.json python src/evaluate.py model.pkl data/features auc.json
```

```
evaluate:
  cmd: python src/evaluate.py model.pkl data/features auc.json
  deps:
  - data/features
  - model.pkl
  - src/evaluate.py
  metrics:
  - auc.json:
      cache: false
```

Mit `dvc metrics show` lassen sich Experimente dann auch über verschiedene Branches und Tags hinweg vergleichen:

```
$ dvc metrics show
       auc.json: 0.514172
```

Um nun unsere erste Version der DVC-Pipeline abzuschließen, fügen wir die Dateien und ein Tag dem Git-Repository hinzu:

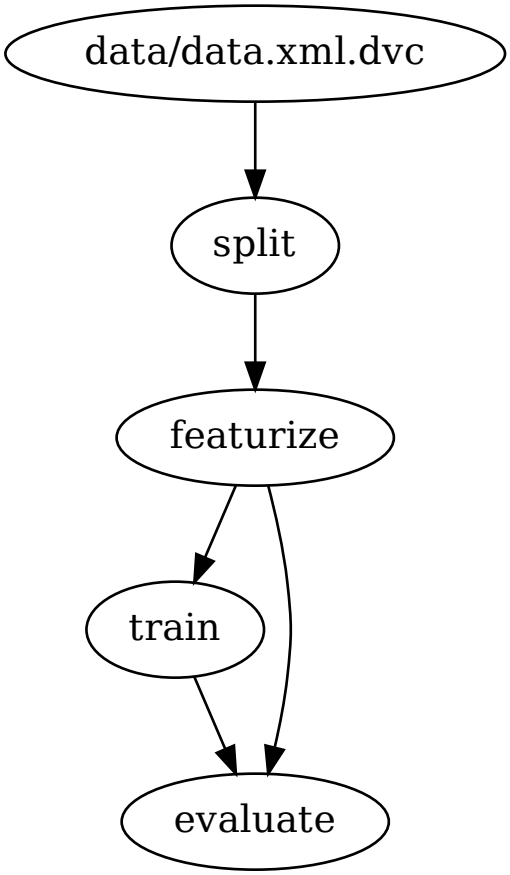
```
$ git add dvc.yaml dvc.lock auc.json
$ git commit -m 'Add stage ‹evaluate›'
$ git tag -a 0.1.0 -m "Initial pipeline version 0.1.0"
```

Pipelines anzeigen

Solche Datenpipelines lassen sich anzeigen oder als Abhängigkeitsgraph darstellen mit `dvc dag`:

```
$ dvc dag

    +------------------+
    | data/data.xml.dvc |
    +------------------+
              *
              *
              *
          +-------+
          | split |
          +-------+
              *
              *
              *
        +-----------+
        | featurize |
        +-----------+
         **        **
       **            *
      *               **
+-------+               *
| train |              **
+-------+             *
         **        **
           **    **
             *  *
        +----------+
        | evaluate |
        +----------+

data/data.xml.dvc
prepare.dvc
featurize.dvc
train.dvc
evaluate.dvc
```

- Mit `dvc dag --dot` kann auch eine `.dot`-Datei für Graphviz generiert werden:



Reproduzieren

Um die Ergebnisse eines Projekts zu reproduzieren, clonen wir den Code und rufen anschließend die mit DVC verwalteten Daten ab:

```
$ git clone https://github.com/veit/dvc-example.git
$ cd dvc-example
$ dvc pull -TR
A       data/data.xml
1 file added
$ ls data/
data.xml    data.xml.dvc
```

Anschließend könnt ihr die Ergebnisse einfach reproduzieren mit dvc repro:

```
$ dvc repro
Verifying data sources in stage: 'data/data.xml.dvc'
Stage 'split' didn't change, skipping
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Stage 'featurize' didn't change, skipping
Stage 'train' didn't change, skipping
Stage 'evaluate' didn't change, skipping
```

Ihr könnt nun z.B. Parameter in der `params.yaml`-Datei ändern und anschließend die Pipeline erneut durchlaufen:

```
$ dvc repro
Stage 'data/data.xml.dvc' didn't change, skipping
Stage 'split' didn't change, skipping
Running stage 'featurize' with command:
    python src/featurization.py data/splitted data/features
...
Stage 'train' didn't change, skipping
Stage 'evaluate' didn't change, skipping
To track the changes with git, run:
    git add dvc.lock
```

In unserem Fall hatte die Änderung der Parameter also keinen Einfluss auf das Ergebnis. Beachtet dabei jedoch, dass DVC Änderungen an Abhängigkeiten und Ausgaben über md5-Hashwerte erkennt, die in der `dvc.lock`-Datei gespeichert sind.

Vim- und IDE-Integration

Vim

Um DVC-Dateien in Vim als YAML zu erkennen, solltet ihr Folgendes in `~/.vimrc` hinzufügen:

```
" DVC
autocmd! BufNewFile,BufRead Dvcfile,*.dvc setfiletype yaml
```

Visual Studio Code

Für Visual Studio Code gibt es eine Erweiterung für DVC, die aus dem Visual Studio Marketplace heruntergeladen werden kann.

IntelliJ IDEs

intellij-dvc ist ein Plugin für IntelliJ IDEs einschließlich PyCharm, IntelliJ IDEA und CLion. Es kann aus dem JetBrains Plugins-Repository heruntergeladen werden.

FastDS

FastDS ist ein Open-Source-Tool, das *Git* und *DVC* kombiniert, um eine einfache Versionierung von Code und Daten zu ermöglichen.

Installation

FastDS kann einfach installiert werden mit:

```
$ pipenv install fastds
```

Einführung

Schon das Erstellen des initialen Repositories wird deutlich vereinfacht:

```
$ git init
$ dvc init
$ git add .
$ dvc add data/data.xml
$ git add data/.gitignore data/data.xml.dvc
$ git commit -m "Initial commit"
$ dvc push -r origin
$ git push origin
```

wird zu:

```
$ fds init
$ fds add .
$ fds save -m "Initial commit"
```

FastDS kürzt Git- und DVC-Befehle ab, um Eingabefehler zu minimieren und sich wiederholende Aufgaben zu automatisieren:

init
: initialisiert sowohl das Git- wie auch das DVC-Repository.

status
: gibt den Status beider Repositories zurück.

add
: fügt Dateien dem Git- oder DVC-Repository hinzu.

commit
: übergibt Änderungen an das Git- oder DVC-Repository.

clone
: klont das Git-Repository und holt Daten vom entfernten DVC-Repository.

push
: überträgt Daten an die entfernten Git- und DVC-Repositories.

save
: fügt Änderungen in das Projekt ein und überträgt diese mit einer Commit-Nachricht an die entfernten Git- und DVC-Repositories.

7.3 Umgebungen reproduzieren

Reproduzierbare und sichere Python-Umgebungen sind nur schwer zu gewährleisten. Mit dem Python Paketmanager pip, würde das so aussehen:

```
$ python -m pip install --no-deps --require-hashes ----only-binary=:all:
```

Dezidierte Umgebungen (z.B. mit *Pipenv*, und *Spack* vereinfachen dies, wenn ihr die Dateien mit den Spezifikationen speichert, also z.B. mit `Pipfile`, `Pipfile.lock`, `package-lock.json` ETC (et cetera). Auf diese Weise könnt ihr und andere eure Umgebungen reproduzieren.

7.3.1 Spack

Modellierungs- und Simulationsumgebungen sind sehr heterogen. Spack unterstützt daher viele verschiedene Produktionsumgebungen:

- 7 verschiedene Compiler: Intel, GCC, Clang, PGI, ...
- Auflösen von Abhängigkeiten
- Auflösen verschiedener Versionen von Abhängigkeiten

Siehe auch:

- Docs
- Tutorial
- Spack Encyclopedia
- GitHub

Bisherige Systeme

Sie bieten meist keine Unterstützung für kombinatorische Versionierung.

- Traditionelle Binärpaketmanager wie RPM, yum, APT, yast, etc.
  - sind konzipiert um einen einzelnen Software-Stack zu verwalten
  - installieren eine Version eines Pakets
  - üblicherweise problemlose Upgrades auf einen stabilen, gut getesteten Stack
- Port-Systeme
  - BSD Ports, portage, NixOS, Macports, Homebrew, etc.
  - meist kaum Unterstützung für Builds, die parametrisiert sind durch Compiler oder abhängige Versionen
- Virtuelle Maschinen und Linux-Container
  - Container erlauben die Erstellung unterschiedlicher Umgebungen für unterschiedliche Anwendungen
  - Sie lösen jedoch nicht das Build-Problem für das Image
  - Performance, Security und Upgrades werden bei vielen unterschiedlichen Builds sehr aufwändig.

Spack-Installation

Anforderungen

- Interpreter für Spack:
- Software erstellen
  - C/C++ Compiler
  - `make`, `patch` und `bash`
- Archive erstellen und extrahieren
  - `tar`, `gzip` und `bzip`
- Verwalten von Software-Repositories
  - `git`
- Signieren und Verifizieren von Build-Caches
  - `gnupg2` für `gpg`-Subcommand

```
$ sudo apt install build-essential patch tar gzip bzip2 git gnupg2
```

```
$ xcode-select --install
$ brew install make bash gzip bzip2 git gnupg
$ brew link gnupg
```

Anschließend wird die Shell konfiguriert, indem z.B. für die Bash folgendes in die Bash-Konfiguration eingetragen wird:

```
$ source /usr/local/opt/modules/init/bash
```

Installation

```
$ git clone https://github.com/spack/spack.git
Cloning into 'spack'...
...
$ cd spack
$ git switch releases/v0.19
```

Shell konfigurieren

1. Zur Konfiguration des Bash-Environment wird folgendes in `~/.bashrc` eingetragen:

   ```
   export SPACK_ROOT=~/spack
   . $SPACK_ROOT/share/spack/setup-env.sh
   ```

2. Die geänderte Konfiguration wird nun übernommen mit

   ```
   $ source ~/.bashrc
   ```

Bootstrapping `clingo`

Spack uses clingo to resolve optimal versions and variants of dependencies when installing packages. To install clingo from pre-built binaries you can simply specify a package:

Spack benutzt clingo um optimale Versionen und Varianten von Abhängigkeiten bei der Installation von Paketen aufzulösen. Um clingo aus vorgefertigten Binärdateien zu installieren, könnt ihr einfach ein Paket angeben:

```
$ spack spec zlib
==> Bootstrapping clingo from pre-built binaries
==> Fetching https://mirror.spack.io/bootstrap/github-actions/v0.4/build_cache/linux-
↪centos7-x86_64-gcc-10.2.1-clingo-bootstrap-spack-idkenmhnscjlu5gjqhpcqa4h7o2a7aow.spec.
↪json
==> Fetching https://mirror.spack.io/bootstrap/github-actions/v0.4/build_cache/linux-
↪centos7-x86_64/gcc-10.2.1/clingo-bootstrap-spack/linux-centos7-x86_64-gcc-10.2.1-
↪clingo-bootstrap-spack-idkenmhnscjlu5gjqhpcqa4h7o2a7aow.spack
==> Installing "clingo-bootstrap@spack%gcc@10.2.1~docs~ipo+python+static_libstdcpp build_
↪type=Release arch=linux-centos7-x86_64" from a buildcache
Input spec
--------------------------------
zlib

Concretized
--------------------------------
zlib@1.2.13%gcc@11.3.0+optimize+pic+shared build_system=makefile arch=linux-ubuntu22.04-
↪sandybridge
```

Bemerkung: Um von vorgefertigten Binärdateien zu booten, benötigt Spack `patchelf` unter Linux oder `otool` unter macOS. Ansonsten baut Spack sie aus den Quellen und mit einem C++ Compiler.

Bootstrap store

Alle Werkzeuge, die Spack benötigt, werden in einem separaten Speicher installiert, der sich im Verzeichnis *$HOME/.spack* befindet. Die dort installierte Software kann abgefragt werden mit:

```
$ spack find --bootstrap
==> Warning: `spack find --bootstrap` is deprecated and will be removed in v0.19.
  Use `spack --bootstrap find` instead.
==> Showing internal bootstrap store at "/srv/jupyter/.spack/bootstrap/store"
-- linux-centos7-x86_64 / gcc@10.2.1 ----------------------------
bison@3.0.4  clingo-bootstrap@spack  python@3.10
==> 3 installed packages
```

Compiler-Konfiguration

```
$ spack compilers
==> Available compilers
-- gcc ubuntu22.04-x86_64 ---------------------------------------
gcc@11.3.0
```

Baut euren eigenen Compiler

```
$ spack install gcc
...
==> gcc: Successfully installed gcc-11.2.0-azhiay4ugfrs634hqlez7u3f2li3wvzd
  Fetch: 12.09s.  Build: 2h 8m 13.92s.  Total: 2h 8m 26.01s.
[+] /Users/veit/spack/opt/spack/darwin-bigsur-cannonlake/apple-clang-13.0.0/gcc-11.2.0-
↪azhiay4ugfrs634hqlez7u3f2li3wvzd
```

Allerdings findet Spack den Compiler zunächst nicht:

```
$ $ spack compilers
==> Available compilers
-- apple-clang bigsur-x86_64 ------------------------------------
apple-clang@13.0.0
```

Ihr könnt ihn jedoch mit `spack compiler find` hinzufügen:

```
$ spack compiler find /srv/jupyter/spack/opt/spack/linux-ubuntu22.04-sandybridge/gcc-11.
↪3.0/gcc-12.2.0-gbaw464qxjuz6i3uud42cd5mb4xujxia/
==> Added 1 new compiler to /srv/jupyter/.spack/linux/compilers.yaml
    gcc@12.2.0
==> Compilers are defined in the following files:
    /srv/jupyter/.spack/linux/compilers.yaml
```

```
$ spack compilers
==> Available compilers
-- gcc ubuntu22.04-x86_64 ---------------------------------------
gcc@12.2.0  gcc@11.3.0
```

Wenn ihr die Standard- und Site-Einstellungen überschreiben möchtet, könnt ihr `$HOME/.spack/packages.yaml` ändern:

```
packages:
  all:
    compiler: [gcc@12.2.0]
```

GPG Signing

Spack unterstützt das Signieren und Verifizieren von Paketen mit GPG-Schlüsseln. Für Spack wird ein separater Schlüsselring verwendet, weswegen keine Schlüssel aus dem Home-Verzeichnis von Nutzern verfügbar sind.

Wenn Spack zum ersten Mal installiert wird, ist dieser Schlüsselring leer. Die in `/var/spack/gpg` gespeicherten Schlüssel sind die Standardschlüssel für eine Spack-Installation. Diese Schlüssel werden durch `spack gpg init` importiert. Dadurch werden die Standardschlüssel als vertrauenswürdige Schlüssel in den Schlüsselbund importiert.

Schlüsseln vertrauen

Zusätzliche Schlüssel können mit `spack gpg trust <keyfile>` dem Schlüsselring hinzugefügt werden. Sobald ein Schlüssel vertrauenswürdig ist, können Pakete, die vom Besitzer dieses Schlüssels signiert wurden, installiert werden.

Schlüssel erstellen

Ihr könnt auch eigene Schlüssel erstellen, um eure eigenen Pakete signieren zu können mit

```
$ spack gpg export <location> [<key>...]
```

Schlüssel auflisten

Die im Schlüsselbund verfügbaren Schlüssel können aufgelistet werden mit

```
$ spack gpg list
```

Schlüssel entfernen

Schlüssel können entfernt werden mit

```
$ spack gpg untrust <keyid>
```

Schlüssel-IDs können E-Mail-Adressen, Namen oder Fingerprints sein.

Kombinatorische Builds

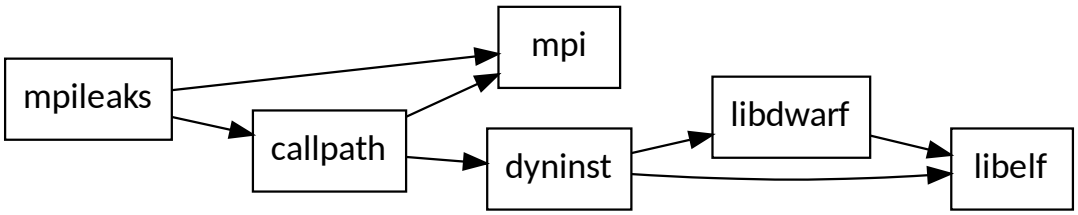
Environment modules

```
$ module avail
-------------------------- /opt/modules/modulefiles ----------------------------
acml-gnu/4.4 intel/12.0 mvapich2-pgi-ofa/1.7
acml-gnu_mp/4.4 intel/13.0 mvapich2-pgi-psm/1.7
acml-intel/4.4 intel/14.0(default) mvapich2-pgi-shmem/1.7...
$ module load intel/13.0
$ module load mvapich2-pgi-shmem/1.7
```

- Vorteile
  - tauschen verschiedene Versionen dynamisch in der Shell aus

  - abstrahieren viel von der Environment-Komplexität
- Nachteile
  - Benutzer müssen daran denken, mit welchen Versionen der Build durchgeführt wurde
  - Es ist einfach, das falsche Modul zu laden und einen Build fehlschlagen zu lassen

Dependency DAG



Installationslayout

```
$ tree /Users/veit/spack/opt/spack/
/Users/veit/spack/opt/spack/
└── darwin-mojave-x86_64
    ├── clang-10.0.1-apple
    │   ├── autoconf-2.69-ymadj7a7gg52r76payi7jd7qu7qcuasp
    │   │   ├── bin
    │   │   │   ├── autoconf
    │   │   │   ├── autoheader
...
```

- Jeder eindeutige Abhängigkeitsgraph erhält eine einzigartige Konfiguration
- Jede Konfiguration ist in einem eindeutigen Verzeichnis installiert
  - Konfigurationen des gleichen Pakets koexistieren nebeneinander
- Der Hash-Wert eines gerichteten azyklischen Graphen wird angehängt
- Installierte Pakete finden automatisch ihre Abhängigkeiten
  - Spack bettet `RPATH` in Binärdateien ein
  - Es besteht keine Notwendigkeit, Module zu verwenden oder `LD_LIBRARY_PATH` zu setzen

`spack list` zeigt die verfügbaren Pakete:

```
$ spack list
==> 3250 packages.
abinit                                py-fiona
abyss                                 py-fiscalyear
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
accfft                                  py-flake8
...
```

Spack bietet eine `spec`-Syntax zum Beschreiben benutzerdefinierter DAGs:

- ohne Einschränkungen

```
$ spack install mpileaks
```

- `@`: Benutzerdefinierte Version

```
$ spack install mpileaks@3.3
```

- `%`: Benutzerdefinierter Compiler

```
$ spack install mpileaks@3.3 %gcc@4.7.3
```

- `+/-`: Build-Option

```
$ spack install mpileaks@3.3 %gcc@4.7.3 +threads
```

- `=`: Cross-compile

```
$ spack install mpileaks@3.3 =bgq
```

- `^`: Version von Abhängigkeiten

```
$ spack install mpileaks %intel@12.1 ^libelf@0.8.12
```

- Spack stellt eine Konfiguration jeder Bibliothek pro DAG sicher
  - gewährleistet die Konsistenz des Application Binary Interface (ABI)
  - Der Benutzer muss die DAG-Struktur nicht kennen, sondern nur die Namen der abhängigen Bibliotheken
- Spack kann sicherstellen, dass Builds den gleichen Compiler verwenden
- Es können auch verschiedene Compiler für verschiedene Bibliotheken eines DAG angegeben werden
- Spack kann auch ABI-inkompatible, versionierte Schnittstellen wie z.B. das Message Passing Interface (MPI) bereitstellen
- So kann z.B. `mpi` auf unterschiedliche Weise erstellt werden:

```
$ spack install mpileaks ^mvapich@1.9
$ spack install mpileaks ^openmpi@1.4
```

- Alternativ kann Spack auch selbst das passende Build wählen, sofern nur das MPI 2-Interface implementiert wird:

```
$ spack install mpileaks ^mpi@2
```

- Spack-Pakete sind einfache Python-Skripte:

```
from spack import *
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
class Dyninst(Package):
    """API for dynamic binary instrumentation."""

    homepage = "https://paradyn.org"

    version(
        "8.2.1",
        "abf60b7faabe7a2e",
        url="http://www.paradyn.org/release8.2/DyninstAPI-8.2.1.tgz",
    )
    version(
        "8.1.2",
        "bf03b33375afa66f",
        url="http://www.paradyn.org/release8.1.2/DyninstAPI-8.1.2.tgz",
    )
    version(
        "8.1.1",
        "d1a04e995b7aa709",
        url="http://www.paradyn.org/release8.1/DyninstAPI-8.1.1.tgz",
    )

    depends_on("libelf")
    depends_on("libdwarf")
    depends_on("boost@1.42:")

    def install(self, spec, prefix):
        libelf = spec["libelf"].prefix
        libdwarf = spec["libdwarf"].prefix

        with working_dir("spack-build", create=True):
            cmake(
                "..",
                "-DBoost_INCLUDE_DIR=%s" % spec["boost"].prefix.include,
                "-DBoost_LIBRARY_DIR=%s" % spec["boost"].prefix.lib,
                "-DBoost_NO_SYSTEM_PATHS=TRUE" * std_cmake_args,
            )
            make()
            make("install")

    @when("@:8.1")
    def install(self, spec, prefix):
        configure("--prefix=" + prefix)
        make()
        make("install")
```

- Abhängigkeiten in Spack können optional sein:
  - Ihr könnt *named variants* definieren, wie z.B. in `~/spack/var/spack/repos/builtin/packages/vim/package.py`:

```
class Vim(AutotoolsPackage):
    ...
    variant("python", default=False, description="build with Python")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    depends_on("python", when="+python")

    variant("ruby", default=False, description="build with Ruby")
    depends_on("ruby", when="+ruby")
```

  - ... und zum Installieren verwenden:

```
$ spack install vim +python
$ spack install vim -python
```

  - Abhängig von anderen Bedingungen können Abhängigkeiten optional gelten, z.B. gcc-Abhängigkeit von mpc ab Version 4.5:

```
depends_on("mpc", when="@4.5:")
```

- DAGs sind nicht immer vollständig, bevor sie konkretisiert werden. Konkretisierungen füllen die fehlenden Konfigurationsdetails aus, wenn ihr sie nicht explizit benennt:

  1. Normalisierung

```
$ spack install mpileaks ^callpath@1.0+debug ^libelf@0.8.11
```

  2. Konkretisierung

     Die detaillierte Herkunft wird mit dem installierten Paket in `spec.yaml` gespeichert:

```
spec:
- mpileaks:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies:
    adept-utils: kszrtkpbzac3ss2ixcjkcorlaybnptp4
    callpath: bah5f4h4d2n47mgycej2mtrnrivvxy77
    mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
  hash: 33hjjhxi7p6gyzn5ptgyes7sghyprujh
  variants: {}
  version: '1.0'
- adept-utils:
  arch: linux-x86_64
  compiler:
    name: gcc
    version: 4.9.2
  dependencies:
    boost: teesjv7ehpe5ksspjim5dk43a7qnowlq
    mpich: aa4ar6ifj23yijqmdabeakpejcli72t3
  hash: kszrtkpbzac3ss2ixcjkcorlaybnptp4
  variants: {}
  version: 1.0.1
- boost:
  arch: linux-x86_64
  compiler:
    name: gcc
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    version: 4.9.2
  dependencies: {}
  hash: teesjv7ehpe5ksspjim5dk43a7qnowlq
  variants: {}
  version: 1.59.0
...
```

1. Wenn unspezifiziert, werden bei der Konkretisierung Werte basierend auf den Nutzereinstellungen gewählt.
2. Bei der Konkretisierung werden neue Abhängigkeiten unter Berücksichtigung der Constraints hinzugefügt.
3. Beim aktuellen Algorithmus kann nicht zurückverfolgt werden, warum eine Entscheidung getroffen wurde.
4. Zukünftig soll es einen *Full constraint solver* geben.

Vorteile der Build-Automatisierung

- Spack erleichtert Teams, ihren Code zu teilen
  - Rezepte für gebräuchliche Bibliotheken
  - reduzieren den Aufwand für reproduzierbare Builds
  - und erleichtert damit das Teilen von Builds.
- Patches erlauben eine schnelle Bereitstellung von Bugfixes
  - Anwendungsentwickler, die eine Bibliothek nutzen, haben häufig keine Schreibrechte auf deren Repositories.
  - Entwickler von Bibliotheken können Probleme evtl. nicht so schnell beheben wie gewünscht.
  - Mit Spack können Anwendungsentwickler schnell Korrekturen vornehmen und Änderungen rückgängig machen.
- Python erlaubt die schnelle Übernahme durch Entwicklungsteams.
  - Viele Anwendungsentwickler kennen Python bereits.
  - Die `yaml`-Syntax der Specs sind ausdrucksstark.

Use Case 1: Verwalten kombinatorischer Installationen

Anzeige aller installierten Konfigurationen

```
$ spack find
==> 103 installed packages.
-- linux-x86_64 / gcc@4.8.2 ---------------------------------
gdk-pixbuf@2.31.2   libpng@1.6.16        otf2@1.4            qhull@1.0
adept-utils@1.0.1   boost@1.55.0         cmake@5.6-special   libdwarf@20130729   mpich@3.
↪0.4
adept-utils@1.0.1   cmake@5.6 dyninst@8.1.2 libelf@0.8.13 openmpi@1.8.2
-- linux-x86_64 / intel@14.0.2 ------------------------------
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
hwloc@1.9           mpich@3.0.4          starpu@1.1.4
-- linux-x86_64 / intel@15.0.0 -----------------------------
adept-utils@1.0.1   boost@1.55.0         libdwarf@20130729   libelf@0.8.13       mpich@3.
↪0.4
-- linux-x86_64 / intel@15.0.1 -----------------------------
adept-utils@1.0.1   callpath@1.0.2       libdwarf@20130729   mpich@3.0.4
boost@1.55.0        hwloc@1.9            libelf@0.8.13       starpu@1.1.4
```

- `spack find` zeigt alle installierten Konfigurationen
- Dabei kann es auch verschiedene Versionen desselben Pakets geben
- Pakete werden differenziert zwischen Architektur und Compiler
- Spack generiert ebenfalls `modulefiles`, diese müssen jedoch nicht genutzt werden

Spack-Syntax zum Einschränken der Anfragen

```
$ spack find mpich
==> 5 installed packages.
-- linux-x86_64 / gcc@4.4.7 --------------------------------
mpich@3.0.4
-- linux-x86_64 / gcc@4.8.2 --------------------------------
mpich@3.0.4
-- linux-x86_64 / intel@14.0.2 -----------------------------
mpich@3.0.4
```

```
$ spack find libelf %intel
-- linux-x86_64 / intel@15.0.0 ------
libelf@0.8.13
-- linux-x86_64 / intel@15.0.1 ------
libelf@0.8.13
```

```
$ spack find libelf %intel@15.0.1
-- linux-x86_64 / intel@15.0.1 ------
libelf@0.8.13
```

Spack-Syntax zum Anzeigen der Abhängigkeiten

```
$ spack find callpath
==> 2 installed packages.
-- linux-x86_64 / clang@3.4 --------    -- linux-x86_64 / gcc@4.9.2 -------------
callpath@1.0.2                         callpath@1.0.2
```

```
$ spack find -dl callpath
==> 2 installed packages.
-- linux-x86_64 / clang@3.4 -----------    -- linux-x86_64 / gcc@4.9.2 -----------
xv2clz2    callpath@1.0.2                  udltshs callpath@1.0.2
ckjazss        ^adept-utils@1.0.1          rfsu7fb ^adept-utils@1.0.1
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3ws43m4        ^boost@1.59.0              ybet64y ^boost@1.55.0
ft7znm6        ^mpich@3.1.4               aa4ar6i ^mpich@3.1.4
qqnuet3        ^dyninst@8.2.1             tmnnge5 ^dyninst@8.2.1
3ws43m4        ^boost@1.59.0              ybet64y ^boost@1.55.0
g65rdud        ^libdwarf@20130729         g2mxrl2 ^libdwarf@20130729
cj5p5fk        ^libelf@0.8.13             ynpai3j ^libelf@0.8.13
cj5p5fk        ^libelf@0.8.13             ynpai3j ^libelf@0.8.13
g65rdud        ^libdwarf@20130729         g2mxrl2 ^libdwarf@20130729
cj5p5fk        ^libelf@0.8.13             ynpai3j ^libelf@0.8.13
cj5p5fk        ^libelf@0.8.13             ynpai3j ^libelf@0.8.13
ft7znm6        ^mpich@3.1.4               aa4ar6i ^mpich@3.1.4
```

Use Case 2: Python und andere interpretierte Sprachen

```
$ spack install python@2.7.10
==> Building python.
==> Successfully installed python.
  Fetch: 5.01s. Build: 97.16s. Total: 103.17s.
[+] /srv/jupyterhub/spack/opt/spack/linux-x86_64/gcc-4.9.2/python-2.7.10-y2zr767
$ spack extensions python@2.7.10
==> python@2.7.10%gcc@4.9.2=linux-x86_64-y2zr767
==> 49 extensions:
geos            py-h5py         py-numpy        py-pypar            py-setuptools
libxml2         py-ipython      py-pandas       py-pyparsing        py-shiboken
py-basemap      py-libxml2      py-pexpect      py-pyqt             py-sip
py-biopython    py-lockfile     py-pil          py-pyside           py-six
py-cffi         py-mako         py-pmw          py-python-daemon    py-sphinx
py-cython       py-matplotlib   py-pychecker    py-pytz             py-sympy
py-dateutil     py-mock         py-pycparser    py-rpy2             py-virtualenv
py-epydoc       py-mpi4py       py-pyelftools   py-scientificpython py-yapf
py-genders      py-mx           py-pygments     py-scikit-learn     thrift
py-gnuplot      py-nose         py-pylint       py-scipy
==> 3 installed:
-- linux-x86_64 / gcc@4.9.2 --------------------------------
py-nose@1.3.6   py-numpy@1.9.2  py-setuptools@18.1
==> None currently activated.
```

```
$ spack activate py-numpy
==> Activated extension py-setuptools-18.1-gcc-4.9.2-ru7w3lx
==> Activated extension py-nose-1.3.6-gcc-4.9.2-vudjpwc
==> Activated extension py-numpy-1.9.2-gcc@4.9.2-45hjazt
```

```
$ spack deactivate -a py-numpy
==> Deactivated extension py-numpy-1.9.2-gcc@4.9.2-45hjazt
==> Deactivated extension py-nose-1.3.6-gcc-4.9.2-vudjpwc
==> Deactivated extension py-setuptools-18.1-gcc-4.9.2-ru7w3lx
```

Zukünftige Features

- Lmod (Lua based module system)-Integration
- Auflösen externer Abhängigkeiten
- Benutzerdefinierte Compiler Flag Injection
- XML-Testergebnisse (JUnit)

Siehe auch:

Pull requests

Spack verwenden

Auflisten der verfügbaren Pakete

```
$ spack list
==> 3247 packages.
abinit                                py-fiona
abyss                                 py-fiscalyear
...
```

oder zum filtern nach bestimmten Paketen, z.B.

```
$ spack list numpy
==> 2 packages.
py-numpy  py-numpydoc
```

Auflisten der installierten Pakete

```
$ spack find
==> 17 installed packages
-- darwin-mojave-x86_64 / clang@10.0.1-apple ---------------------
bzip2@1.0.8    libffi@3.2.1    perl@5.26.2           python@3.7.4   zlib@1.2.11
diffutils@3.7  ncurses@6.1     pkgconf@1.6.1         readline@7.0
expat@2.2.5    openblas@0.3.6  py-numpy@1.16.4       sqlite@3.28.0
gdbm@1.18.1    openssl@1.1.1b  py-setuptools@41.0.1  xz@5.2.4
```

spack info

```
$ spack info py-numpy
PythonPackage:   py-numpy

Description:
    NumPy is the fundamental package for scientific computing with Python.
    It contains among other things: a powerful N-dimensional array object,
    sophisticated (broadcasting) functions, tools for integrating C/C++ and
    Fortran code, and useful linear algebra, Fourier transform, and random
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    number capabilities

Homepage: http://www.numpy.org/

Tags:
    None

Preferred version:
    1.16.4    https://pypi.io/packages/source/n/numpy/numpy-1.16.4.zip

Safe versions:
    1.16.4    https://pypi.io/packages/source/n/numpy/numpy-1.16.4.zip
    1.16.3    https://pypi.io/packages/source/n/numpy/numpy-1.16.3.zip
    1.16.2    https://pypi.io/packages/source/n/numpy/numpy-1.16.2.zip
    1.16.1    https://pypi.io/packages/source/n/numpy/numpy-1.16.1.zip
    1.16.0    https://pypi.io/packages/source/n/numpy/numpy-1.16.0.zip
    1.15.4    https://pypi.io/packages/source/n/numpy/numpy-1.15.4.zip
    1.15.3    https://pypi.io/packages/source/n/numpy/numpy-1.15.3.zip
    1.15.2    https://pypi.io/packages/source/n/numpy/numpy-1.15.2.zip
    1.15.1    https://pypi.io/packages/source/n/numpy/numpy-1.15.1.zip
    1.15.0    https://pypi.io/packages/source/n/numpy/numpy-1.15.0.zip
    1.14.6    https://pypi.io/packages/source/n/numpy/numpy-1.14.6.zip
    1.14.5    https://pypi.io/packages/source/n/numpy/numpy-1.14.5.zip
    1.14.4    https://pypi.io/packages/source/n/numpy/numpy-1.14.4.zip
    1.14.3    https://pypi.io/packages/source/n/numpy/numpy-1.14.3.zip
    1.14.2    https://pypi.io/packages/source/n/numpy/numpy-1.14.2.zip
    1.14.1    https://pypi.io/packages/source/n/numpy/numpy-1.14.1.zip
    1.14.0    https://pypi.io/packages/source/n/numpy/numpy-1.14.0.zip
    1.13.3    https://pypi.io/packages/source/n/numpy/numpy-1.13.3.zip
    1.13.1    https://pypi.io/packages/source/n/numpy/numpy-1.13.1.zip
    1.13.0    https://pypi.io/packages/source/n/numpy/numpy-1.13.0.zip
    1.12.1    https://pypi.io/packages/source/n/numpy/numpy-1.12.1.zip
    1.12.0    https://pypi.io/packages/source/n/numpy/numpy-1.12.0.zip
    1.11.3    https://pypi.io/packages/source/n/numpy/numpy-1.11.3.zip
    1.11.2    https://pypi.io/packages/source/n/numpy/numpy-1.11.2.zip
    1.11.1    https://pypi.io/packages/source/n/numpy/numpy-1.11.1.zip
    1.11.0    https://pypi.io/packages/source/n/numpy/numpy-1.11.0.zip
    1.10.4    https://pypi.io/packages/source/n/numpy/numpy-1.10.4.zip
    1.9.3     https://pypi.io/packages/source/n/numpy/numpy-1.9.3.zip
    1.9.2     https://pypi.io/packages/source/n/numpy/numpy-1.9.2.zip
    1.9.1     https://pypi.io/packages/source/n/numpy/numpy-1.9.1.zip

Variants:
    Name [Default]    Allowed values    Description


    blas [on]         True, False       Build with BLAS support
    lapack [on]       True, False       Build with LAPACK support

Installation Phases:
    build    install
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Build Dependencies:
    blas  lapack  py-setuptools  python

Link Dependencies:
    blas  lapack  python

Run Dependencies:
    python

Virtual Packages:
    None
```

spack version

`spack version` zeigt die verfügbaren Versionen an, z.B.

```
$ spack versions python
==> Safe versions (already checksummed):
  3.7.4  3.7.0  3.6.5  3.6.1  3.5.1   3.3.6   2.7.15  2.7.11
  3.7.3  3.6.8  3.6.4  3.6.0  3.5.0   3.2.6   2.7.14  2.7.10
  3.7.2  3.6.7  3.6.3  3.5.7  3.4.10  3.1.5   2.7.13  2.7.9
  3.7.1  3.6.6  3.6.2  3.5.2  3.4.3   2.7.16  2.7.12  2.7.8
==> Remote versions (not yet checksummed):
  3.8.0b2   3.6.9     3.5.7rc1  3.5.0a2    3.4.0     3.1.2      2.7    2.4.3
  3.8.0b1   3.6.8rc1  3.5.6rc1  3.5.0a1    3.3.7rc1  3.1.1      2.6.9  2.4.2
  ...
```

Installation bestimmter Pakete

z.B.

```
$ spack install python@3.7.4
```

oder um `py-numpy` für Python 3.7.4 zu installieren:

```
$ spack install py-numpy ^python@3.7.4
```

Anschließend kann die Installation überprüft werden mit

```
$ spack find --deps py-numpy
==> 1 installed package
-- darwin-mojave-x86_64 / clang@10.0.1-apple ---------------------
    py-numpy@1.16.4
        ^openblas@0.3.6
        ^python@3.7.4
            ^bzip2@1.0.8
            ^expat@2.2.5
            ^gdbm@1.18.1
                ^readline@7.0
                    ^ncurses@6.1
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            ^libffi@3.2.1
            ^openssl@1.1.1b
                ^zlib@1.2.11
            ^sqlite@3.28.0
            ^xz@5.2.4
```

Deinstallieren

```
$ spack uninstall py-numpy
```

oder

```
$ spack uninstall --dependents py-numpy
```

Extensions und Python-Support

Das Installationsmodell von Spack geht davon aus, dass jedes Paket in einem eigenen Installations-Präfix lebt. Module in interpretierten Sprachen wie Python werden typischerweise im `$prefix/lib/python-3.7/site-packages/` installiert, also z.B. `/Users/veit/spack/opt/spack/darwin-mojave-x86_64/clang-10.0.1-apple/py-numpy-1.16.4-45sqnufha2yprpx6rxyelsokky65ucdy/lib/python3.7/site-packages/numpy`. Es können jedoch auch Pakete verwendet werden, die in einem anderen Präfix installiert wurden. In Spack wird ein solches Paket als *Extension* bezeichnet.

Angenommen, Python wurde installiert mit

```
$ spack find python
==> 1 installed package
-- darwin-mojave-x86_64 / clang@10.0.1-apple ---------------------
python@3.7.4
```

so können *Extensions* gefunden werden mit

```
$ spack extensions python
==> python@3.7.4%clang@10.0.1-apple+bz2+ctypes+dbm+lzma~nis~optimizations␣
↪patches=210df3f28cde02a8135b58cc4168e70ab91dbf9097359d05938f1e2843875e57␣
↪+pic+pyexpat+pythoncmd+readline~shared+sqlite3+ssl~tix~tkinter~ucs4~uuid+zlib␣
↪arch=darwin-mojave-x86_64/jqlxzxp
==> 623 extensions:
adios2                                  py-munch
antlr                                   py-mx
...

==> 2 installed:
-- darwin-mojave-x86_64 / clang@10.0.1-apple ---------------------
py-numpy@1.16.4  py-setuptools@41.0.1

==> None activated.
```

`numpy` kann dem `PYTHONPATH` der aktuellen Shell hinzugefügt werden mit `load`:

```
$ spack load python
$ spack load py-numpy
$ python
Python 3.7.4 (default, Jul 28 2019, 20:00:06)
[Clang 10.0.1 (clang-1001.0.46.4)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy
>>>
```

Oft sollen jedoch bestimmte Pakete dauerhaft einer Python-Installation zur Verfügung stehen. Spack bietet hierfür `activate` an:

```
$ spack activate py-numpy
==> Activating extension py-numpy@1.16.4%clang@10.0.1-apple+blas+lapack arch=darwin-
↪mojave-x86_64/45sqnuf for python@3.7.4%clang@10.0.1-apple+bz2+ctypes+dbm+lzma~nis~
↪optimizations patches=210df3f28cde02a8135b58cc4168e70ab91dbf9097359d05938f1e2843875e57␣
↪+pic+pyexpat+pythoncmd+readline~shared+sqlite3+ssl~tix~tkinter~ucs4~uuid+zlib␣
↪arch=darwin-mojave-x86_64/jqlxzxp
```

Environments, `spack.yaml` und `spack.lock`

1. Erstellen einer virtuellen Umgebung:

   ```
   ==> Created environment 'python-311' in /srv/jupyter/spack/var/spack/environments/
   ↪python-311
   ==> You can activate this environment with:
   ==>   spack env activate python-311
   ```

   Alternativ kann sie auch an beliebigen anderen Orten gespeichert werden, z.B.:

   ```
   $ cd spackenvs/
   $ spack env create -d python-311
   ==> Created environment in /srv/jupyter/jupyter-tutorial/spackenvs/python-311
   ==> You can activate this environment with:
   ==>   spack env activate /srv/jupyter/jupyter-tutorial/spackenvs/python-311
   ```

2. Überprüfen der virtuellen Umgebung:

   ```
   $ spack env list
   ==> 1 environments
       python-311
   ```

3. Aktivieren der virtuellen Umgebung:

   ```
   $ spack env activate python-311
   ```

4. Überprüfen der Aktivierung:

   Wenn ihr eine Umgebung aktiviert habt, wird euch nur das angezeigt, was sich in der aktuellen Umgebung befindet. Das sollte unmittelbar nach der Aktivierung nichts sein:

   ```
   $ spack find
   ==> In environment python-311
   ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
==> No root specs
==> 0 installed packages
```

Und wenn ihr überprüfen möchtet, in welcher Umgebung ihr euch befindet, dann könnt ihr dies abfragen mit:

```
$ spack env status
==> In environment python-311
```

5. Schließlich könnt ihr die aktivierte Umgebung verlassen mit `spack env deactivate` oder kurz `despacktivate`.

```
$ despacktivate
$ spack env status
==> No active environment
```

Pakete installieren

```
$ spack env activate python-311
$ spack add python@3.11.0
$ spack install
==> Concretized python@3.11.0
 -   4nvposf  python@3.11.0%gcc@11.3.0+bz2+ctypes+dbm~debug+libxml2+lzma~nis~
↪optimizations+pic+pyexpat+pythoncmd+readline+shared+sqlite3+ssl~tix~tkinter~
↪ucs4+uuid+zlib build_system=generic patches=13fa8bf,b0615b2,f2fd060 arch=linux-
↪ubuntu22.04-sandybridge
 -   6fefzf3      ^bzip2@1.0.8%gcc@11.3.0~debug~pic+shared build_system=generic␣
↪arch=linux-ubuntu22.04-sandybridge
 -   27f7g74          ^diffutils@3.8%gcc@11.3.0 build_system=autotools arch=linux-
↪ubuntu22.04-sandybridge
...
==> python: Successfully installed python-3.11.0-4nvposf6bicf5ogp6nqacfo4dfvwm7zv
  Fetch: 5.19s.  Build: 3m 48.84s.  Total: 3m 54.03s.
[+] /srv/jupyter/spack/opt/spack/linux-ubuntu22.04-sandybridge/gcc-11.3.0/python-3.11.0-
↪4nvposf6bicf5ogp6nqacfo4dfvwm7zv
==> Updating view at /srv/jupyter/python-311/.spack-env/view
$ spack find
==> In environment /home/veit/python-311
==> Root specs
python@3.11.0

==> Installed packages
-- linux-ubuntu22.04-sandybridge / gcc@11.3.0 --------------------
berkeley-db@18.1.40                libiconv@1.16   readline@8.1.2
bzip2@1.0.8                        libmd@1.0.4     sqlite@3.39.4
ca-certificates-mozilla@2022-10-11 libxml2@2.10.1  tar@1.34
diffutils@3.8                      ncurses@6.3     util-linux-uuid@2.38.1
expat@2.4.8                        openssl@1.1.1s  xz@5.2.7
gdbm@1.23                          perl@5.36.0     zlib@1.2.13
gettext@0.21.1                     pigz@2.7        zstd@1.5.2
libbsd@0.11.5                      pkgconf@1.8.0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
libffi@3.4.2                          python@3.11.0
==> 25 installed packages
```

Mit `spack cd -e python-311` könnt ihr in dieses Verzeichnis wechseln, z.B.:

```
$ spack cd -e python-311
$ pwd
/srv/jupyter/spack/var/spack/environments/python-311
```

Dort befinden sich die beiden Dateien `spack.yaml` und `spack.lock`.

`spack.yaml`
: ist die Konfigurationsdatei für die virtuelle Umgebung. Sie wird in `~/spack/var/spack/environments/` beim Aufruf von `spack env create` erstellt.

  Alternativ zu `spack install` können in `spack.yaml` auch der `specs`-Liste `python@3.11.0`, `py-numpy` etc. hinzugefügt werden:

  ```
  specs: [python@3.11.0, ...]
  ```

`spack.lock`
: Mit `spack install` werden die Specs konkretisiert, in `spack.lock` geschrieben und installiert. Im Gegensatz zu `spack.yaml` ist `spack.lock` im `json`-Format geschrieben und enthält die notwendigen Informationen, um reproduzierbare Builds der Umgebung erstellen zu können:

  ```
  {
    "_meta": {
      "file-type": "spack-lockfile",
      "lockfile-version": 4,
      "specfile-version": 3
    },
    "roots": [
      {
        "hash": "4nvposf6bicf5ogp6nqacfo4dfvwm7zv",
        "spec": "python@3.11.0"
      }
    ],
    "concrete_specs": {
      "4nvposf6bicf5ogp6nqacfo4dfvwm7zv": {
        "name": "python",
        "version": "3.11.0",
        "arch": {
          "platform": "linux",
          "platform_os": "ubuntu22.04",
          "target": {
            "name": "sandybridge",
            "vendor": "GenuineIntel",
            "features": [
              "aes",
              "avx",
              ...
            ]
          }
        }
  ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    }
  }
}
```

Installation zusätzlicher Pakete

Zusätzliche Pakete können in der virtuellen Umgebung mit `spack add` und `spack install` installiert werden. Für Matplotlib sieht dies z.B. folgendermaßen aus:

```
$ spack add py-numpy
==> Adding py-numpy to environment /srv/jupyter/jupyter-tutorial/spackenvs/python-311
$ spack install
==> Concretized python@3.11.0
[+]  4nvposf  python@3.11.0%gcc@11.3.0+bz2+ctypes+dbm~debug+libxml2+lzma~nis~
↪optimizations+pic+pyexpat+pythoncmd+readline+shared+sqlite3+ssl~tix~tkinter~
↪ucs4+uuid+zlib build_system=generic patches=13fa8bf,b0615b2,f2fd060 arch=linux-
↪ubuntu22.04-sandybridge
[+]  6fefzf3      ^bzip2@1.0.8%gcc@11.3.0~debug~pic+shared build_system=generic␣
↪arch=linux-ubuntu22.04-sandybridge
[+]  27f7g74          ^diffutils@3.8%gcc@11.3.0 build_system=autotools arch=linux-
↪ubuntu22.04-sandybridge
...
==> Installing environment /srv/jupyter/jupyter-tutorial/spackenvs/python-311
...
==> Successfully installed py-numpy
```

Bemerkung: Falls von diesem Spack-Environment bereits ein *Pipenv-Environment* abgeleitet wurde, muss dieses neu gebaut werden um das zusätzliche Spack-Paket zu erhalten:

```
$ pipenv install --python=/srv/jupyter/spack/var/spack/environments/python-311/.spack-
↪env/view/bin/python
Creating a virtualenv for this project...
Pipfile: /srv/jupyter/jupyter-tutorial/pipenvs/python-311/Pipfile
Using /srv/jupyter/spack/var/spack/environments/python-311/.spack-env/view/bin/python (3.
↪11.0) to create virtualenv...
 Creating virtual environment...Using base prefix '/srv/jupyter/jupyterhub/spackenvs/
↪python-374/.spack-env/view'
  creator Venv(dest=/srv/jupyter/.local/share/virtualenvs/python-311-aGnPz55z,␣
↪clear=False, no_vcs_ignore=False, global=False, describe=CPython3Posix)
  seeder FromAppData(download=False, pip=bundle, setuptools=bundle, wheel=bundle,␣
↪via=copy, app_data_dir=/srv/jupyter/.local/share/virtualenv)
    added seed packages: pip==22.3.1, setuptools==65.5.1, wheel==0.38.4
  activators BashActivator,CShellActivator,FishActivator,NushellActivator,
↪PowerShellActivator,PythonActivator

✓ Successfully created virtual environment!
Virtualenv location: /srv/jupyter/.local/share/virtualenvs/python-311-aGnPz55z
Creating a Pipfile for this project...
Pipfile.lock not found, creating...
Locking [packages] dependencies...
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Locking [dev-packages] dependencies...
Updated Pipfile.lock (a3aa656db1de341c375390e74afd03f09eb681fe6881c58a71a85d6e08d77619)!
Installing dependencies from Pipfile.lock (d77619)...
To activate this project's virtualenv, run pipenv shell.
Alternatively, run a command inside the virtualenv with pipenv run.
```

Anschließend kann die Installation überprüft werden mit:

```
$ pipenv run python
Python 3.11.0 (main, Nov 19 2022, 11:29:15) [GCC 12.2.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> import matplotlib.pyplot as plt
```

Konfiguration

`spack spec` spezifiziert die Abhängigkeiten bestimmter Pakete, z.B.:

```
$ spack spec py-matplotlib

Input spec
--------------------------------
py-matplotlib

Concretized
--------------------------------
py-matplotlib@3.6.2%gcc@11.3.0~animation~fonts~latex~movies backend=agg build_
↪system=python_pip arch=linux-ubuntu22.04-sandybridge
    ^freetype@2.11.1%gcc@11.3.0 build_system=autotools arch=linux-ubuntu22.04-sandybridge
        ^bzip2@1.0.8%gcc@11.3.0~debug~pic+shared build_system=generic arch=linux-
↪ubuntu22.04-sandybridge
            ^diffutils@3.8%gcc@11.3.0 build_system=autotools arch=linux-ubuntu22.04-
↪sandybridge
    ^libpng@1.6.37%gcc@11.3.0 build_system=autotools arch=linux-ubuntu22.04-sandybridge
    ...
```

Mit `spack config get` könnt ihr euch die Konfiguration einer bestimmten Umgebung anschauen:

```
$ spack config get
# This is a Spack Environment file.
#
# It describes a set of packages to be installed, along with
# configuration settings.
spack:
  # add package specs to the `specs` list
  specs: [python@3.11.0, py-numpy]
  view: true
  concretizer:
    unify: true
```

Mit `spack config edit` kann die Konfigurationsdatei `spack.yaml` editiert werden.

Bemerkung: Wenn in der Umgebung bereits Pakete installiert sind, sollten mit `spack concretize -f` alle Abhängigkeiten erneut spezifiziert werden.

Laden der Module

Mit `spack env loads -r` <env> werden alle Module mit ihren Abhängigkeiten geladen.

Bemerkung: Aktuell funktioniert dies jedoch nicht beim Laden der Module aus Environments, die nicht in `$SPACK_ROOT/var/environments` liegen.

Daher ersetzen wir das Verzeichnis `$SPACK_ROOT/var/environments` durch einen symbolischen Link:

```
$ rm $SPACK_ROOT/var/environments
$ cd $SPACK_ROOT/var/
$ ln -s /srv/jupyter/jupyter-tutorial/spackenvs environments
```

Siehe auch:

- Environments Tutorial

Spack Mirrors

Einige Maschinen haben möglicherweise keinen Internetzugang, um Pakete abzurufen. Sie benötigen dann ein lokales Repository mit Tarballs, aus denen sie ihre Dateien abrufen können. Spack unterstützt dies mit *Spack Mirrors*. Ein Mirror ist eine URL, die auf ein Verzeichnis im lokalen Dateisystem oder auf einem Server verweist und Tarballs für alle Pakete von Spack enthält.

Hier ist ein Beispiel für die Verzeichnisstruktur eines Mirror:

```
$ tree /path/to/mirror/
/path/to/mirror/
├── autoconf
│   └── autoconf-2.69.tar.gz
├── automake
│   └── automake-1.16.1.tar.gz
├── bzip2
│   └── bzip2-1.0.8.tar.gz
├── diffutils
│   └── diffutils-3.7.tar.xz
├── expat
│   └── expat-2.2.5.tar.bz2
├── gcc
│   └── gcc-9.1.0.tar.xz
...
```

spack mirror create

Ihr könnt mit dem Befehl `spack mirror create` einen Mirror erstellen, vorausgesetzt, ihr befindet euch auf einer Maschine, die auf das Internet zugreifen kann. Der Befehl durchläuft alle Pakete von Spack und lädt die gewünschten herunter.

spack mirror add

Sobald ihr einen Spiegel erstellt habt, müsst ihr Spack darüber informieren. Das ist relativ einfach. Ermittelt zunächst die URL eures Mirrors. Wenn es sich um ein Verzeichnis handelt, könnt ihr eine Datei-URL wie die folgende verwenden:

```
$ spack mirror add local_filesystem file://$HOME/spack-mirror
```

Reihenfolge der Mirrors

`spack mirror ad` fügt eine Zeile hinzu in `~/.spack/mirrors.yaml`:

```
mirrors:
  local_filesystem: file:///home/veit/spack-mirror
  remote_server: https://spack-mirror.cusy.io
```

Wenn ihr die Reihenfolge ändern möchtet, in der Mirrors nach Paketen durchsucht werden, könnt ihr diese Datei bearbeiten und die Abschnitte neu anordnen: Spack durchsucht diese von oben nach unten bis ein passender Eintrag gefunden wird.

Lokaler Standardcache

Spack erstellt einen Zwischenspeicher für Ressourcen, die im Rahmen von Installationen heruntergeladen werden. Dieser Cache ist ein gültiger Spack-Mirror: er verwendet dieselbe Verzeichnisstruktur und dasselbe Namensschema wie andere Spack-Mirror. Der Mirror wird lokal im Spack-Installationsverzeichnis verwaltet unter `~/spack/var/spack/cache/`.

7.3.2 Pipenv

Pipenv ist ein Python-Paketmanager. Er nutzt Pip zum Installieren von Python-Paketen, vereinfacht jedoch auch die Verwaltung und Pflege von Abhängigkeiten.

Installation

Dieser Abschnitt behandelt die Grundlagen zur Installation von Python-Paketen.

Voraussetzungen für die Installation von Paketen

Vor der Installation von Python-Paketen müssen einige Voraussetzungen erfüllt sein.

1. Stellt sicher, dass ihr die gewünschte Python-Version verwendet:

```
$ python3 --version
Python 3.10.6
```

Bemerkung: In iPython oder einem Jupyter Notebook könnt ihr die Version folgendermaßen herausbekommen:

```
In [1]: import sys


        sys.version_info
sys.version_info(major=3, minor=10, micro=6, releaselevel='final', serial=0)
```

Bemerkung: Falls ihr das System-Python eurer Linux-Distribution verwendet, solltet ihr zunächst eine virtuelle Umgebung mit Python 3 und Pip erstellen.

---

2. Stellt sicher, dass Pip installiert ist:

```
$ pip --version
pip 22.0.2 from /usr/lib/python3/dist-packages/pip (python 3.10)
```

    1. Falls Pip noch nicht installiert ist, könnt ihr es installieren mit:

```
$ sudo apt install python3-venv python3-pip
```

```
$ sudo apt install python-pip
```

Pipenv installieren

pipenv ist ein Abhängigkeitsmanager für Python-Projekte. Er nutzt Pip zum Installieren von Python-Paketen, er vereinfacht jedoch die Verwaltung von Abhängigkeiten. Pip kann zum Installieren von Pipenv verwendet werden, es sollte jedoch das `--user`-Flag verwendet werden, damit es nur für diesen Nutzer bereitsteht. Dadurch soll verhindert werden, dass versehentlich systemweite Pakete überschrieben werden:

```
$ python3 -m pip install --user pipenv
...
Successfully installed distlib-0.3.4 filelock-3.4.2 pipenv-2022.1.8 platformdirs-2.4.1␣
↪virtualenv-20.13.0 virtualenv-clone-0.5.7
```

Bemerkung: Wenn pipenv nach der Installation nicht in der Shell verfügbar ist, muss ggf. das `USER_BASE/bin`-Verzeichnis in `PATH` angegeben werden.

`USER_BASE` lässt sich ermitteln mit:

```
$ python3 -m site --user-base
/srv/jupyter/.local
```

Anschließend muss noch das `bin`-Verzeichnis angehängt und zu `PATH` hinzugefügt werden. Alternativ kann `PATH` dauerhaft gesetzt werden, indem `~/.profile` oder `~/.bash_profile` geändert werden, in meinem Fall also:

```
export PATH=/srv/jupyter/.local/bin:$PATH
```

Das Verzeichnis kann ermittelt werden mit `py -m site --user-site` und anschließend `site-packages` durch `Scripts` ersetzt werden. Dies ergibt dann z.B.:

```
C:\Users\veit\AppData\Roaming\Python38\Scripts
```

Um dauerhaft zur Verfügung zu stehen, kann dieser Pfad unter `PATH` im Control Panel eingetragen werden.

Siehe auch:

Weitere Informationen zur nutzerspezifischen Installation findet ihr in User Installs.

Virtuelle Umgebungen erstellen

Virtuelle Python-Umgebungen ermöglichen die Installation von Python-Paketen an einem isolierten Ort für eine bestimmte Anwendung, anstatt sie global zu installieren. Ihr habt also eure eigenen Installationsverzeichnisse und teilt keine Bibliotheken mit anderen virtuellen Umgebungen:

```
$ mkdir myproject
$ cd !$
cd myproject
$ pipenv install requests
Creating a virtualenv for this project...
...
Virtualenv location: /srv/jupyter/.local/share/virtualenvs/myproject-CZKj6mqJ
Installing requests...
Adding requests to Pipfile's [packages]...
...
```

Nutzung

Beispiel

Nachdem nun `requests` installiert ist, kann es verwendet werden.

1. Exemplarisch legen wir hierfür die Datei `main.py` mit folgendem Inhalt an:

```
import requests


response = requests.get("https://cusy.io")

print(response.status_code)
```

1. Anschließend kann das Skript ausgeführt werden mit:

```
$ pipenv run python main.py
```

2. Als Ergebnis des Aufrufs solltet ihr den HTTP-Status-Code `200` erhalten.

Die Verwendung von `pipenv run` stellt sicher, dass eure installierten Pakete für euer Skript verfügbar sind.

Alternativ könnt ihr euch auch eine neue Shell mit `pipenv shell` erstellen, mit der auf alle installierten Pakete zugegriffen werden kann:

```
$ pipenv shell
Launching subshell in virtual environment...
 . /srv/jupyter/.local/share/virtualenvs/myproject-CZKj6mqJ/bin/activate
```

Optionen

`-venv`
: gibt den Pfad zum Virtualenv an, üblicherweise in `~/.local/share/virtualenvs/`. Falls ihr jedoch ein Verzeichnis `myproject/.venv` angelegt habt, verwendet `pipenv` diesen Ordner um dort die zugehörige Python-Umgebung anzulegen.

`--py`
: gibt den Pfad zum Python-Interpreter an.

`--envs`
: gibt Optionen der Environment-Variablen aus.

  Für `PIPENV_DONT_LOAD_ENV`, `PIPENV_DONT_USE_PYENV` und `PIPENV_DOTENV_LOCATION` siehe *Umgebungsvariablen*.

  Wenn ihr diese Umgebungsvariablen pro Projekt festlegen möchtet, könnt ihr direnv verwenden.

  Beachtet auch, dass pip selbst Umgebungsvariablen unterstützt, falls ihr zusätzliche Anpassungen benötigt: Pip Environment Variables.

  Hier noch ein Beispiel:

  ```
  $ PIP_INSTALL_OPTION="-- -DCMAKE_BUILD_TYPE=Release" pipenv install -e .
  ```

  Weitere Informationen hierzu findet ihr unter Configuration With Environment Variables

`--three, --two, --python`
: verwendet Python 2 oder Python 3 oder ein spezifisches Python, zu dem der Pfad angegeben wird.

`--site-packages`
: aktiviert site packages für das virtual environment.

`--pypi-mirror`
: gibt einen PyPI-Mirror an. Der Standard ist der Python Package Index (PyPI).

  Ihr könnt jedoch auch eure eigenen Mirror angeben:

  - mit der Umgebungsvariablen `PIPENV_PYPI_MIRROR`
  - in der Kommandozeile, z.B. mit:

    ```
    $ pipenv install --pypi-mirror https://pypi.cusy.io/simple
    $ pipenv update --pypi-mirror https://pypi.cusy.io/simple
    ...
    ```

- oder im `pipfile`:

```
[[source]]
url = "https://pypi.python.org/simple"
verify_ssl = true
name = "pypi"

[[source]]
url = "https://pypi.cusy.io/simple"
verify_ssl = true
name = "cusy-mirror"

[dev-packages]

[packages]
requests = {version="*", index="cusy-mirror"}
maya = {version="*", index="pypi"}
records = "*"
```

Bemerkung: Wird ein privater Index verwendet, kommt es aktuell noch zu Problemen mit dem Hashing der Pakete.

Weitere Optionen findet ihr unter pipenv.

check

`pipenv check` prüft auf Sicherheitslücken und auf **PEP 508**-Marker im Pipfile. Hierzu verwendet es safety.

Beispiel:

```
$ pipenv install django==1.10.1
Installing django==1.10.1...
...
$ pipenv check
Checking PEP 508 requirements...
Passed!
Checking installed package safety...

33075: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django before 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3, when␣
↪settings.DEBUG is True, allow remote attackers to conduct DNS rebinding attacks by␣
↪leveraging failure to validate the HTTP Host header against settings.ALLOWED_HOSTS.

33076: django >=1.10,<1.10.3 resolved (1.10.1 installed)!
Django 1.8.x before 1.8.16, 1.9.x before 1.9.11, and 1.10.x before 1.10.3 use a␣
↪hardcoded password for a temporary database user created when running tests with an␣
↪Oracle database, which makes it easier for remote attackers to obtain access to the␣
↪database server by leveraging failure to manually specify a password in the database␣
↪settings TEST dictionary.

33300: django >=1.10,<1.10.7 resolved (1.10.1 installed)!
CVE-2017-7233: Open redirect and possible XSS attack via user-supplied numeric redirect␣
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪URLs
==========================================================================================

Django relies on user input in some cases  (e.g.
:func:`django.contrib.auth.views.login` and :doc:`i18n </topics/i18n/index>`)
to redirect the user to an "on success" URL. The security check for these
redirects (namely ``django.utils.http.is_safe_url()``) considered some numeric
URLs (e.g. ``http:999999999``) "safe" when they shouldn't be.

Also, if a developer relies on ``is_safe_url()`` to provide safe redirect
targets and puts such a URL into a link, they could suffer from an XSS attack.

CVE-2017-7234: Open redirect vulnerability in ``django.views.static.serve()``
============================================================================

A maliciously crafted URL to a Django site using the
:func:`~django.views.static.serve` view could redirect to any other domain. The
view no longer does any redirects as they don't provide any known, useful
functionality.

Note, however, that this view has always carried a warning that it is not
hardened for production use and should be used only as a development aid.
```

Bemerkung: Pipenv bettet hierfür einen API-Clientschlüssel von pyup.io ein, anstatt eine vollständige Kopie der CC BY-NC-SA 3.0 lizenzierten Datenbank aufzunehmen.

Um nun die vollständige Datenbank zu installieren könnt ihr diese auschecken mit:

```
$ pipenv install -e git+https://github.com/pyupio/safety-db.git#egg=safety-db
```

Um die lokale Datenbank zu verwenden, müsst ihr den Pfad zu dieser Datenbank angeben, in meinem Fall also:

```
$ pipenv check --db /Users/veit/.local/share/virtualenvs/myproject-9TTuTZjx/src/safety-
↪db/data
```

```
╒══════════════════════════════════════════════════════════════════════════════╕
│                                                                              │
│                                /$$$$$$            /$$                        │
│                               /$$__  $$          | $$                        │
│         /$$$$$$$  /$$$$$$ | $$  \__//$$$$$$  /$$$$$$   /$$   /$$           │
│        /$$_____/ |____  $$| $$$$   /$$__  $$|_  $$_/  | $$  | $$           │
│       |  $$$$$$   /$$$$$$$| $$_/  | $$$$$$$$  | $$    | $$  | $$           │
│        \____  $$ /$$__  $$| $$    | $$_____/  | $$ /$$| $$  | $$           │
│        /$$$$$$$/|  $$$$$$$| $$    |  $$$$$$$  |  $$$$/|  $$$$$$$           │
│       |_______/  \_______/|__/     \_______/   \___/   \____  $$           │
│                                                            /$$  | $$           │
│                                                           |  $$$$$$/           │
│  by pyup.io                                                \______/           │
│                                                                              │
╘══════════════════════════════════════════════════════════════════════════════╛
╒══════════════════════════════════════════════════════════════════════════════╕
│ REPORT                                                                       │
│ checked 21 packages, using local DB                                          │
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
No known security vulnerabilities found.
```

clean

`pipenv clean` deinstalliert alle Pakete, die nicht in `Pipfile.lock` angegeben sind.

graph

`pipenv graph` zeigt für die aktuell installierten Pakete die Abhängigkeitsgrapheninformationen an.

install

`pipenv install` installiert bereitgestellte Pakete und fügt sie dem Pipfile hinzu. `pipenv install` kennt die folgenden Optionen:

-d, --dev
: installiert die Pakete in `[dev-packages]`, z.B.:

```
$ pipenv install --dev pytest
...
$ cat Pipfile
...
[dev-packages]
pytest = "*"
```

--deploy
: bricht ab, wenn `Pipfile.lock` nicht aktuell ist oder eine falsche Python-Version verwendet wird.

-r, --requirements <requirements.txt>
: importiert eine `requirements.txt`-Datei

--sequential
: installiert die Abhängigkeit in einer bestimmten Reihenfolge, nicht gleichzeitig.

  Dies verlangsamt zwar die Installation, erhöht jedoch die Determinierbarkeit der Builds.

sdist vs. wheel

Pip kann sowohl Pakete als Source Distribution (sdist) oder wheel installieren. Wenn beide auf PyPI vorhanden sind, wird pip ein kompatibles wheel bevorzugen.

Bemerkung: Abhängigkeiten von Wheels werden jedoch nicht erfasst von `$ pipenv lock`.

Requirement specifier

Dabei konkretisieren Requirement specifier das jeweilige Paket.

- Die aktuelleste Version kann installiert werden, z.B.:

```
$ pipenv install requests
```

- Eine spezifische Version kann installiert werden, z.B.:

```
$ pipenv install requests==2.18.4
```

- Soll die Version in einem bestimmten Versionsbereich liegen, kann dies ebenfalls angegeben werden:

```
$ pipenv install requests>=2,<3
```

- Auch eine kompatible Version lässt sich installieren:

```
$ pipenv install requests~=2.18
```

  Dies ist kompatibel mit ==2.18.*.

- Für einige Pakete können auch Installationsoptionen mit Extras mit eckigen Klammern angegeben werden:

```
$ pipenv install requests[security]
```

- Es kann auch angegeben werden, dass bestimmte Pakete nur auf bestimmten Systemen installiert werden, so wird bei folgendem `Pipfile` das Modul `pywinusb` nur auf Windows-Systemen installiert:

```
[packages]
pywinusb = {version = "*", sys_platform = "== 'win32'"}
```

  Ein komplexeres Beispiel unterscheidet, welche Modul-Versionen mit welchen Python-Versionen installiert werden soll:

```
[packages]
unittest2 = {version = ">=1.0,<3.0", markers="python_version < '2.7.9' or (python_
↪version >= '3.0' and python_version < '3.4')"}
```

VCS

Ihr könnt auch Python-Pakete aus Versionsverwaltungen installieren, z.B.:

```
$ pipenv install -e git+https://github.com/requests/requests.git#egg=requests
```

Bemerkung: Wenn `editable=false`, werden Unterabhängigkeiten nicht aufgelöst.

Weitere Informationen zu pipenv und VCS erhaltet ihr in Pipfile spec.

Auch die Credentials der Versionsverwaltung lassen sich im Pipfile angeben, z.B.:

```
[[source]]
url = "https://$USERNAME:${PASSWORD}@pypi.cusy.io/simple"
verify_ssl = true
name = "cusy-pypi"
```

Bemerkung: `pipenv` hasht das `Pipfile`, bevor die Umgebungsvariablen ermittelt werden, und auch in `Pipfile.lock` werden die Umgebungsvariablen geschrieben, sodass keine Credentials in der Versionsverwaltung gespeichert werden müssen.

`lock`

`pipenv lock` generiert die Datei `Pipfile.lock`, die alle Abhängigkeiten und Unterabhängigkeiten eures Projekts aufführt inklusive der neuesten verfügbaren Versionen und der aktuellen Hashwerte für die heruntergeladenen Dateien. Dies stellt wiederholbare und vor allem deterministische Builds sicher.

Bemerkung: Um den Determinismus zu erhöhen, kann neben den Hashwerten auch die Installationsreihenfolge gewährleistet werden. Hierfür gibt es das `--sequential`-Flag.

Security Features

`pipfile.lock` nutzt einige Sicherheitsverbesserungen von `pip`. So werden standardmäßig sha256-Hashes jedes heruntergeladenen Pakets generiert.

Wir empfehlen dringend, `lock` zum Deployment von Entwicklungsumgebungen in die Produktion zu verwenden. Hierbei verwendet ihr `pipenv lock` zum Kompilieren eurer Abhängigkeiten in der Entwicklungsumgebung und anschließend könnt ihr die kompilierte `Pipfile.lock`-Datei in der Produktionsumgebung für reproduzierbare Builds zu verwenden.

`open`

`pipenv open MODULE` zeigt ein bestimmtes Modul in eurem Editor an.

Falls ihr PyCharm verwendet, müsst ihr `pipenv` für euer Python-Projekt konfigurieren. Wie dies geht, ist in Configuring Pipenv Environment beschrieben.

`run`

`pipenv run` spawnt einen Befehl, der im virtual environment installiert ist, z.B.:

```
$ pipenv run python main.py
```

shell

`pipenv shell` spawnt eine Shell, im virtual environment. Damit erhaltet ihr einen Python-Interpreter, der alle Python-Pakete enthält und sich somit hervorragend z.B. zum Debugging und Testen eignet:

```
$ pipenv shell --fancy
Launching subshell in virtual environment...
bash-4.3.30$ python
Python 3.6.4 (default, Jan  6 2018, 11:51:59)
>>> import requests
>>>
```

Bemerkung: Shells sind meist nicht so konfiguriert, dass eine Subshell verwendet werden kann. Dies kann dazu führen, dass `pipenv shell --fancy` zu unerwarteten Ergebnissen führt. In diesen Fällen sollte `pipenv shell` verwendet werden, da diese einen Kompatibilitätsmodus verwendet.

---

sync

`pipenv sync` installiert alle in `Pipfile.lock` angegebenen Pakete.

uninstall

`pipenv uninstall` deinstalliert alle bereitgestellten Pakete und entfernt sie aus dem `Pipfile`. `uninstall` unterstützt alle Parameter von *install* und darüberhinaus die folgenden beiden Optionen:

`--all`
: löscht alle Dateien aus der virtuellen Umgebung, lässt aber `Pipfile` unberührt.

`--all-dev`
: entfernt alle Entwicklungspakete aus der virtuellen Umgebung und entfernt sie aus `Pipfile`.

update

`pipenv update` führt zunächst `pipenv lock` aus, dann `pipenv sync`.

`pipenv update` hat u.a. folgende Optionen:

`--clear`
: löscht den *Dependency Cache*.

`--outdated`
: listet veraltete Abhängigkeiten auf.

Deterministische Builds

Ihr müsst nur spezifizieren, was ihr wollt:

Mit `pipenv install requests` wird z.B. ein `Pipfile` erzeugt wie das folgende:

```
[[source]]
url = "https://pypi.org/simple"
verify_ssl = true
name = "pypi"

[packages]
requests = "*"

[dev-packages]

[requires]
python_version = "3.6"
```

Die zugehörige `Pipfile.lock`-Datei spezifiziert jedoch die Pakete exakt, z.B.:

```
{
    "default": {
        "requests": {
            "hashes": [
                "sha256:63b52e3c866428a224f97cab011de738c36aec0185aa91cfacd418b5d58911d1
↪",
                "sha256:ec22d826a36ed72a7358ff3fe56cbd4ba69dd7a6718ffd450ff0e9df7a47ce6a"
            ],
            "index": "pypi",
            "version": "==2.19.1"
        },
        "urllib3": {
            "hashes": [
                "sha256:a68ac5e15e76e7e5dd2b8f94007233e01effe3e50e8daddf69acfd81cb686baf
↪",
                "sha256:b5725a0bd4ba422ab0e66e89e030c806576753ea3ee08554382c14e685d117b5"
            ],
            "markers": "python_version != '3.2.*' and python_version != '3.1.*' and␣
↪python_version < '4' and python_version != '3.3.*' and python_version >= '2.6' and␣
↪python_version != '3.0.*'",
            "version": "==1.23"
        }
    },
    "develop": {}
}
```

`Pipfile.lock` spezifiziert auch alle Abhängigkeiten eures Projekts, wobei die Hashwerte der heruntergeladenen Dateien gespeichert werden. Dies soll wiederholbare und deterministische Builds gewährleisten.

Workflows

Im- und Export von `requirements.txt`-Dateien

Falls ihr in einem bestehenden Projekt bereits eine `requirements.txt`-Datei habt, so kann `pipenv` diese Abhängigkeiten auflösen. Sofern die `requirements.txt`-Datei im selben Verzeichnis liegt, einfach mit `$ pipenv install` oder falls sie in einem anderen Verzeichnis liegt mit `$ pipenv install -r /path/to/requirements.txt`.

Umgekehrt könnt ihr auch aus einer bestehenden Pipenv-Umgebung eine `requirements.txt`-Datei erzeugen mit:

```
$ pipenv run pip freeze > requirements.txt
```

Upgrade-Workflow

1. Findet heraus, was sich *Upstream* geändert hat:

   ```
   $ pipenv update --outdated
   Package 'requests' out-of-date: '==2.13.0' installed, '==2.19.1' available.
   ```

2. Um die Python-Pakete dann zu aktualisieren, habt ihr die folgenden beiden Optionen:

   - alles aktualisieren mit `$ pipenv update`
   - einzelne Pakete aktualisieren, z.B. `requests` mit `$ pipenv update requests`

`Pipfile` vs. `setup.py`

Dabei ist zu unterscheiden, ob ihr eine Anwendung oder eine Bibliothek entwickelt.

Bibliotheken
: Sie bieten wiederverwendbare Funktionen für andere Bibliotheken und Anwendungen/Projekte. Sie müssen mit anderen Bibliotheken zusammenarbeiten, die alle ihre eigenen Abhängigkeiten haben. Um Versionskonflikte in Abhängigkeiten verschiedener Bibliotheken innerhalb eines Projekts zu vermeiden, sollten Bibliotheken niemals Abhängigkeitsversionen festschreiben. Sie können jedoch Unter- oder Obergrenzen angeben, wenn sie sich auf ein bestimmtes Feature oder einen Bugfix verlassen. Bibliotheksabhängigkeiten werden über `install_requires` in der `setup.py` angegeben.

Anwendungen
: Sie verwenden Bibliotheken und sind meist nicht von anderen Projekten abhängig. Sie sollen in eine bestimmte Umgebung implementiert werden und erst dann sollten die genauen Versionen aller ihrer Abhängigkeiten und Subabhängigkeiten konkretisiert werden. Diesen Prozess zu erleichtern, ist das Hauptziel von Pipenv.

Umgebungsvariablen

`pipenv`-Umgebungsvariablen

`pipenv --envs` gibt Optionen der Environment-Variablen aus.

Weitere Informationen hierzu findet ihr unter Configuration With Environment Variables.

.env-Datei

Wenn eine `.env`-Datei in eurer virtuellen Umgebung vorhanden ist, werden `$ pipenv shell` und `$ pipenv run` diese automatisch laden:

```
$ cat .env
USERNAME=veit

$ pipenv run python
Loading .env environment variables...
...
```

```
>>> import os
>>> os.environ["USERNAME"]
'veit'
```

Auch die Credentials, z.B. der Versionsverwaltung lassen sich in der `Pipfile`-Datei angeben, z.B.:

```
[[source]]
url = "https://$USERNAME:${PASSWORD}@ce.cusy.io/api/v4/projects/$PROJECT_ID/packages/
↪pypi/simple"
verify_ssl = true
name = "gitlab"
```

Bemerkung: `pipenv` hasht die `Pipfile`-Datei, bevor die Umgebungsvariablen ermittelt werden, und auch die Umgebungsvariablen aus der `Pipfile.lock`-Datei werden ersetzt, sodass keine Credentials in der Versionsverwaltung gespeichert werden müssen.

Ihr könnt die `.env`-Datei auch außerhalb eures Virtual Environments speichern. Ihr müsst dann nur den Pfad zu dieser Datei angeben in `PIPENV_DOTENV_LOCATION`:

```
$ PIPENV_DOTENV_LOCATION=/path/to/.env pipenv shell
```

Ihr könnt auch verhindern, dass `pipenv` eine vorhandene `.env`-Datei verwenet mit:

```
$ PIPENV_DONT_LOAD_ENV=1 pipenv shell
```

Pipenv und Spack

Pipenv wurde bereits zur Installation von Jupyter Notebooks verwendet. Wir benötigen hier jedoch Pipenv für unsere *Spack-Environments* um einerseits binärkompatible Builds mit Spack erzeugen zu können und andererseits Python-Pakete für die Datenerhebung, -Visualisierung ETC. einfach nutzen zu können.

Aktiviert hierfür zunächst die passende Python-Version aus dem Spack-Environment:

```
$  spack env activate python-311
$ spack env status
==> In environment python-311
$ which python
/srv/jupyter/spack/var/spack/environments/python-311/.spack-env/view/bin/python
```

Das bestehende Pipenv-Environment könnt ihr anschließend installieren mit:

```
$ cd ~/jupyter-tutorial/pipenvs/python-374/
$ pipenv --python=/Users/veit/jupyter-tutorial/spackenvs/python-311/.spack-env/
↪view/bin/python --site-packages
$ pipenv install
Creating a virtualenv for this project...
Pipfile: /Users/veit/jupyter-tutorial/pipenvs/python-311/Pipfile
Using /Users/veit/jupyter-tutorial/spackenvs/python-311/.spack-env/view/bin/
↪python3.11 (3.11.4) to create virtualenv...
...
```

Dies verwendet das mit Spack installierte Environment und installiert weitere Pakete.

Siehe auch:

- Pipenv and Other Python Distributions

7.4 Erstellen von Programmbibliotheken und -paketen

Wie ihr Programmbibliotheken und -pakete erstellen könnt, erfahrt ihr in unserem Python Basics Tutorial.

7.5 Dokumentieren

Damit euer Produkt sinnvoll genutzt werden kann, sind Dokumentationen sowohl für die Zielgruppen in den Daten-Wissenschaften und im Data-Engineering, als auch im System-Engineering erforderlich:

- Die Zielgruppe in den Daten-Wissenschaften wollen dokumentiert sehen
  - welche Probleme euer Produkt löst und was die Hauptfunktionen und Limitationen der Software sind (`README`)
  - wie das Produkt beispielhaft verwendet werden kann
  - welche Veränderungen in aktuelleren Software-Versionen gekommen sind (`CHANGELOG`)
- Die Zielgruppe im Daten-Engineering wollen wissen, wie sie mit Fehlerbehebungen zur Verbesserung des Produkts beitragen können (`CONTRIBUTING`) und wie sie mit anderen kommunizieren (`CODE_OF_CONDUCT`) können
- Die Zielgruppe im System-Engineering benötigt eine Installationsanleitung für euer Produkt und für die erforderlichen Abhängigkeiten

Alle gemeinsam benötigen Informationen, wie das Produkt lizenziert ist (`LICENSE`-Datei oder `LICENSES`-Ordner) und wie sie bei Bedarf Hilfe erhalten können.

Siehe auch:

- Dokumentieren
- Read the Docs for Science

7.6 Lizenzieren

Damit andere eure Software verwenden können, sollte sie eine oder mehrere Lizenzen erhalten, die die Nutzungsbedingungen beschreiben. Andernfalls dürfte sie meist urheberrechtlich geschützt sein. Urheber sind diejenigen, die zur Software originär beigetragen haben. Wenn eine Software relizenziert werden soll, ist häufig die Zustimmung aller Personen erforderlich, die Urheberschaft beanspruchen können.

Bemerkung: Dies stellt keine Rechtsberatung dar. Wendet euch im Zweifelsfall bitte an eine Rechtsvertretung oder die Rechtsabteilung eures Unternehmens.

Siehe auch:

- The Whys and Hows of Licensing Scientific Code
- A Quick Guide to Software Licensing for the Scientist-Programmer
- Karl Fogel: Producing Open Source Software
- Forschungsdaten veröffentlichen

7.6.1 Proprietäre Softwarelizenzen

Proprietäre Softwarelizenzen sind selten standardisiert; sie können kommerziell, Shareware oder Freeware sein.

7.6.2 Freie und Open-Source Softwarelizenzen

Sie werden von der Free Software Foundation (FSF) und der Open Source Initiative (OSI) definiert. Dabei kann im Wesentlichen unterschieden werden zwischen Copyleft-, freizügigen- und gemeinfreien Lizenzen.

Copyleft- oder reziproke Lizenzen

Copyleft-Lizenzen verpflichten die Lizenznehmer, jegliche Bearbeitung der Software (sog. Derivate, unter die Lizenz des ursprünglichen Werks zu stellen. Dies soll Nutzungseinschränkungen der Software verhindern. Die bekannteste Copyleft-Lizenz ist die GPL (GNU General Public License). Dabei wird das Copyleft der GPL (GNU General Public License) als sehr stark, das der Mozilla Public License hingegen als sehr schwach angesehen.

Da die Lizenzgeber nicht selbst an ihr eigenes Copyleft gebunden sind, können sie neue Versionen auch unter proprietärer Lizenz veröffentlichen oder Dritten dies erlauben (Mehrfachlizenzierung).

Durch Copyleft-Lizenzen können bei der Verbreitung zusammen mit Software unter anderen freien Lizenzen jedoch schnell Inkompatibilitäten entstehen. So ist beispielsweise die 3-Clause-BSD-Lizenz mit der GPL inkompatibel.

Die EUPL ist hingegen eine reziproke Lizenz, die zumindest mit den meisten anderen offenen reziproken Lizenzen kompatibel und interoperabel ist: Die kompatiblen Lizenzverpflichtungen haben Vorrang, wenn sie mit den sich aus der EUPL ergebenden Verpflichtungen in Konflikt geraten.

Freizügige Open-Source-Lizenzen

Freizügige oder permissive Open-Source-Lizenzen erlauben eine breitere Wiederverwendung als die Copyleft-Lizenzen. Ableitungen und Kopien des Quellcodes können unter Bedingungen verbreitet werden, die grundlegend andere Eigenschaften haben als die der Originallizenz. Die bekanntesten Beispiele solcher Lizenzen sind MIT und BSD.

Gemeinfreie Lizenzen

Bei gemeinfreien oder Public Domain-Lizenzen gehen die Urheberrechte an die Allgemeinheit über. Zur Kennzeichnung der Gemeinfreiheit von Software wurde die WTFPL erstellt.

7.6.3 Nicht-Software-Lizenzen

Open-Source-Software-Lizenzen können auch für Werke verwendet werden, die nicht Software sind. Oft sind sie auch die beste Wahl, insbesondere wenn die betreffenden Werke als Quelltext bearbeitet und versioniert werden.

Daten, Medien, etc.

CC0 1.0, CC BY 4.0 und CC BY-SA 4.0 sind offene Lizenzen, die für Nicht-Software-Material verwendet werden, von Datensätzen bis zu Videos. Sie sind jedoch nicht für Software empfohlen.

Die Open Knowledge Foundation hat ebenfalls eine Reihe von Open Data Commons-Lizenzen für Daten/Datenbanken veröffentlicht:

Open Data Commons Open Database License (ODbL) v1.0
: Namensnennung und Weitergabe unter gleichen Bedingungen.

Open Data Commons Attribution License (ODC-By) v1.0
: Namensnennung.

Open Data Commons Public Domain Dedication and License (PDDL) v1.0
: Die PDDL stellt die Daten in den öffentlichen Bereich und verzichtet auf alle Rechte.

GovData hat die *Datenlizenz Deutschland* in zwei Varianten vorgelegt:

- Datenlizenz Deutschland – Namensnennung – Version 2.0
- Datenlizenz Deutschland – Zero – Version 2.0

Bei der Verwendung des Community Data License Agreement – Permissive, Version 2.0 müssen die Urheberrechtshinweise beibehalten werden.

Eine weitere mögliche Lizenz für künstlerische Werke ist die Free Art License 1.3.

Dokumentation

Jede Open-Source-Softwarelizenz oder offene Lizenz für Medien gilt auch für Software-Dokumentation. Wenn ihr unterschiedliche Lizenzen für eure Software und deren Dokumentation verwendet, solltet ihr darauf achten, dass die Quellcode-Beispiele in der Dokumentation auch unter der Software-Lizenz lizenziert sind. Neben den oben bereits genannten Creative Commons-Lizenzen gibt es speziell für freie Dokumentationen folgende Lizenzen.

GNU Free Documentation License (FDL)
: Copyleft-Lizenz für Dokumentationen, die für alle GNU-Handbücher verwendet werden soll. Ihre Anwendbarkeit ist auf textuelle Werke (Bücher) beschränkt.

FreeBSD Documentation License
Freizügige Dokumentationslizenz mit Copyleft, die mit der GNU FDL vereinbar ist.

Open Publication License, Version 1.0
freie Dokumentationslizenz mit Copyleft, sofern keine der Lizenzoptionen aus Abschnitt VI der Lizenz wahrgenommen werden. In jedem Fall ist sie mit der GNU FDL unvereinbar.

Schriftarten

SIL Open Font License 1.1
Schriftlizenz, die in anderen Werken frei verwendet werden kann.

GNU General Public License 3
Sie kann auch für Schriften verwendet werden, sie darf jedoch nur mit der Schriftausnahme in Dokumente eingebunden werden.

Siehe auch:

- Font Licensing

LaTeX ec fonts
Freie *European Computer Modern- und Text Companion*-Schriften, die üblicherweise mit Latex verwendet werden.

Arphic Public License
Freie Lizenz mit Copyleft.

IPA Font license
Freie Lizenz mit Copyleft, deren abgeleitete Werte jedoch nicht den Namen des Originals verwenden oder beinhalten dürfen.

Hardware

Entwürfe für Open-Source-Hardware werden von den CERN Open Hardware Lizenzen abgedeckt:

CERN-OHL-P-2.0
permissiv

CERN-OHL-W-2.0
schwach reziprok

CERN-OHL-S-2.0
stark reziprok

7.6.4 Auswahl geeigneter Lizenzen

Übersichten über mögliche Lizenzen findet ihr in SPDX License List oder OSI Open Source Licenses by Category. Bei der Wahl geeigneter Lizenzen unterstützt euch die Website Choose an open source license und Comparison of free and open-source software licenses.

Wenn ihr z.B. eine möglichst große Verbreitung eures Pakets erreichen wollt, sind MIT- oder die BSD-Varianten eine gute Wahl. Die Apache-Lizenz schützt euch besser vor Patentverletzungen, ist jedoch nicht kompatibel mit der GPL v2.

Abhängigkeiten überprüfen

Zudem solltet ihr schauen, welche Lizenzen diejenigen Pakete haben, von denen ihr abhängt und zu denen ihr kompatibel sein solltet:

Abb. 5: Lizenzkompatibilität für abgeleitete Werke oder kombinierte Werke aus eigenem Code und externem Code, der unter einer Open-Source-Lizenz steht (aus License compatibility, in Anlehnung an The Rise of Open Source Licensing S. 119).

Siehe auch:

Um Lizenzen zu analysieren, könnt ihr euch license compatibility anschauen.

Mit liccheck könnt ihr Python-Pakete und ihre Abhängigkeiten mit einer `requirement.txt`-Datei überprüfen z.B.:

```
liccheck -s liccheck.ini -r requirements.txt
gathering licenses...
3 packages and dependencies.
check unknown packages...
3 packages.
    cffi (1.15.1): ['MIT']
      dependency:
          cffi << cryptography
    cryptography (41.0.3): ['Apache Software', 'BSD']
      dependency:
          cryptography
    pycparser (2.21): ['BSD']
      dependency:
              pycparser << cffi << cryptography
```

Darüberhinaus kann es auch sinnvoll sein, ein Package unter mehreren Lizenzen zu veröffentlichen. Ein Beispiel hierfür ist cryptography/LICENSE:

> This software is made available under the terms of *either* of the licenses found in LICENSE.APACHE or LICENSE.BSD. Contributions to cryptography are made under the terms of *both* these licenses.
>
> The code used in the OpenSSL locking callback and OS random engine is derived from the same in CPython, and is licensed under the terms of the PSF License Agreement.

7.6.5 GitHub

Auf GitHub könnt ihr euch eine Open Source-Lizenz in eurem Repository erstellen lassen.

1. Geht zur Hauptseite eures Repository.
2. Klickt auf *Create new file* und gebt anschließend als Dateiname `LICENSE` oder `LICENSE.md` ein.
3. Anschließend könnt ihr rechts neben dem Feld für den Dateinamen auf *Choose a license template* klicken.
4. Nun könnt ihr die für euer Repository passende Open Source-Lizenz auswählen.
5. Ihr werdet nun zu zusätzlichen Angaben aufgefordert, sofern die gewählte Lizenz dies erfordert.
6. Nachdem ihr eine Commit-Message angegeben habt, z.B. `Add license`, könnt ihr auf *Commit new file* klicken.

Falls ihr in eurem Repository bereits eine `/LICENSE`-Datei hinzugefügt habt, verwendet GitHub licensee um die Datei mit einer kurzen Liste von Open-Source-Lizenzen abzugleichen. Falls GitHub die Lizenz eures Repository nicht erkennen kann, enthält es möglicherweise mehrere Lizenzen oder ist zu komplex. Überlegt Euch dann, ob ihr die Lizenz vereinfachen könnt, z.B. indem ihr Komplexität in die `/README`-Datei auslagert.

Umgekehrt könnt ihr auf GitHub auch nach Repositories mit bestimmten Lizenzen oder Lizenzfamilien suchen. Eine Übersicht über die Lizenz-Schlüsselwörter erhaltet ihr in Searching GitHub by license type.

Schließlich könnt ihr euch von Shields.io ein License-Badge generieren lassen, das ihr z.B. auf eurer `README`-Datei einbinden könnt:

```
|License|

.. |License| image:: https://img.shields.io/github/license/veit/python4datascience.svg
   :target: https://github.com/veit/python4datascience/blob/main/LICENSE
```

7.6.6 Standardformat für die Lizenzierung

SPDX steht für *Software Package Data Exchange* und definiert eine standardisierte Methode zum Austausch von Urheberrechts- und Lizenzinformationen zwischen Projekten und Personen. Die passenden SPDX-Identifier könnt ihr aus der SPDX License List auswählen und dann in den Kopf eurer Lizenzdateien eintragen:

```
# SPDX-FileCopyrightText: [year] [copyright holder] <[email address]>
#
# SPDX-License-Identifier: [identifier]
```

7.6.7 Konformität überprüfen

REUSE

REUSE wurde von der FSFE (Free Software Foundation Europe) initiiert, um die Lizenzierung freier Software-Projekte zu erleichtern. Das REUSE tool überprüft Lizenzen und unterstützt euch bei der Einhaltung der Lizenzkonformität, z.B.:

```
$ cd cryptography
$ reuse lint
# FEHLENDE URHEBERRECHTS- UND LIZENZINFORMATIONEN

Die folgenden Dateien haben keine Urheberrechts- und Lizenzinformationen:
* .gitattributes
* .github/ISSUE_TEMPLATE/openssl-release.md
...
* vectors/cryptography_vectors/x509/wosign-bc-invalid.pem
* vectors/pyproject.toml

Die folgenden Dateien haben keine Lizenzinformationen:
* docs/_ext/linkcode_res.py
* src/cryptography/__about__.py


# ZUSAMMENFASSUNG

* Falsche Lizenzen: 0
* Veraltete Lizenzen: 0
* Lizenzen ohne Dateiendung: 0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
* Fehlende Lizenzen: 0
* Unbenutzte Lizenzen: 0
* Verwendete Lizenzen: 0
* Read errors: 0
* files with copyright information: 2 / 2806
* files with license information: 0 / 2806

Leider ist Ihr Projekt nicht konform mit Version 3.0 der REUSE-Spezifikation :-(
```

Mit der REUSE API könnt ihr euch auch ein dynamisches Compliance-Badge generieren:

GitLab-CI-Workflow

Ihr könnt REUSE problemlos in euren Continuous Integration-Workflow integrieren:

Ihr könnt `reuse lint` automatisch als *Pre-Commit-Hook* bei jedem Commit ausführen lassen, indem ihr Folgendes zu eurer `.pre-commit-config.yaml`-Datei hinzufügt:

```yaml
repos:
- repo: https://github.com/fsfe/reuse-tool
  rev: v2.1.0
  hooks:
  - id: reuse
```

Fügt der `.gitlab-ci.yml`-Datei Folgendes hinzu:

```yaml
reuse:
  image:
    name: fsfe/reuse:latest
    entrypoint: [""]
  script:
    - reuse lint
```

Auf GitHub könnt ihr die REUSE-Aktion mit der GitHub-Aktion REUSE Compliance Check in euren Workflow integrieren, indem ihr z.B. Folgendes zu eurer `workflow .yml`-Datei hinzufügt:

```yaml
name: REUSE Compliance Check
on: [push, pull_request]
jobs:
  test:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v3
    - name: REUSE Compliance Check
      uses: fsfe/reuse-action@v2
```

Alternativen

ISO/IEC 5230/OpenChain
: empfiehlt *REUSE* als eine Komponente, um die Klarheit der Lizenz- und Urheberrechtssituation zu verbessern, stellt jedoch höhere Anforderungen, um eine vollständige Konformität zu erreichen.

  Sie basiert auf der OpenChain Specification 2.1 und ist ein internationaler Standard zu Software-Lieferketten, vereinfachter Beschaffung und Open-Source-Lizenz-Compliance.

  Siehe auch:

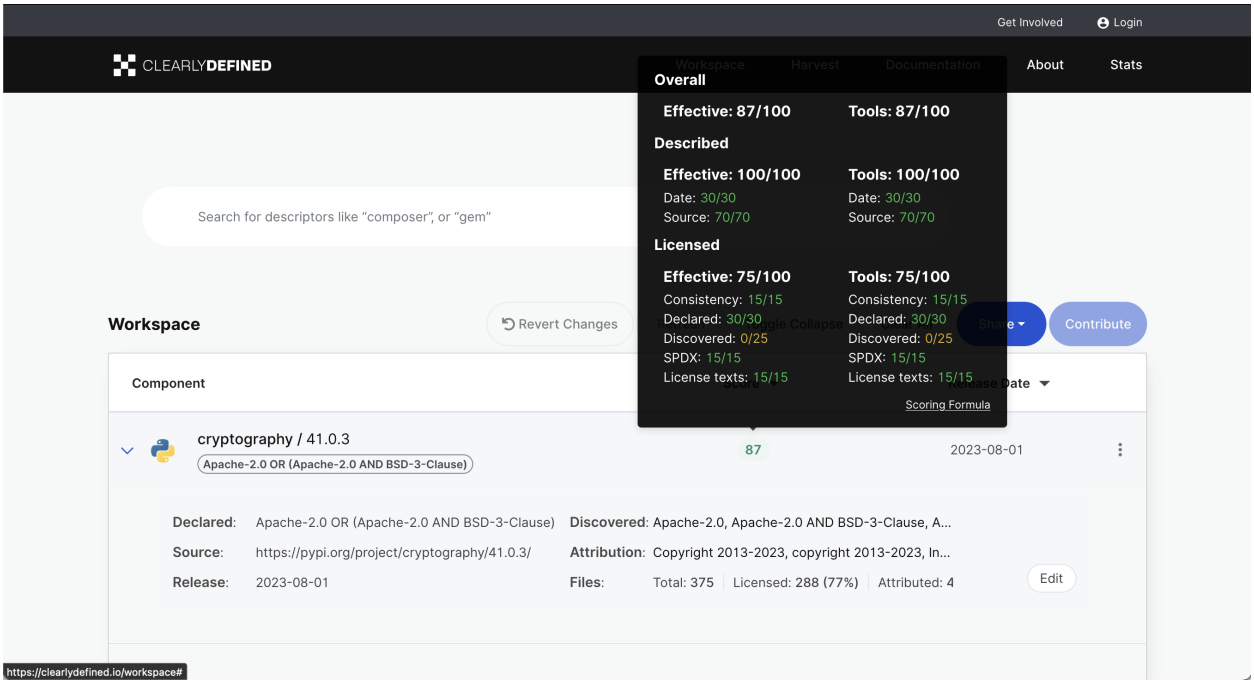
  - OpenChain project
  - OpenChain Self Certification
  - Reference-Material

AboutCode
: ist eine Community von Open-Source-Entwicklern, die die Nutzung von Open Source durch die Entwicklung von Open-Source-Tools für die Software Composition Analysis (SCA) erleichtern.

  ScanCode
  : bietet eine Reihe von Tools und Anwendungen zum Scannen von Software-Codebasen und -paketen, um den Ursprung und die Lizenz (Provenienz) von Open-Source-Software (und anderer Software von Drittanbietern) zu ermitteln.

  DeltaCode
  : vergleicht zwei Codebase-Scans, um signifikante Änderungen zu erkennen.

ClearlyDefined
: sammelt und zeigt Informationen über die Lizenzierungs- und Urheberrechtssituation eines Software-Projekts an.



FOSSology
: ist ein Toolkit für die Einhaltung freier Software, das Informationen in einer Datenbank mit Lizenz-, Copyright- und Exportscanner speichert.

OSS Review Toolkit (ORT)
ist ein Toolkit zur Automatisierung und Orchestrierung von FOSS-Richtlinien, mit dem ihr eure (Open-Source-)Software-Abhängigkeiten verwalten könnt. Es

- generiert OWASP CycloneDX, SPDX Software Bill of Materials (SBOM) oder benutzerdefinierte FOSS-Attributionsdokumentation für euer Softwareprojekt
- automatisiert eure FOSS-Policy, um euer Softwareprojekt und seine Abhängigkeiten auf Lizenzierung, Sicherheitslücken, Quellcode und technische Standards zu prüfen
- erstellt ein Quellcode-Archiv für euer Softwareprojekt und seine Abhängigkeiten, um bestimmte Lizenzen einzuhalten
- korrigiert Paket-Metadaten oder Lizenzfeststellungen selbst

Siehe auch:

- GitHub Action for ORT
- ORT for GitLab

licensechecker
Ein Kommandozeilenwerkzeug, das Installationsverzeichnisse nach Lizenzen durchsucht.

7.6.8 Python-Paket-Metadaten

Mit **PEP 658** wird die `METADATA`-Datei aus Distributionen in der **PEP 503**-Repository-API auf PyPI verfügbar. Damit können die Metadaten der Verteilungspakete analysiert werden ohne dass das ganze Paket heruntergeladen werden muss.

In Python-Paketen gibt es noch weitere Felder, in denen Lizenzinformationen gespeichert werden, wie die Core metadata specifications, die zudem limitiert sind. Dies führt nicht nur zu Problemen für die Autoren, die richtige Lizenz angeben zu können, sondern auch zu Problemen beim Re-Paketieren für diverse Linux-Distributionen.

Aktuell werden zwar einige häufige Fälle abgedeckt und die Lizenzklassifizierung kann auch erweitert werden, es gibt jedoch einige beliebte Klassifizierungen wie `License :: OSI Approved :: BSD License`, die abgeschafft werden. Damit ist dann jedoch die Abwärtskompatibilität nicht mehr gewährleistet und die Pakete müssen relizensiert werden. Immerhin habt ihr mit trove-classifiers auch eine Möglichkeit, eure Trove-Klassifizierungen zu überprüfen.

Siehe auch:

- **PEP 639** – Improving License Clarity with Better Package Metadata
- **PEP 621** – Storing project metadata in pyproject.toml
- **PEP 643** – Metadata for Package Source Distributions

7.7 Zitieren

Heute sind Software und Daten integraler Bestandteil der wissenschaftlichen Forschung. Mit Software werden Forschungsdaten erstellt, verarbeitet und analysiert sowie komplexe Prozesse modelliert und simuliert. Trotz ihrer zunehmenden Bedeutung in der Forschung ist wenig bekannt, wie sie in die wissenschaftlichen Anerkennungs- und Reputationssysteme eingebettet werden können. Zitate sind eine wesentliche Option in diesen Systemen, aber nur wenige Forschende wissen, wie Software und Daten zitiert werden können.

Zudem gibt es bedauerlicherweise keine allgemein anerkannten Richtlinien für die Urheberschaft von Software. Neben der Rolle *programmers* können auch andere Rollen wie z.B. *software architects*, *technical writers* und *maintainers* definiert werden.

Siehe auch:

- ICMJE: Defining the Role of Authors and Contributors
- Bot Recognize All Contributors

7.7.1 Daten zitieren

DataCite Metadata Schema

Die DataCite Metadata Working Group veröffentlichte 2019 die DataCite Metadata Schema zum Veröffentlichen und Zitieren von Forschungsdaten zusammen mit einer XSD (XML Schema Definition): metadata.xsd.

Ein einfaches Datacite-Beispiel kann folgendermaßen aussehen:

```
<?xml version="1.0" encoding="UTF-8"?>
<resource xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns="http://datacite.
↪org/schema/kernel-4" xsi:schemaLocation="http://datacite.org/schema/kernel-4 http://
↪schema.datacite.org/meta/kernel-4.3/metadata.xsd">
  <identifier identifierType="DOI">10.5072/D3P26Q35R-Test</identifier>
  <creators>
    <creator>
      <creatorName nameType="Personal">Fosmire, Michael</creatorName>
      <givenName>Michael</givenName>
      <familyName>Fosmire</familyName>
    </creator>
    <creator>
      <creatorName nameType="Personal">Wertz, Ruth</creatorName>
      <givenName>Ruth</givenName>
      <familyName>Wertz</familyName>
    </creator>
    <creator>
      <creatorName nameType="Personal">Purzer, Senay</creatorName>
       <givenName>Senay</givenName>
      <familyName>Purzer</familyName>
    </creator>
  </creators>
  <titles>
    <title xml:lang="en">Critical Engineering Literacy Test (CELT)</title>
  </titles>
  <publisher xml:lang="en">Purdue University Research Repository (PURR)</publisher>
  <publicationYear>2013</publicationYear>
  <subjects>
    <subject xml:lang="en">Assessment</subject>
    <subject xml:lang="en">Information Literacy</subject>
    <subject xml:lang="en">Engineering</subject>
    <subject xml:lang="en">Undergraduate Students</subject>
    <subject xml:lang="en">CELT</subject>
    <subject xml:lang="en">Purdue University</subject>
  </subjects>
  <language>en</language>
  <resourceType resourceTypeGeneral="Dataset">Dataset</resourceType>
  <version>1.0</version>
  <descriptions>
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    <description xml:lang="en" descriptionType="Abstract">
      We developed an instrument, Critical Engineering Literacy Test (CELT), which is a␣
↪multiple choice instrument designed to measure undergraduate students' scientific and␣
↪information literacy skills. It requires students to first read a technical memo
      and, based on the memo's arguments, answer eight multiple choice and six open-
↪ended response questions. We collected data from 143 first-year engineering students␣
↪and conducted an item analysis. The KR-20 reliability of the instrument was .39. Item
      difficulties ranged between .17 to .83. The results indicate low reliability index␣
↪but acceptable levels of item difficulties and item discrimination indices. Students␣
↪were most challenged when answering items measuring scientific and mathematical
      literacy (i.e., identifying incorrect information).
    </description>
  </descriptions>
</resource>
```

W3C-PROV

Die PROV-Dokumentenfamilie der W3C-Arbeitsgruppe definiert verschiedene Aspekte, die erforderlich sind, um Herkunftsinformationen interoperabel austauschen zu können.

Siehe auch:

- Luc Moreau, Paul Groth: Provenance: An Introduction to PROV
- Provenance storage and distribution
- ProvStore's API documentation

Python prov

Mit prov steht eine Python3-Bibliothek zur Verfügung, die den Im- und Export des PROV-Datenmodells in folgende Serialisierungsformate unterstützt:

- PROV-O (RDF)
- PROV-XML
- PROV-JSON

Zudem können mit NetworkX MultiDiGraph PROV-Dokumente erstellt werden und umgekehrt. Schließlich können PROV-Dokumente auch als Graphen in den Formaten PDF, PNG und SVG generiert werden.

Siehe auch:

- Dong Huynh: A Short Tutorial for Prov Python
- PROV Tutorial.ipynb

7.7.2 Software zitieren

James Howison und Julia Bullard führten in ihrem 2016 veröffentlichten Artikel Software in the scientific literature folgende Beispiele in absteigender Reputation auf:

1. zitieren von Veröffentlichungen, die die jeweilige Software beschreiben
2. zitieren von Bedienungsanleitungen
3. zitieren der Software-Projekt-Website
4. Link auf eine Software-Projekt-Website
5. erwähnen des Software-Namens

Die Situation bleibt für die Autor*innen von Software dennoch unbefriedigend, zumal wenn sie sich von den Autor*innen der Software-Beschreibung unterscheiden. Umgekehrt ist Forschungssoftware leider auch nicht immer gut geeignet um zitiert zu werden. So werden andere eure Software kaum direkt zitieren können, wenn ihr ihnen die Software als Anhang von E-Mails schickt. Auch ein Download-Link ist hier noch nicht wirklich zielführend. Besser stellt ihr einen Persistent Identifier (PID) bereit, um die langfristige Verfügbarkeit eurer Software sicherzustellen. Sowohl Zenodo- als auch das figshare-Repository akzeptieren Quellcode einschließlich Binärdateien und stellen Digital Object Identifier (DOI) hierfür bereit. Gleiches gilt für CiteAs, mit dem sich Zitierinformationen für Software abrufen lassen.

Siehe auch:

- Should I cite?
- How to cite software "correctly"
- Daniel S. Katz: Compact identifiers for software: The last missing link in user-oriented software citation?
- Neil Chue Hong: How to cite software: current best practice
- Recognizing the value of software: a software citation guide
- Stephan Druskat, Radovan Bast, Neil Chue Hong, Alexander Konovalov, Andrew Rowley, Raniere Silva: A standard format for CITATION files
- Module-5-Open-Research-Software-and-Open-Source
- Software Heritage: Save and reference research software
- Mining software metadata for 80 M projects and even more
- Extensions to schema.org to support structured, semantic, and executable documents
- Guide to Citation File Format schema
- schema.json

Erstellen eines DOI mit Zenodo

Zenodo ermöglicht die Archivierung von Software und die Bereitstellung eines DOI für diese Software. Im Folgenden werde ich am Beispiel des Jupyter-Tutorials zeigen, welche Schritte hierzu erforderlich sind:

1. Wenn ihr noch keinen Account für Zenodo habt, erstellt einen, bevorzugt mit GitHub.
2. Aktiviert in *Upload* → *New Upload* unter *Basic information* den Button *Reserve DOI* um einen DOI (Digital Object Identifier) für euren Upload zu reservieren. Lasst das Formular offen um später eure Software hochladen zu können.
3. Erstellt oder ändert die *CodeMeta*- und *Citation File Format*-Dateien in eurem Software-Verzeichnis.
4. Bindet den Badge in der `README`-Datei eurer Software ein:

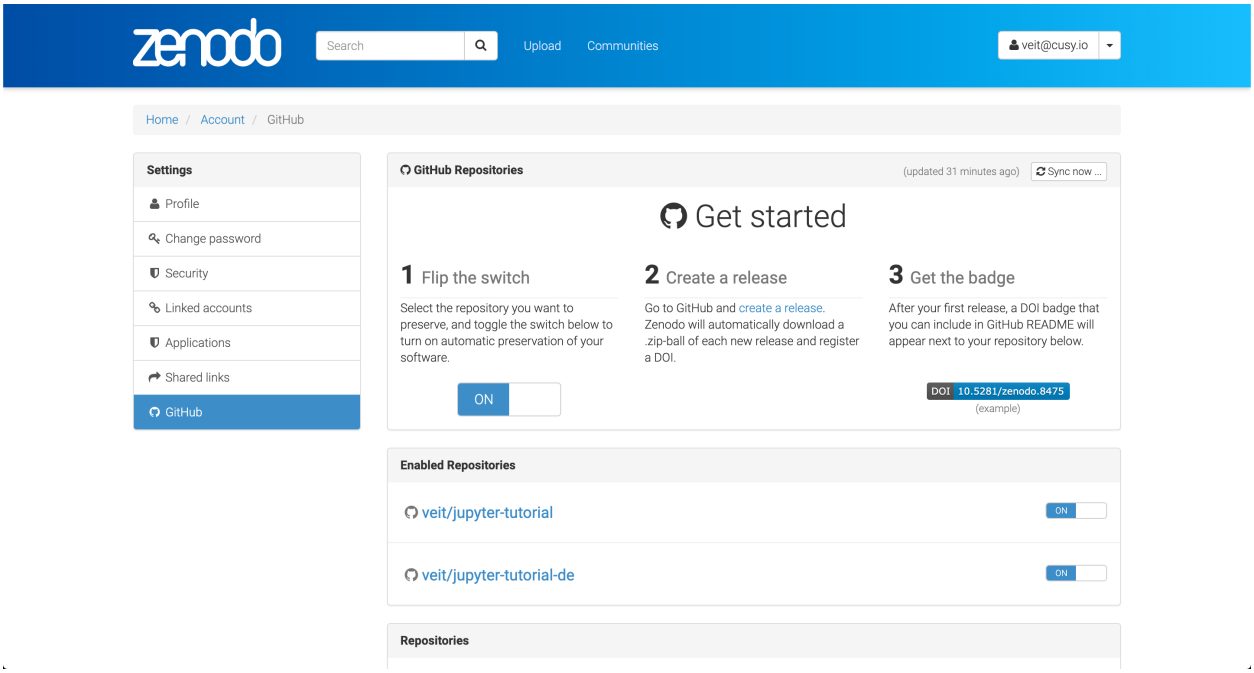
Markdown:

```
[![DOI](https://zenodo.org/badge/307380211.svg)](https://zenodo.org/badge/
↪latestdoi/307380211)
```

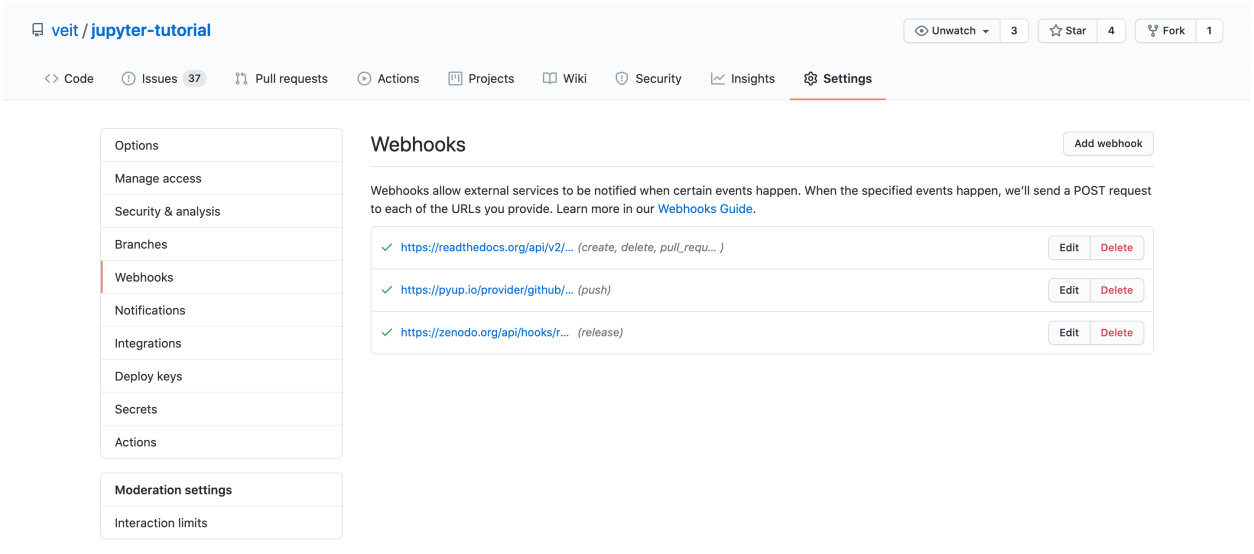
reStructedText:

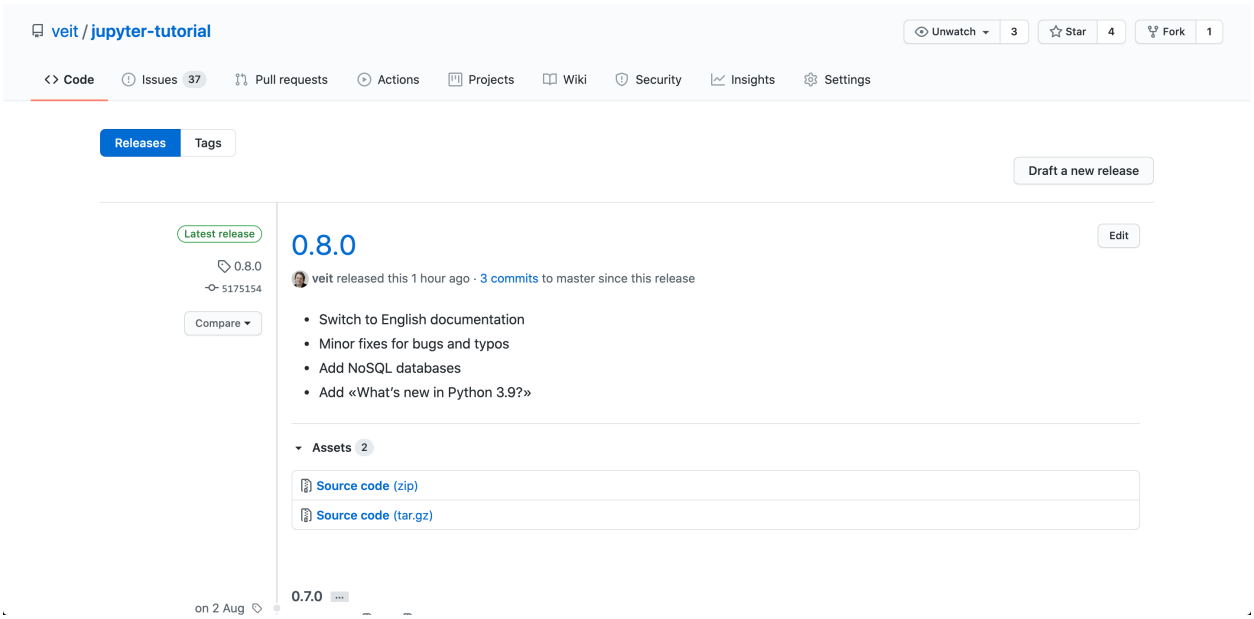
```
.. image:: https://zenodo.org/badge/307380211.svg
   :target: https://zenodo.org/badge/latestdoi/307380211
```

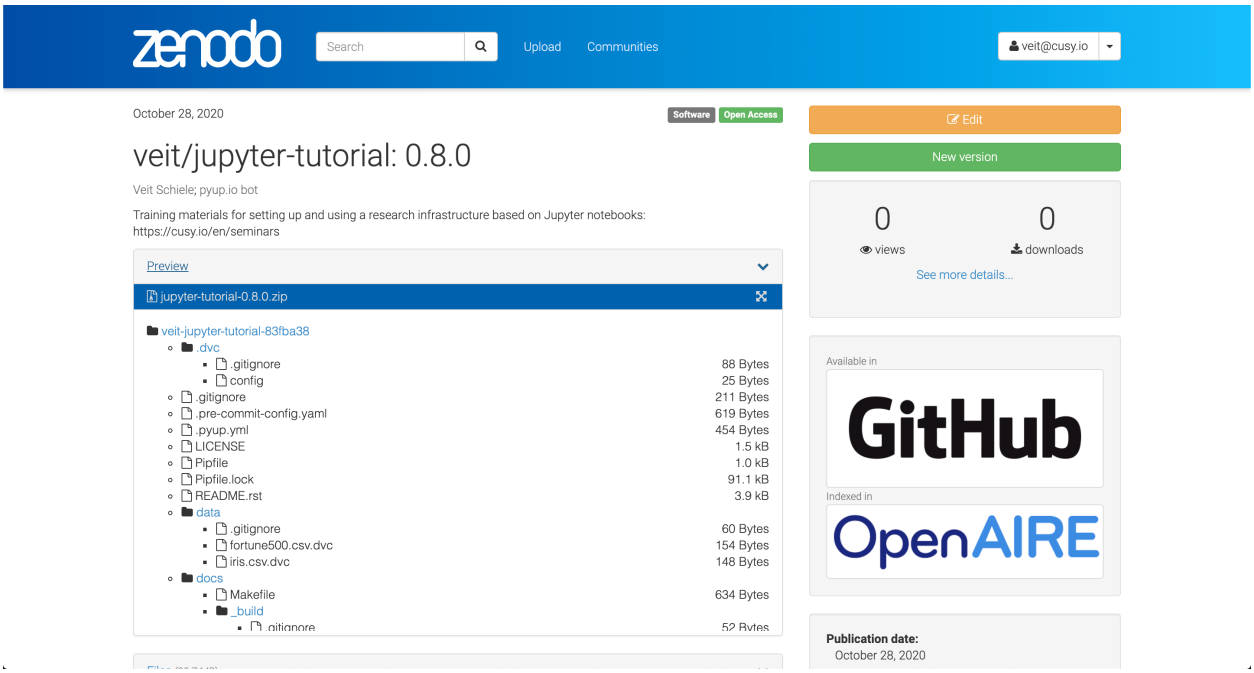
5. Nun wählt das Repository aus, das ihr archivieren wollt:



6. Überprüft, ob Zenodo einen Webhook in eurem Repository für das *Releases*-Event erstellt hat:
7. Erstellt ein neues Release:
8. Überprüft, ob der DOI korrekt erstellt wurde:


veit / jupyter-tutorial
Unwatch 3
Star 4
Fork 1
Code
Issues 37
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Options
Manage access
Security & analysis
Branches
Webhooks
Notifications
Integrations
Deploy keys
Secrets
Actions
Moderation settings
Interaction limits
Webhooks
Add webhook
Webhooks allow external services to be notified when certain events happen. When the specified events happen, we'll send a POST request to each of the URLs you provide. Learn more in our Webhooks Guide.
https://readthedocs.org/api/v2/... (create, delete, pull_requ...)
Edit
Delete
https://pyup.io/provider/github/... (push)
Edit
Delete
https://zenodo.org/api/hooks/r... (release)
Edit
Delete


veit / jupyter-tutorial
Unwatch 3
Star 4
Fork 1
Code
Issues 37
Pull requests
Actions
Projects
Wiki
Security
Insights
Settings
Releases
Tags
Draft a new release
Latest release
0.8.0
5175154
Compare
0.8.0
Edit
veit released this 1 hour ago · 3 commits to master since this release
Switch to English documentation
Minor fixes for bugs and typos
Add NoSQL databases
Add «What's new in Python 3.9?»
Assets 2
Source code (zip)
Source code (tar.gz)
0.7.0
on 2 Aug



Metadaten-Formate

Die FORCE11 -Arbeitsgruppe hat ein Paper veröffentlicht, in denen Prinzipien des wissenschaftlichen Software-zitierens dargelegt werden: Arfon Smith, Daniel Katz, Kyle Niemeyer: FORCE11 Software Citation Working Group, 2016. Dabei kristallisieren sich aktuell zwei Projekte für strukturierte Metadaten heraus:

CodeMeta

CodeMeta ist ein Austauschschema für allgemeine Software-Metadaten und Referenzimplementierung für JSON for Linking Data (JSON-LD).

Dabei wird eine `codemeta.json`-Datei im Stammverzeichnis des Software-Repository erwartet. Die Datei kann z.B. so aussehen:

```
{
    "@context": "https://doi.org/10.5063/schema/codemeta-2.0",
    "@type": "SoftwareSourceCode",
    "author": [{
        "@type": "Person",
        "givenName": "Stephan",
        "familyName": "Druskat",
        "@id": "http://orcid.org/0000-0003-4925-7248"
    }],
    "name": "My Research Tool",
    "softwareVersion": "2.0",
    "identifier": "https://doi.org/10.5281/zenodo.1234",
    "datePublished": "2017-12-18",
    "codeRepository": "https://github.com/research-software/my-research-tool"
}
```

Siehe auch:

- CodeMeta generator
- Codemeta Terms
- GitHub Repository

Citation File Format

(Fortsetzung der vorherigen Seite)

```
    type: doi
    value: "10.5281/zenodo.4147287"
keywords:
  - "data-science"
  - jupyter
  - "jupyter-notebooks"
  - "jupyter-kernels"
  - ipython
  - pandas
  - spack
  - pipenv
  - ipywidgets
  - "ipython-widget"
  - dvc
title: "Jupyter tutorial"
version: "0.8.0"
date-released: 2020-10-08
license: "BSD-3-Clause"
repository-code: "https://github.com/veit/jupyter-tutorial"
```

Ihr könnt einfach das obige Beispiel anpassen um eure eigene `CITATION.cff`-Datei zu erzeugen oder die Website cffinit verwenden.

Mit cff-validator steht euch eine GitHub-Action zur Verfügung, die `CITATION.cff`-Dateien mit dem R-Paket `V8` überprüft.

Es gibt auch einige Tools zum Workflow von *CITATION.cff* -Dateien:

- cff-converter-python konvertiert `CITATION.cff`-Dateien in BibTeX, RIS, *CodeMeta*- und andere Dateiformate
- doi2cff erstellt eine `CITATION.cff`-Datei aus einem Zenodo DOI

Auch GitHub bietet einen Service um die Informationen aus der `CITATION.cff`-Datei eures GitHub-Repository im APA- und BibTex-Format zu kopieren.

Siehe auch:

- GitHub Docs: About CITATION files

Wenn ihr einen DOI mit Zenodo registriert, wird die `CITATION.cff`-Datei aus dem GitHub-Repository ebenfalls verwendet. Auch Zotero interpretiert die *Citation File Format*-Datei in GitHub-Repositories; Zotero kann jedoch auch ohne *Citation File Format*-Datei Metainformationen des Repository, wie Unternehmen, Programmiersprache ETC., übernehmen.

Git2PROV

Git2PROV generiert PROV-Daten aus den Informationen eines Git-Repository.

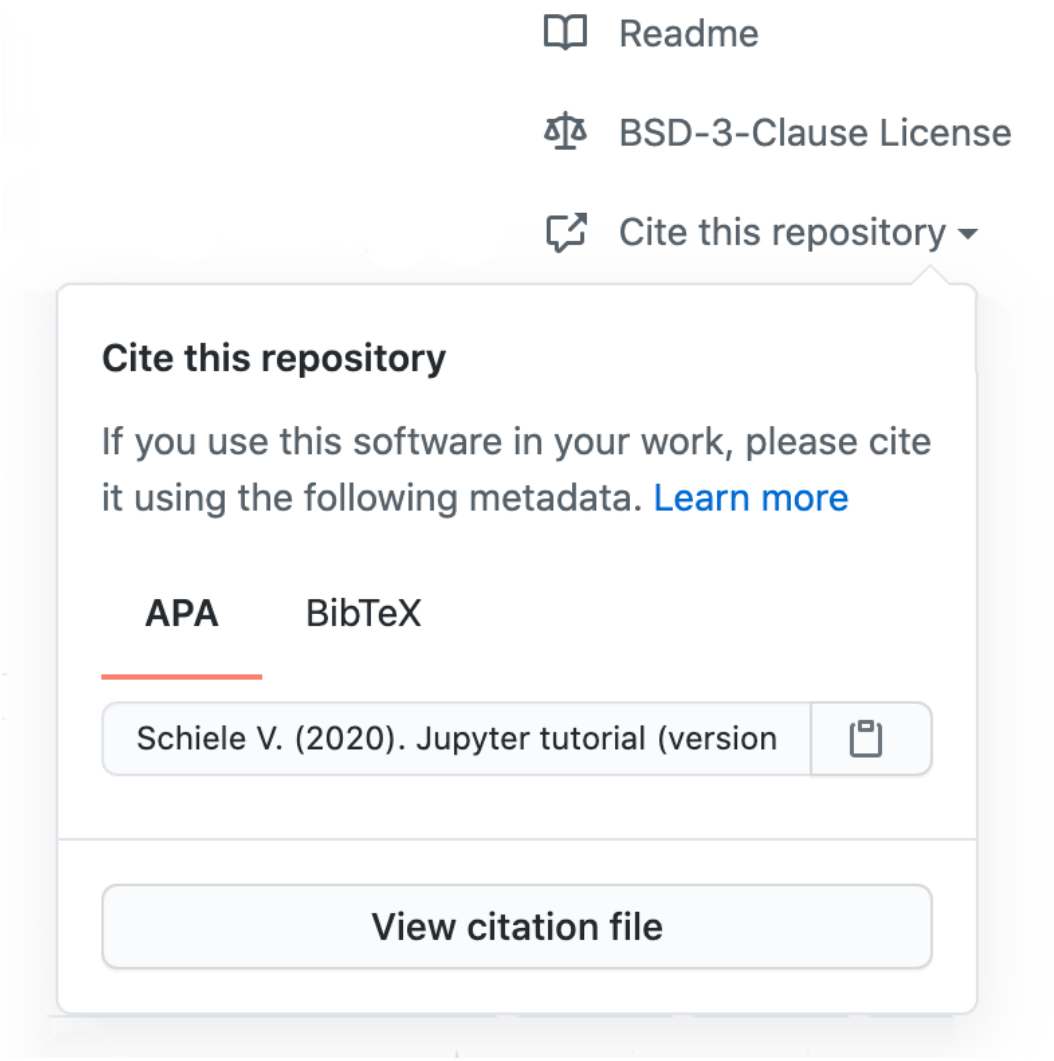
Auf der Kommandozeile kann die Konvertierung einfach ausgeführt werden mit:

```
$ git2prov git_url [serialization]
```

Zum Beispiel:

```
$ git2prov git@github.com:veit/python4datascience.git PROV-JSON
```

Insgesamt stehen die folgenden Serialisierungsformate zur Verfügung:


Readme
BSD-3-Clause License
Cite this repository
Cite this repository
If you use this software in your work, please cite it using the following metadata. Learn more
APA
BibTeX
Schiele V. (2020). Jupyter tutorial (version
View citation file

- `PROV-N`
- `PROV-JSON`
- `PROV-O`
- `PROV-XML`

Alternativ stellt Git2PROV auch einen Web-Server bereit mit:

```
$ git2prov-server [port]
```

Siehe auch:

- Git2PROV: Exposing Version Control System Content as W3C PROV
- GitHub-Repository

HERMES

HERMES vereinfacht die Publikation von Forschungssoftware, indem kontinuierlich in *Citation File Format*, *CodeMeta* und *Git* vorhandene Metadaten abgerufen werden. Anschließend werden die Metadaten auch für InvenioRDM und Dataverse passend zusammengestellt. Schließlich werden auch *CITATION.cff* und *codemeta.json* für die Publikationsrepositories aktualisiert.

1. `.hermes/` in der *.gitignore*-Datei hinzufügen
2. *CITATION.cff*-Datei mit zusätzlichen Metadaten bereitstellen

   ---

   Wichtig: Stellt sicher, dass `license` in der Datei *CITATION.cff* definiert ist; andernfalls wird eure Veröffentlichung von der *Zenodo*-Sandbox nicht als Open Access akzeptiert.

   ---

3. HERMES-Workflow konfigurieren

   Der HERMES-Workflow wird konfiguriert in der *TOML*-Datei `hermes.toml`, wobei jeder Schritt einen eigenen Abschnitt erhält.

   Wenn ihr HERMES so konfigurieren wollt, dass die Metadaten aus *Git* und *CITATION.cff* verwendet werden und die Ablage in der Zenodo Sandbox, die auf InvenioRDM aufbaut, erfolgen soll, sieht die `hermes.toml`-Datei folgendermaßen aus:

   Quellcode 2: hermes.toml

```
# SPDX-FileCopyrightText: 2023 Veit Schiele
#
# SPDX-License-Identifier: BSD-3-Clause

[harvest]
from = [ "git", "cff" ]

[deposit]
mapping = "invenio"
target = "invenio"

[deposit.invenio]
site_url = "https://sandbox.zenodo.org"
access_right = "open"
```

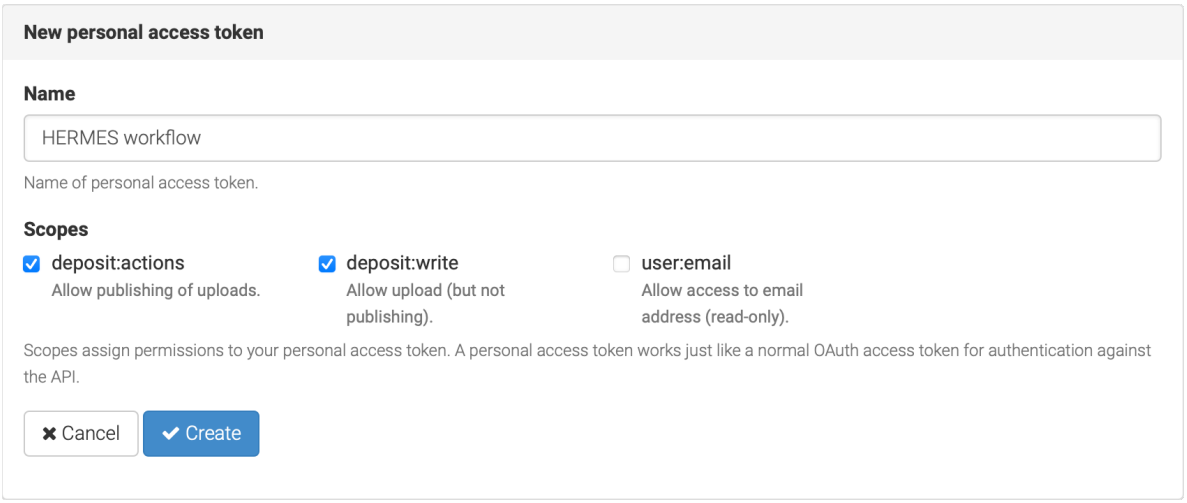
(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

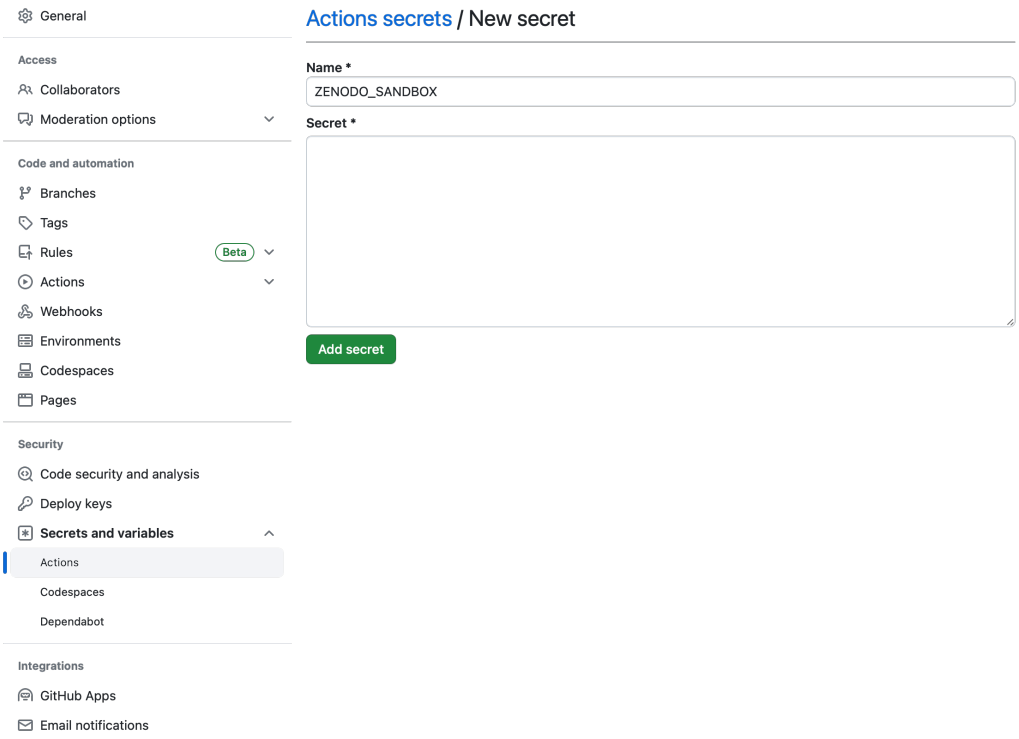
```
[postprocess]
execute = [ "config_record_id" ]
```

4. Zugangstoken für Zenodo Sandbox

   Damit GitHub Actions euer Repository in der Zenodo Sandbox veröffentlichen kann, benötigt ihr ein persönliches Zugangstoken. Hierfür müsst ihr euch bei der Zenodo Sandbox anmelden, um dann in eurem Benutzerprofil einen persönliches Zugangstoken mit dem Namen `HERMES workflow` und den Geltungsbereichen *deposit:actions* und *deposit:write* zu erstellen:



5. Kopiert das neu erstellte Token in ein neues GitHub Secret namens `ZENODO_SANDBOX` in Ihrem Repository: *Settings* → *Secrets and Variables* → *Actions* → *New repository secret*:



6. Konfiguriert die GitHub-Aktion

   Das HERMES-Projekt stellt Vorlagen zur kontinuierlichen Integration in einem speziellen Repository bereit: hermes-hmc/ci-templates. Kopiert die Vorlagendatei TEMPLATE_hermes_github_to_zenodo.yml in das Verzeichnis `.github/workflows/` eures Repository und benennt sie um, z.B. in `hermes_github_to_zenodo.yml`.

   Anschließend solltet ihr die Datei durchgehen und nach Kommentaren, die mit `# ADAPT` gekennzeichnet sind, suchen. Passt die Datei an eure Bedürfnisse an.

   Schließlich fügt ihr die Workflow-Datei zur Versionskontrolle hinzu und schiebt sie auf den GitHub-Server:

   ```
   $ git add .github/workflows/hermes_github_to_zenodo.yml
   $ git commit -m ":construction_worker: GitHub action for automatic publication with␣
   ↪HERMES"
   $ git push
   ```

7. GitHub-Actions sollen Pull Requests in eurem Repository erstellen dürfen

   Der HERMES-Workflow wird keine Metadaten ohne eure Zustimmung veröffentlichen. Stattdessen erstellt er einen Pull-Request, damit ihr die hinterlegten Metadaten genehmigen oder ändern könnt. Um dies zu aktivieren, geht in eurem Repository zu *Settings* → *Actions* → *General* und aktiviert im Abschnitt *Workflow permissions* *Allow GitHub Actions to create and approve pull requests*.

7.7.3 Software-Journale

Allgemein

- IEEE Computer Society Digital Library
- Wiley Online Library
- Journal of Open Source Software
- Journal of Open Research Software (JORS)
- Journal of Software: Practice and Experience
- Nature Toolbox
- Research Ideas and Outcomes (RIO)
- SIAM Journal on Scientific Computing (SISC) Software section
- SoftwareX

Bildverarbeitung

- Image Processing On Line
- Insight Journal

Biologie

- American Journal of Human Genetics
- Artificial Life
- Psychonomic Society: Behaviour Research Methods
- Oxford Academic: Bioinformatics
- Bioinformatics and Biology Insights
- Biophysical Journal
- BMC Bioinformatics
- BMC Systems Biology
- Bone
- Computer Methods and Programs in Biomedicine
- Current Protocols in Bioinformatics
- Database: The Journal of Biological Databases and Curation
- Ecography
- eLife
- Epidemiology
- Evolutionary Bioinformatics
- F1000 Research
- Frontiers in Neuroinformatics
- Gigascience
- Methods in Ecology and Evolution
- Nature Methods
- Neurocomputing
- Neuroinformatics
- Nucleic Acids Research
- PeerJ
- PLoS Computational Biology: Software collection
- PLoS ONE
- Trends in Parasitology

Chemie

- International Journal of Quantum Chemistry
- Journal of Applied Crystallography
- Journal of Chemical Theory and Computation
- Journal of Chemical Information and Modelling
- Journal of Cheminformatics
- Journal of Computational Chemistry
- Molecular Simulation
- Wiley Interdisciplinary Reviews: Computational Molecular Science

Geistes- und Sozialwissenschaften

- Digital Humanities Quarterly
- Journal of Artificial Societies and Social Simulation
- Journal of Economic Dynamics and Control

Ingenieurwesen

- Advances in Engineering Software
- Coastal Engineering
- Renewable Energy

Informatik, Mathematik und Statistik

- ACM Transactions on Mathematical Software
- The Archive of Numerical Software
- Future Generation Computer Systems
- Journal of Machine Learning Research: Machine Learning Open Source Software track
- Journal of Multiscale Modelling and Simulation
- Journal of Parallel and Distributed Computing
- Journal of Software for Algebra and Geometry
- Journal of Statistical Softwarer
- Knowledge-Based Systemsr
- LMS Journal of Computation and Mathematics
- The Mathematica Journal
- Mathematical Programming Computation
- Numerical Algorithms
- PeerJ Computer Science

- The R Journal
- Science of Computer Programming
- The Stata Journal

Physik und Geowissenschaften

- AAS: The Astronomy Journal
- AAS: The Astrophysical Journal
- AAS: The Astrophysical Journal Supplement Series
- Astronomy and Computing
- Communications in Computational Physics
- Computational Astrophysics and Cosmology
- Computer Physics Communications
- Computers and Geosciences
- Computing and Software for Big Science
- Environmental Modelling & Software
- Geoscientific Model Development

7.8 Testen

Alle Möglichkeiten, die ihr zum Testen eurer Notebooks habt, stehen euch auch für Python-Pakete zur Verfügung. Darüberhinaus könnt ihr auch die Testabdeckung eures Pakets überprüfen und regelmäßig automatisiert eure Tests ausführen lassen.

Siehe auch:

Testen

7.9 Logging

Das logging-Modul ist Teil der Python-Standardbibliothek. Es ist beschrieben in **PEP 0282**. Eine erste Einführung in das Modul erhaltet ihr in Basic Logging Tutorial.

Logging erfüllt üblicherweise zwei verschiedene Zwecke:

- Diagnose:
  - Ihr könnt euch den Kontext von bestimmten Ereignissen anzeigen lassen.
  - Tools wie Sentry gruppieren zusammengehörende Ereignisse und erleichtern die Benutzeridentifikation etc., sodass Entwickler die Fehlerursache schneller finden können.
- Monitoring:
  - Das Logging zeichnet Ereignisse für benutzerdefinierten Heuristiken auf, z.B. für Geschäftsanalysen. Diese Aufzeichnungen können für Berichte oder Optimierungen der Geschäftsziele verwendet und ggf. visualisiert werden.

Welche Vorteile bietet `logging` nun gegenüber `print`?

- Die Logdatei enthält alle verfügbaren Diagnoseinformationen wie Dateiname, Pfad, Funktion und Zeilennummer
- Alle Ereignisse sind über den Root-Logger automatisch verfügbar, sofern sie nicht explizit herausgefiltert werden.
- Logging kann wahlweise durch eine der folgenden beiden Methoden stummgeschaltet werden: logging.Logger.setLevel() oder logging.disabled.

Siehe auch:

- loguru macht das Protokollieren fast so einfach wie die Verwendung von `print`-Anweisungen.
- structlog fügt euren Log-Einträgen Struktur hinzu.

7.9.1 Logging-Beispiele

Erstellen einer Log-Datei

[1]:
```
import logging


logging.warning("This is a warning message")
logging.critical("This is a critical message")
logging.debug("debug")
```

```
WARNING:root:This is a warning message
CRITICAL:root:This is a critical message
```

Logging-Ebenen

Ebene	Beschreibung
`CRITICAL`	Das Programm wurde angehalten
`ERROR`	Ein schwerwiegender Fehler ist aufgetreten
`WARNING`	Ein Hinweis darauf, dass etwas Unerwartetes passiert ist (Standardstufe)
`INFO`	Bestätigung, dass die Dinge wie erwartet funktionieren.
`DEBUG`	Detaillierte Informationen, die in der Regel nur bei der Diagnose von Problemen von Interesse sind.

Setzen der Logging-Ebene

[2]:
```
import logging


logging.basicConfig(filename="example.log", filemode="w", level=logging.INFO)

logging.info("Informational message")
logging.error("An error has happened!")
```

```
ERROR:root:An error has happened!
```

Erstellen eines Logger-Objekts

[3]:
```
import logging


logging.basicConfig(filename="example.log")
logger = logging.getLogger("example")
logger.setLevel(logging.INFO)

try:
    raise RuntimeError
except Exception:
    logger.exception("Error!")
```

```
ERROR:example:Error!
Traceback (most recent call last):
  File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_65477/2646645271.py",␣
↪line 9, in <module>
    raise RuntimeError
RuntimeError
```

Ausnahmen (engl.: *exceptions*) loggen

[4]:
```
try:
    1 / 0
except ZeroDivisionError:
    logger.exception("You can't do that!")
```

```
ERROR:example:You can't do that!
Traceback (most recent call last):
  File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_65477/760044062.py",␣
↪line 2, in <module>
    1 / 0
    ~~^~~
ZeroDivisionError: division by zero
```

Logging-Handler

Handler-Typen

Handler	Beschreibung
`StreamHandler`	`stdout`, `stderr` oder dateiähnliche Objekte
`FileHandler`	für das Schreiben auf die Festplatte
`RotatingFileHandler`	unterstützt die Protokollrotation
`TimedRotatingFileHandle`	unterstützt die Rotation von Protokolldateien auf der Festplatte in bestimmten Zeitabständen
`SocketHandler`	sendet Logging-Ausgaben an einen Netzwerk-Socket
`SMTPHandler`	unterstützt das Senden von Logging-Nachrichten an eine E-Mail-Adresse über SMTP

Siehe auch

Weitere Handler können gefunden werden unter Logging handlers.

StreamHandler

[5]:
```
import logging


logger = logging.getLogger("stream_logger")
logger.setLevel(logging.INFO)

console = logging.StreamHandler()

logger.addHandler(console)
logger.info("This is an informational message")
```

```
This is an informational message
INFO:stream_logger:This is an informational message
```

SMTPHandler

[6]:
```
import logging
import logging.handlers


logger = logging.getLogger("email_logger")
logger.setLevel(logging.INFO)
fh = logging.handlers.SMTPHandler(
    "localhost",
    fromaddr="python-log@localhost",
    toaddrs=["logs@cusy.io"],
    subject="Python log",
)
logger.addHandler(fh)
logger.info("This is an informational message")
```

```
--- Logging error ---
Traceback (most recent call last):
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/logging/handlers.py", line 1081, in emit
    smtp = smtplib.SMTP(self.mailhost, port, timeout=self.timeout)
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/smtplib.py", line 255, in __init__
    (code, msg) = self.connect(host, port)
                  ^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/smtplib.py", line 341, in connect
    self.sock = self._get_socket(host, port, self.timeout)
                ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪11/lib/python3.11/smtplib.py", line 312, in _get_socket
    return socket.create_connection((host, port), timeout,
           ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/socket.py", line 851, in create_connection
    raise exceptions[0]
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/socket.py", line 836, in create_connection
    sock.connect(sa)
ConnectionRefusedError: [Errno 61] Connection refused
Call stack:
  File "<frozen runpy>", line 198, in _run_module_as_main
  File "<frozen runpy>", line 88, in _run_code
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel_launcher.py", line 17, in <module>
    app.launch_new_instance()
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/traitlets/config/application.py", line 1043, in launch_instance
    app.start()
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/kernelapp.py", line 736, in start
    self.io_loop.start()
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/tornado/platform/asyncio.py", line 195, in start
    self.asyncio_loop.run_forever()
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/asyncio/base_events.py", line 607, in run_forever
    self._run_once()
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/asyncio/base_events.py", line 1922, in _run_once
    handle._run()
  File "/opt/homebrew/Cellar/python@3.11/3.11.4_1/Frameworks/Python.framework/Versions/3.
↪11/lib/python3.11/asyncio/events.py", line 80, in _run
    self._context.run(self._callback, *self._args)
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/kernelbase.py", line 516, in dispatch_queue
    await self.process_one()
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/kernelbase.py", line 505, in process_one
    await dispatch(*args)
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/kernelbase.py", line 412, in dispatch_shell
    await result
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/kernelbase.py", line 740, in execute_request
    reply_content = await reply_content
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/ipkernel.py", line 422, in do_execute
    res = shell.run_cell(
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/ipykernel/zmqshell.py", line 546, in run_cell
    return super().run_cell(*args, **kwargs)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/IPython/core/interactiveshell.py", line 3009, in run_cell
    result = self._run_cell(
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/IPython/core/interactiveshell.py", line 3064, in _run_cell
    result = runner(coro)
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/IPython/core/async_helpers.py", line 129, in _pseudo_sync_runner
    coro.send(None)
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/IPython/core/interactiveshell.py", line 3269, in run_cell_async
    has_raised = await self.run_ast_nodes(code_ast.body, cell_name,
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/IPython/core/interactiveshell.py", line 3448, in run_ast_nodes
    if await self.run_code(code, result, async_=asy):
  File "/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/IPython/core/interactiveshell.py", line 3508, in run_code
    exec(code_obj, self.user_global_ns, self.user_ns)
  File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_65477/3660210047.py",␣
↪line 14, in <module>
    logger.info("This is an informational message")
Message: 'This is an informational message'
Arguments: ()
INFO:email_logger:This is an informational message
```

Log-Formattierung

Mit Formatierern könnt ihr den Log-Meldungen Formatierungen hinzufügen.

[7]:
```
formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
```

Neben `%(asctime)s`, `%(name)s` und `%(message)s` findet ihr noch weitere Attribute in LogRecord attributes.

[8]:
```
import logging


logger = logging.getLogger("stream_logger")
logger.setLevel(logging.INFO)

console = logging.StreamHandler()
formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
console.setFormatter(formatter)

logger.addHandler(console)
logger.info("This is an informational message")
```

```
This is an informational message
2023-08-21 18:08:14,555 - stream_logger - This is an informational message
INFO:stream_logger:This is an informational message
```

Bemerkung

Das Logging-Modul ist thread-sicher. Logging funktioniert jedoch möglicherweise nicht in asynchronen Kontexten. In solchen Fällen könnt ihr jedoch den QueueHandler verwenden.

Siehe auch

Logging to a single file from multiple processes

Logging an mehrere Handler

[9]:
```
import logging


def log(path, multipleLocs=False):
    logger = logging.getLogger("Example_logger_%s" % fname)
    logger.setLevel(logging.INFO)
    fh = logging.FileHandler(path)
    formatter = logging.Formatter("%(asctime)s - %(name)s - %(message)s")
    fh.setFormatter(formatter)
    logger.addHandler(fh)

    if multipleLocs:
        console = logging.StreamHandler()
        console.setLevel(logging.INFO)
        console.setFormatter(formatter)
        logger.addHandler(console)

    logger.info("This is an informational message")
    try:
        1 / 0
    except ZeroDivisionError:
        logger.exception("You can't do that!")

    logger.critical("This is a no-brainer!")
```

Logging konfigurieren

Siehe auch

logging configuration

... in einer INI-Datei

Im folgenden Beispiel wird die Datei `development.ini` in diesem Verzeichnis geladen:

```
[loggers]
keys=root

[handlers]
keys=stream_handler

[formatters]
keys=formatter
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
[logger_root]
level=DEBUG
handlers=stream_handler

[handler_stream_handler]
class=StreamHandler
level=DEBUG
formatter=formatter
args=(sys.stderr,)

[formatter_formatter]
format=%(asctime)s %(name)-12s %(levelname)-8s %(message)s
```

[10]:
```
import logging
import logging.config

from logging.config import fileConfig


logging.config.fileConfig("development.ini")
logger = logging.getLogger("example")

logger.info("Program started")
logger.info("Done!")
```

Pro:

- Möglichkeit, die Konfiguration während des Betriebs zu aktualisieren, indem die Funktion `logging.config.listen()` verwendet wird um an einem Socket zu lauschen.
- In verschiedenen Umgebungen können unterschiedliche Konfigurationen verwendet werden, also z.B. kann in der `development.ini` `DEBUG` als Log-Level angegeben werden, während in der `production.ini` `WARN` verwendet wird.

Con:

- Weniger Kontrolle z.B. gegenüber benutzerdefinierten Filtern oder Logger, die im Code konfiguriert sind.

... in einer dictConfig

[11]:
```
import logging
import logging.config


dictLogConfig = {
    "version": 1,
    "handlers": {
        "fileHandler": {
            "class": "logging.FileHandler",
            "formatter": "exampleFormatter",
            "filename": "dict_config.log",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        }
    },
    "loggers": {
        "exampleApp": {
            "handlers": ["fileHandler"],
            "level": "INFO",
        }
    },
    "formatters": {
        "exampleFormatter": {
            "format": "%(asctime)s - %(name)s - %(levelname)s - %(message)s"
        }
    },
}
```

[12]:
```
logging.config.dictConfig(dictLogConfig)

logger = logging.getLogger("exampleApp")

logger.info("Program started")
logger.info("Done!")
```

```
2023-08-21 18:08:14,573 exampleApp   INFO     Program started
2023-08-21 18:08:14,574 exampleApp   INFO     Done!
```

Pro:

- Aktualisieren während des Betriebs

Con:

- Weniger Kontrolle als beim Konfigurieren eines Loggers im Code

... direkt im Code

[13]:
```
logger = logging.getLogger()
handler = logging.StreamHandler()
formatter = logging.Formatter(
    "%(asctime)s %(name)-12s %(levelname)-8s %(message)s"
)
handler.setFormatter(formatter)
logger.addHandler(handler)
logger.setLevel(logging.DEBUG)
```

Magic Commands

Befehl	Beschreibung
`%logstart`	Startet das Logging irgendwo in einer Session
	`%logstart [-o|-r|-t|-q] [log_name [log_mode]]`
	Wenn kein Name angegeben wird, wird `ipython_log.py` im aktuellen Verzeichnis verwendet.
	`log_mode` ist ein optionaler Parameter. Folgende Modi können angegeben werden:
	* `append` hängt die Logging-Informationen am Ende einer vorhandenen Datei an
	* `backup` benennt die vorhandene Datei um in `name~` und schreibt in `name`
	* `global` hängt die Logging-Informationen am Ende einer vorhandenen Datei im
	* `over` überschreibt eine existierende Log-Datei
	* `rotate` erstellt rotierende Log-Dateien: `name.1~`, `name.2~`, etc.
	Optionen:
	* `-o` logt auch den Output von IPython
	* `-r` logt *raw* Output
	* `-t` schreibt einen Zeitstempel vor jeden Logeintrag
	* `-q` unterdrückt die Logging-Ausgabe
`%logon`	Neustart des Logging
`%logoff`	Temporäres Beenden des Logging

Pro:

- Vollständige Kontrolle über die Konfiguration

Con:

- Änderungen in der Konfiguration erfordern eine Änderung des Quellcodes

Logs rotieren

[14]:
```
import logging
import time

from logging.handlers import RotatingFileHandler


def create_rotating_log(path):
    logger = logging.getLogger("Rotating Log")
    logger.setLevel(logging.INFO)

    handler = RotatingFileHandler(path, maxBytes=20, backupCount=5)
    logger.addHandler(handler)

    for i in range(10):
        logger.info("This is an example log line %s" % i)
        time.sleep(1.5)


if __name__ == "__main__":
    log_file = "rotated.log"
    create_rotating_log(log_file)
```

```
2023-08-21 18:08:14,580 Rotating Log INFO     This is an example log line 0
2023-08-21 18:08:14,580 Rotating Log INFO     This is an example log line 0
2023-08-21 18:08:16,086 Rotating Log INFO     This is an example log line 1
2023-08-21 18:08:16,086 Rotating Log INFO     This is an example log line 1
2023-08-21 18:08:17,595 Rotating Log INFO     This is an example log line 2
2023-08-21 18:08:17,595 Rotating Log INFO     This is an example log line 2
2023-08-21 18:08:19,103 Rotating Log INFO     This is an example log line 3
2023-08-21 18:08:19,103 Rotating Log INFO     This is an example log line 3
2023-08-21 18:08:20,612 Rotating Log INFO     This is an example log line 4
2023-08-21 18:08:20,612 Rotating Log INFO     This is an example log line 4
2023-08-21 18:08:22,122 Rotating Log INFO     This is an example log line 5
2023-08-21 18:08:22,122 Rotating Log INFO     This is an example log line 5
2023-08-21 18:08:23,632 Rotating Log INFO     This is an example log line 6
2023-08-21 18:08:23,632 Rotating Log INFO     This is an example log line 6
2023-08-21 18:08:25,137 Rotating Log INFO     This is an example log line 7
2023-08-21 18:08:25,137 Rotating Log INFO     This is an example log line 7
2023-08-21 18:08:26,646 Rotating Log INFO     This is an example log line 8
2023-08-21 18:08:26,646 Rotating Log INFO     This is an example log line 8
2023-08-21 18:08:28,155 Rotating Log INFO     This is an example log line 9
2023-08-21 18:08:28,155 Rotating Log INFO     This is an example log line 9
```

Logs zeitgesteuert rotieren

[15]:
```python
import logging
import time

from logging.handlers import TimedRotatingFileHandler


def create_timed_rotating_log(path):
    """"""
    logger = logging.getLogger("Rotating Log")
    logger.setLevel(logging.INFO)

    handler = TimedRotatingFileHandler(
        path, when="s", interval=5, backupCount=5
    )
    logger.addHandler(handler)

    for i in range(6):
        logger.info("This is an example!")
        time.sleep(75)


if __name__ == "__main__":
    log_file = "timed_rotation.log"
    create_timed_rotating_log(log_file)
```

```
2023-08-21 18:08:29,674 Rotating Log INFO     This is an example!
2023-08-21 18:08:29,674 Rotating Log INFO     This is an example!
2023-08-21 18:09:44,681 Rotating Log INFO     This is an example!
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
2023-08-21 18:09:44,681 Rotating Log INFO     This is an example!
2023-08-21 18:10:59,688 Rotating Log INFO     This is an example!
2023-08-21 18:10:59,688 Rotating Log INFO     This is an example!
2023-08-21 18:12:14,697 Rotating Log INFO     This is an example!
2023-08-21 18:12:14,697 Rotating Log INFO     This is an example!
2023-08-21 18:13:29,702 Rotating Log INFO     This is an example!
2023-08-21 18:13:29,702 Rotating Log INFO     This is an example!
2023-08-21 18:14:44,710 Rotating Log INFO     This is an example!
2023-08-21 18:14:44,710 Rotating Log INFO     This is an example!
```

Erstellen eines Logging-Dekorators

Siehe auch

How to Create an Exception Logging Decorator

Einen Logging-Filter erstellen

[16]:
```
import logging
import sys


class ExampleFilter(logging.Filter):
    def filter(self, record):
        if record.funcName == "foo":
            return False
        return True


logger = logging.getLogger("filter_example")
logger.addFilter(ExampleFilter())


def foo():
    """
    Ignore this function's log messages
    """
    logger.debug("Message from function foo")


def bar():
    logger.debug("Message from bar")


if __name__ == "__main__":
    logging.basicConfig(stream=sys.stderr, level=logging.DEBUG)
    foo()
    bar()
```

```
2023-08-21 18:15:59,730 filter_example DEBUG    Message from bar
2023-08-21 18:15:59,730 filter_example DEBUG    Message from bar
```

7.10 Code-Qualität und Komplexität überprüfen und verbessern

Bevor ihr mit dem Refactoring beginnt, solltet ihr die Komplexität eures Codes messen. Im Folgenden möchte ich euch einige Werkzeuge und Konzepte vorstellen, die die Komplexität eures Codes überprüfen und die Wartung und Pflege von Python-Paketen und anderem Quellcode vereinfachen. Häufig lässt sich zusammen mit dem *pre-commit-Framework* die Code-Qualität auch automatisiert überprüfen und verbessern.

Siehe auch:

- PyCQA Meta Documentation
- github.com/PyCQA

7.10.1 Checker

`flake8`

flake8 stellt sicher, dass euer Code größtenteils **PEP 8** folgt. Eine automatische Formatierung, z.B. mit *Black*, ist jedoch noch komfortabler. Zudem prüft `flake8` auf nicht verwendete Importe.

Installation

```
$ spack env activate python-311
$ spack install py-flake8
```

Überprüfen

```
$ flake8 path/to/your/code
```

`flake8` kann für tox konfiguriert werden in der `tox.ini`-Datei eines Pakets, z.B.:

```
[tox]
envlist = py38, py311, flake8, docs

[testenv:flake8]
basepython = python
deps =
    flake8
    flake8-isort
commands =
    flake8 src tests setup.py conftest.py docs/conf.py
```

Siehe auch:

- Configuring flake8
- flake8 error/violation codes
- pycodestyle error codes

check-manifest

check-manifest ist ein Werkzeug, mit dem ihr schnell überprüfen könnt, ob die Datei `Manifest.in` für Python-Pakete vollständig ist.

Installation

```
$ pipenv install check-manifest
```

Überprüfen

```
$ cd /path/to/MANIFEST.in
$ pipenv run check-manifest
```

oder für eine automatische Aktualisierung

```
$ pipenv run check-manifest -uv
listing source files under version control: 6 files and directories
building an sdist: check-manifest-0.7.tar.gz: 4 files and directories
lists of files in version control and sdist do not match!
missing from sdist:
  tests.py
  tox.ini
suggested MANIFEST.in rules:
  include *.py
  include tox.ini
updating MANIFEST.in

$ cat MANIFEST.in
include *.rst

# added by check_manifest.py
include *.py
include tox.ini
```

Konfiguration

Ihr könnt `check-manifest` so konfigurieren, dass bestimmte Dateimuster ignoriert werden, indem ihr einen Abschnitt `[tool.check-manifest]` in eurer `pyproject.toml`-Datei oder einen Abschnitt `[check-manifest]` in eurer `setup.cfg` oder `tox.ini`-Datei anlegt, z.B.:

```
[tool.check-manifest]
ignore = [".travis.yml"]

# setup.cfg or tox.ini
[check-manifest]
ignore =
    .travis.yml
```

`check-manifest` kennt die folgenden Optionen:

ignore
: Eine Liste von Dateinamenmustern, die von `check-manifest` ignoriert werden. Verwendet diese Option, wenn ihr Dateien in eurem Versionskontrollsystem behalten möchtet, die nicht in eurem Quelldistributionen enthalten sein sollen. Die Standardliste ist:

```
PKG-INFO
* .egg-info
* .egg-info / *
setup.cfg
.hgtags
.hgsigs
.hgignore
.gitignore
.bzrignore
.gitattributes
.github / *
.travis.yml
Jenkinsfile
* .mo
```

ignore-default-rules
: wenn `true`, dann ersetzen deine `ignore`-Angaben die Standardliste, anstatt sie zu ergänzen.

ignore-bad-ideas
: Eine Liste von Dateinamenmustern, de von der Prüfung der generierten Dateien ignoriert werden. Damit könnt ihr generierte Dateien in eurem Versionskontrollsystem behalten, auch wenn dies üblicherweise eine schlechte Idee ist.

Integration in die Versionskontrolle

Mit *pre-commit-Framework* kann *check-manifest* Teil eures Git-Workflows sein. Fügt hierfür eurer *.pre-commit-config.yaml*-Datei folgendes hinzu:

```
repos:
-   repo: https://github.com/mgedmin/check-manifest
    rev: "0.39"
    hooks:
    -   id: check-manifest
```

Mypy

Mit Mypy könnt ihr eine statische Typüberprüfung vornehmen.

Siehe auch:

- Home
- GitHub
- Docs
- PyPI
- Using Mypy in production at Spring

Installation

Mypy erfordert Python3.5. Dann kann es installiert werden, z.B. mit:

```
$ pipenv install mypy
```

Überprüfen

Dann könnt ihr es überprüfen, z.B. mit:

```
$ pipenv run mypy myprogram.py
```

Bemerkung: Obwohl Mypy mit Python3 installiert werden muss, kann es auch Python2-Code analysieren, z.B. mit:

```
$ pipenv run mypy --py2 myprogram.py
```

Pytype

Pytype ist ein statisches Analysewerkzeug, das Typen aus eurem Python-Code ableitet ohne dass Typanmerkungen notwendig sind. Es kann jedoch auch im Code stehende Type Annotations erzwingen. Obwohl Annotationen für Pytype optional sind, werden sie geprüft und angewendet, sofern sie vorhanden sind. Die von Pytype erzeugten Typ-Annotationen werden in eigenständigen `.pyi`-Dateien gespeichert, die mit merge-pyi wieder in den Python-Code eingebunden werden können. Schließlich markiert es häufige Fehler wie falsch geschriebene Attributnamen oder Funktionsaufrufe und vieles mehr, auch über Dateigrenzen hinweg.

Siehe auch:

- Home
- GitHub
- PyPI
- User guide
- FAQ

Anforderungen

- Alle gängigen Linux-Distributionen werden unterstützt
- macOS 10.7 und Xcode 8
- Windows mit WSL. Zusätzlich müssen folgende Bibliotheken installiert werden:

  ```
  $ sudo apt install build-essential python3-dev libpython3-dev
  ```

Installation

Pytype kann einfach installiert werden mit

```
$ pipenv install pytype
```

Anschließend kann die Installation überprüft werden mit

```
$ pipenv run pytype file_or_directory
```

Konfiguration

Für ein Python-Paket könnt ihr Pytype einrichten indem ihr eine `pytype.cfg`-Datei anlegt mit

```
$ pipenv run pytype --generate-config pytype.cfg
```

Diese beginnt dann z.B. mit

```
# NOTE: All relative paths are relative to the location of this file.

[pytype]

# Space-separated list of files or directories to exclude.
exclude =
    **/*_test.py
    **/test_*.py

# Space-separated list of files or directories to process.
inputs =
    .
```

Nun könnt ihr die Konfigurationsdatei euren Anforderungen entsprechend ampassen.

Zusätzliche Skripte

annotate-ast
: in-progress Type-Annotator für ASTs

merge-pyi
: Zusammenführen von Typinformationen aus einer `.pyi`- in eine Python-Datei

pytd-tool
: Parser für `.pyi`-Dateien

pytype-single
: Debugging-Tool für Pytype-Entwickler, das eine einzelne Python-Datei unter der Annahme analysiert, dass für alle Abhängigkeiten bereits `.pyi`-Dateien generiert wurden

pyxref
: cross-References-Generator

Wily

Das *Zen of Python*[1] betont in vielfältiger Weise die Komplexitätsreduktion:

> Einfach ist besser als komplex.
>
> Komplex ist besser als kompliziert.
>
> Flach ist besser als verschachtelt.

Wily ist ein Kommandozeilenwerkzeug zum Überprüfen der Komplexität von Python-Code in Tests und Anwendungen. Hierfür verwendet Wily folgende Metriken:

McCabe-Metrik
: auch zyklomatische Komplexität genannt, misst die Komplexität von Code durch die Anzahl linear unabhängiger Pfade im Kontrollflussgraphen.

  Das Software Engineering Institute der Carnegie Mellon University unterscheidet die folgenden vier Risikostufen[2]:

Zyklomatische Komplexität	Risikobewertung
1–10	einfaches Programm ohne großes Risiko
11–20	mäßiges Risiko
21–50	komplexes, hochriskantes Programm
> 50	untestbares Programm mit sehr hohem Risiko

Halstead-Metrik
: statisch analysierendes Verfahren, das aus der Anzahl der Operatoren und Operanden die Schwierigkeit des Programms, den Aufwand und die Implementierungszeit berechnet.

Wartbarkeitsindex (engl. Maintainability Index)
: basiert auf den McCabe- und Halstead-Metriken sowie der Anzahl der Codezeilen[3]:

Index	Wartbarkeit
0–25	unwartbar
25–50	besorgniserregend
50–75	verbesserungsbedürftig
75–100	Superhelden-Code

Siehe auch:

- Docs
- GitHub
- PyPI
- wily-pycharm

[1] **PEP 20** – The Zen of Python
[2] C4 Software Technology Reference Guide, S. 147
[3] Using Metrics to Evaluate Software System Maintainability

Installation

Wily kann einfach installiert werden mit

```
$ pipenv install wily
```

Anschließend könnt ihr die Installation überprüfen mit

```
$ pipenv run wily --help
Usage: wily [OPTIONS] COMMAND [ARGS]...
  Version: 1.19.0
   Inspect and search through the complexity of your source code. To get
  started, run setup:
    $ wily setup ...
```

Konfiguration

Im Projektverzeichnis kann eine `wily.cfg`-Datei angelegt werden mit der Liste der verfügbaren Operatoren:

```
[wily]
# list of operators, choose from cyclomatic, maintainability, mccabe and raw
operators = cyclomatic,raw
# archiver to use, defaults to git
archiver = git
# path to analyse, defaults to .
path = /path/to/target
# max revisions to archive, defaults to 50
max_revisions = 20
```

Auch Python-Code in `.ipynb`-Dateien wird üblicherweise automatisch erkannt. Ihr könnt dies jedoch ggf. unterbinden für ein Jupyter Notebook mit

```
ipynb_support = false
```

oder für einzelne Zellen mit

```
ipynb_cells = false
```

Verwendung

... als Kommandozeilenwerkzeug

1. Aufbau eines Caches mit den Statistiken des Projekts

   Bemerkung: Wily geht davon aus, dass euer Projektordner ein *Git*-Repository ist. Wily erstellt jedoch keinen Cache, wenn das Arbeitsverzeichnis verschmutzt ist.

   ```
   $ pipenv run wily build
   ```

2. Metrik anzeigen

```
    $ pipenv run wily report

Dies gibt sowohl die Metrik wie auch das Delta zur vorherigen Revision aus.
```

3. Rangfolge anzeigen

```
$ pipenv run wily rank
```

   Dies zeigt die Rangfolge aller Dateien in einem Verzeichnis oder einer einzelnen Datei an basierend auf der angegebenen Metrik, sofern diese in `.wily/` vorhanden ist.

4. Diagramm anzeigen

```
$ pipenv run wily graph
```

   Dies zeigt ein Diagramm im Standard-Browser an.

5. Informationen zum Build-Verzeichnis anzeigen

```
$ pipenv run wily index
```

6. Auflisten der in den Wily-Operatoren verfügbaren Metriken

```
$ pipenv run wily list-metrics
```

... als pre-commit Hook

Ihr könnt Wily auch als *pre-commit-Framework* verwenden. Hierzu müsstet ihr in der `pre-commit-config.yaml`-Konfigurationsdatei z.B. folgendes hinzufügen:

```
repos:
- repo: local
  hooks:
  - id: wily
    name: wily
    entry: wily diff
    verbose: true
    language: python
    additional_dependencies: [wily]
```

... in einer CI/CD-Pipeline

Üblicherweise vergelicht Wily die Komplexität mit der vorherigen Revision. Ihr könnt jedoch auch andere Referenzen angeben, z.B. `HEAD^1` mit

```
$ pipenv run wily build src/
$ pipenv run wily diff src/ -r HEAD^1
```

PyRe

PyRe (Python Reliability) analysiert die strukturelle Zuverlässigkeit von Python-Code und fasst sie in einem Report zusammen. Aktuell werden jedoch nur Zufälligkeitsmethoden erster Ordnung unterstützt wie Crude Monte-Carlo-Simulation und Importance Sampling.

Siehe auch:

- Docs
- GitHub

Installation

```
$ pipenv install git+git://github.com/hackl/pyre.git
```

Zuverlässigkeitsanalyse

Eine FORM (first-order reliability method)-Analyse kann z.B. zu folgendem Ergebnis führen:

```
================================================

 RESULTS FROM RUNNING FORM RELIABILITY ANALYSIS

 Number of iterations:      17
 Reliability index beta:    1.75397614074
 Failure probability:       0.039717297753
 Number of calls to the limit-state function: 164

================================================
```

Pysa

Der Python Static Analyzer Pysa führt Taint-Analysen durch um potenzielle Sicherheitsprobleme zu identifizieren. Dabei verfolgt Pysa Datenströme von ihrem Ursprung zu ihrem Endpunkt und identifiziert dabei anfälligen Code.

Siehe auch:

- What Is Taint Analysis and Why Should I Care?
- How Pysa works
- Running Pysa
- Pysa Tutorial

Konfiguration

Pysa verwendet zwei Dateitypen für die Konfiguration:

- eine `taint.config`-Datei im JSON-Format, in der `sources`, `sinks`, `features` und `rules` definiert werden.

```
{
  "comment": "UserControlled, Test, Demo sources are predefined. Same for Demo,␣
↪Test and RemoteCodeExecution sinks",
  "sources": [],
  "sinks": [],
  "features": [],
  "rules": []
}
```

- Dateien mit der Endung `.pysa` in einem Verzeichnis, das mit `taint_models_path` in eurer `.pyre_configuration`-Datei konfiguriert wurde.

Praktische Beispiele findet ihr im Pyre-Repository.

Verwendung

Pyre kann aufgerufen werden, z.B. mit

```
$ $ pipenv run pyre analyze --save-results-to ./
```

Die Option `--save-results-to` speichert detaillierte Ergebnisse in `./taint-output.json`.

Pysa Postprozessor

Installation

```
$ pipenv install fb-sapp
```

Verwendung

1. Parsen der JSON-Datei, z.B. mit

   ```
   $ pipenv run sapp --database-name sapp.db analyze ./taint-output.json
   ```

   Die Ergebnisse werden in der lokalen SQLite-Datei `sapp.db` gespeichert.

2. Erkunden der Probleme mit

   ```
   $ pipenv run sapp --database-name sapp.db explore
   ```

   Dies startet ein IPython-Interface, das mit der SQLite-Datenbank verbunden ist:

   `issues`
   : listet alle Probleme auf

   `issue 1`
   : wählt das erste Problem aus

`trace`
: zeigt den Datenfluss von `source` bis `sink` an

`n`
: springt zum nächsten Aufruf

`list`
: zeigt den Quellcode des Aufrufs

`jump 1`
: springt zum ersten Aufruf und zeigt den Quellcode an

Weitere Kommandos erhaltet ihr in Commands.

7.10.2 Formatter

Black

Black formatiert euren Code in ein schönes und deterministisches Format.

Siehe auch:

Was lesbaren Code auszeichnet, ist gut beschrieben im Trey Hunners Blog-Post Craft Your Python Like Poetry.

Installation

```
$ pipenv install black
```

Überprüfen

Anschließend könnt ihr die Installation überprüfen mit

```
$ pipenv run black /PATH/TO/YOUR/SOURCE/FILE
```

Integration

Mit jupyter-black könnt ihr Black auch bereits in euren Jupyter Notebooks verwenden.

Siehe auch:

Auch die Integration in andere Editoren wie PyCharm, Wing IDE oder Vim ist möglich, s. Editor integration

Konfiguration

Im Gegensatz zur Standardformatierung von Black mit 88 Zeichen bevorzuge ich jedoch eine Zeilenlänge von 79 Zeichen.

Hierfür könnt ihr in `pyproject.toml` folgendes eintragen:

```
[tool.black]
line-length = 79
```

Siehe auch:

Weitere Informationen zur Konfiguration von Black in der Toml-Datei erhaltet ihr in pyproject.toml.

isort

isort formatiert eure `import`-Anweisungen in getrennte und sortierte Blöcke.

Installation

```
$ pipenv install isort
```

Konfiguration

`isort` lässt sich z.B. in der `pyproject.toml`-Datei konfigurieren:

```
[tool.isort]
atomic=true
force_grid_wrap=0
include_trailing_comma=true
lines_after_imports=2
lines_between_types=1
multi_line_output=3
not_skip="__init__.py"
use_parentheses=true

known_first_party=["MY_FIRST_MODULE", "MY_SECOND_MODULE"]
known_third_party=["mpi4py", "numpy", "requests"]
```

Um Pakete von Drittanbietern gegenüber euren Projektimporten zu erkennen, könnt ihr euer Projekt zusammen mit `isort` installieren.

Bemerkung: Mit isort 5 könnt ihr Profile verwenden. Dies erleichtert die Konfiguration von isort, um auch zukünftig mit *Black* zusammenzuspielen:

```
isort --profile black .
```

---

prettier

prettier bietet automatische Formatierer für andere Dateitypen, u.a. für TypeScript, JSON, Vue, YAML, TOML und XML an.

Installation

```
$ npm install prettier --save-dev --save-exact
```

Konfiguration

```
$ npx prettier --write path/to/my/file.js
```

Pre-commit-Hook für `prettier`

Installation

```
$ npm install pretty-quick husky --save-dev
```

Konfiguration

In der `package.json`-Datei kann der Pre-commit-Hook folgendermaßen konfiguriert werden:

```
{ "husky": { "hooks": { "pre-commit": "pretty-quick --staged" } } }
```

Siehe auch:

- Prettier docs

7.10.3 Refactoring

Code-Smells und Anti-Patterns

Code-Smells sind Codierungsmuster, die darauf hinweisen, dass mit dem Entwurf eines Programms etwas nicht stimmt. Zum Beispiel ist die übermäßige Verwendung von `isinstance`-Prüfungen gegen konkrete Klassen ein Code-Smell, da das Programm dadurch schwieriger zu erweitern ist, um mit neuen Typen in der Zukunft umzugehen.

Erkennen von Code-Smells

Eine Möglichkeit, Code-Smells besser zu erkennen, besteht darin, die Merkmale von Code zu beschreiben. Notiert euch die Dinge, die euch auffallen; fügt alle Muster hinzu, die ihr seht, die ihr mögt oder nicht versteht. Möglicherweise können euch die folgenden Fragen zu weiteren Überlegungen anregen:

- Gibt es Methoden, die die gleiche Form haben?
- Gibt es Methoden, die ein Argument mit demselben Namen haben?
- Bedeuten gleichnamige Argumente immer das Gleiche?
- Wenn ihr eine private Methode einer Klasse hinzufügen wollt, wo würde sie hingehören?
- Wenn ihr die Klasse in zwei Teile aufteilen würdet, wo wäre die Trennlinie?
- Haben die Tests in den Bedingungen etwas gemeinsam?

- Wie viele Verzweigungen haben die Bedingungen?
- Enthalten die Methoden außer der Bedingung noch weiteren Code?
- Hängt jede Methode mehr vom übergebenen Argument ab oder von der Klasse als Ganzes?

SOLID-Prinzipien

SOLID ist ein Akronym für:

S – ***Single-Responsibility-Prinzip***
: Die Methoden einer Klasse sollten auf einen einzigen Zweck ausgerichtet sein.

O – ***Open-Closed-Prinzip***
: Objekte sollten offen für Erweiterungen, aber geschlossen für Änderungen sein.

L – ***Liskovsches Substitutionsprinzip***
: Unterklassen sollten durch ihre Oberklassen substituierbar sein.

I – ***Interface-Segregation-Prinzip***
: Objekte sollten nicht von Methoden abzuhängen, die sie nicht verwenden.

D – ***Dependency-Inversion-Prinzip***
: Abstraktionen sollten nicht von Details abhängen.

Open-Closed-Prinzip

Die Entscheidung, ob eine Refaktorierung durchgeführt werden soll, sollte davon abhängen, ob euer Code bereits *offen* für neue Anforderungen ist, ohne hierfür bestehenden Code ändern zu müssen. Refaktorierungen sollten nicht mit dem Hinzufügen neuer Funktionen vermischt sondern beide Vorgänge voneinander getrennt werden. Wenn ihr mit einer neuen Anforderung konfrontiert werdet, ordnet zunächst den vorhandenen Code so um, dass er für die neue Funktion offen ist, und fügt den neuen Code erst hinzu, wenn dies abgeschlossen ist.

> Unter Refaktorierung versteht man den Prozess, ein Softwaresystem so zu verändern, dass das äußere Verhalten des Codes nicht verändert, aber seine innere Struktur verbessert wird.

– Martin Fowler: Refactoring

Bemerkung: Sicheres Refactoring ist auf Tests angewiesen. Wenn ihr den Code wirklich umgestaltet, ohne das Verhalten zu ändern, sollten die vorhandenen Tests bei jedem Schritt weiterhin erfolgreich sein. Die Tests sind ein Sicherheitsnetz, das das Vertrauen in die neue Anordnung des Codes rechtfertigt. Wenn sie versagen,

- habt ihr den Code versehentlich beschädigt,
- oder die vorhandenen Tests sind fehlerhaft.

Single-Responsibility-Prinzip

Das Single-Responsibility-Prinzip besagt, dass jede Klasse nur eine Aufgabe erfüllen soll:

> Es sollte nie mehr als einen Grund geben, eine Klasse zu ändern.

– Robert C. Martin: SRP: The Single Responsibility Principle

Liskovsches Substitutionsprinzip

Das Liskovsche Substitutionsprinzip besagt, dass Unterklassen durch ihre Oberklassen ersetzbar sein müssen. Das Liskov-Substitutionsprinzip gilt auch für Duck-Typing: jedes Objekt, das behauptet, eine Ente zu sein, muss die API der Ente vollständig implementieren. Duck-Types sollten gegeneinander austauschbar sein. Die Logik über verschiedene Datentypen von Objekten hinweg anzuwenden, nennt sich Polymorphie.

Interface-Segregation-Prinzip

Das Interface-Segregation-Prinzip wendet das *Single-Responsibility-Prinzip* auf Schnittstellen an um ein bestimmtes Verhalten zu isolieren. Wenn eine Änderung an einem Teil eures Codes erforderlich ist, eröffnet die Extraktion eines Objekts, das eine Rolle spielt, die Möglichkeit, das neue Verhalten unterstützen, ohne dass der bestehende Code geändert werden muss. Dies ist kodierten Konkretisierungen vorzuziehen.

In diesem Zusammenhang ist auch das Gesetz von Demeter interessant, das besagt, dass Objekte nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren sollen. Damit wird die Liste der anderen Objekte wirksam eingeschränkt, an die ein Objekt eine Nachricht senden kann und die Kopplung zwischen Objekten verringert: ein Objekt kann nur mit seinen Nachbarn sprechen, nicht aber mit den Nachbarn seiner Nachbarn; Objekte können nur Nachrichten an direkt Beteiligte senden.

Dependency-Inversion-Prinzip

Das Dependency-Inversion-Prinzip kann definiert werden als

> Abstraktionen sollten nicht von Details abhängen. Details sollten von Abstraktionen abhängen.

– Robert C. Martin: The Dependency Inversion Principle

Typische Code-Smells in Python

Siehe auch:

- Effective Python by Brett Slatkin
- When Python Practices Go Wrong by Brandon Rhodes

Funktionen, die Objekte sein sollten

Python unterstützt neben der objektorientierten auch die prozedurale Programmierung mithilfe von Funktionen und vererbbaren Klassen. Beide Paradigmen sollten jedoch auf die passenden Probleme angewendet werden.

Typische Symptome von funktionalem Code, der in Klassen umgestaltet werden sollte, sind

- ähnliche Argumente über Funktionen hinweg
- hohe Anzahl eindeutiger Halstead-Operanden
- Mix aus mutable und immutable Funktionen

So können z.B. drei Funktionen mit unklarer Verwendung so reorganisiert werden, dass `load_image()` durch `.__init__()` ersetzt wird, `crop()` eine Klassenmethode wird und `get_thumbnail()` eine Eigenschaft:

```
class Image(object):
    thumbnail_resolution = 128
    def __init__(self, path):
        ...

    def crop(self, width, height):
        ...

    @property
    def thumbnail(self):
        ...
        return thumb
```

Objekte, die Funktionen sein sollten

Manchmal sollte jedoch auch objektorientierter Code besser in Funktionen aufgelöst werden, z.B. wenn in einer Klasse außer `.__init__()` nur eine weitere Methode oder nur statische Methoden enthalten sind.

Bemerkung: Ihr müsst nicht händisch nach solchen Klassen suchen, sondern es gibt eine pylint-Regel dafür:

```
$ pipenv run pylint --disable=all --enable=R0903 requests
************* Module requests.auth
requests/auth.py:72:0: R0903: Too few public methods (1/2) (too-few-public-methods)
requests/auth.py:100:0: R0903: Too few public methods (1/2) (too-few-public-methods)
************* Module requests.models
requests/models.py:60:0: R0903: Too few public methods (1/2) (too-few-public-methods)


-----------------------------------
Your code has been rated at 9.99/10
```

Dies zeigt uns, dass in `auth.py` zwei Klassen mit nur einer öffentlichen Methode definiert wurden und zwar in den Zeilen 72ff. und 100ff. Auch in `models.py` gibt es ab Zeile 60 eine Klasse mit nur einer öffentlichen Methode.

Verschachtelter Code

> *«Flat is better than nested.»*

– Tim Peters, Zen of Python

Verschachtelter Code erschwert das Lesen und Verstehen. Ihr müsst die Bedingungen verstehen und merken, wenn ihr durch die Zweige geht. Objektiv erhöht sich die zyklomatische Komplexität bei steigender Anzahl der Code-Verzweigungen.

Ihr könnt verschachtelte Methoden mit mehreren ineinandergesteckten `if`-Anweisungen reduzieren, indem ihr Ebenen durch Methoden ersetzt, die GGF. `False` zurückgeben. Anschließend könnt ihr mit `.count()` überprüfen, ob die Anzahl der Fehler > `0` ist.

Eine andere Möglichkeit besteht in der Verwendung von *List Comprehensions*. So kann der Code

```
results = []
for item in iterable:
    if item == match:
        results.append(item)
```

ersetzt werden durch:

```
results = [item for item in iterable if item == match]
```

Bemerkung: Die itertools der Python-Standardbibliothek sind häufig ebenfalls gut geeignet, um die Verschachtelungstiefe zu reduzieren indem Funktionen zum Erstellen von Iteratoren aus Datenstrukturen erstellt werden.

Bemerkung: Zudem könnt ihr mit den itertools auch filtern, z.B. mit filterfalse:

```
>>> from itertools import filterfalse
>>> from math import isnan
>>> from statistics import median
>>> data = [20.7, float('NaN'),19.2, 18.3, float('NaN'), 14.4]
>>> sorted(data)
[20.7, nan, 14.4, 18.3, 19.2, nan]
>>> median(data)
16.35
>>> sum(map(isnan, data))
2
>>> clean = list(filterfalse(isnan, data))
>>> clean
[20.7, 19.2, 18.3, 14.4]
>>> sorted(clean)
[14.4, 18.3, 19.2, 20.7]
>>> median(clean)
18.75
```

Query-Tools für komplexe Dicts

JMESPath, glom, asq und flupy können die Abfrage von Dicts in Python deutlich vereinfachen.

Code reduzieren mit `dataclasses` und `attrs`

dataclasses
: sollen die Definition von Klassen vereinfachen, die hauptsächlich zum Speichern von Werten erstellt werden, und auf die dann über die Attributsuche zugegriffen werden kann. Einige Beispiele sind `collections.namedtuple()`, `typing.NamedTuple`, Rezepte zu Records und Verschachtelte Dicts. `dataclasses` ersparen euch das Schreiben und Verwalten dieser Methoden.

  Siehe auch:

  - **PEP 557** – Data Classes

attrs
: ist ein Python-Paket, das es schon viel länger als `dataclasses` gibt, umfangreicher ist und auch mit älteren Versionen von Python verwendet werden kann.

Rope

Rope ist eine Python-Refactoring-Bibliothek.

Installation

Rope kann einfach installiert werden mit

```
$pipenv install rope
```

Nutzung

Nun importieren wir zunächst den `Project`-Typ und instanziieren ihn mit dem Pfad zum Projekt:

```
[1]: from rope.base.project import Project


     proj = Project("requests")
```

Dies erstellt dann einen Projektordner mit dem Namen `.ropeproject` in unserem Projekt.

```
[2]: [f.name for f in proj.get_files()]
```

```
[2]: ['hooks.py',
      'utils.py',
      '_internal_utils.py',
      'status_codes.py',
      '__version__.py',
      'sessions.py',
      'api.py',
      'cookies.py',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
 'adapters.py',
 'certs.py',
 'exceptions.py',
 'api_v1.py',
 'auth.py',
 'help.py',
 'structures.py',
 'compat.py',
 'packages.py',
 '__init__.py',
 'models.py']
```

Die `proj`-Variable kann eine Reihe von Befehlen ausführen wie `get_files` und `get_file`. Im folgenden Beispiel nutzen wir dies um der Datei `api.py` die Variable `api` zuzuweisen.

[3]:
```
!git clone -q https://github.com/psf/requests.git && cp requests/src/requests/api.py␣
↪requests/src/requests/api_v1.py
```

[4]:
```
api = proj.get_file("api.py")
```

[5]:
```
from rope.refactor.rename import Rename


change = Rename(proj, api).get_changes("api_v1")

proj.do(change)
```

[6]:
```
!cd requests && git status
```

```
Auf Branch main
Änderungen, die nicht zum Commit vorgemerkt sind:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
  (benutzen Sie "git restore <Datei>...", um die Änderungen im Arbeitsverzeichnis zu␣
↪verwerfen)
        geändert:       __init__.py

Unversionierte Dateien:
  (benutzen Sie "git add <Datei>...", um die Änderungen zum Commit vorzumerken)
        .ropeproject/
        api_v1.py

keine Änderungen zum Commit vorgemerkt (benutzen Sie "git add" und/oder "git commit -a")
```

[7]:
```
!cd requests && git diff __init__.py
```

```
diff --git a/__init__.py b/__init__.py
index f8f9429..502e33a 100644
--- a/__init__.py
+++ b/__init__.py
@@ -118,7 +118,7 @@ from .__version__ import __copyright__, __cake__
 from . import utils
 from . import packages
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
 from .models import Request, Response, PreparedRequest
-from .api import request, get, head, post, patch, put, delete, options
+from .api_v1 import request, get, head, post, patch, put, delete, options
 from .sessions import session, Session
 from .status_codes import codes
 from .exceptions import (
```

Mit `proj.do(change)` ist also die Datei `requests/__init__.py` so geändert worden, dass von `new_api` anstatt von `api` importiert wird.

Rope kann nicht nur zum Umbenennen von Dateien verwendet werden, sondern auch für hunderte anderer Fälle; siehe auch Rope Refactorings.

Siehe auch:

- Martin Fowler: Refactoring

7.11 Sicherheit

In den vorherigen Kapiteln haben wir schon einige Hinweise gegeben, die einen sichereren Betrieb ermöglichen. Hier wollen wir die einzelnen Elemente nun nochmal zusammenfassen und erweitern. Dabei orientieren wir uns an der OpenSSF Scorecard. Alternativ könnt ihr euch auch an *ISO/IEC 5230/OpenChain* orientieren.

7.11.1 Schwachstellen überprüfen

Risiko: Hoch

Mit dieser Prüfung wird festgestellt, ob das Projekt offene, nicht behobene Sicherheitslücken in seiner eigenen Codebasis oder in seinen Abhängigkeiten aufweist. Eine offene Sicherheitslücke kann leicht ausgenutzt werden und sollte so schnell wie möglich geschlossen werden.

Für eine solche Überprüfung könnt ihr z.B. *pipenv check* verwenden, das die Python-Bibliothek safety verwendet. Alternativ könnt ihr auch osv oder pip-audit verwenden, das auf die Open Source Vulnerability Database zurückgreift.

Wenn eine Schwachstelle in einer Abhängigkeit gefunden wird, solltet ihr auf eine Version nicht-anfällige Version aktualisieren; wenn kein Update verfügbar ist, solltet ihr überlegen, die Abhängigkeit zu entfernen.

Wenn ihr glaubt, dass die Sicherheitslücke euer Projekt nicht betrifft, kann für `osv` eine `osv-scanner.toml`-Datei erstellt werden, U.A. mit der zu ignorierenden ID und einer Begründung, z.B.:

```
[[IgnoredVulns]]
id = "GO-2022-1059"
# ignoreUntil = 2022-11-09 # Optional exception expiry date
reason = "No external http servers are written in Go lang."
```

7.11.2 Wartung

Werden die Abhängigkeiten automatisch aktualisiert?

Risiko: Hoch

Veraltete Abhängigkeiten machen ein Projekt anfällig für Angriffe auf bekannte Schwachstellen. Daher sollte der Prozess der Aktualisierung von Abhängigkeiten automatisiert werden, indem nach veralteten oder unsicheren Anforderungen gesucht und GGF. aktualisiert werden. Hierfür könnt ihr z.B. dependabot oder PyUp verwenden.

Ihr könnt eure *Pipenv*-Umgebungen auch automatisch mit *pipenv update* aktualisieren.

Werden die Abhängigkeiten noch gewartet?

Risiko: Hoch

Dies weist auf möglicherweise ungepatchte Sicherheitslücken hin. Daher sollte regelmäßig überprüft werden, ob ein Projekt archiviert wurde. Umgekehrt wird bei der OSSF-Scorecard davon ausgegangen, dass bei mindestens einem Commit in der Woche über 90 Tage hinweg das Projekt sehr aktiv gewartet wird. Ein Mangel an aktiver Wartung ist jedoch nicht unbedingt immer ein Problem: insbesondere kleinere Dienstprogramme müssen normalerweise nicht oder nur sehr selten gewartet werden. Fehlende aktive Wartung weist euch also nur darauf hin, dass ihr die Situation genauer untersuchen solltet.

Ihr könnt euch die Aktivitäten eines Projekts auch mit Badges anzeigen lassen, z.B.:

Gibt es ein Sicherheitskonzept für das Projekt?

Risiko: Mittel

Idealerweise sollte mit dem Projekt eine Datei `SECURITY.md` O.Ä. (oder ähnliches) veröffentlicht worden sein. Diese Datei sollte Informationen enthalten,

- wie eine Sicherheitslücke gemeldet werden kann ohne dass sie öffentlich sichtbar wird,
- über den Ablauf und den Zeitplan für die Offenlegung der Schwachstelle,
- zu Links, z.B. URLs und E-Mails, unter denen Unterstützung angefragt werden kann.

Siehe auch:

- Guide to implementing a coordinated vulnerability disclosure process for open source projects
- Adding a security policy to your repository
- Runbook

Enthält das Projekt eine verwendbare Lizenz?

Risiko: Niedrig

Eine *Lizenz* weist darauf hin, wie der Quellcode verwendet werden darf oder nicht. Das Fehlen einer Lizenz erschwert jede Art von Sicherheitsüberprüfung oder Audit und stellt ein rechtliches Risiko für die potenzielle Nutzung dar.

OSSF-Scorecard verwendet die GitHub License API für auf GitHub gehostete Projekte, ansonsten eine eigene Heuristik, um eine veröffentlichte Lizenzdatei zu erkennen. Dateien in einem `LICENSES`-Verzeichnis sollten mit mit ihrem *SPDX*-Lizenzbezeichner benannt werden, gefolgt von einer entsprechenden Dateierweiterung, wie in der *REUSE*-Spezifikation beschrieben.

Wird nach den Best Practices der Core Infrastructure Initiative (CII) gehandelt?

Risiko: Niedrig

Das Core Infrastructure Initiative (CII) Best Practices Program umfasst eine Reihe von sicherheitsorientierten Best Practices für die Entwicklung von Open-Source-Software:

- das Verfahren zur Meldung von Schwachstellen ist auf der Projektseite veröffentlicht
- ein funktionierendes Build-System erstellt die Software automatisch aus dem Quellcode neu
- eine allgemeine Richtlinie, dass Tests zu einer automatisierten Testsuite hinzugefügt werden, wenn wichtige neue Funktionen hinzukommen
- GGF. verschiedene Kryptographie-Kriterien werden erfüllt
- mindestens ein statisches Code-Analyse-Tool, das auf jede geplante größere Produktionsversion angewendet wird

Mit dem OpenSSF Best Practices Badge Programm könnt ihr euch auch ein entsprechendes Badge holen.

7.11.3 Kontinuierliches Testen

Werden im Projekt CI-Tests durchgeführt?

Risiko: Niedrig

Bevor Code in Pull- oder Merge-Requests zusammengeführt wird, sollten Tests durchgeführt werden, die dabei helfen, Fehler frühzeitig zu erkennen und die Anzahl der Schwachstellen in einem Projekt zu reduzieren.

Verwendet das Projekt Fuzzing-Tools?

Risiko: Mittel

Fuzzing oder Fuzz-Testing übergibt unerwartete oder zufällige Daten an euer Programm, um Fehler zu entdecken. Regelmäßiges Fuzzing ist wichtig, um Schwachstellen aufzuspüren, die von anderen ausgenutzt werden können, zumal auch bei einem Angriff Fuzzing genutzt werden kann, um dieselben Schwachstellen zu finden.

- Verwendet euer Projekt Fuzzing?
- Ist der Name des Repository in der OSS-Fuzz-Projektliste enthalten?
- Wird ClusterFuzzLite im Repository eingesetzt?
- Sind benutzerdefinierte sprachenspezifische Fuzzing-Funktionen im Repository vorhanden, z.B. mit atheris oder OneFuzz?

Verwendet euer Projekt Werkzeuge zur statischen Codeanalyse?

Risiko: Mittel

Statische Codeanalysen testen den Quellcode, bevor die Anwendung ausgeführt wird. Dies kann verhindern, dass bekannte Fehlerklassen versehentlich in die Codebasis eingeführt werden.

Um Schwachstellen zu überprüfen, könnt ihr bandit verwenden, das ihr auch in eure `.pre-commit-hooks.yaml` integrieren könnt:

```
repos:
- repo: https://github.com/PyCQA/bandit
  rev: '1.7.5'
  hooks:
  - id: bandit
```

Zudem könnt ihr *Pysa* für Taint-Analysen verwenden.

Für GitHub-Repositories könnt ihr alternativ auch CodeQL verwenden; s.a. codeql-action.

7.11.4 Risikobewertung des Quellcodes

Ist das Projekt frei von eingecheckten Binärdateien?

Risiko: Hoch

Generierte ausführbare Dateien im Quellcode-Repository (z.B. Java `.class`-Dateien, Python `.pyc` Dateien) erhöhen das Risiko, da sie schwer überprüft werden können, so dass sie veraltet oder böswillig manipuliert sein können. Diesen Problemen kann mit verifizierten, reproduzierbaren Builds begegnet werden, deren ausführbare Dateien jedoch nicht wieder im Quellcode-Repository landen sollten.

Siehe auch:

- Reproducible Builds
- Python 3.12.0 from a supply chain security perspective
- Defending against the PyTorch supply chain attack PoC

Ist der Entwicklungsprozess anfällig für das Einschleusen von bösartigem Code?

Risiko: Hoch

Mit *geschützten Git-Zweigen* können Regeln für die Übernahme von Änderungen in Standard- und Veröffentlichungszweige definiert werden, z.B. automatisierte statische Code-Analysen mit *flake8*, *Pysa*, *Wily* und *Code-Reviews* über sog. *Merge-Requests*.

Werden Code-Reviews durchgeführt?

Risiko: Hoch

Mit Code-Reviews lassen sich unbeabsichtigte Schwachstellen oder das mögliche Einschleusen von bösartigem Code erkennen. Ggf. können so Angriffe aufgespürt werden, bei denen das Konto eines Teammitglieds unterwandert wurde.

Wirken an dem Projekt Personen aus mehreren Organisationen mit?

Risiko: Niedrig

Dies wird als Indiz für eine geringere Anzahl von vertrauenswürdigen Code-Reviewers gewertet. Hierfür kann in den Profilen nach unterschiedlichen Einträgen im Feld *Unternehmen* gesucht werden. Wünschenswert sind mindestens drei verschiedene Unternehmen in den letzten 30 Commits, wobei jedes dieser Teammitglieder mindestens fünf Commits gemacht haben sollte.

7.11.5 Risikobewertung der Builds

Werden im Projekt Abhängigkeiten deklariert und festgeschrieben?

Risiko: Mittel

In eurem Projekt sollten Abhängigkeiten, die während des Build- und Release-Prozesses verwendet werden, festgeschrieben werden. Dabei sollte eine *gepinnte Abhängigkeit* explizit auf einen bestimmten Hash gesetzt sein und nicht nur auf eine veränderbare Version oder einen Versionsbereich.

Spack schreibt für die jeweilige Umgebung diese Hashes in *spack.lock*, *Pipenv* in *Pipfile.lock* fest. Diese Dateien sollten daher ebenfalls mit dem Quellcode eingecheckt werden.

Hierdurch können die folgenden Sicherheitsrisiken verringert werden:

- Die Prüfung und Bereitstellung erfolgt mit derselben Software, was die Risiken beim Deployment verringert, die Fehlersuche vereinfacht und Reproduzierbarkeit ermöglicht.
- Kompromittierte Abhängigkeiten untergraben nicht die Sicherheit des Projekts.
- Substitutionsangriffe, also Angriffe, die auf die Verwechslung von Abhängigkeiten abzielen, kann so entgegengewirkt werden.

Das Festschreiben der Abhängigkeiten sollte jedoch Software-Updates nicht verhindern. Ihr könnt dieses Risiko verringern durch

- automatisierte Werkzeuge, die euch benachrichtigen, wenn Abhängigkeiten in eurem Projekt veraltet sind
- Anwendungen, die Abhängigkeiten festhalten, schnell aktualisieren.

KAPITEL 8

Web-Applikationen erstellen

Im Folgenden stelle ich euch drei verschiedene Arten von Web-Anwendungen vor:

- aus Jupyter-Notebooks generierte Dashboards
- über die Notebooks hinausgehende Web-Anwendungen, die z.B. Bokeh-Plots einbinden, wie in Bokeh-Plots in Flask einbinden demonstriert
- schließlich die Bereitstellung eurer Daten über eine RESTful API, z.B. mit dem *FastAPI-Framework*.

KAPITEL 9

Stichwortverzeichnis

Sonderzeichen

A

B

C

E

F

G

H

I

K

L

M

O

P

R

S

T

V

W

X

Z