PyViz Tutorial

Release 24.1.0

Veit Schiele

16.04.2024

Inhaltsverzeichnis

Das Tutorial gibt einen Überblick über die verschiedenen Python-Bibliotheken, die ihr zur Datenvisualisierung einsetzen könnt. Es ist entstanden aus den cusy-Schulungen zur Datenvisualisierung mit Python.

Aktuell ist es sehr schwierig, einen *Überblick über die Bibliotheken zur Datenvisualisierung* zu erhalten. Im Folgenden versuchen wir, die Suche nach der passenden Bibliothek zu vereinfachen, indem wir einige Aspekte genauer beleuchten:

- *Technologien*
- *Zentrale Bibliotheken*
- *pandas .plot()-API*
- *Weitere High-Level-APIs*
- *Rendern großer Datenmengen*
- *Diagrammtypen* (Statistische Darstellungen, regelmäßige und unregelmäßige Gitter ETC. (et cetera))

Anschließend geben wir eine praktische Einführung in die gebräuchlichsten Python-Bibliotheken.

Wir behandeln hier jedoch nicht Design-Prinzipien, Data-Storytelling oder die Auswahl des passenden Diagrammtyps. Hierzu verweisen wir jedoch gerne auf die folgenden Quellen:

Siehe auch:

- Data Visualisation Guide
- Graphical Perception: Theory, Experimentation, and Application to the Development of Graphical Methods
- Financial Times Chart Doctor: Visual vocabluary
- The Data Visualisation Catalogue
- Cartography Guide
- Xenographics

KAPITEL 1

Schnelleinstieg

1.1 Installation

1. Herunterladen und Auspacken:

```
$ curl -O https://codeload.github.com/veit/pyviz-tutorial/zip/refs/heads/main
$ unzip main
Archive:  main
...
   creating: pyviz-tutorial-main/
...
```

2. Pandoc installieren:

   - für Ubuntu und Debian:

     ```
     $ sudo apt install pandoc
     ```

   - für Mac OSX:

     ```
     $ brew install pandoc
     ```

3. Python-Pakete installieren:

```
$ cd pyviz-tutorial
$ python3 -m venv .
$ bin/python -m pip install  --upgrade pip setuptools
$ bin/python -m pip install -r requirements.txt
```

4. HTML-Dokumentation erstellen:

```
$ bin/sphinx-build -b html docs/ docs/_build/
```

5. PDF erstellen:

   Für die Erstellung von PDFs benötigt ihr weitere Pakete.

   Für Debian/Ubuntu erhaltet ihr diese mit:

```
$ apt-get install texlive-latex-recommended texlive-latex-extra texlive-fonts-
↪recommended latexmk
```

   oder für Mac OS X mit:

```
$ brew cask install mactex
...
  mactex was successfully installed!
$ curl --remote-name https://www.tug.org/fonts/getnonfreefonts/install-
↪getnonfreefonts
$ sudo texlua install-getnonfreefonts
...
mktexlsr: Updating /usr/local/texlive/2020/texmf-dist/ls-R...
mktexlsr: Done.
```

   Anschließend könnt ihr ein PDF generieren mit:

```
$ cd docs/
$ make latexpdf
...
The LaTeX files are in _build/latex.
Run 'make' in that directory to run these through (pdf)latex
...
```

   Das PDF findet ihr anschließend in `docs/_build/latex/pyviz-tutorial.pdf`.

1.2 Folge uns

- GitHub
- Twitter
- Mastodon

1.3 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.

KAPITEL 2

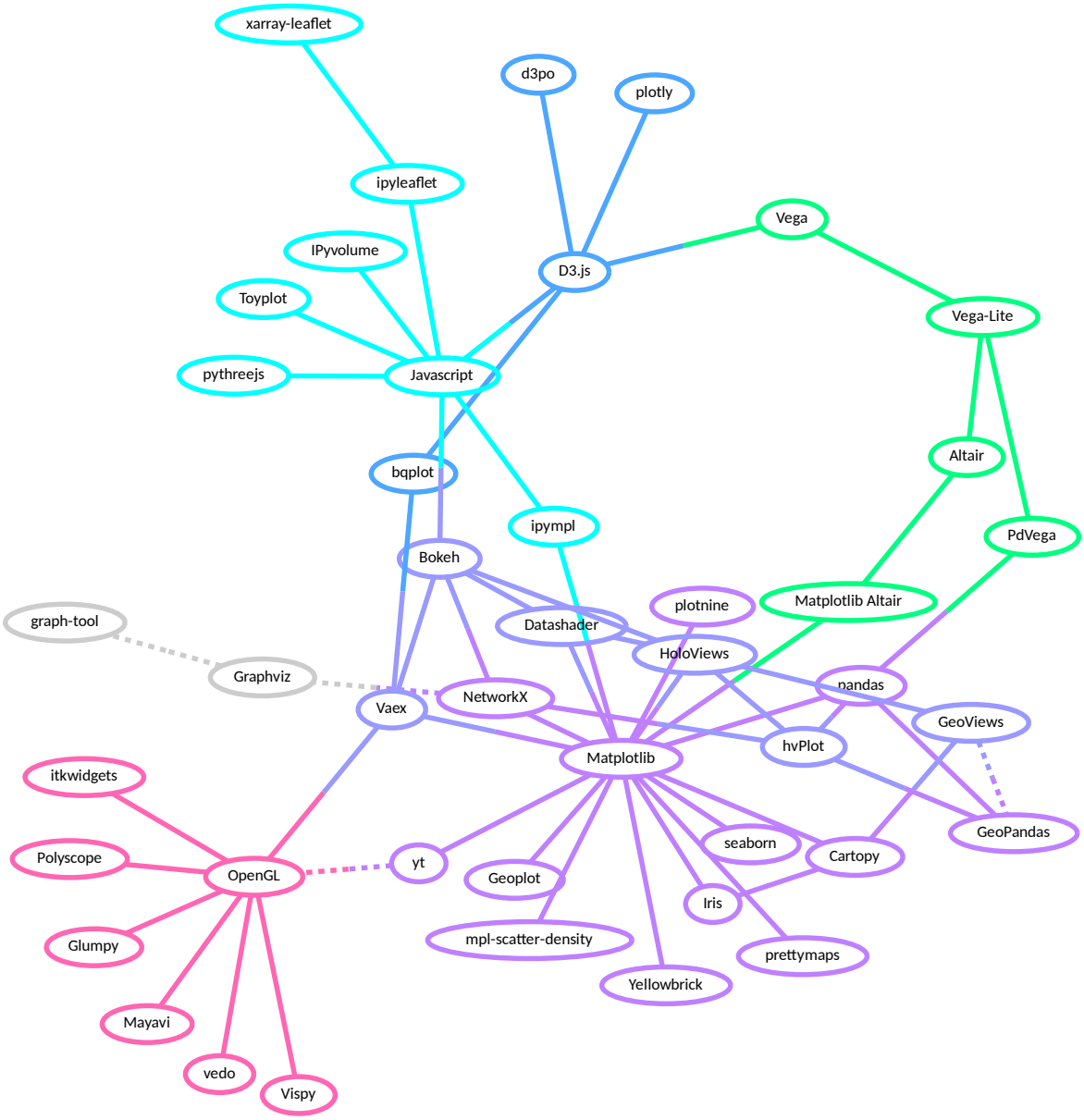
Überblick über die Bibliotheken zur Datenvisualisierung

Zunächst erhaltet ihr einen grafischen Überblick auf Basis der zentralen Python-Bibliotheken zur Datenvisualisierung.

Siehe auch:

- Jake VanderPlas: Python's Visualization Landscape (PyCon 2017)
- The Data Visualisation Catalogue

2.1 Technologien



Im Folgenden findet ihr tabellarische Übersichten über die verschiedenen Python-Bibliotheken zur Datenvisualisierung, ihre Aktivitäten und Lizenzen, so dass ihr einen ersten Anhaltspunkt erhalten könnt, ob die Bibliotheken euren Anforderungen entspricht.

Siehe auch:

Lizenzieren

2.2 Zentrale Bibliotheken

Tab. 1: GitHub-Insights: Zentrale Bibliotheken

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Matplotlib	Stars 19k	contributors 417	commit activity 3.1k/year	license not specified
bokeh	Stars 19k	contributors 391	commit activity 380/year	license BSD-3-Clause
plotly				

2.3 pandas `.plot()`-API

Tab. 2: GitHub-Insights: pandas `.plot()`-API

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
pandas	Stars 42k	contributors 412	commit activity 2.6k/year	license BSD-3-Clause
xarray	Stars 3.4k	contributors 414	commit activity 545/year	license Apache-2.0
Pandas-Bokeh	Stars 876	contributors 10	commit activity 0/year	license MIT
hvplot	Stars 935	contributors 45	commit activity 106/year	license BSD-3-Clause

2.4 Weitere High-Level-APIs

Tab. 3: GitHub-Insights: Weitere High-Level-APIs

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
seaborn	Stars 12k	contributors 185	commit activity 98/year	license BSD-3-Clause
altair	Stars 8.9k	contributors 144	commit activity 200/year	license BSD-3-Clause
perspective	Stars 7.5k	contributors 81	commit activity 404/year	license Apache-2.0
plotnine	Stars 3.8k	contributors 85	commit activity 393/year	license MIT
bqplot	Stars 3.6k	contributors 53	commit activity 18/year	license Apache-2.0
chartify	Stars 3.5k	contributors 17	commit activity 4/year	license Apache-2.0
holoviews	Stars 2.6k	contributors 127	commit activity 256/year	license BSD-3-Clause
vega	Stars 11k	contributors 133	commit activity 91/year	license BSD-3-Clause
Vega-Lite	Stars 4.5k	contributors 185	commit activity 275/year	license BSD-3-Clause
AutoViz	Stars 1.6k	contributors 11	commit activity 49/year	license Apache-2.0
Lets-Plot	Stars 1.4k	contributors 19	commit activity 854/year	license MIT
proplot	Stars 1.1k	contributors 13	commit activity 3/year	license MIT
ipyvizzu	Stars 923	contributors 19	commit activity 259/year	license Apache-2.0
ipyvizzu-story	Stars 318	contributors 8	commit activity 126/year	license Apache-2.0
toyplot	Stars 425	contributors 9	commit activity 14/year	license not identifiable by github
quibbler	Stars 305	contributors 4	commit activity 0/year	license MIT
omniplot	Stars 16	contributors 1	commit activity 89/year	license Apache-2.0

2.5 Rendern großer Datenmengen

Die Architektur und die zugrundeliegende Technologie für jede Bibliothek bestimmen die unterstützten Datengrößen und somit, ob die Bibliothek für mehrdimensionale Arrays, lange Zeitreihen oder andere große Datasets geeignet ist:

2.5.1 OpenGL-basierte Bibliotheken

Sie können I.A. (im Allgemeinen) sehr große Datensätze (mehrere Gigabyte) verarbeiten.

Tab. 4: GitHub-Insights: OpenGL-basierte Bibliotheken

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
VisPy	Stars 3.2k	contributors 159	commit activity 193/year	license not identifiable by github
vedo	Stars 1.9k	contributors 29	commit activity 539/year	license MIT
Polysco-pe	Stars 1.6k	contributors 23	commit activity 150/year	license MIT
Mayavi	Stars 1.2k	contributors 58	commit activity 60/year	license not identifiable by github
Glumpy	Stars 1.2k	contributors 51	commit activity 4/year	license BSD-3-Clause
itkwid-gets	Stars 571	contributors 7	commit activity 260/year	license Apache-2.0

2.5.2 Matplotlib-basierte Bibliotheken

Sie können I.D.R. (in der Regel) Hunderttausende von Punkten mit angemessener Leistung verarbeiten oder in bestimmten Sonderfällen (z.B. (zum Beispiel) abhängig vom *Backend*) mehr.

Tab. 5: GitHub-Insights: Matplotlib-basierte Bibliotheken

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
mpl-scatter-density	Stars 492	contributors 3	commit activity 12/year	license BSD-2-Clause

2.5.3 Javascript-basierte Bibliotheken

Sie sind ohne besondere Behandlung beschränkt auf einige tausend bis hunderttausend Punkte. ipywidgets, *Bokeh* und Plotly nutzen statt JSON jedoch spezielle Transportmechanismen für Binärdaten, sodass sie hunderttausende bis Millionen von Datenpunkten verarbeiten können. Andere Bibliotheken wie *ipyvolume*, Plotly und in einigen Fällen *Bokeh* nutzen WebGL, sodass sie bis zu einer Millionen Datenpunkte verarbeiten können.

Tab. 6: GitHub-Insights: WebGL-basierte Bibliotheken

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Bokeh	Stars 19k	contributors 391	commit activity 380/year	license BSD-3-Clause
Plotly				
pythreejs	Stars 923	contributors 25	commit activity 0/year	license not identifiable by github
jupyter-scatter	Stars 296	contributors 7	commit activity 96/year	license Apache-2.0

2.5.4 Server-side Rendering

datashader oder Vaex ermöglichen Milliarden, Billionen oder mehr Datenpunkte.

Tab. 7: GitHub-Insights: Server-side Rendering

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
vaex	Stars 8.2k	contributors 63	commit activity 27/year	license MIT
datashader	Stars 3.2k	contributors 49	commit activity 83/year	license BSD-3-Clause

2.6 Weitere Bibliotheken

Tab. 8: GitHub-Insights: Weitere Bibliotheken

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Facets	Stars 7.3k	contributors 31	commit activity 3/year	license Apache-2.0
scikit-image	Stars 5.9k	contributors 358	commit activity 409/year	license not identifiable by github
Yellow-brick	Stars 4.2k	contributors 109	commit activity 1/year	license Apache-2.0
missingno	Stars 3.8k	contributors 16	commit activity 0/year	license MIT
napari	Stars 2k	contributors 153	commit activity 584/year	license BSD-3-Clause
Hyper-Tools	Stars 1.8k	contributors 17	commit activity 16/year	license MIT
ipympl	Stars 1.5k	contributors 32	commit activity 10/year	license BSD-3-Clause
ArviZ	Stars 1.5k	contributors 150	commit activity 55/year	license Apache-2.0
MetPy	Stars 1.2k	contributors 75	commit activity 790/year	license BSD-3-Clause
iris	Stars 610	contributors 95	commit activity 340/year	license BSD-3-Clause
yt	Stars 439	contributors 147	commit activity 748/year	license not identifiable by github

2.7 Farbkarten

Tab. 9: GitHub-Insights: Farbkarten

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
palettable	Stars 737	contributors 13	commit activity 8/year	license not identifiable by github
colorcet	Stars 664	contributors 15	commit activity 9/year	license not identifiable by github
CMasher	Stars 383	contributors 8	commit activity 132/year	license BSD-3-Clause
cmocean	Stars 221	contributors 21	commit activity 30/year	license MIT
distincti-py	Stars 220	contributors 4	commit activity 21/year	license MIT
viscm	Stars 158	contributors 14	commit activity 56/year	license MIT
cmcra-meri	Stars 112	contributors 9	commit activity 29/year	license not identifiable by github

2.8 Diagrammtypen

2.8.1 Statistische Darstellungen

Streudiagramme, Linien, Flächen, Balken, Histogramme

Tab. 10: GitHub-Insights: Statistische Darstellungen

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
seaborn	Stars 12k	contributors 185	commit activity 98/year	license BSD-3-Clause
bqplot	Stars 3.6k	contributors 53	commit activity 18/year	license Apache-2.0
Matplotlib Al-tair	Stars 12	contributors 5	commit activity 0/year	license not identifiable by github

2.8.2 Regelmäßige Gitter mit rechteckigen Maschen

Tab. 11: GitHub-Insights: Regelmäßige Gitter mit rechteckigen Maschen

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Matplotlib	Stars 19k	contributors 417	commit activity 3.1k/year	license not specified
bokeh	Stars 19k	contributors 391	commit activity 380/year	license BSD-3-Clause
Plotly				
datashader	Stars 3.2k	contributors 49	commit activity 83/year	license BSD-3-Clause
Holo-Views	Stars 2.6k	contributors 127	commit activity 256/year	license BSD-3-Clause

2.8.3 Unregelmäßige 2D-Netze (Dreiecksgitter)

Tab. 12: GitHub-Insights: Unregelmäßige 2D-Netze (Dreiecksgitter)

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Matplotlib	Stars 19k	contributors 417	commit activity 3.1k/year	license not specified
bokeh	Stars 19k	contributors 391	commit activity 380/year	license BSD-3-Clause
datashader	Stars 3.2k	contributors 49	commit activity 83/year	license BSD-3-Clause
Holo-Views	Stars 2.6k	contributors 127	commit activity 256/year	license BSD-3-Clause

2.8.4 Geographie

Tab. 13: GitHub-Insights: Geographie

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Plotly				
prettymaps	Stars 11k	contributors 13	commit activity 0/year	license AGPL-3.0
kepler.gl				
folium	Stars 6.7k	contributors 153	commit activity 159/year	license MIT
OSMnx	Stars 4.7k	contributors 70	commit activity 761/year	license MIT
geopandas	Stars 4.2k	contributors 210	commit activity 184/year	license BSD-3-Clause
datashader	Stars 3.2k	contributors 49	commit activity 83/year	license BSD-3-Clause
geemap	Stars 3.2k	contributors 49	commit activity 271/year	license MIT
leafmap	Stars 2.9k	contributors 31	commit activity 238/year	license MIT
ipyleaflet	Stars 1.4k	contributors 79	commit activity 29/year	license MIT
cartopy	Stars 1.3k	contributors 111	commit activity 258/year	license BSD-3-Clause
geoplot	Stars 1.1k	contributors 6	commit activity 2/year	license MIT
PyGMT	Stars 713	contributors 48	commit activity 504/year	license BSD-3-Clause
GeoViews	Stars 562	contributors 29	commit activity 94/year	license BSD-3-Clause
Pyrosm	Stars 339	contributors 4	commit activity 33/year	license MIT
EOmaps	Stars 310	contributors 6	commit activity 994/year	license BSD-3-Clause
mapwidget	Stars 209	contributors 1	commit activity 0/year	license MIT
splot	Stars 97	contributors 40	commit activity 2/year	license BSD-3-Clause
Gspatial Plot	Stars 9	contributors 1	commit activity 10/year	license MIT
xarray-leaflet	Stars 7	contributors 5	commit activity 0/year	license MIT

2.8.5 Graphen und Netzwerke

Tab. 14: GitHub-Insights: Graphen und Netzwerke

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Plotly				
networkx	Stars 14k	contributors 423	commit activity 412/year	license not identifiable by github
datashader	Stars 3.2k	contributors 49	commit activity 83/year	license BSD-3-Clause
HoloViews	Stars 2.6k	contributors 127	commit activity 256/year	license BSD-3-Clause
graphviz	Stars 1.6k	contributors 22	commit activity 49/year	license MIT
python-igraph	Stars 1.2k	contributors 65	commit activity 257/year	license GPL-2.0
Pandas-Bokeh	Stars 876	contributors 10	commit activity 0/year	license MIT
pyvis	Stars 909	contributors 25	commit activity 0/year	license BSD-3-Clause
pydot	Stars 863	contributors 26	commit activity 42/year	license MIT
PyGraphviz	Stars 729	contributors 47	commit activity 54/year	license not identifiable by github
nxviz	Stars 448	contributors 20	commit activity 0/year	license MIT
py4cytoscape	Stars 60	contributors 9	commit activity 33/year	license not identifiable by github
ipycytoscape	Stars 258	contributors 30	commit activity 1/year	license BSD-3-Clause
Py3Plex	Stars 155	contributors 5	commit activity 14/year	license BSD-3-Clause
ipysigma	Stars 169	contributors 5	commit activity 17/year	license MIT
ipydagred3	Stars 68	contributors 3	commit activity 38/year	license Apache-2.0

2.8.6 3D-Darstellungen (Netze, Streudiagramme)

Tab. 15: GitHub-Insights: 3D-Darstellungen (Netze, Streudiagramme)

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Matplotlib	Stars 19k	contributors 417	commit activity 3.1k/year	license not specified
Plotly				
HoloViews	Stars 2.6k	contributors 127	commit activity 256/year	license BSD-3-Clause
ipyvolume	Stars 1.9k	contributors 44	commit activity 12/year	license MIT
pythreejs	Stars 923	contributors 25	commit activity 0/year	license not identifiable by github
mpl-scatter-density	Stars 492	contributors 3	commit activity 12/year	license BSD-2-Clause

2.9 Ruhende Projekte

Tab. 16: GitHub-Insights: Ruhende Projekte (Stand: 18.12.2023)

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
graph-tool	Stars 83	contributors 5	commit activity 0/year	license GPL-3.0
ggpy	Stars 3.7k	contributors 12	commit activity 0/year	license BSD-2-Clause
cufflinks	Stars 3k	contributors 29	commit activity 0/year	license MIT
scikit-plot	Stars 2.4k	contributors 12	commit activity 0/year	license MIT
mpld3	Stars 2.3k	contributors 44	commit activity 10/year	license BSD-3-Clause
vincent	Stars 2k	contributors 21	commit activity 0/year	license MIT
geoplotlib	Stars 1k	contributors 7	commit activity 0/year	license MIT
gmplot	Stars 813	contributors 25	commit activity 0/year	license MIT
plotly_express	Stars 701	contributors 6	commit activity 0/year	license MIT
PyGSP	Stars 465	contributors 13	commit activity 0/year	license BSD-3-Clause
PdVega	Stars 345	contributors 8	commit activity 0/year	license MIT
Clustergrammer2	Stars 113	contributors 4	commit activity 0/year	license MIT
chart	Stars 56	contributors 1	commit activity 0/year	license MIT
xtrude	Stars 32	contributors 2	commit activity 0/year	license MIT
Matplotlib Altair	Stars 12	contributors 5	commit activity 0/year	license not identifiable by github
d3po	Stars 2	contributors 1	commit activity 0/year	license BSD-3-Clause

KAPITEL 3

Matplotlib

Matplotlib ist eine 2D-Plot-Bibliothek, die in Python-Skripten, iPython-Shells und Jupyter-Notebooks verwendet werden kann. Mit ihr lassen sich Daten als Diagramme, Histogramme, Leistungsspektren, Balkendiagramme, Streudiagramme etc. visualisieren. Einen Überblick erhaltet ihr in Gallery. Es ist eine Low-Level-Bibliothek mit einem Matlab ähnlichen Schnittstelle, die zwar einerseits viele Freiheiten bietet, andererseits jedoch viel Code erfordert.

Pros	Cons
Ähnliches Design wie Matlab; der Wechsel ist einfach	Imperative API und häufig sind sehr ausführliche Angaben nötig
Viele Rendering-Backends, siehe *Matplotlib-Backends*	Häufig ungenügende Standarddarstellung

Siehe auch:

- Pyplot function overview
- User's Guide
- Toolkits
- Third party packages
- Nicolas P. Rougier: Scientific Visualization – Python & Matplotlib
- Nicolas P. Rougier: Matplotlib cheat sheet
- Learning matplotlib
- SciTools courses: An introduction to matplotlib

3.1 Matplotlib-Installation

Mit Spack könnt ihr Matplotlib in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install py-matplotlib
```

Alternativ könnt ihr Matplotlib auch mit anderen Paketmanagern installieren, z.B. mit Pipenv:

```
$ pipenv install matplotlib
```

Die Installation könnt ihr dann überprüfen mit:

```
>>> import matplotlib.pyplot as plt
```

Bemerkung: Wenn ihr den Fehler `TclError: no display name and no $DISPLAY environment variable` erhaltet, müsst ihr vermutlich das iPython-Backend für Matplotlib verwenden mit

```
import matplotlib.pyplot as plt

# iPython backend for matplotlib
%matplotlib inline
```

Sofern Ihr Matplotlib in einer Python-Datei importiert, müsst ihr stattdessen folgendes einfügen:

```
import matplotlib.pyplot as plt


# iPython backend for matplotlib
get_ipython().run_line_magic("matplotlib", "inline")
```

3.2 Matplotlib-Backends

Matplotlib stellt viele verschiedene Rendering-Backends zur Verfügung:

[1]:
```
from matplotlib import rcsetup


rcsetup.all_backends
```

[1]:
```
['GTK3Agg',
 'GTK3Cairo',
 'GTK4Agg',
 'GTK4Cairo',
 'MacOSX',
 'nbAgg',
 'QtAgg',
 'QtCairo',
 'Qt5Agg',
 'Qt5Cairo',
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
 'TkAgg',
 'TkCairo',
 'WebAgg',
 'WX',
 'WXAgg',
 'WXCairo',
 'agg',
 'cairo',
 'pdf',
 'pgf',
 'ps',
 'svg',
 'template']
```

Alternativ können auch nur die interaktiven oder nicht-interaktiven Backends aufgelistet werden:

[2]:
```
rcsetup.interactive_bk
```

[2]:
```
['GTK3Agg',
 'GTK3Cairo',
 'GTK4Agg',
 'GTK4Cairo',
 'MacOSX',
 'nbAgg',
 'QtAgg',
 'QtCairo',
 'Qt5Agg',
 'Qt5Cairo',
 'TkAgg',
 'TkCairo',
 'WebAgg',
 'WX',
 'WXAgg',
 'WXCairo']
```

[3]:
```
rcsetup.non_interactive_bk
```

[3]:
```
['agg', 'cairo', 'pdf', 'pgf', 'ps', 'svg', 'template']
```

3.3 Matplotlib-Beispiel

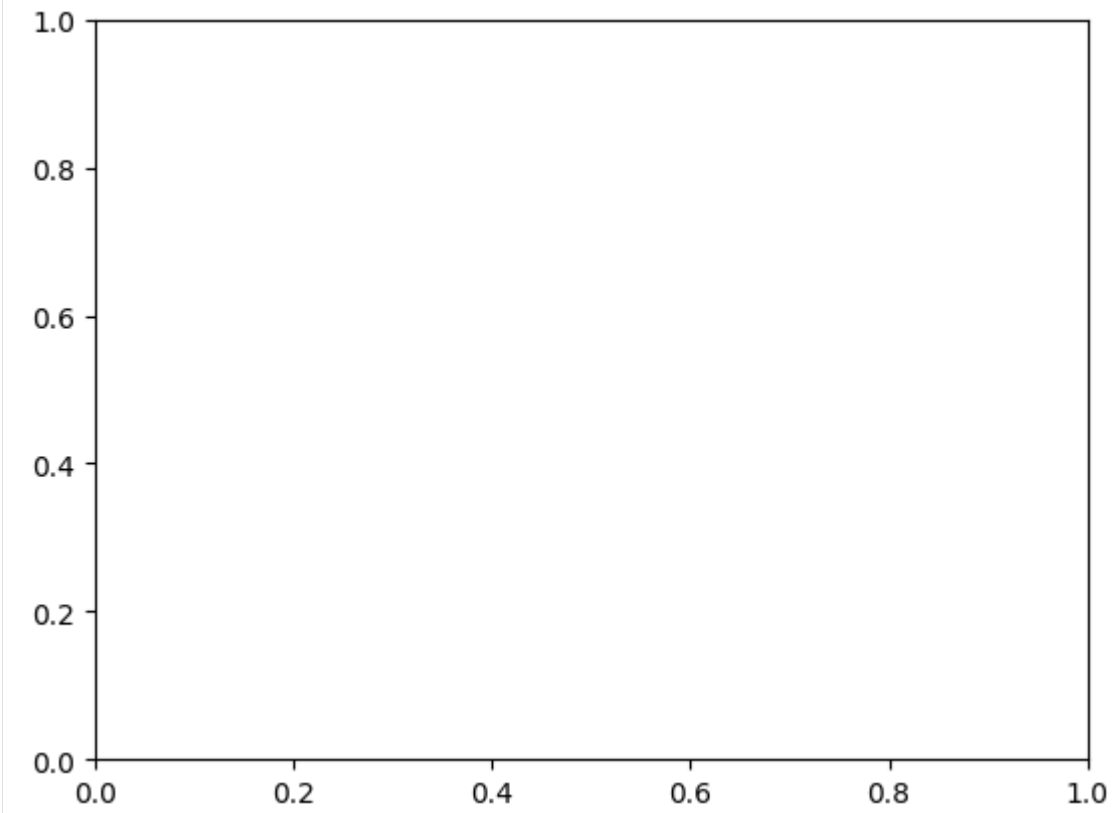
Matplotlib stellt Daten in Figuren (`Figures`) dar, d.h. in Fenstern, Jupyter-Widgets usw., von denen jede eine oder mehre Achsen (`Axes`) enthalten kann. Die einfachste Möglichkeit, eine Figur mit Achsen zu erstellen, ist die Verwendung von `pyplot.subplots` um dann mit `Axes.plot` einige Daten anzugeben.

1. Import

[1]:
```
import matplotlib.pyplot as plt
```

2. Figur mit Achsen erstellen

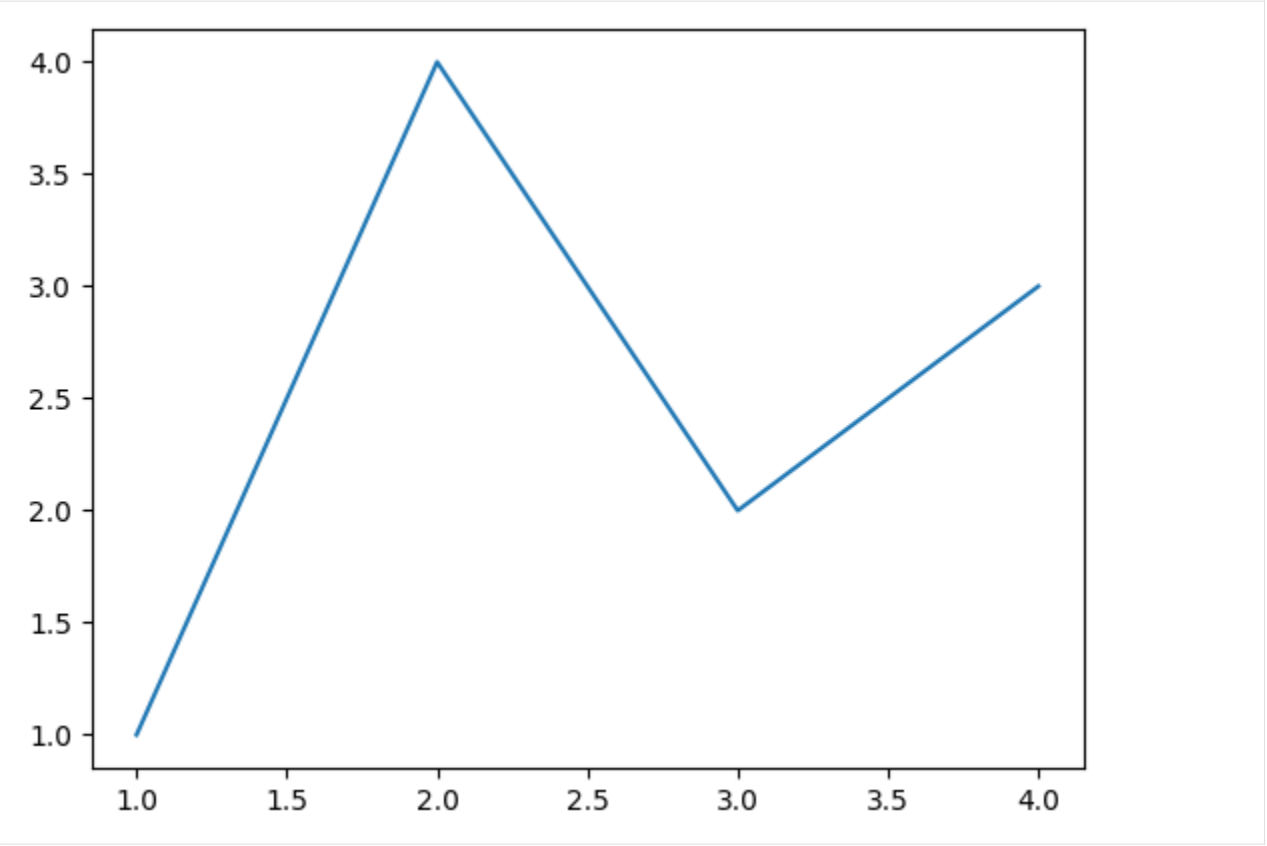
[2]:
```
fig, ax = plt.subplots()
```



3. Einige Daten auf den Achsen zeichnen

[3]:
```
fig, ax = plt.subplots()
ax.plot([1, 2, 3, 4], [1, 4, 2, 3])
```

[3]:
```
[<matplotlib.lines.Line2D at 0x11b315350>]
```



3.4 mpl-scatter-density

mpl-scatter-density erleichtert das Erstellen interaktiver Streudichtekarten aus großen Datenmengen.

3.4.1 Installation

```
$ pipenv install mpl-scatter-density
```

Damit werden automatisch auch die Abhängigkeiten Numpy, *Matplotlib* und fast-histogram installiert.

3.4.2 Verwendungszweck

1. Importe

[1]:
```
import matplotlib.pyplot as plt
import mpl_scatter_density
import numpy as np
```

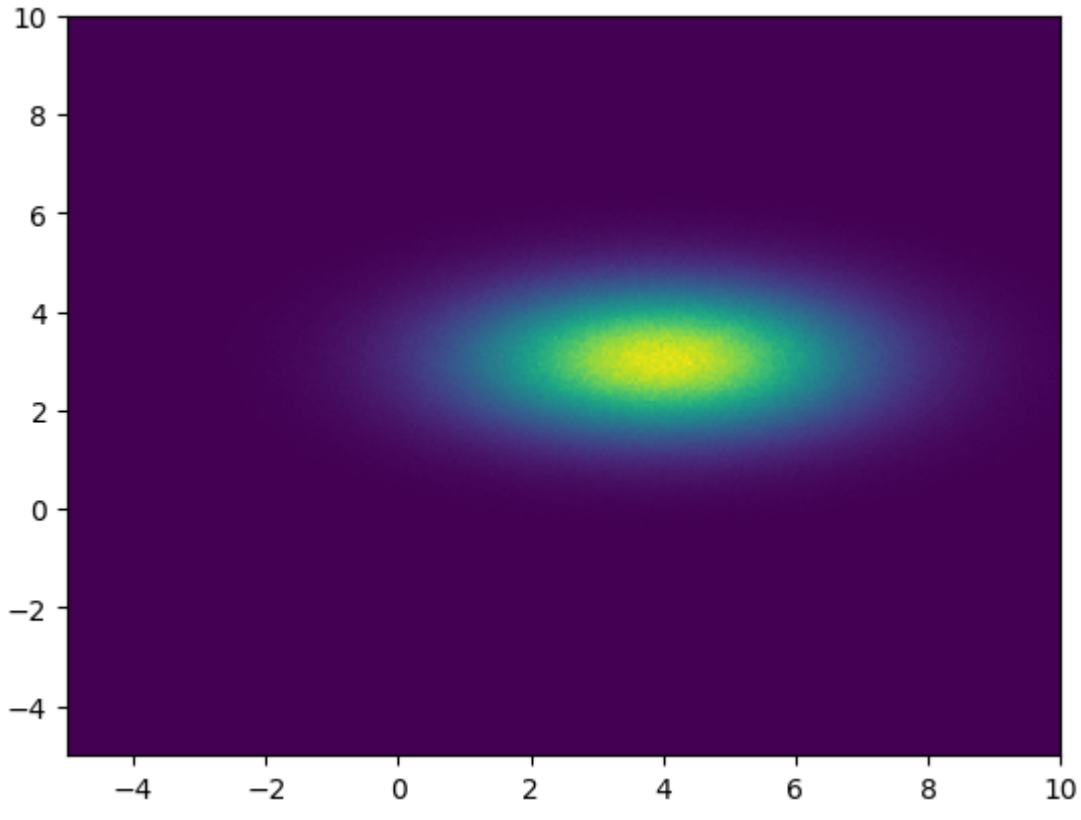
2. Beispieldaten generieren

[2]:
```
N = 10000000
x = np.random.normal(4, 2, N)
y = np.random.normal(3, 1, N)
```

3. Es gibt im Wesentlichen zwei Verwendungszwecke für mpl-scatter-density:
    1. `projection='scatter_density'`
    2. `ScatterDensityArtist`
3. 1. `projection='scatter_density'`

[3]:
```
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, projection="scatter_density")
ax.scatter_density(x, y)
ax.set_xlim(-5, 10)
ax.set_ylim(-5, 10)
fig.savefig("gaussian.png")
```

```
/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
↪mpl_scatter_density/generic_density_artist.py:77: RuntimeWarning: All-NaN slice␣
↪encountered
  vmin = self._density_vmin(array)
/Users/veit/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
↪mpl_scatter_density/generic_density_artist.py:82: RuntimeWarning: All-NaN slice␣
↪encountered
  vmax = self._density_vmax(array)
```



3. 2. `ScatterDensityArtist`

Auch im vorhergehenden Beispiel wird `ScatterDensityArtist` verwendet, ohne es jedoch direkt anzugeben. Im folgenden Beispiel fügen wir es explizit den Achsen hinzu:

[4]:
```
from mpl_scatter_density import ScatterDensityArtist


a = ScatterDensityArtist(ax, x, y)
ax.add_artist(a)
```

[4]:
```
<mpl_scatter_density.scatter_density_artist.ScatterDensityArtist at 0x10bc0f150>
```

3.5 pandas

pandas-Visualization baut auf *Matplotlib* auf, bietet jedoch eine einfachere API zum Plotten von pandas-Dataframes und -Series.

Siehe auch:

- pandas Plotting

3.5.1 pandas-Installation

Mit Spack könnt ihr pandas in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install py-pandas
```

Alternativ könnt ihr pandas auch mit anderen Paketmanagern installieren, z.B. mit Pipenv.

Bemerkung: Falls ihr pipenv noch nicht installiert habt, findet ihr eine Anleitung hierzu unter Pipenv-Installation.

```
$ pipenv install pandas
```

Die Installation könnt Ihr dann überprüfen mit:

```
>>> import pandas as pd
```

3.5.2 Pandas-Beispiele

Importe

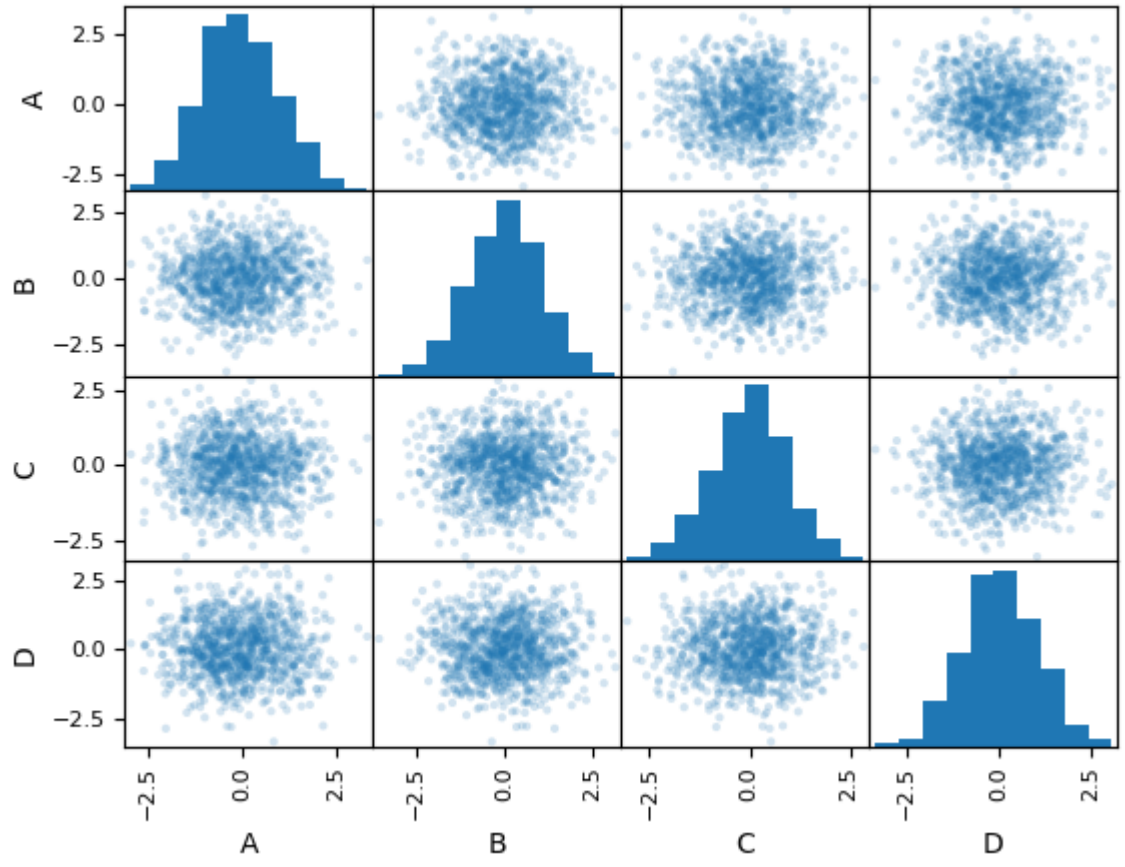
[1]:
```
import numpy as np
import pandas as pd
```

Scatter matrix

Mit pandas.plotting.scatter_matrix lässt sich eine Streumatrix (Scatter-Matrix erstellen, z.B.:

[2]:
```
df = pd.DataFrame(np.random.randn(1000, 4), columns=["A", "B", "C", "D"])
pd.plotting.scatter_matrix(df, alpha=0.2)
```

[2]:
```
array([[<Axes: xlabel='A', ylabel='A'>, <Axes: xlabel='B', ylabel='A'>,
        <Axes: xlabel='C', ylabel='A'>, <Axes: xlabel='D', ylabel='A'>],
       [<Axes: xlabel='A', ylabel='B'>, <Axes: xlabel='B', ylabel='B'>,
        <Axes: xlabel='C', ylabel='B'>, <Axes: xlabel='D', ylabel='B'>],
       [<Axes: xlabel='A', ylabel='C'>, <Axes: xlabel='B', ylabel='C'>,
        <Axes: xlabel='C', ylabel='C'>, <Axes: xlabel='D', ylabel='C'>],
       [<Axes: xlabel='A', ylabel='D'>, <Axes: xlabel='B', ylabel='D'>,
        <Axes: xlabel='C', ylabel='D'>, <Axes: xlabel='D', ylabel='D'>]],
      dtype=object)
```



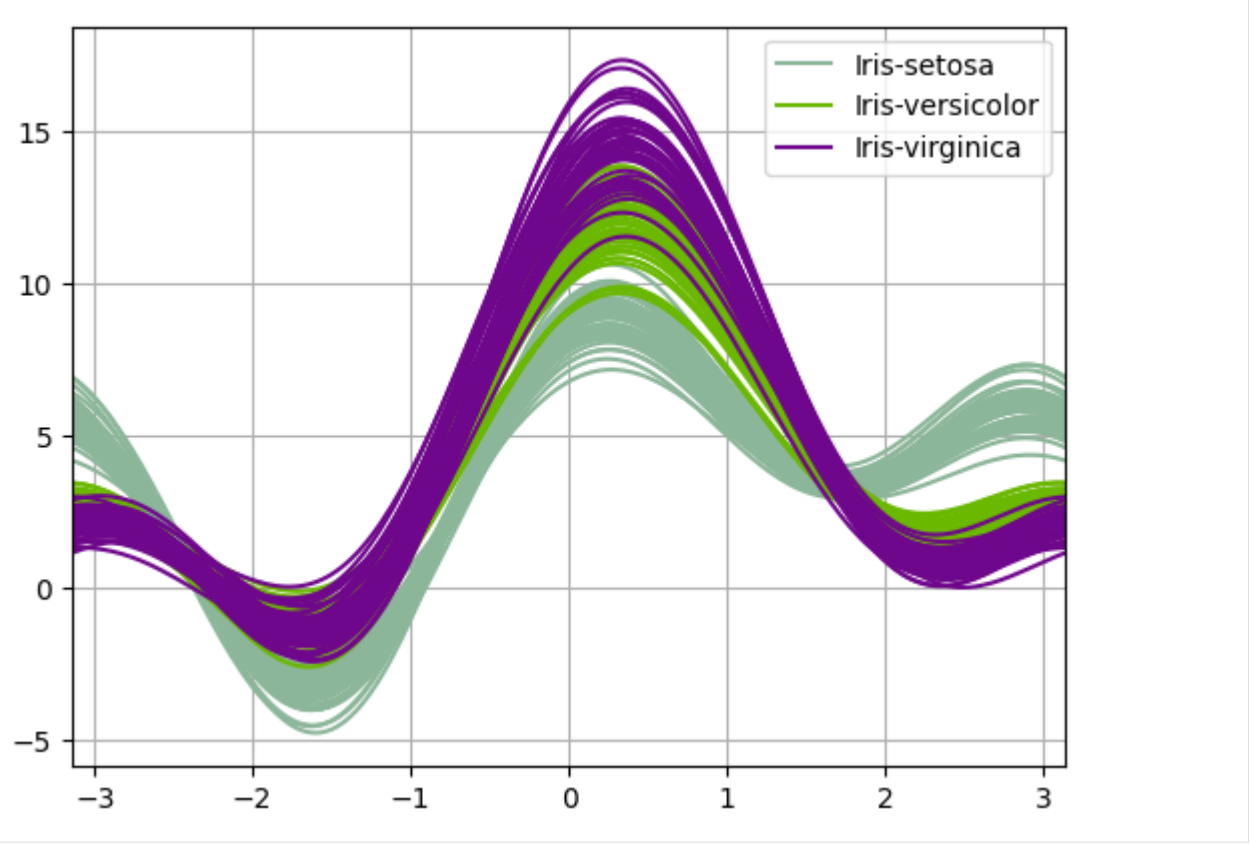
numpy.random.randn gibt eine Stichprobe (oder mehrere Stichproben) mit Standardnormalverteilung zurück. Die Parameter `(d0, d1, ..., dn)` sind optionale Ganzzahlen, die die Dimensionen des zurückgegebenen Arrays bestimmen.

Andrews plot

In den letzten Jahren kamen weitere ausgefeilte statistische Visualisierungswerkzeuge hinzu, unter anderem Andrews plot für die Visualisierung mehrdimensionaler Daten:

[3]:
```
df = pd.read_csv(
    "https://raw.githubusercontent.com/pandas-dev/pandas/master/pandas/tests/io/data/csv/
↪iris.csv"
)
pd.plotting.andrews_curves(df, "Name")
```

[3]:
```
<Axes: >
```



3.6 GeoPandas

GeoPandas erweitert *pandas* um Datentypen, die räumliche Operationen an geometrischen Typen ermöglicht, wobei die geometrischen Operationen von Shapely ausgeführt werden. Für den Datenzugriff verwendet GeoPandas Fiona und zum Plotten descartes und *Matplotlib*.

Siehe auch:

- Docs
- Gallery
- GeoPandas Introduction
- dask-geopandas

- MovingPandas

3.6.1 GeoPandas-Installation

Mit Spack könnt ihr GeoPandas in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install py-geopandas
```

Alternativ könnt ihr GeoPandas auch mit anderen Paketmanagern installieren, z.B. mit Pipenv:

```
$ pipenv install fiona matplotlib descartes geopandas
```

Bemerkung: Falls ihr pipenv noch nicht installiert habt, findet ihr eine Anleitung hierzu unter Pipenv-Installation.

Die Installation könnt ihr dann überprüfen mit:

```
>>> import geopandas as gpd
```

3.6.2 GeoPandas-Beispiele

In den folgenden Beispielen verwenden wir den nybb-Datensatz.

GeoPandas-Plot

1. Import

[1]:
```
import geopandas as gpd

from geodatasets import get_path
```

2. Daten lesen

[2]:
```
nybb = get_path("nybb")
```

[3]:
```
df = gpd.read_file(nybb)
```

3. Anzeigen der ersten Zeilen

[4]:
```
df.head()
```

[4]:
```
   BoroCode       BoroName     Shape_Leng    Shape_Area  \
0         5  Staten Island  330470.010332  1.623820e+09
1         4         Queens  896344.047763  3.045213e+09
2         3       Brooklyn  741080.523166  1.937479e+09
3         1      Manhattan  359299.096471  6.364715e+08
4         2          Bronx  464392.991824  1.186925e+09

                                            geometry
0  MULTIPOLYGON (((970217.022 145643.332, 970227...
```

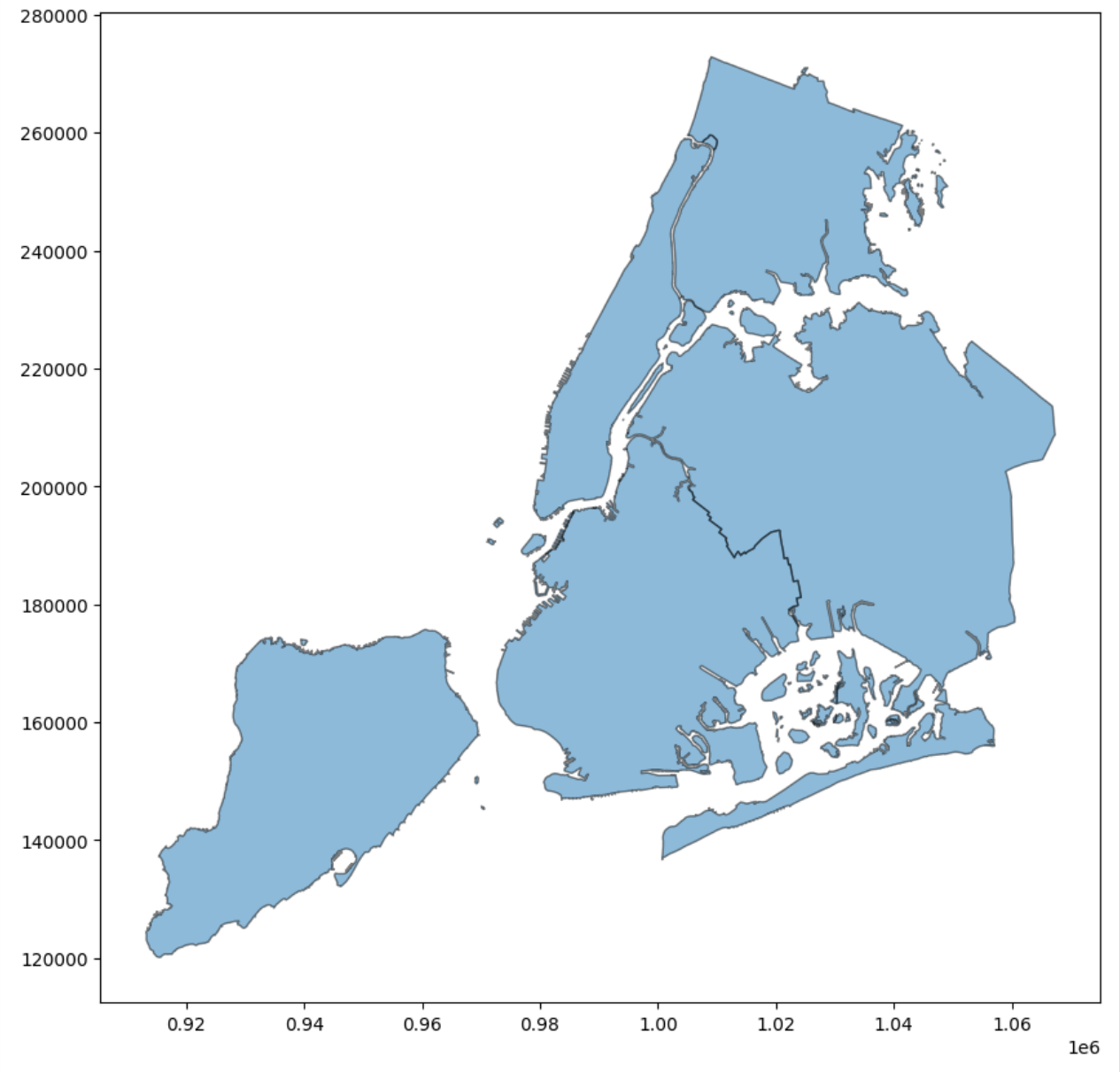
(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
1  MULTIPOLYGON (((1029606.077 156073.814, 102957...
2  MULTIPOLYGON (((1021176.479 151374.797, 102100...
3  MULTIPOLYGON (((981219.056 188655.316, 980940...
4  MULTIPOLYGON (((1012821.806 229228.265, 101278...
```

4. Plotten

[5]:
```
ax = df.plot(figsize=(10, 10), alpha=0.5, edgecolor="k")
```



`figsize` gibt die Größe des Plots, `alpha` die Transparenz und `edgecolor` die Kantenfarbe an.

Plotten in Folium

Folium oder genauer das mit Folium verwendete leaflet.js verwendet standardmäßig die Werte für Breiten- und Längengrade. Daher müssen wir unsere Werte erst konvertieren:

1. Koordinaten bestimmen

   Um die Daten mit EPSG-Codes zu verwenden zu können, bietet GeoPandas geopandas.GeoDataFrame.crs (Coordination Reference System; deutsch: Koordinatenreferenzsystem).

[6]:
```
print(df.crs)
```

```
EPSG:2263
```

[7]:
```
df = df.to_crs(epsg=4326)
print(df.crs)
df.head()
```

```
EPSG:4326
```

[7]:

	BoroCode	BoroName	Shape_Leng	Shape_Area	geometry
0	5	Staten Island	330470.010332	1.623820e+09	MULTIPOLYGON (((-74.05051 40.56642, -74.05047 ...
1	4	Queens	896344.047763	3.045213e+09	MULTIPOLYGON (((-73.83668 40.59495, -73.83678 ...
2	3	Brooklyn	741080.523166	1.937479e+09	MULTIPOLYGON (((-73.86706 40.58209, -73.86769 ...
3	1	Manhattan	359299.096471	6.364715e+08	MULTIPOLYGON (((-74.01093 40.68449, -74.01193 ...
4	2	Bronx	464392.991824	1.186925e+09	MULTIPOLYGON (((-73.89681 40.79581, -73.89694 ...

[8]:
```
import folium
```

[9]:
```
m = folium.Map(
    location=[40.70, -73.94], zoom_start=10, tiles="CartoDB positron"
)
for _, r in df.iterrows():
    # without simplifying the representation of each borough, the map might not be␣
↪displayed
    sim_geo = gpd.GeoSeries(r["geometry"]).simplify(tolerance=0.001)
    geo_j = sim_geo.to_json()
    geo_j = folium.GeoJson(
        data=geo_j,
    )
    folium.Popup(r["BoroName"]).add_to(geo_j)
    geo_j.add_to(m)
m
```

[9]:
```
<folium.folium.Map at 0x15ad69310>
```

3.7 seaborn

Ähnlich wie *pandas* basiert seaborn auf *Matplotlib* und bietet ein High-Level-API für die Visualisierung statistischer Daten. Zudem bietet seaborn schöne Farbpaletten und Diagrammstile.

Siehe auch:

- Tutorial
- Example Gallery
- Three common seaborn difficulties

3.7.1 seaborn-Installation

Mit Spack könnt ihr seaborn in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install py-seaborn
```

Alternativ könnt ihr seaborn auch mit anderen Paketmanagern installieren, z.B. mit Pipenv.

Bemerkung: Falls ihr pipenv noch nicht installiert habt, findet ihr eine Anleitung hierzu unter Pipenv-Installation.

```
$ pipenv install seaborn
```

Die Installation könnt ihr überprüfen mit

```
>>> import seaborn as sns
```

3.7.2 seaborn-Beispiel

1. Importe

```
[1]: import matplotlib.pyplot as plt
     import numpy as np
     import seaborn as sns
```

2. Stil definieren

   seaborn.set ist ein Alias für seaborn.set_theme, mit dem sich unterschiedliche Stile, Paletten,Schriften und Farben definieren lassen, z.B.:

```
[2]: sns.set(style="white", color_codes=True)
```

3. Erzeugen eines bivariaten Zufallsdatensatzes

   Hierfür verwenden wir numpy.random.RandomState.multivariate_normal aus numpy.random.RandomState, wobei ein *Seed* von 9, `0` als Mittelwerte und 100 Proben eine nicht ganz symmetrische Kovarianzmatrix ergeben:

```
[3]: rs = np.random.RandomState(9)
     mean = [0, 0]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

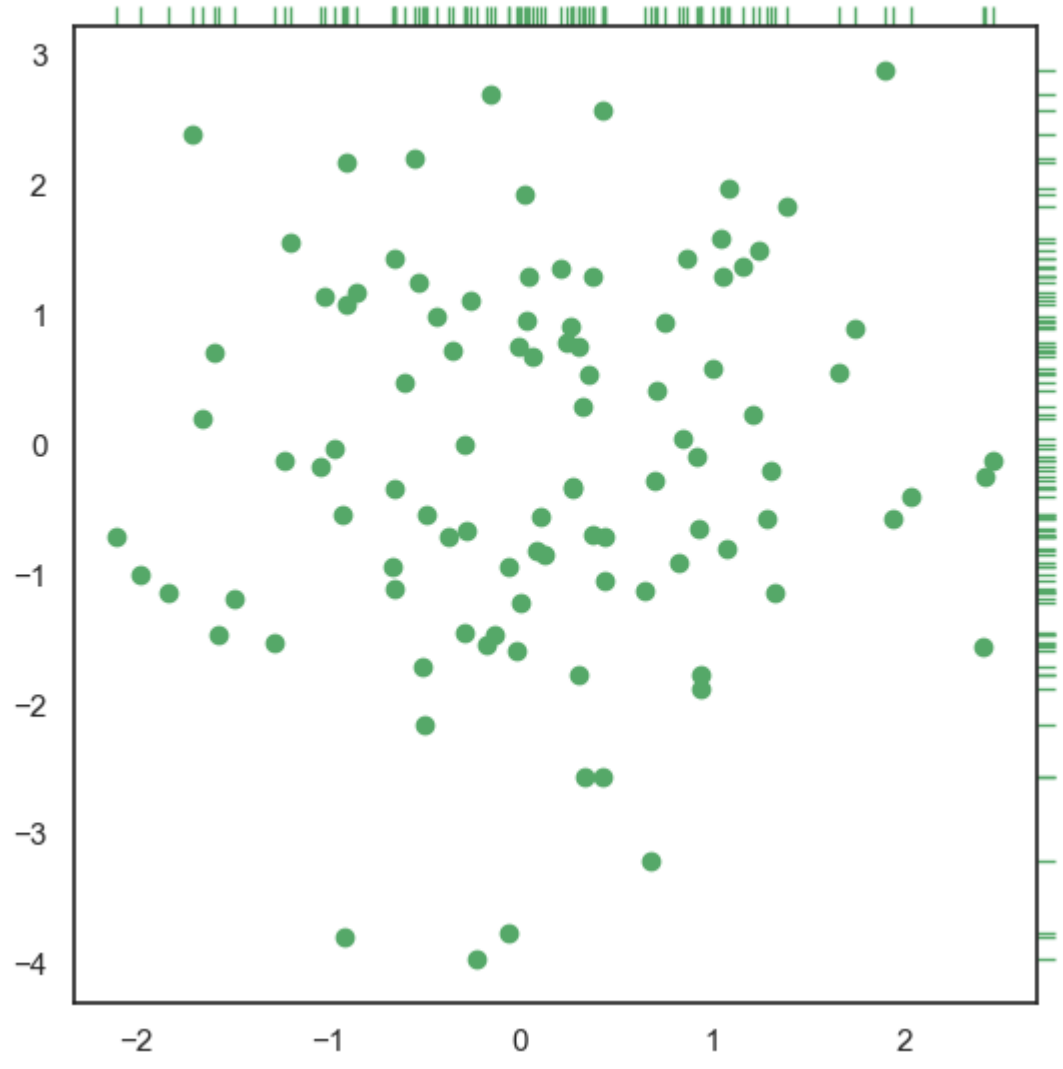
```
cov = [(1, 0), (0, 2)]
x, y = rs.multivariate_normal(mean, cov, 100).T
```

4. `JointGrid` für bivariate Plots

   Mit seaborn.JointGrid lassen sich bivariate Plots erzeugen. Wenn beide Funktionen unterschiedliche Schlüsselwortargumente übergeben, muss JointGrid.plot_joint() und JointGrid.plot_marginals() verwendet werden:

[4]:
```
grid = sns.JointGrid(x=x, y=y, space=0, height=6, ratio=10)
grid.plot_joint(plt.scatter, color="g")
grid.plot_marginals(sns.rugplot, height=0.2, color="g")
```

[4]:
```
<seaborn.axisgrid.JointGrid at 0x154b964d0>
```



3.8 plotnine

plotnine implementiert The Grammar of Graphics in Python basierend auf ggplot2. Die Grammatik erlaubt die einfache Beschreibung auch von komplexen Grafiken.

Siehe auch:

plotnine documentation
API-Referenz, Gallerie, Tutorials etc.

ggplot2 documentation
plotnine verwendet eine ähnliche API und Pipeline wie ggplt2

Grammer of graphics with plotnine
Gutes Tutorial zur Einführung in plotnine als Teil eines Datenvisualisierung-Track von kaggle.

Paul Teehan: Plotnine is the best Python implementation of R's ggplot2
Vergleich zwischen plotnine und ggplot2, v.a. (vor allem) in Bezug auf die API-Kompatibilität.

Python Plotting for Exploratory Analysis
Eine Liste von Plots für die explorative Datenanalyse und wie sie mit verschiedenen Bibliotheken erstellt werden können.

Introduction to Plotnine
Erläutert die Hauptaspekte von plotnine und zeigt, wie die Bibliothek verwendet werden kann.

Making Plots With plotnine
Eine Einführung in The Grammar of Graphics und die Verwendung von plotnine. Dies ist Teil des Kurses von Data Carpentry Data Analysis and Visualization in Python for Ecologists.

3.8.1 plotnine-Installation

In den meisten Fällen sollte folgende Installation hinreichend sein:

```
$ pipenv install plotnine
```

Für die Verwendung zusammen mit scikit-learn und scikit-misc können Extras installiert werden mit

```
$ pipenv install "plotnine[all]"
```

```
$ jupyter nbextension enable --py widgetsnbextension
```

```
$ jupyter labextension install @jupyter-widgets/jupyterlab-manager
```

3.8.2 plotnine-Beispiele

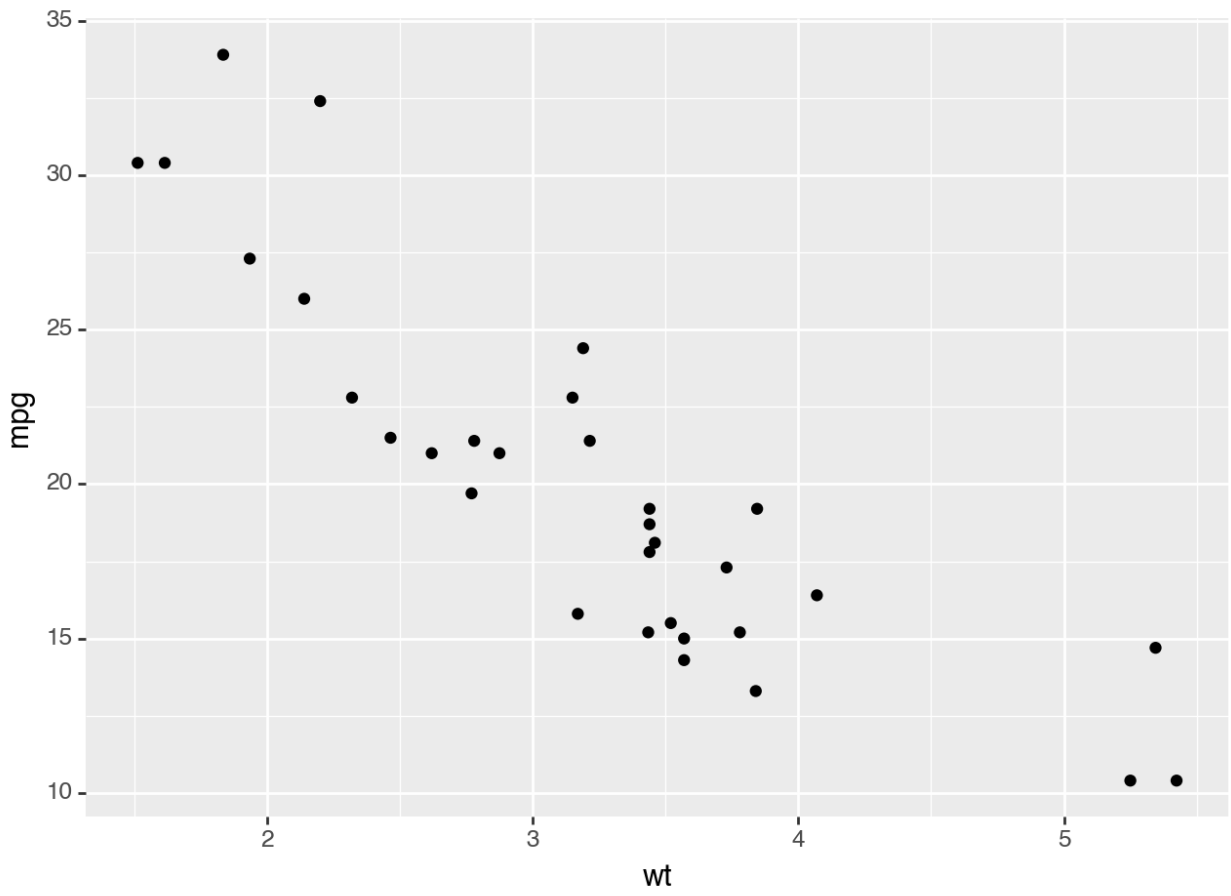
Einfaches Streudiagramm

1. Importe

[1]:
```
from plotnine import *
from plotnine.data import mtcars
```

2. Streudiagramm

[2]:
```
(
    ggplot(mtcars, aes("wt", "mpg"))
    + geom_point()
)
```

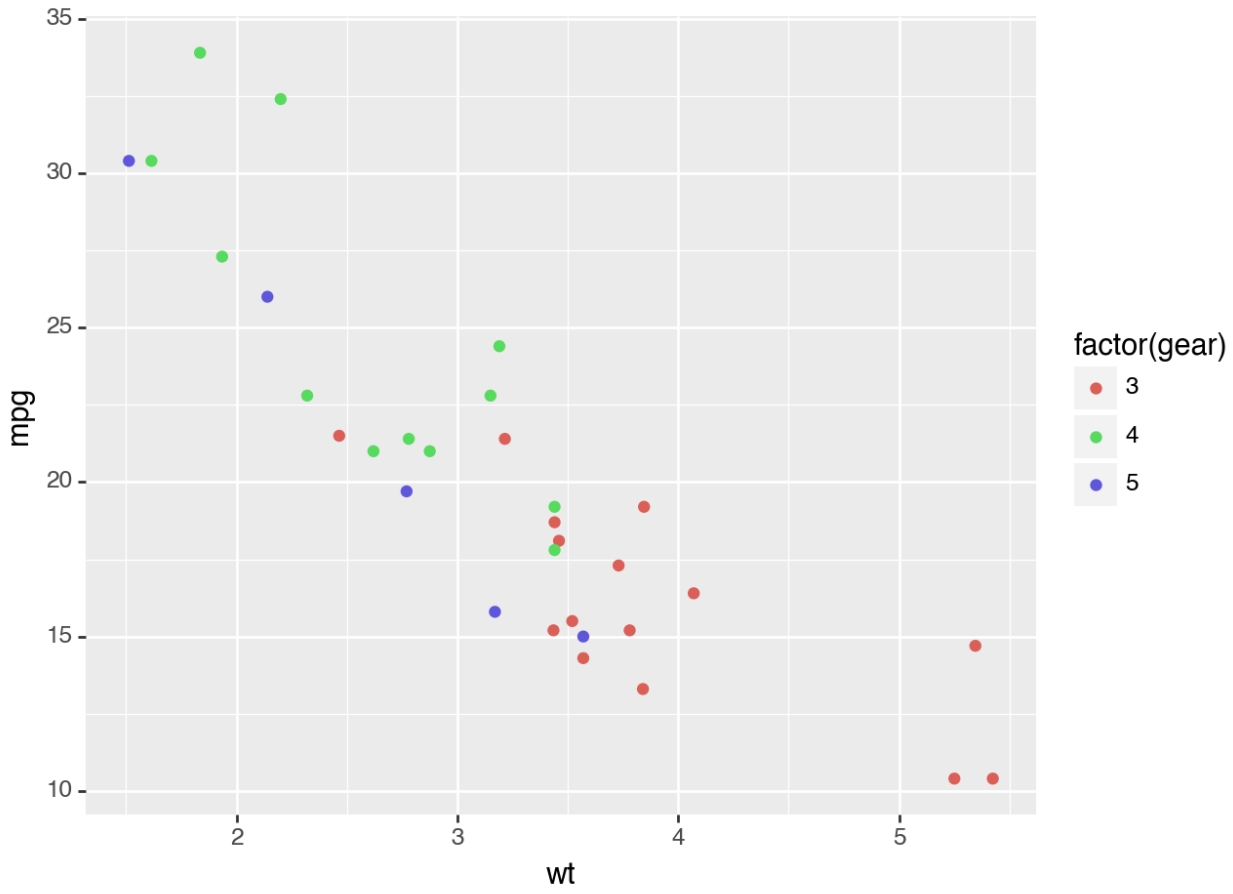


[2]:
```
<Figure Size: (640 x 480)>
```

plotnine.mapping.aes erstellt ästhetische Zuordnungen mit *Meilen je Gallone* `mpg` auf der y-Achse und *Gewicht der Autos* `wt` auf der x-Achse. plotnine.geoms.geom_point erstellt dann ein Streudiagramm.

3. Farbliche Unterscheidung der Variablen

[3]:
```
(
    ggplot(mtcars, aes("wt", "mpg", color="factor(gear)"))
    + geom_point()
)
```

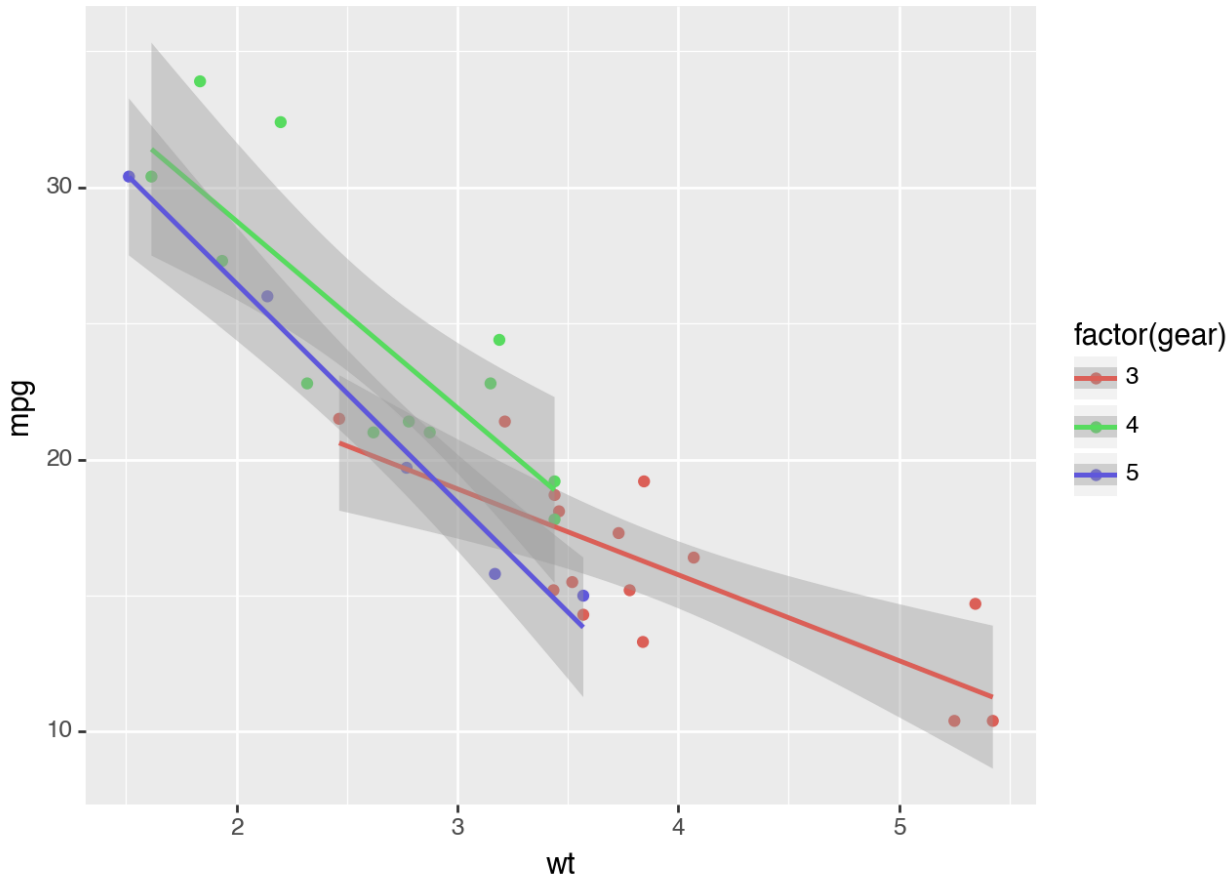


[3]:
```
<Figure Size: (640 x 480)>
```

4. Geglättetes lineares Modell mit Konfidenzintervallen

   Mit plotnine.stats.stat_smooth lassen sich geglättete bedingte Mittelwerte berechnen, wobei `lm` ein lineares Modell zugrunde liegt:

[4]:
```
(
    ggplot(mtcars, aes("wt", "mpg", color="factor(gear)"))
    + geom_point()
    + stat_smooth(method="lm")
)
```

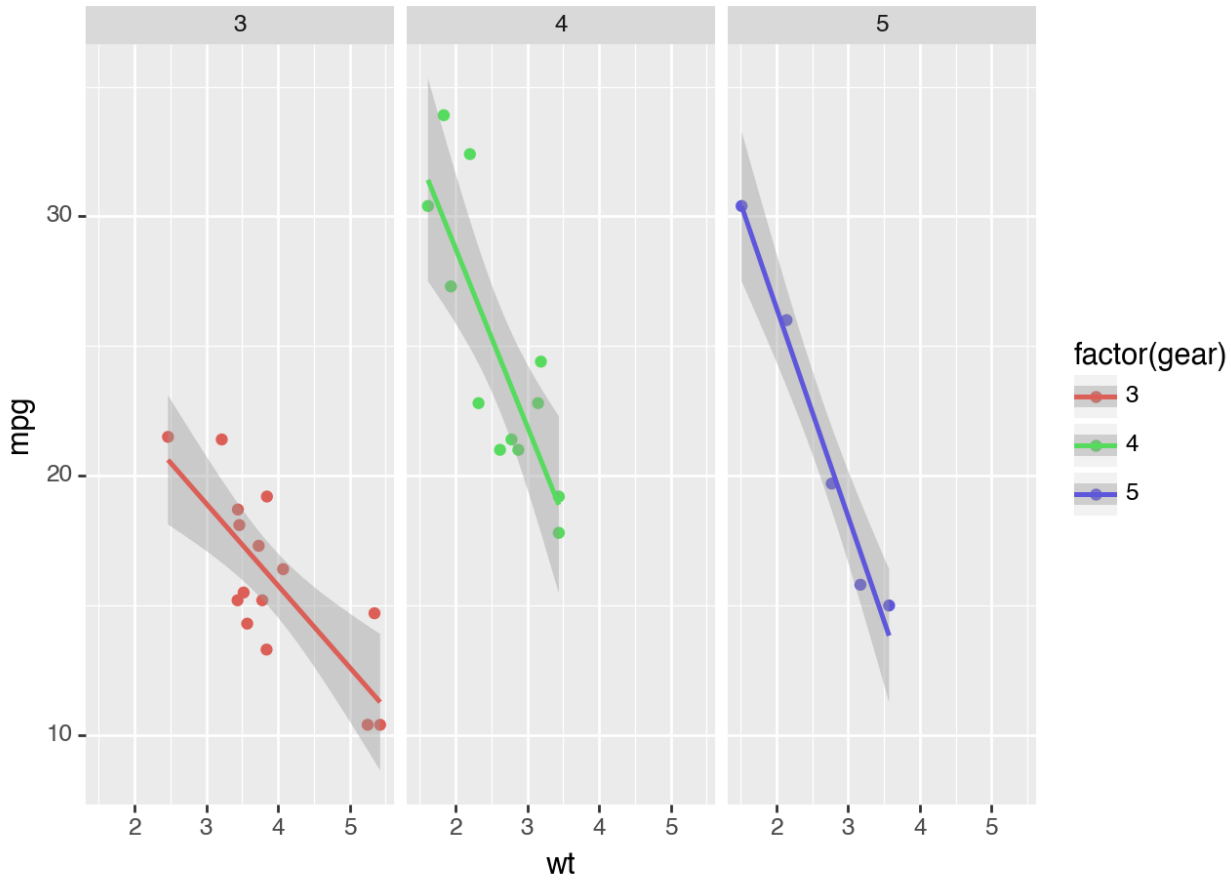


[4]:
```
<Figure Size: (640 x 480)>
```

5. Darstellung in separierten Feldern

   Mit plotnine.facets.facet_wrap lassen sich die Felder trennen.

[5]:
```
(
    ggplot(mtcars, aes("wt", "mpg", color="factor(gear)"))
    + geom_point()
    + stat_smooth(method="lm")
    + facet_wrap("~gear")
)
```



[5]: <Figure Size: (640 x 480)>

Interaktive Diagramme

Zusammen mit ipywidgets lassen sich auch interaktive Diagramme erstellen.

1. Importe

[6]:
```
import itertools

from copy import copy

import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import plotnine as p9

from IPython.display import display
from ipywidgets import widgets
from plotnine.data import mtcars
```

2. Interaktives Streudiagramm erstellen

Im folgenden betrachten wir PS auf der x-Achse, Meilen je Gallone auf der y-Achse und unterscheiden farblich das Gewicht der Autos:

[7]:
```
%matplotlib notebook

p = (ggplot(mtcars, aes(x="hp", y="mpg", color="wt"))
     + p9.geom_point()
     + p9.theme_linedraw()
    )
p
```

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

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

[7]:
```
<Figure Size: (640 x 480)>
```

3. Nun wählen wir die Autos anhand der Zylinderanzahl aus.

3. 1. Zunächst bereiten wir die Liste vor, die wir verwenden werden, um Teilmengen von Daten auf der Grundlage der Anzahl von Zylindern auszuwählen:

[8]:
```
cylList = np.unique(mtcars["cyl"])
```

3. 2. Diese Liste verwenden wir nun für ein Dropdown-Menü mit der Anzahl der Zylinder:

[9]:
```
cylSelect = widgets.Dropdown(
    options=list(cylList),
    value=cylList[1],
    description="Cylinders:",
    disabled=False,
)
```

4. Die Widgets sollen dieselbe Darstellung aktualisieren können und nicht bei jeder Änderung einen neuen Plot erstellen.

4. 1. Zunächst ermitteln wir die maximalen Bereiche der relevanten Variablen, um die Achsen bei Aktualisierungen konstant zu halten.

   Mit `minWt` und `maxWt` erhalten wir die Spanne der Gewichte.

   Mit `wtOptions` wandeln wir das NumPy-Array in eine sortierte Python-Liste von eindeutigen Werte um.

   In `cylSelect` wird die erste Auswahl für das Dropdown-Menü der Zylinder-Anzahl festgelegt.

[10]:
```
minWt = min(mtcars["wt"])
maxWt = max(mtcars["wt"])
wtOptions = list(
    np.sort(np.unique(mtcars.loc[mtcars["cyl"] == cylList[0], "wt"]))
)

minHP = min(mtcars["hp"])
maxHP = max(mtcars["hp"])

minMPG = min(mtcars["mpg"])
maxMPG = max(mtcars["mpg"])

cylSelect = widgets.Dropdown(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    options=list(cylList),
    value=cylList[1],
    description="Cylinders:",
    disabled=False,
)
```

4. 2. Mit `get_current_artists` ermitteln wir dann alle Objekte, die gerendert werden sollen:

[11]:
```
def get_current_artists():
    # Return artists attached to all the matplotlib axes
    axes = plt.gca()
    return itertools.chain(
        axes.lines, axes.collections, axes.artists, axes.patches, axes.texts
    )
```

4. 3. Nun erstellen wir die Funktion `plotUpdate`, die aufgerufen wird, um die Darstellung jedes Mal zu aktualisieren, wenn wir eine Auswahl ändern.

[12]:
```
fig = None
axs = None


def plotUpdate(*args):
    # Use global variables for matplotlib's figure and axis.
    global fig, axs

    # Get current values of the selection widget
    cylValue = cylSelect.value

    # Create a temporary dataset that is constrained by the user's selections.
    tmpDat = mtcars.loc[(mtcars["cyl"] == cylValue), :]

    # Create plotnine's plot

    # Using the maximum and minimum values we gatehred before, we can keep the plot axis␣
↪from
    # changing with the cyinder selection
    p = (
        ggplot(tmpDat, aes(x="hp", y="mpg", color="wt"))
        + p9.geom_point()
        + p9.theme_linedraw()
    )
    if fig is None:
        # If this is the first time a plot is made in the notebook, we let plotnine␣
↪create a new
        # matplotlib figure and axis.
        fig = p.draw()
        axs = fig.axes
    else:
        # p = copy(p)
        # This helps keeping old selected data from being visualized after a new␣
↪selection is made.
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        # We delete all previously reated artists from the matplotlib axis.
        for artist in get_current_artists():
            artist.remove()

        # If a plot is being updated, we re-use the figure an axis created before.
        p._draw_using_figure(fig, axs)
```

4. 4. Nun beobachten wir, ob sich der Wert in unserem Dropdown-Menü der Zylinder-Anzahl ändert:

[13]:
```
cylSelect.observe(plotUpdate, "value")
```

4. 5. Mit `display` wird das Widget dargestellt:

[14]:
```
display(cylSelect)
```

```
Dropdown(description='Cylinders:', index=1, options=(4, 6, 8), value=6)
```

4. 6. Nun plotten wir das erste Bild mit Anfangswerten.

[15]:
```
plotUpdate()
```

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

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

4. 7. Mit der Matplotlib-Funktion `tight_layout()` passen wir die Figur an die Abmessungen des Plots an:

[16]:
```
plt.tight_layout()
```

```
/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_23701/2925123646.py:2:␣
↪UserWarning: The figure layout has changed to tight
```

4. 8. Trick, damit das erste gerenderte Bild dem `tight_layout` entspricht. Ohne diesen Befehl würde die Figur erst nach der ersten Aktualisierung in ihre Abmessungen passen.

[17]:
```
cylSelect.value = cylList[0]
```

5. Bereichsregler hinzufügen

   Nun schränken wir mit einem Bereichsregler die Daten basierend auf dem Fahrzeuggewicht ein:

[18]:
```
# The first selection is a drop-down menu for number of cylinders
cylSelect = widgets.Dropdown(
    options=list(cylList),
    value=cylList[1],
    description="Cylinders:",
    disabled=False,
)

# The second selection is a range of weights
wtSelect = widgets.SelectionRangeSlider(
    options=wtOptions,
    index=(0, len(wtOptions) - 1),
    description="Weight",
    disabled=False,
)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
widgetsCtl = widgets.HBox([cylSelect, wtSelect])


# The range of weights needs to always be dependent on the cylinder selection.
def updateRange(*args):
    """Updates the selection range from the slider depending on the cylinder selection.""
↪"
    cylValue = cylSelect.value

    wtOptions = list(
        np.sort(np.unique(mtcars.loc[mtcars["cyl"] == cylValue, "wt"]))
    )

    wtSelect.options = wtOptions
    wtSelect.index = (0, len(wtOptions) - 1)


cylSelect.observe(updateRange, "value")

# For the widgets to update the same plot, instead of creating one new image every time
# a selection changes. We keep track of the matplotlib image and axis, so we create only␣
↪one
# figure and set of axis, for the first plot, and then just re-use the figure and axis
# with plotnine's "_draw_using_figure" function.
fig = None
axs = None


# This is the main function that is called to update the plot every time we chage a␣
↪selection.
def plotUpdate(*args):
    # Use global variables for matplotlib's figure and axis.
    global fig, axs

    # Get current values of the selection widgets
    cylValue = cylSelect.value
    wrRange = wtSelect.value

    # Create a temporary dataset that is constrained by the user's selections.
    tmpDat = mtcars.loc[
        (mtcars["cyl"] == cylValue)
        & (mtcars["wt"] >= wrRange[0])
        & (mtcars["wt"] <= wrRange[1]),
        :,
    ]

    # Create plotnine's plot

    p = (
        ggplot(tmpDat, aes(x="hp", y="mpg", color="wt"))
        + p9.geom_point()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        + p9.theme_linedraw()
        + p9.lims(x=[minHP, maxHP], y=[minMPG, maxMPG])
        + p9.scale_color_continuous(limits=(minWt, maxWt))
    )

    if fig is None:
        fig = p.draw()
        axs = fig.axes
    else:
        for artist in get_current_artists():
            artist.remove()
        p._draw_using_figure(fig, axs)


cylSelect.observe(plotUpdate, "value")
wtSelect.observe(plotUpdate, "value")

# Display the widgets
display(widgetsCtl)

# Plots the first image, with inintial values.
plotUpdate()

# Matplotlib function to make the image fit within the plot dimensions.
plt.tight_layout()

# Trick to get the first rendered image to follow the previous "tight_layout" command.
# without this, only after the first update would the figure be fit inside its␣
↪dimensions.
cylSelect.value = cylList[0]
```

```
HBox(children=(Dropdown(description='Cylinders:', index=1, options=(4, 6, 8), value=6),␣
↪SelectionRangeSlider(d...
```

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

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

```
/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_23701/1448775928.py:89:␣
↪UserWarning: The figure layout has changed to tight
```

6. Diagramm optimieren

   Schließlich ändern wir noch einige Diagrammeigenschaften, um eine verständlichere Abbildung zu erhalten:

[19]:
```
# The first selection is a drop-down menu for number of cylinders
cylSelect = widgets.Dropdown(
    options=list(cylList),
    value=cylList[1],
    description="Cylinders:",
    disabled=False,
)

# The second selection is a range of weights
wtSelect = widgets.SelectionRangeSlider(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    options=wtOptions,
    index=(0, len(wtOptions) - 1),
    description="Weight",
    disabled=False,
)

widgetsCtl = widgets.HBox([cylSelect, wtSelect])


# The range of weights needs to always be dependent on the cylinder selection.
def updateRange(*args):
    """Updates the selection range from the slider depending on the cylinder selection."
↪"
    cylValue = cylSelect.value

    wtOptions = list(
        np.sort(np.unique(mtcars.loc[mtcars["cyl"] == cylValue, "wt"]))
    )

    wtSelect.options = wtOptions
    wtSelect.index = (0, len(wtOptions) - 1)


cylSelect.observe(updateRange, "value")

fig = None
axs = None


# This is the main function that is called to update the plot every time we chage a␣
↪selection.
def plotUpdate(*args):
    # Use global variables for matplotlib's figure and axis.
    global fig, axs

    # Get current values of the selection widgets
    cylValue = cylSelect.value
    wrRange = wtSelect.value

    # Create a temporary dataset that is constrained by the user's selections of
    # number of cylinders and weight.
    tmpDat = mtcars.loc[
        (mtcars["cyl"] == cylValue)
        & (mtcars["wt"] >= wrRange[0])
        & (mtcars["wt"] <= wrRange[1]),
        :,
    ]

    # Create plotnine's plot showing all data ins smaller grey points, and
    # the selected data with coloured points.
    p = (
        ggplot(tmpDat, aes(x="hp", y="mpg", color="wt"))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        + p9.geom_point(mtcars, color="grey")
        + p9.geom_point(size=3)
        + p9.theme_linedraw()
        + p9.xlim([minHP, maxHP])
        + p9.ylim([minMPG, maxMPG])
        + p9.scale_color_continuous(
            name="spring", limits=(np.floor(minWt), np.ceil(maxWt))
        )
        + p9.labs(x="Horse-Power", y="Miles Per Gallon", color="Weight")
    )

    if fig is None:
        fig = p.draw()
        axs = fig.axes
    else:
        for artist in get_current_artists():
            artist.remove()
        p._draw_using_figure(fig, axs)


cylSelect.observe(plotUpdate, "value")
wtSelect.observe(plotUpdate, "value")

# Display the widgets
display(widgetsCtl)

# Plots the first image, with inintial values.
plotUpdate()

# Matplotlib function to make the image fit within the plot dimensions.
plt.tight_layout()

# Trick to get the first rendered image to follow the previous "tight_layout" command.
# without this, only after the first update would the figure be fit inside its␣
↪dimensions.
cylSelect.value = cylList[0]
```

```
HBox(children=(Dropdown(description='Cylinders:', index=1, options=(4, 6, 8), value=6),␣
↪SelectionRangeSlider(d...
```

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

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

```
/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_23701/2074626009.py:91:␣
↪UserWarning: The figure layout has changed to tight
```

3.9 NetworkX

3.9.1 Beispiel

Im folgenden Beispiel erstellen wir eine einfache Benutzeroberfläche zum Erkunden von Zufallsgraphen mit NetworkX.

[1]:
```
import networkx as nx
```

Zunächst erstellen wir einige Funktionen zur Erzeugung von Graphen, die alle die gleiche Signatur haben:

[2]:
```
import matplotlib.pyplot as plt


def random_lobster(n, m, k, p):
    return nx.random_lobster(n, p, p / m)


def powerlaw_cluster(n, m, k, p):
    return nx.powerlaw_cluster_graph(n, m, p)


def erdos_renyi(n, m, k, p):
    return nx.erdos_renyi_graph(n, p)


def newman_watts_strogatz(n, m, k, p):
    return nx.newman_watts_strogatz_graph(n, k, p)


def plot_random_graph(n, m, k, p, generator):
    g = generator(n, m, k, p)
    nx.draw(g)
    plt.show()
```

Nun erstellen wir Schieberegler und ein Aufklappmenü zur Interaktion mit Plot:

[3]:
```
from ipywidgets import interact

%matplotlib inline

interact(plot_random_graph, n=(2,30), m=(1,10), k=(1,10), p=(0.0, 1.0, 0.001),
         generator={
             'lobster': random_lobster,
             'power law': powerlaw_cluster,
             'Newman-Watts-Strogatz': newman_watts_strogatz,
             u'Erdős-Rényi': erdos_renyi,
         });
```

```
interactive(children=(IntSlider(value=16, description='n', max=30, min=2),␣
↪IntSlider(value=5, description='m',...
```

3.9.2 Graphviz

Graphviz kann auf zweierlei Arten verwendet werden:

- zusammen mit Matplotlib und pygraphviz, siehe Drawing NetworkX with Matplotlib
- zusammen mit Graphviz AGraph, siehe Drawing NetworkX with Graphviz AGraph

3.10 Graphviz

`Graph`- und `Digraph`-Objekte haben eine `_repr_svg_()`-Methode, sodass sie direkt in einem Jupyter-Notebook gerendert und dargestellt werden können.

3.10.1 Installation

Mit Spack könnt ihr Graphviz in eurem Kernel bereitstellen:

```
$ spack env activate python-311
$ spack env status
==> In environment python-311
$ spack install py-graphviz
...
==> Successfully installed py-graphviz
...
==> Updating view at /srv/jupyter/spack/var/spack/environments/python-311/.spack-env/view
```

Zunächst wird Graphviz installiert mit:

```
$ sudo apt install graphviz
```

Anschließend könnt ihr dann in eurem Kernel das zugehörige Python-Paket installieren:

```
$ pipenv install graphviz
```

Zunächst wird Graphviz mit Homebrew installiert:

```
$ brew install graphviz
```

Anschließend könnt ihr dann in eurem Kernel das zugehörige Python-Paket installieren:

```
$ pipenv install graphviz
```

Beispiele

`Graph`- und `Digraph`-Objekte haben eine `_repr_svg_()`-Methode, sodass sie direkt in einem Jupyter-Notebook gerendert und dargestellt werden können.

Einfaches Beispiel

[1]:
```
import graphviz
```

[2]:
```
dot = graphviz.Digraph("hello-pythonistas", comment="Hello world example")

dot.edge("Hello", "Pythonistas!")

dot
```

[2]:

Ihr könnt euch auch den Quelltext ausgeben lassen mit:

[3]:
```
print(dot.source)
```

```
// Hello world example
digraph "hello-pythonistas" {
        Hello -> "Pythonistas!"
}
```

Auch die Ausgabe des Kommentars oder anderer Elemente des Quelltexts sind möglich, z.B. mit:

[4]:
```
print(dot.comment)
```

```
Hello world example
```

Ihr könnt auch Daten aus einem pandas DataFrame verwenden, z.B.:

[5]:
```
import pandas as pd


j = {
    "action": [
        "single use",
        "teamwork",
        "convert",
        "Java, R, Julia etc.",
        "extend",
    ],
    "view": ["Jupyter", "JupyterHub", "nbconvert", "kernels", "extensions"],
}

df = pd.DataFrame(j)

df
```

[5]:

	action	view
0	single use	Jupyter
1	teamwork	JupyterHub
2	convert	nbconvert
3	Java, R, Julia etc.	kernels
4	extend	extensions

[6]:
```
jm = graphviz.Graph("jupyter_moons", comment="Jupyter moons")

jm.node("What do you want to do?")

for index, row in df.iterrows():
    jm.edge(
        "What do you want to do?", str(row["view"]), label=(str(row["action"]))
    )

jm
```

[6]:

Styling

Ihr könnt `graph_attr`-, `node_attr`- und `edge_attr`-Argumente der `Graph`- und `Digraph`-Konstuktoren verwenden, um die Standardattribute von Graphviz für eure Graphen, Knoten und Kanten zu ändern, z.B.:

[7]:
```
dot = graphviz.Digraph(
    "hello-pythonistas",
    comment="Hello world example",
    node_attr={"shape": "plaintext"},
)

dot.edge("Hello", "Pythonistas!")

dot
```

[7]:

Die `graph_attr`-, `node_attr`- und `edge_attr`-Argumente können auch auf Instanzen angewendet werden:

[8]:
```
dot.graph_attr["rankdir"] = "LR"

dot
```

[8]:

Um `att_stmt`-Attributanweisungen direkt hinzuzufügen, ruft die `attr()`-Methode der `Graph`- oder `Digraph`-Instanz mit dem gewünschten Ziel als erstes Argument und den Attributen als Schlüsselwort-Argument auf.

Hinweis:

Attribut-Anweisungen wirken sich auf alle späteren Graphen, Knoten oder Kanten innerhalb desselben (Sub-)Graphen aus.

Engines

Neben Dot können auch verschiedene andere Layout Engines verwendet werden.

[9]:
```
pvl = graphviz.Graph("python_visualisation_landscape", engine="neato")

pvl.edge("Matplotlib", "pandas")
pvl.edge("pandas", "GeoPandas")
pvl.edge("Matplotlib", "Geoplot")
pvl.edge("Matplotlib", "descartes")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
pvl.edge("Matplotlib", "seaborn")
pvl.edge("Matplotlib", "ggpy")
pvl.edge("Matplotlib", "plotnine")
pvl.edge("Matplotlib", "scikit_plot")

pvl
```

[9]:

Ihr könnt auch das `engine`-Attribut einer bestehenden Instanz ändern:

[10]:
```
pvl.engine = "circo"

pvl
```

[10]:

Eine vollständige Übersicht über die Engines erhaltet ihr mit:

[11]:
```
for engine in sorted(graphviz.ENGINES):
    print(engine)
```

```
circo
dot
fdp
neato
osage
patchwork
sfdp
twopi
```

Dateien schreiben und lesen

[12]:
```
pvl.save()
```

[12]:
```
'python_visualisation_landscape.gv'
```

[13]:
```
graphviz.Source.from_file("python_visualisation_landscape.gv")
```

[13]:

Dot-Dateien konvertieren

Eine Dot-Datei kann in ein anderes Format, z.B. PDF, PNG, SVG etc., umgewandelt werden mit `render`:

[14]:
```
from graphviz import render


render("dot", "svg", "python_visualisation_landscape.gv")
```

[14]:
```
'python_visualisation_landscape.gv.svg'
```

Siehe auch:

- graphviz manual

- examples/graphviz-notebook
- examples/graphviz-engines
- examples/graphviz-escapes.ipynb
- Graphviz Online

Graphviz-Bibliotheken und -Werkzeuge

Python-Bibliotheken

pydot
Python-Schnittstelle zu Graphviz

PyGraphviz
Python-Schnittstelle zu Graphviz ähnlich wie *NetworkX*

GvGen
Python-Klasse zur Erzeugung von Dot-Dateien

pytm
Framework für die Modellierung von Buffer Overflows, SQL Injections, CSRF ETC.

graph-tool
Python-Modul zur Manipulation und statistischen Analyse von Graphen

Netzwerk-Tools

dnsviz
Tool-Suite zur Analyse und Visualisierung des DNS (Domain Name Systems), einschließlich seiner Sicherheitserweiterungen (DNSSEC)

TraceViz
Visualisierung der Traceroute-Ausgabe mit Graphivz

Safe Mapping and Reporting Tool (SMART)
Passives Netzwerkfluss-Visualisierungstool für kleine bis mittelgroße IP-Netzwerke mit Geräte- und Betriebssystem-Identifikation und Aufzählung von Netzwerkdiensten.

dockviz
Visualisierung von Docker-Daten

docker-compose-dot
erzeugt Graphviz-Dot-Dateien aus docker-compose `yaml`-Dateien

Dotler
Web-Crawler und Graph-Generator

sabaviz
visualisiert Server-Verbindungen anhand der Netstat-Ausgabe

Kafka Streams Topology Visualizer
visualisiert Kafka-Streams-Topologien mithilfe von Viz.js und Rough.js

Konfigurationsmanagement

terraform graph
: überführt die Konfiguration in eine Dot-Datei

Ansible Playbook Grapher
: Kommandozeilentool, um ein Diagramm aus Ansible-Playbooks zu erstellen

ansible-inventory-grapher
: erstellt Dot-Dateien aus Ansible-Inventories

Profiler

pprof
: visualisiert und analysiert Profildaten

Build-Systeme

CMake
: kann Dot-Dateien erzeugen, die die Abhängigkeiten in einem Projekt sowie zu externen Bibliotheken, gegen die gelinkt wird, anzeigen

makefile2graph
: erzeugt einen Abhängigkeitsgraphen aus GNU-Make-Dateien

3.11 graph-tool

graph-tool stellt eine Graph-Klasse und mehrere Algorithmen zur Verfügung, die mit ihr arbeiten. Die Interna dieser Klasse und der meisten Algorithmen sind aus Leistungsgründen in C++ geschrieben und verwenden die Boost Graph Library.

3.11.1 Installation

graph-tool ist eine in Python verpackte C++-Bibliothek mit vielen C++-Abhängigkeiten wie Boost, CGAL und expat. Daher lässt sich graph-tool am einfachsten installieren mit einem Paketmanager für Linux-Distributionen und macOS:

Linux

Für Debian oder Ubuntu könnt ihr die folgende Zeile zu eurer `/etc/apt/sources.list` hinzufügen:

```
deb [arch=amd64] https://downloads.skewed.de/apt DISTRIBUTION main
```

wobei `DISTRIBUTION` einer der folgenden Werte sein kann:

```
bullseye, buster, sid, bionic, eoan, focal, groovy
```

Anschließend solltet ihr den öffentlichen Schlüssel 612DEFB798507F25 herunterladen, um die Pakete mit dem folgenden Befehl zu überprüfen:

```
$ apt-key adv --keyserver keys.openpgp.org --recv-key 612DEFB798507F25
```

Nach dem Ausführen von `apt update` kann das Paket installiert werden mit

```
$ apt install python3-graph-tool
```

macOS

Mit Homebrew ist die Installation ebenfalls unkompliziert:

```
$ brew install graph-tool
```

Anschließend müssen wir dem Python-Interpreter noch mitteilen, wo er `graph-tool` finden kann:

```
$ export PYTHONPATH="$PYTHONPATH:/usr/local/Cellar/graph-tool/2.43/lib/python3.9/site-
↪packages"
```

Testen der Installation

[1]:
```
from graph_tool.all import *
```

3.11.2 Erstellen und Manipulieren von Graphen

Ein leerer Graph kann durch Instanziierung einer Graph-Klasse erstellt werden:

[2]:
```
g = Graph()
```

Ein Graph kann mit der Methode set_directed() jederzeit von gerichtet auf ungerichtet umgestellt werden. Dabei kann die Richtung des Graphen mit der Methode is_directed() abgefragt werden:

[3]:
```
ug = Graph()
ug.set_directed(False)
assert ug.is_directed() == False
```

Sobald ein Graph erstellt ist, kann er mit Knoten und Kanten gefüllt werden. Ein Knoten kann mit der Methode add_vertex() hinzugefügt werden, die eine Instanz einer Vertex-Klasse, auch *vertex descriptor* genannt, zurückgibt. Der folgende Code erstellt beispielsweise zwei Knoten und gibt *vertex descriptors* zurück, die in den Variablen `v1` und `v2` gespeichert sind:

[4]:
```
v1 = g.add_vertex()
v2 = g.add_vertex()
```

[5]:
```
e = g.add_edge(v1, v2)
```

[6]:
```
graph_draw(g, vertex_text=g.vertex_index, output="two-nodes.svg")
```

[6]:
```
<VertexPropertyMap object with value type 'vector<double>', for Graph 0x1814b13d0, at␣
↪0x181ea0520>
```

[7]:
```
from IPython.display import SVG
SVG('two-nodes.svg')
```

[7]:

3.12 Cartopy

Cartopy erstellt Karten auf Basis von *Matplotlib* und rechnet Punkte, Linien und Vektoren zwischen den unterschiedlichen Projektionen um. Es basiert auf PROJ.4, NumPy und Shapely. Pyresample verwendet Cartopy um SatPy-Daten in die verschiedenen Projektionen umzurechnen.

Siehe auch:

- Docs
- Gallery
- Cartopy Tutorial
- SciTools courses: Cartopy Intro

3.12.1 Cartopy-Installation

Mit Spack könnt ihr Cartopy in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install py-cartopy~epsg~ows~plotting
```

Dies installiert Cartopy mit Unterstützung von:

- epsg
- Open Geospatial Consortium (OGC)
- Plot-Funktionalität

Zusätzlich werden folgende Pakete mitinstalliert:

- gdal
- *Matplotlib*
- OWSLib
- Pillow
- pyepsg
- PyShp
- shapely
- six

Cartopy kann alternativ auch mit Linux-Paketmanagern installiert werden, z.B. für Debian9 (Stretch) mit:

```
$ sudo apt install proj-bin
$ sudo apt install libproj-dev
$ sudo apt install libgeos-dev
```

```
$ brew install proj
$ brew install geos
```

Anschließend kann Cartopy für euren Kernel installiert werden, z.B. mit:

```
$ export PIP_NO_BINARY=:shapely:
$ pipenv install cython numpy cartopy
```

Für die *Cartopy-Beispiele* benötigt ihr dann zusätzlich die folgenden beiden Python-Pakete:

```
$ pipenv install matplotlib scipy
```

Optionale Anforderungen

- rtree für libspatialindex
- psycopg2 für PostGIS
- geopy

Anforderungen zum Plotten

- *Matplotlib*
- descartes
- mapclassify

Überprüfen

Schließlich könnt ihr die Installation überprüfen mit:

```
>>> import cartopy
```

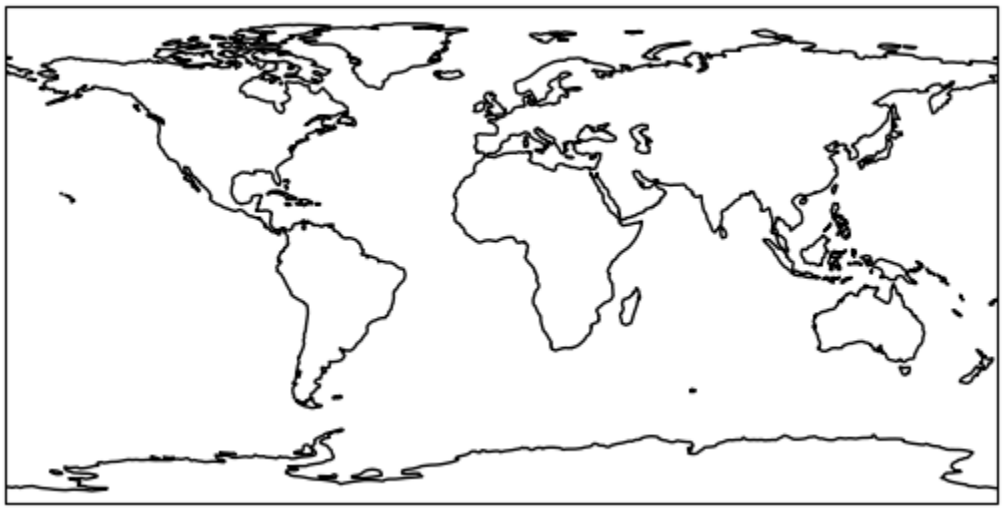
3.12.2 Cartopy-Beispiele

[1]:
```
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
```

Abstandstreue Plate-Carrée-Projektion

[2]:
```
ax = plt.axes(projection=ccrs.PlateCarree())
ax.coastlines()

plt.show()
```



Flächentreue Mollweide-Projektion

[3]:
```
ax = plt.axes(projection=ccrs.Mollweide())
ax.coastlines()

plt.show()
```

3.13 Iris

Iris implementiert ein Datenmodell basierend auf Climate and Forecast (CF) Conventions, wobei die Visualisierung auf *Matplotlib* und *Cartopy* basiert.

Siehe auch:

- User guide
- Gallery
- General visualisation examples
- Meteorology visualisation examples
- Oceanography visualisation examples
- SciTools courses: Iris course
- SciTools: Iris example code
  - GitHub
  - nbviewer

3.14 yt

yt analysiert und visualisiert Volumendaten.

3.14.1 Installation

```
$ pipenv install yt
```

Bemerkung: Falls ihr pipenv noch nicht installiert hab, findet ihr eine Anleitung hierzu unter Installation.

Die Installation könnt ihr dann überprüfen mit:

```
$ yt -h
usage: yt [-h] [--config CONFIG] [--paste] [--paste-detailed] [--detailed]
...
```

3.14.2 Beispiel

```
>>> import yt
>>> ds = yt.load("IsolatedGalaxy/galaxy0030/galaxy0030")
yt : [INFO     ] 2019-06-05 13:10:39,581 Parameters: current_time              = 0.
↪0060000200028298
yt : [INFO     ] 2019-06-05 13:10:39,583 Parameters: domain_dimensions         = [32 32␣
↪32]
yt : [INFO     ] 2019-06-05 13:10:39,585 Parameters: domain_left_edge          = [0. 0.␣
↪0.]
yt : [INFO     ] 2019-06-05 13:10:39,586 Parameters: domain_right_edge         = [1. 1.␣
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
↪1.]
yt : [INFO     ] 2019-06-05 13:10:39,588 Parameters: cosmological_simulation   = 0.0
>>> ds.r[0.45:0.55,:,:].sum("cell_mass").in_units("Mjup")
Parsing Hierarchy : 100%|                                                    | 173/173 [00:00<00:00, 3427.19it/s]
yt : [INFO     ] 2019-06-05 13:10:39,676 Gathering a field list (this may take a moment.)
9985379895930.627 Mjup
```

Bemerkung: In Version 3.3 von yt wurde ein experimenteller hardware-beschleunigter interaktiver Volume-Renderer eingeführt. Um die Interactive Data Visualization (IDV) verwenden zu können, müssen zusätzlich PyOpenGL und cyglfw3 mit ihren jeweiligen Abhängigkeiten installiert werden.

Siehe auch:

- Quickstart
- Cookbook
- Docs
- Extensions

KAPITEL 4

Vega

Vega ist eine deklarative Sprache zum Erstellen, Speichern und Teilen interaktiver Visualisierungsdesigns. Mit Vega könnt ihr das visuelle Erscheinungsbild und das interaktive Verhalten einer Visualisierung in einem JSON-Format beschreiben und webbasierte Ansichten mit Canvas oder SVG generieren.

Siehe auch:

- Example Gallery
- Tutorials
- Documentation
- Usage
- Vega and D3

4.1 Altair

Altair ist eine deklarative statistische Visualisierungsbibliothek für Python, die auf *Vega* und Vega-Lite basiert.

See also:

- Docs
- Example Gallery
- User Guide

4.1.1 Installation

```
$ pipenv install altair
```

Altair hat die folgenden Abhängigkeiten, die automatisch mit dem oben genannten Installationsbefehl installiert werden:

- importlib_metadata
- entrypoints
- typing_extensions
- jinja2
- jsonschema
- NumPy
- pandas
- Toolz

4.1.2 Beispiel

Hier ist ein Beispiel für die Verwendung der Altair-API zur schnellen Visualisierung eines Datensatzes mit einem interaktiven Streudiagramm:

1. Import

[1]:
```
import altair as alt
```

2. Laden eines einfachen Datensets als pandas DataFrame

[2]:
```
from vega_datasets import data

cars = data.cars()
```

[3]:
```
cars.head()
```

[3]:
```
                       Name  Miles_per_Gallon  Cylinders  Displacement  \
0  chevrolet chevelle malibu              18.0          8         307.0
1          buick skylark 320              15.0          8         350.0
2         plymouth satellite              18.0          8         318.0
3              amc rebel sst              16.0          8         304.0
4                ford torino              17.0          8         302.0

   Horsepower  Weight_in_lbs  Acceleration       Year Origin
0       130.0           3504          12.0 1970-01-01    USA
1       165.0           3693          11.5 1970-01-01    USA
2       150.0           3436          11.0 1970-01-01    USA
3       150.0           3433          12.0 1970-01-01    USA
4       140.0           3449          10.5 1970-01-01    USA
```

3. Erstellen eines interaktiven Streudiagramms

[4]:
```
alt.Chart(cars).mark_point().encode(
    x="Horsepower",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    y="Miles_per_Gallon",
    color="Origin",
).interactive()
```

```
[4]: alt.Chart(...)
```

Die wesentliche Idee besteht darin, dass ihr Verknüpfungen zwischen Datenspalten und visuellen Dimensionen wie der x-Achse, der y-Achse, der Farbe usw. deklarieren könnt. Die restlichen Details der Darstellung werden automatisch gehandhabt. Aufbauend auf dieser deklarativen Plot-Idee lässt sich mit einer relativ prägnanten Grammatik eine erstaunliche Bandbreite von einfachen bis hin zu anspruchsvollen Plots und Visualisierungen erstellen.

4.2 PdVega

PdVega ist eine Bibliothek, mit der interaktive Vega-Lite-Plots aus pandas-Dataframes unter Verwendung des pandas-Plot-API erstellt werden können.

Siehe auch:

- Advanced Plotting: Using Vega-Lite Directly
- API Reference

4.2.1 PdVega-Installation

PdVega kann installiert werden mit

```
$ pipenv install pdvega
Installing pdvega...
Adding pdvega to Pipfile's [packages]...
✓ Installation Succeeded
...
$ pipenv run jupyter nbextension install --sys-prefix --py vega3
Installing /srv/jupyter/.local/share/virtualenvs/python-374-c_ntaqVM/lib/python3.7/site-
↪packages/vega3/static -> jupyter-vega3
...
- Validating: OK

    To initialize this nbextension in the browser every time the notebook (or other app)␣
↪loads:

          jupyter nbextension enable vega3 --py --sys-prefix
```

Siehe auch:

- Installing and Using pdvega

Abhängigkeiten

Um Plots als PNG oder SVG speichern zu können, müssen die Kommandozeilenwerkzeuge `vl2png` und `vl2svg` aus dem vega-lite-npm-Paket vorhanden sein.

4.2.2 PdVega-Beispiele

Importe

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

Deklarative Beschreibung der Datenvisualisierung

Mit Vega-Lite lässt sich deklarativ beschreiben, wie die Daten auf Visualisierungsfunktionen abgebildet werden sollen. Mit `pdvega` wird diese Spezifikation ähnlich einfach wie über die Matplotlib-API verfügbar: `data.plot` muss lediglich durch `data.vgplot` ersetzt werden, wobei sich `data` auf pandas-`Series` oder `DataFrame`-Objekte bezieht.

Laden eines `DataFrame` mit Zeitreihen von Aktienkursen:

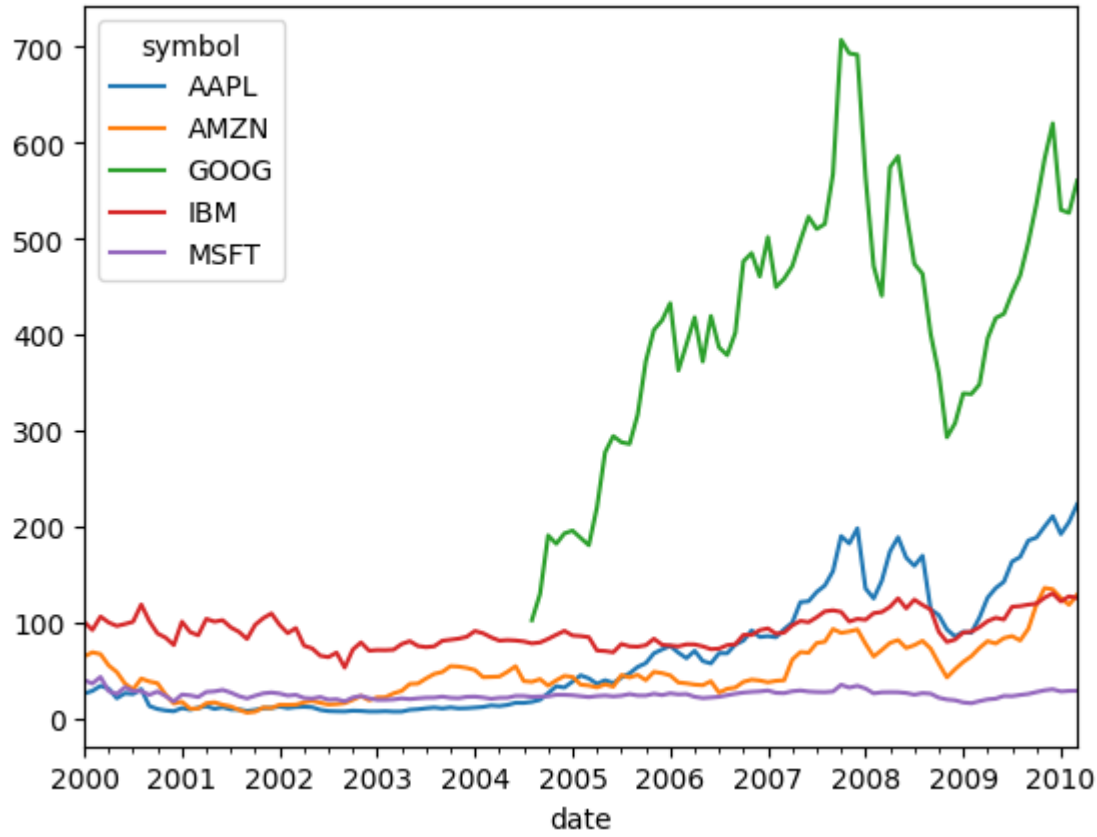
[2]:
```
from vega_datasets import data


stocks = data.stocks(pivoted=True)
```

Matplotlib-API:

[3]:
```
stocks.plot.line()
```

[3]:
```
<Axes: xlabel='date'>
```

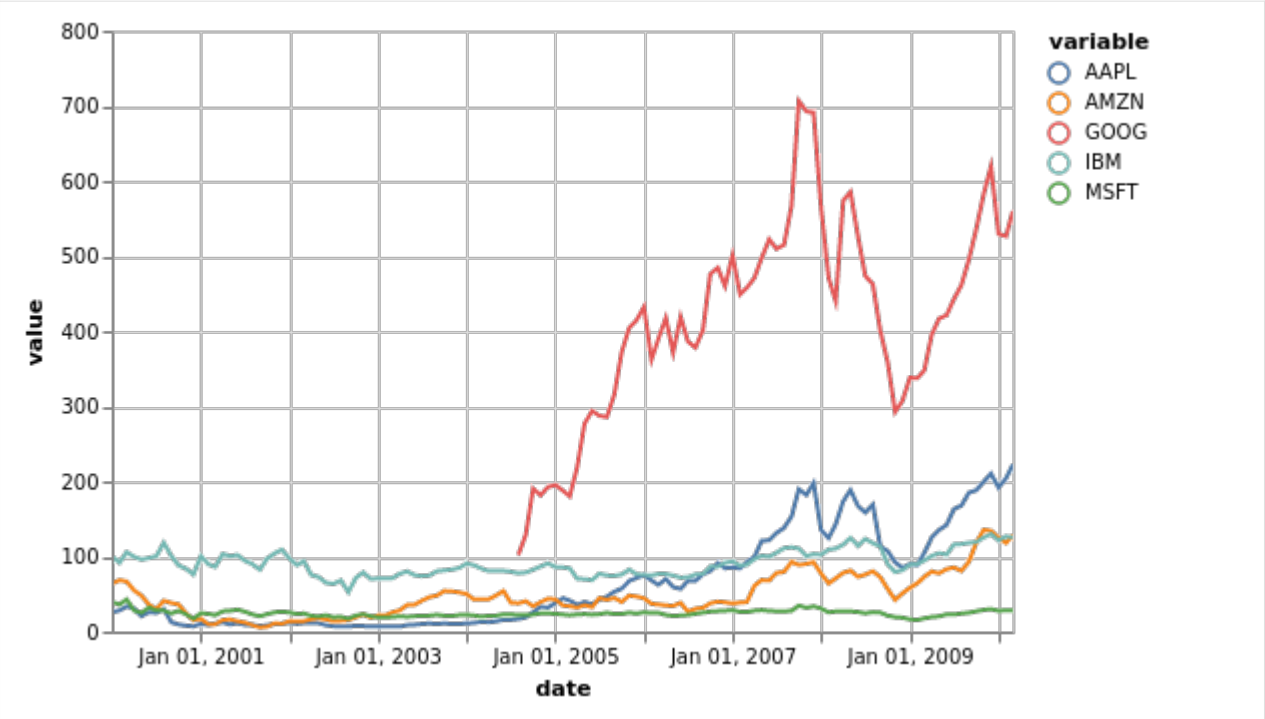


pdvega-API

Hinweis:

vegalite no longer working with pandas 1.0 or later version

[5]:
```
stocks.vgplot.line()
```



Das Ergebnis sind schöne Datenvisualisierungen mit einem Minimum an Boilerplate. Zudem sind die aus `pdvega` erstellten Diagramme interaktiv und lassen sich Verschieben und Vergrößern/Verkleinern.

Einfache Datenvisualisierungen mit `data.vgplot`

Die zentrale Schnittstelle von `pdvega` ist das `vgplot`-Attribut, das pandas-`DataFrame` und `Series`-Objekten hinzugefügt wird.

Wie bei dem pandas-Plots gibt es zwei Möglichkeiten, Diagramme zu erstellen:

1. das `vgplot`-Attribut eines pandas-Objekts kann direkt aufgerufen werden, also z.B.

```
iris.vgplot(kind="scatter", x="sepalLength", y="petalLength", c="species")
```

2. Alternativ kann auch die spezifische Methode aufgerufen werden, die jedem Diagrammtyp zugeordnet ist:

```
iris.vgplot.scatter(x="sepalLength", y="petalLength", c="species")
```

Dieser Ansatz bietet den Vorteil, dass verfügbare Plott-Typen über die Tabulatorvervollständigung untersucht werden können. Die einzelnen Funktionen bieten auch eine detailliertere Dokumentation der für jede Methode verfügbaren Argumente.

Diagrammtypen

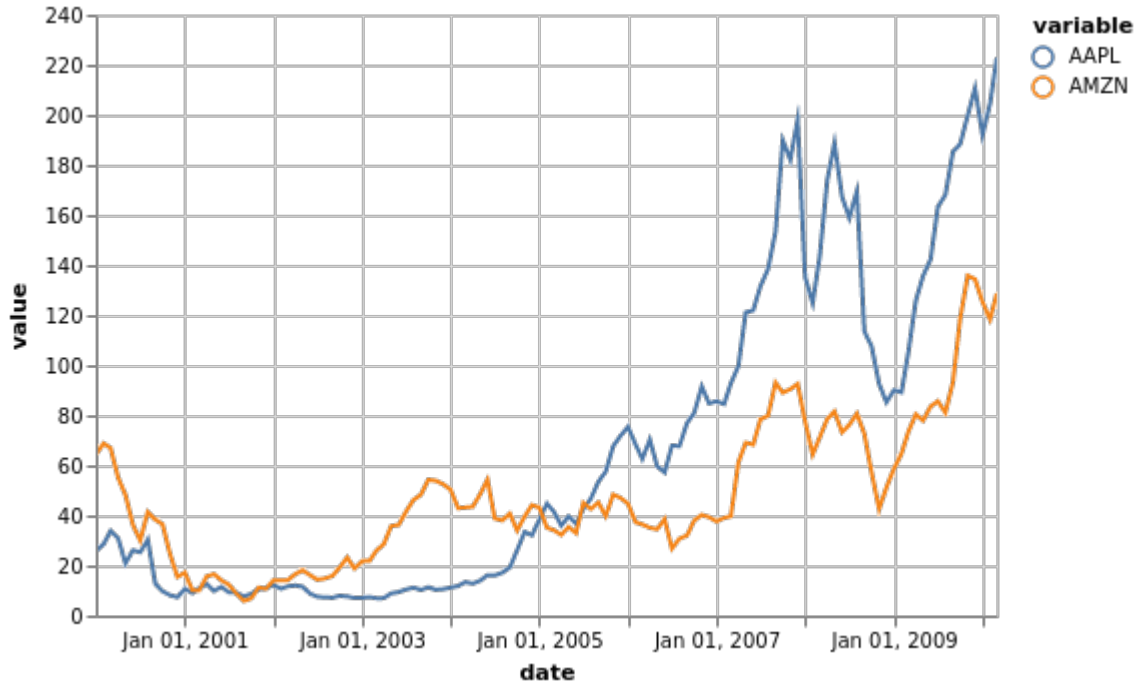
Die `vgplot`-API stellt neun grundlegende Diagrammtypen bereit:

Liniendiagramme mit `vgplot.line`

Der Standard-Diagrammtyp für `vgplot` ist ein Liniendiagramm.

Sofern nichts anders angegeben ist, wird der Index von `DataFrame` oder `Series` als x-Achsenvariable verwendet, und eine separate Linie für die y-Werte jeder Spalte des `DataFrame`. Wenn Sie ihr nur eine Teilmenge der Spalten plotten lassen möchtet, könnt ihr mithilfe der pandas-Indizierung die Spalten auswählen, an denen ihr interessiert seid:

[6]:
```
stocks[['AAPL', 'AMZN']].vgplot.line()
```

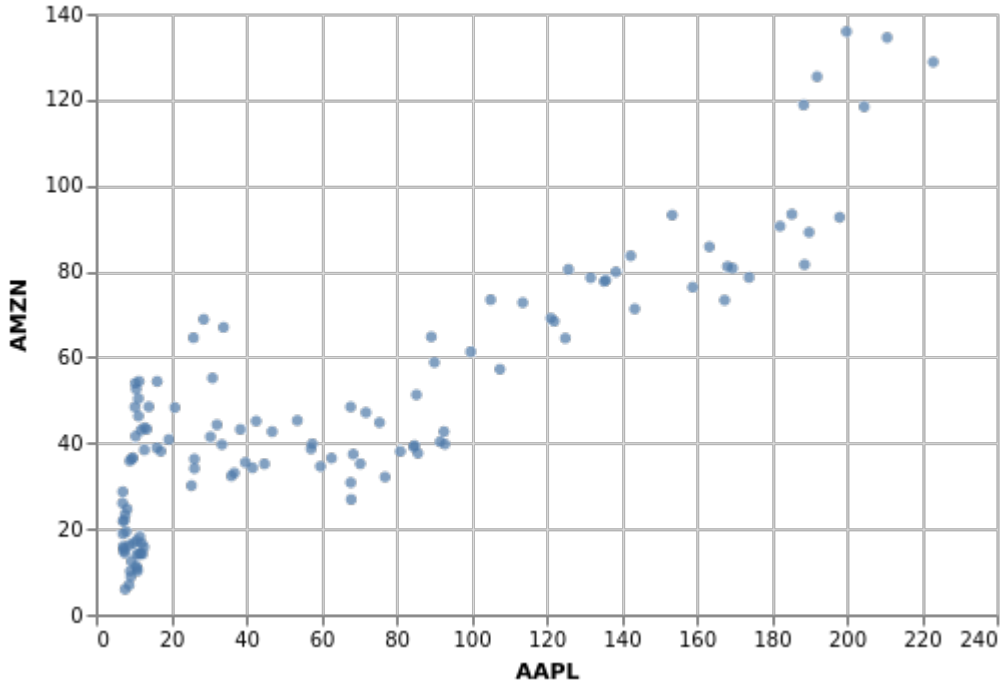


Liniendiagramme können weiter angepasst werden. Informationen hierzu findet ihr in der Dokumentation:

- pdvega.FramePlotMethods.line()
- pdvega.SeriesPlotMethods.line()

Streudiagramme mit `vgplot.scatter`

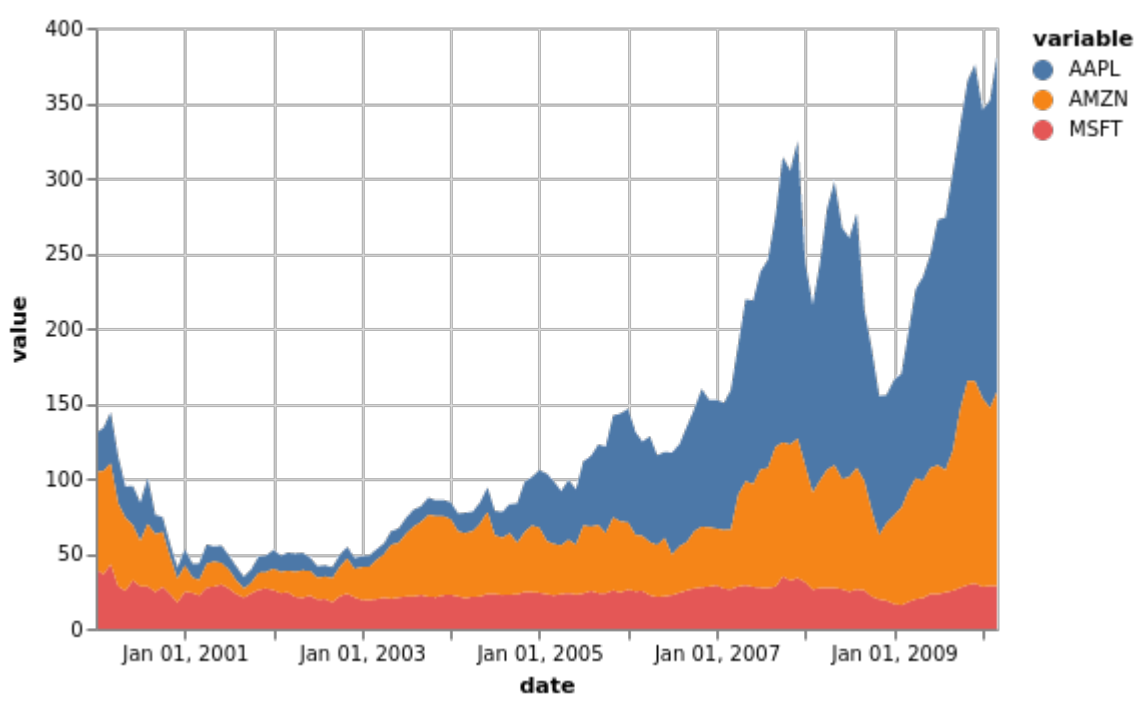
[7]:
```
stocks.vgplot.scatter(x="AAPL", y="AMZN")
```



Um Streudiagramme weiter anzupassen, schaut euch pdvega.FramePlotMethods.scatter() an.

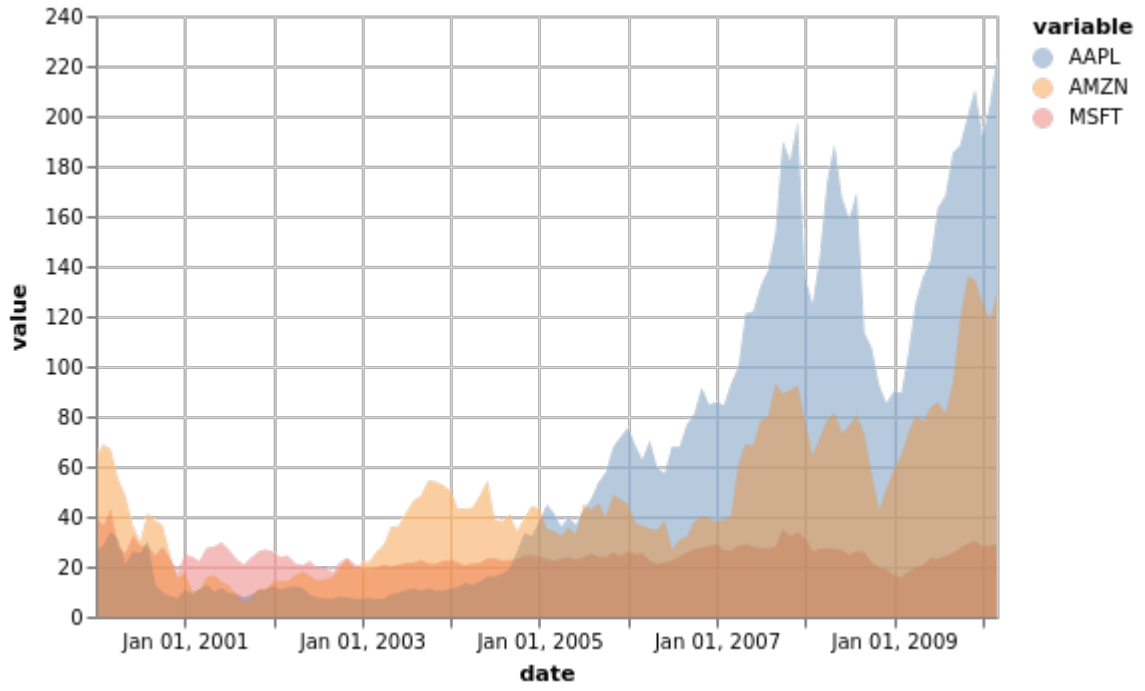
Flächendiagramme mit `vgplot.area`

[8]:
```
stocks[["MSFT", "AAPL", "AMZN"]].vgplot.area()
```



Flächendiagramme können auch gestapelt werden. In diesem Fall sind transparente Flächen häufig hilfreich.

[9]:
```
stocks[["MSFT", "AAPL", "AMZN"]].vgplot.area(stacked=False, alpha=0.4)
```



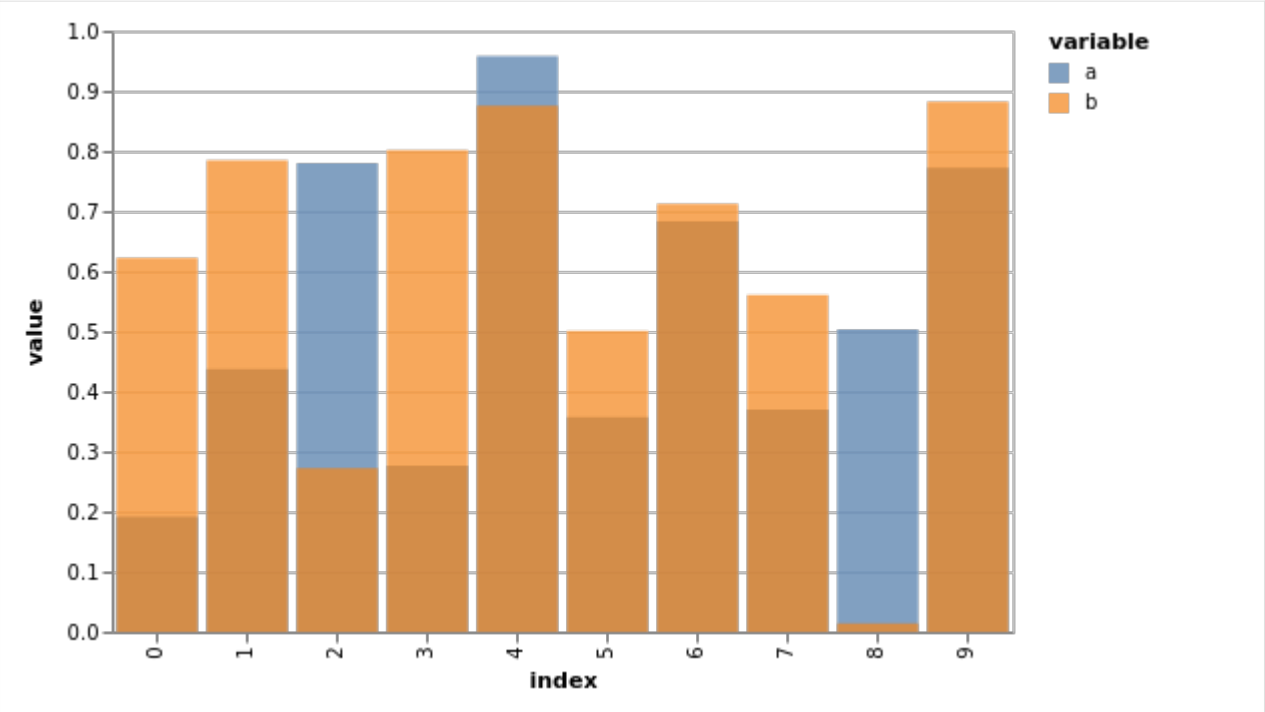
Flächendiagramme können weiter angepasst werden, siehe

- pdvega.FramePlotMethods.area()
- pdvega.SeriesPlotMethods.area()

Balkendiagramme mit `vgplot.bar`

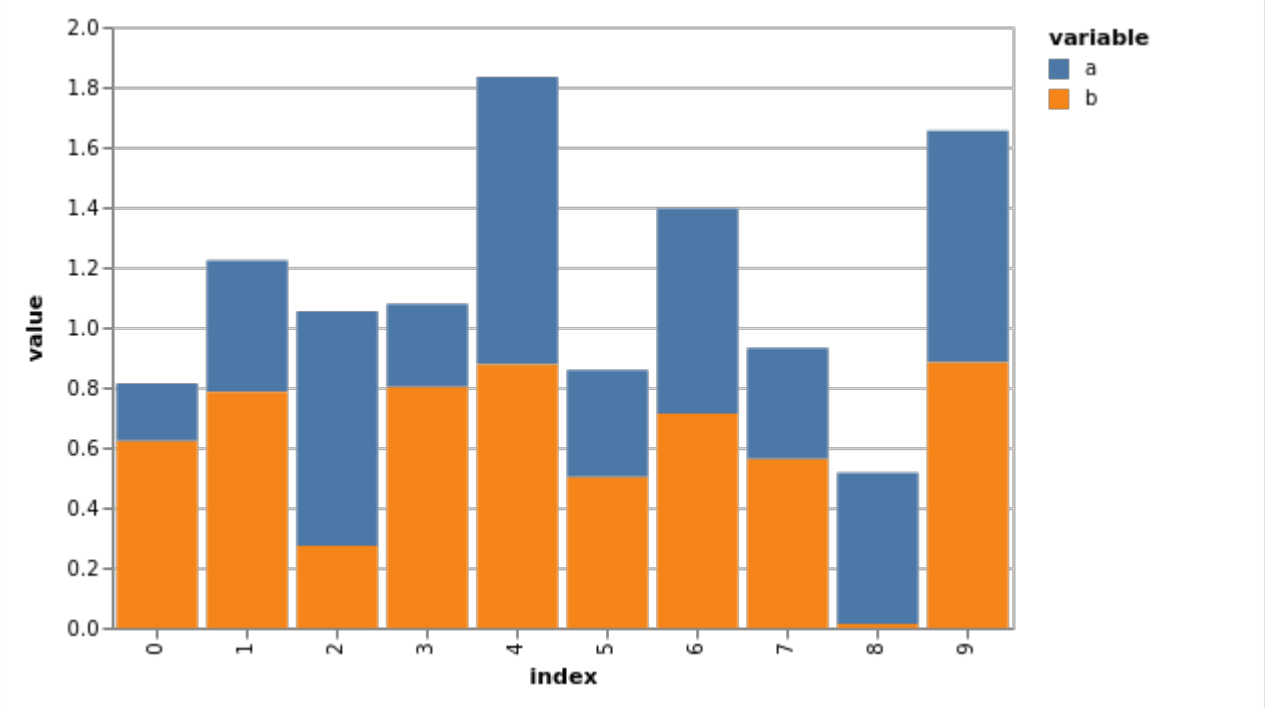
[10]:
```
np.random.seed(1234)
df = pd.DataFrame(np.random.rand(10, 2), columns=["a", "b"])

df.vgplot.bar()
```



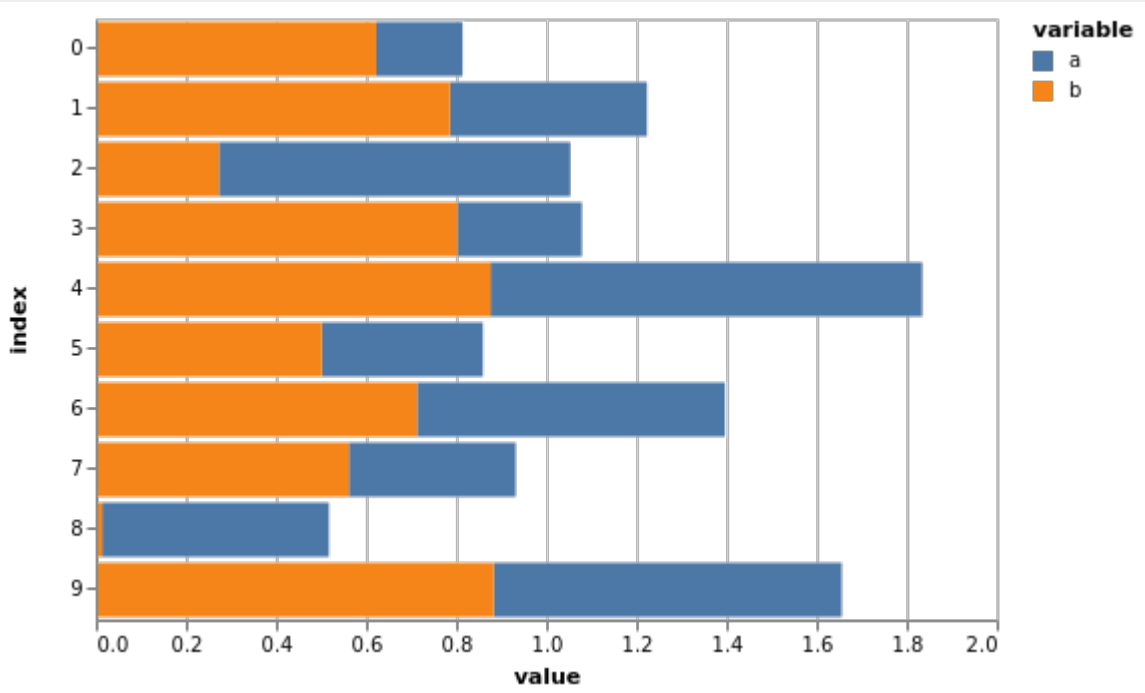
Wie bei Flächendiagrammen könnt ihr mit `stacked=True` die Balken stapeln:

[11]:
```
df.vgplot.bar(stacked=True)
```



Darüberhinaus können horizontale Balkendiagramme erstellt werden mit `barh`:

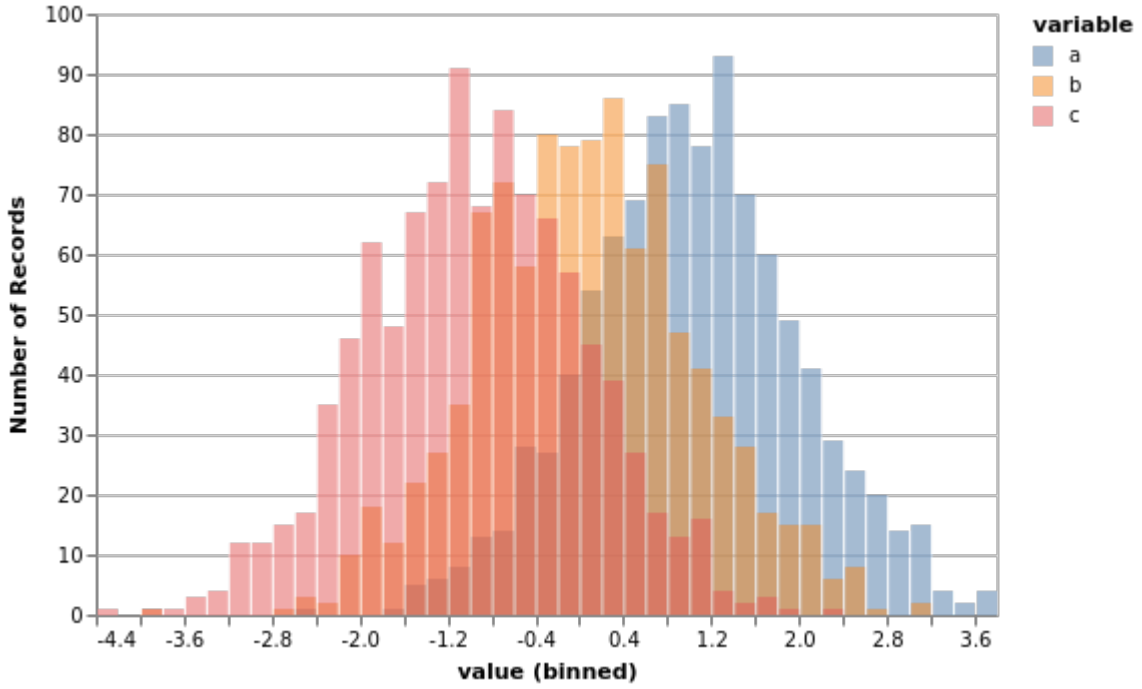
[12]:
```
df.vgplot.barh(stacked=True)
```



Histogramme mit `vgplot.hist`

[13]:
```
df = pd.DataFrame(
    {
        "a": np.random.randn(1000) + 1,
        "b": np.random.randn(1000),
        "c": np.random.randn(1000) - 1,
    },
    columns=["a", "b", "c"],
)

df.vgplot.hist(bins=50, alpha=0.5)
```



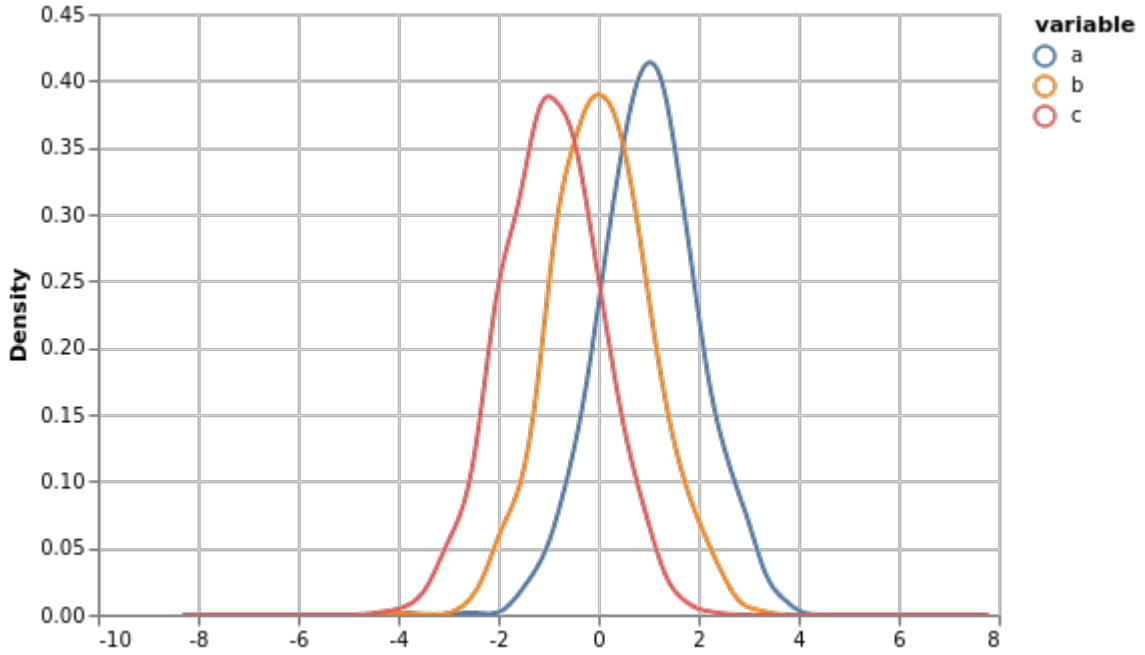
Histogramme können weiter angepasst werden, siehe

- pdvega.FramePlotMethods.hist()
- pdvega.SeriesPlotMethods.hist()

Kerndichteschätzdiagramme mit `vgplot.kde`

Kerndichteschätzdiagramme (englisch *kernel density estimation*, KDE) erzeugen ähnlich wie Histogramme glatte Kurven, die die Dichte der Messpunkte angeben.

[14]:
```
df.vgplot.kde()
```



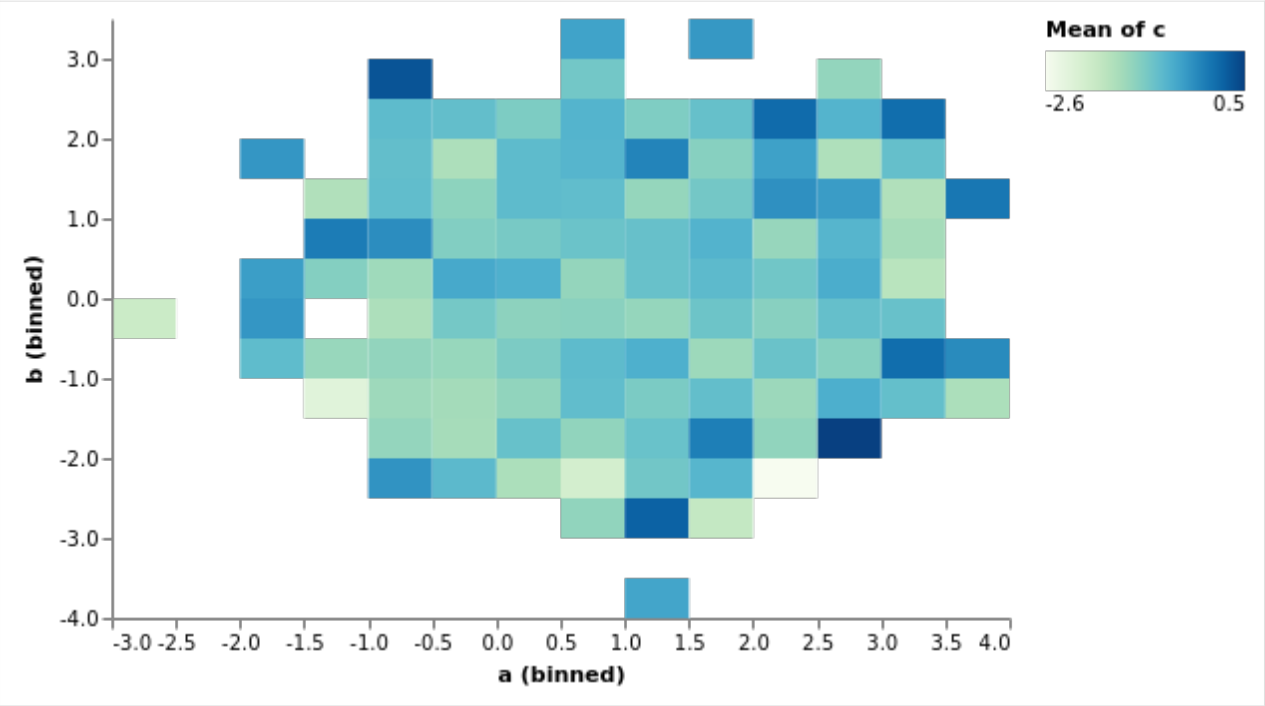
KDE-Diagramme können weiter angepasst werden mit

- pdvega.FramePlotMethods.kde()
- pdvega.SeriesPlotMethods.kde()

Heatmaps mit `vgplot.heatmap`

pandas-Plotting hat eine Funktion zum Erstellen einer hexagonal-gruppierten Heatmap zweidimensionaler Daten. Leider unterstützt derzeit weder Vega noch Vega-Lite diese hexagonalen Heatmaps. Sie unterstützen jedoch kartesische Heatmaps,und diese Funktionalität ist auch enthalten in `pdvega`:

[15]:
```
df.vgplot.heatmap(x="a", y="b", C="c", gridsize=20)
```

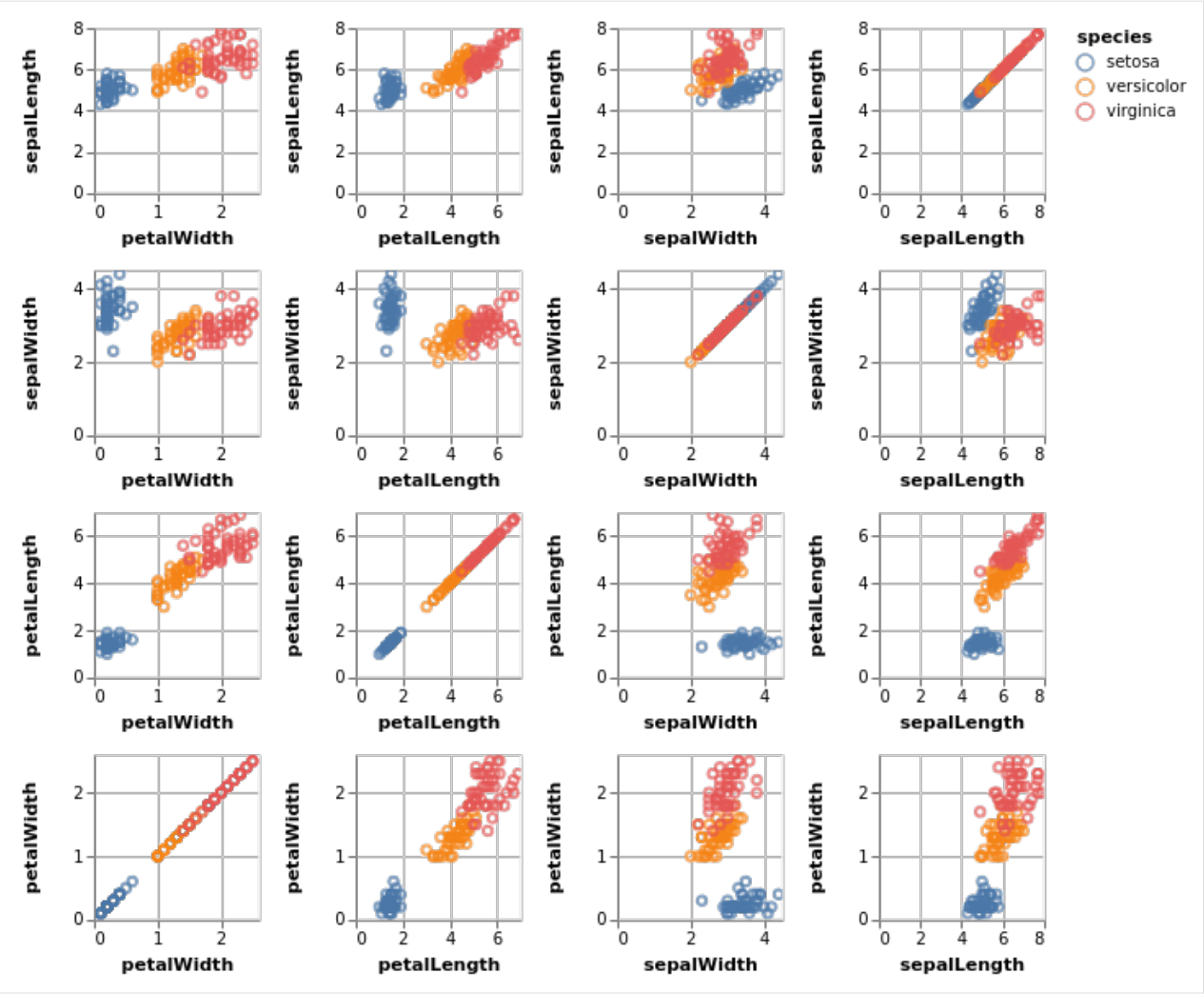


Heatmap-Diagramme können weiter angepasst werden, siehe pdvega.FramePlotMethods.heatmap().

Statistische Visualisierung mit `pdvega.plotting`

pdvegaunterstützt auch viele der komplexeren Plot-Routinen, die im pandas.plotting-Submodul verfügbar sind. Im Folgenden zeigen wir das Beispieleiner Multi-Panel-Streudiagramm-Matrix aus Fisher's Iris-Datensatz:

[16]:
```
iris = data.iris()
pdvega.scatter_matrix(iris, "species", figsize=(7, 7))
```

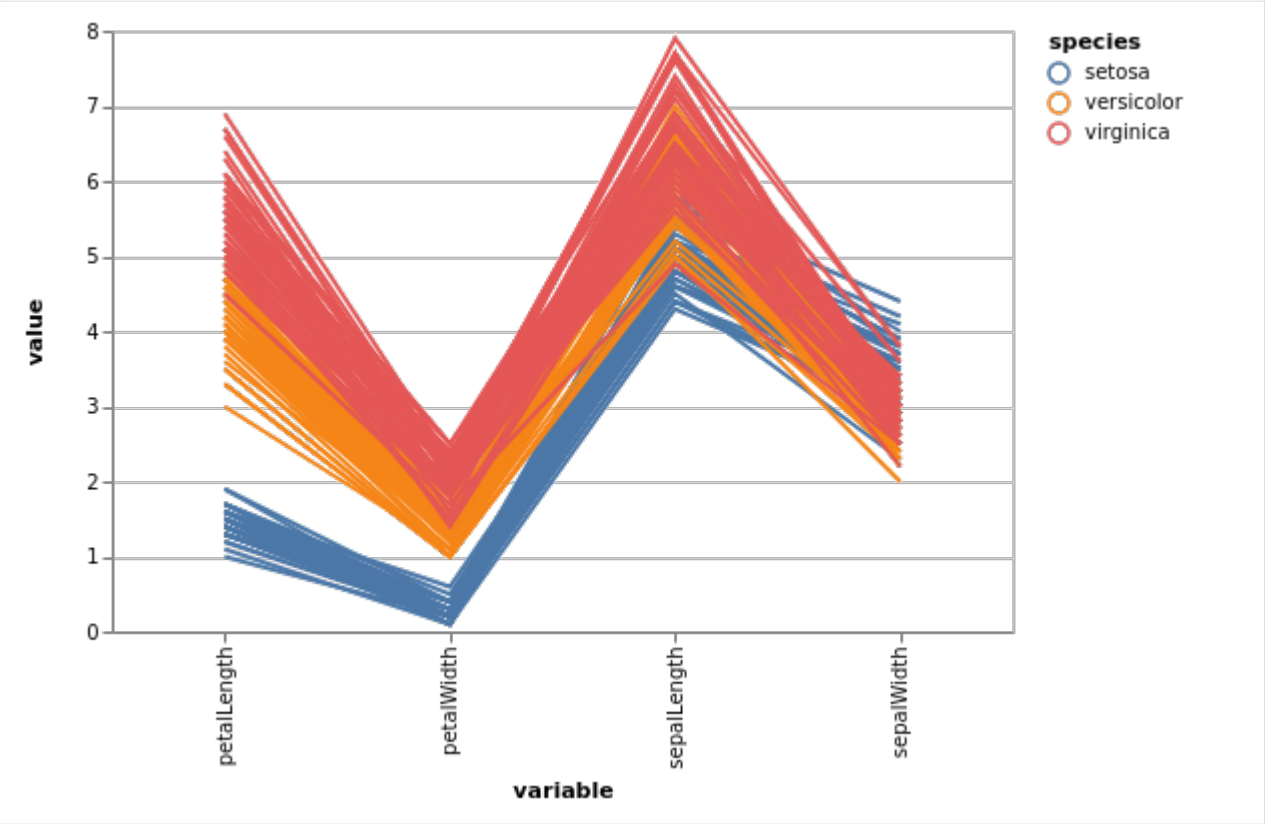


In diesem Diagramm könnt ihr interaktiv Verschieben und Verkleinern/Vergrößern. Mit gedrückter Umschalttaste könnt ihr auch einzelne Messpunkte auswählen.

Parallele Koordinaten

Eine andere Möglichkeit, mehrdimensionale Daten zu visualisieren, besteht darin, jede Dimension unabhängig voneinander mithilfe eines Diagramms mit parallelen Koordinaten zu betrachten. Dies kann mit `pdvega.parallel_coordinates()` realisiert werden, wobei die API `pandas.plotting.parallel_coordinates()` entspricht:

[17]:
```
pdvega.parallel_coordinates(iris, "species")
```

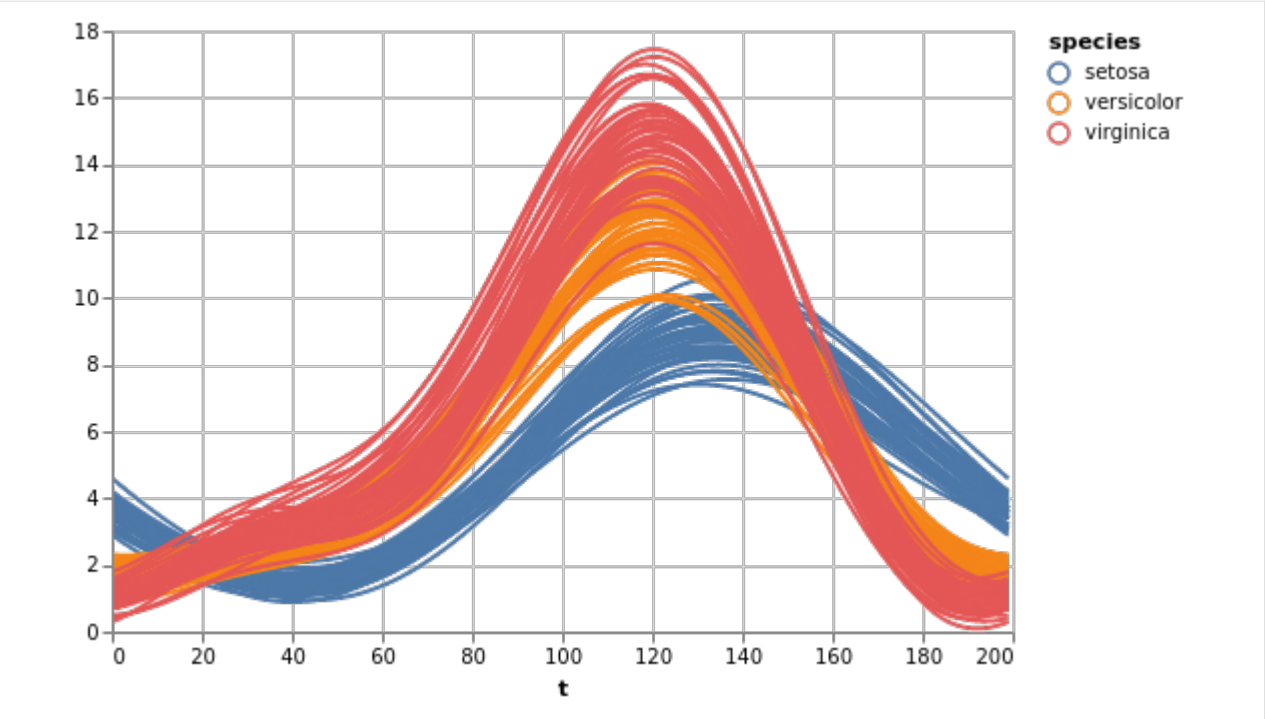


Auf einen Blick könnt ihr Beziehungen zwischen Punkten erkennen und insbesondere deutlich machen, dass sich die „setosa“-Art in Breite und Länge der Blütenblätter deutlich von den beiden anderen Arten unterscheidet.

Andrews-Kurven

Ein ähnlicher Ansatz zur Visualisierung von Datendimensionen ist als *Andrews-Kurve* bekannt: Die Idee besteht darin, aus den Merkmalen jedes Objekts eine Fourier-Reihe zu konstruieren, um die aggregierten Unterschiede zwischen Klassen qualitativ zu visualisieren. Dies kann mit der Funktion `pdvega.andrews_curves()` erfolgen, die der API von `pandas.plotting.andrews_curves()` entspricht:

[17]:
```
pdvega.andrews_curves(iris, "species")
```

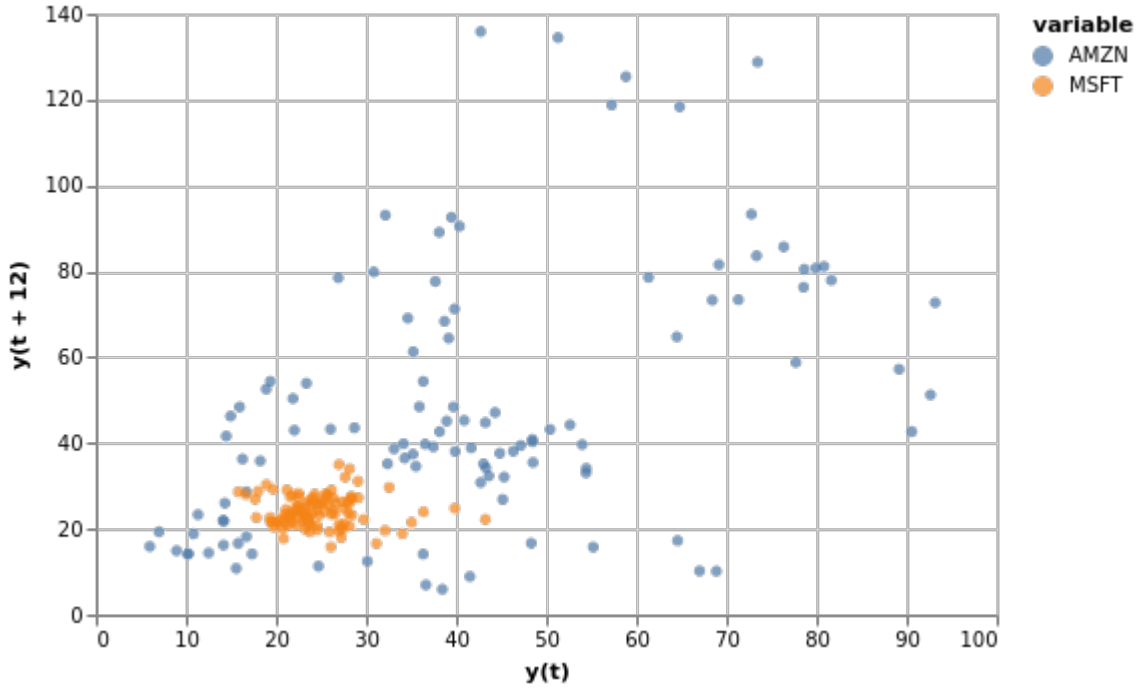


Dies ergibt einen ähnlichen Eindruck wie in der Darstellung der parallelen Koordinaten, gibt jedoch weniger quantitative Hinweise auf die Merkmale, die zu dieser Unterscheidung führen.

Korrelogramm (Lag-Plots)

Korrelogramme sind implementiert mit `pdvega.plotting.lag_plot()` wobei die API `pandas.plotting.lag_plot()` entspricht. Im Folgenden werden die Aktienkurse von Amazon und Microsoft von 1998 bis 2010 mit einer Verzögerung von 12 Monaten visualisiert:

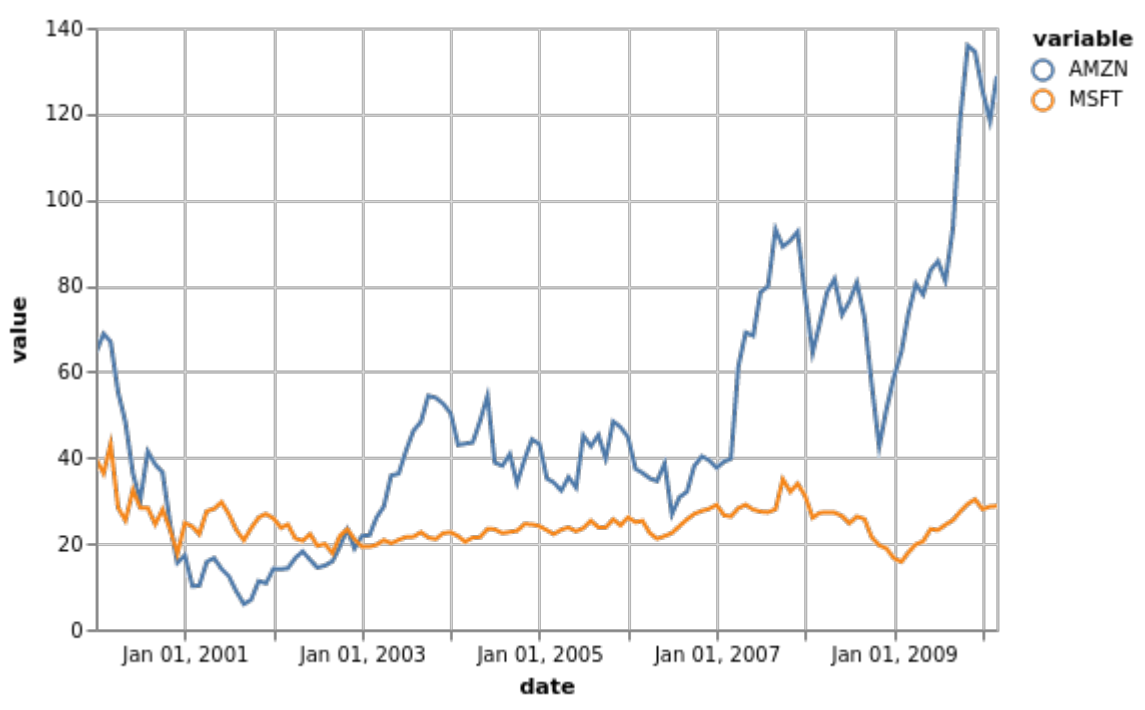
[18]:
```
pdvega.lag_plot(stocks[["AMZN", "MSFT"]], lag=12)
```



Aus diesem Plot geht sofort hervor, dass Amazon in diesem Zeitraum weitaus volatiler war: Die Werte zeigten zu jedem Zeitpunkt nur eine sehr geringe Korrelation mit dem Wert ein Jahr später. Umgekehrt war der Wert von Microsoft in diesem Jahrzehnt sehr viel stabiler.

Wir können diese Interpretation auch im einfachen Zeitreihendiagramm des Aktienkurses jedes Unternehmens sehen:

[19]:
```
stocks[["AMZN", "MSFT"]].vgplot.line()
```



Siehe auch:

- Statistical Visualization with pdvega.plotting

KAPITEL 5

Bokeh

Bokeh ist eine interaktive Visualisierungsbibliothek für moderne Webbrowser. Ihr Ziel ist es, vielseitige Grafiken bereitzustellen und diese Fähigkeit durch performante Interaktivität auf sehr große und Streaming-Datasets zu erweitern. Bokeh ist hilfreich um schnell und einfach interaktive Diagramme, Dashboards und Datenanwendungen zu erstellen.

Um sowohl einfache als auch leistungsstarke und flexible Funktionen zu bieten, die für erweiterbare Anpassungen erforderlich sind, stellt Bokeh zwei Interfaces zur Verfügung:

bokeh.models
: Low-Level-Interface, das Anwendungsentwicklern die größtmögliche Flexibilität bietet

bokeh.plotting
: High-Level-Interface für die Erstellung visueller Glyphen

Siehe auch:

- Home
- User Guide
- Gallery
- Reference Guide
- Source code
- Examples
  - github.com/bokeh/bokeh/tree/main/examples
  - cdn.pydata.org/bokeh/examples/examples-x.y.z.zip

5.1 Installation

Mit Spack könnt ihr Bokeh in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install   py-bokeh
```

Alternativ könnt ihr Bokeh auch mit anderen Paketmanagern installieren, z.B.

```
$ pipenv install bokeh
```

5.1.1 Optionale Erweiterungen

Es gibt Erweiterungen für Bokeh für die folgenden Funktionen:

NodeJS
: Notwendig zum Erweitern von Bokeh oder zum Definieren von `CustomJS`-Implementierungen in CoffeeScript oder TypeScript

pandas
: Notwendig für die Hexbin-Funktion. Einige Anwendungen werden durch die Verwendung von pandas vereinfacht, z.B. werden pandas DataFrames durch Glyph-Funktionen automatisch in Bokeh-Datenquellen konvertiert

Psutil
: Erforderlich, um eine detaillierte Speicherprotokollierung im Bokeh-Server zu ermöglichen

NetworkX
: Mit `from_networkx` lässt sich der Bokeh-Diagramm-Renderer direkt auf NetworkX-Daten anwenden

Selenium, **PhantomJS**
: Notwendig für das Exportieren von Plots in PNG- und SVG-Bilder

5.2 Beispiele

Bei der Installation mit `pip` werden die Beispiele nicht mitinstalliert. Ihr könnt jedoch das Git-Repository klonen und euch das Verzeichnis `examples/` anschauen um die Beispiele zu sehen.

Die meisten dieser Beispiele nutzen Beispieldaten, die ebenfalls separat zur Verfügung gestellt werden müssen. Um diese Dateien herunterzuladen, gebt einfach folgendes ein:

```
$ pipenv run bokeh sampledata
```

5.2.1 Styling und Theming

Mit Bokeh lassen sich verschiedene visuelle Aspekte der Diagramme konfigurieren.

Zuerst machen wir die Standardimporte:

[1]:
```
from bokeh.io import output_notebook, show
from bokeh.plotting import figure
```

[2]:
```
output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Farben

Bokeh kann Farben auf verschiedene Arten annehmen:

- eine der 147 namentlichen CSS-Farben, z.B. `green`, `indigo`
- ein RGB(A)-Hexadezimalwert, z.B. `#FF0000`, `#44444444`
- ein 3-Tupel ganzer Zahlen `(r,g,b)` zwischen 0 und 255
- ein 4-Tupel von `(r,g,b,a)` wobei `r`, `g` und `b` ganze Zahlen zwischen 0 und 255 sind und `a` ein Gleitkommawert zwischen 0 und 1 ist

Eigenschaften

Unabhängig davon, wie ein Bokeh-Plot erstellt wird, kann die Gestaltung immer durch Attribute für die Bokeh-Objekte festgelegt werden, aus denen der resultierende Plot besteht. Dabei gibt drei Arten von visuellen Eigenschaften: `line`-, `fill`- und `text`-Eigenschaften. Vollständige Informationen mit Code und Beispielen findet ihr im Abschnitt Styling Visual Properties des Benutzerhandbuchs.

Plots

Viele Top-Level-Attribute von Plots (`outline`, `border` usw.) können konfiguriert werden. Ausführliche Informationen findet ihr Sie im Abschnitt Plots.

Hier ist ein Beispiel, das den Plot-Rahmen beschreibt:

[3]:
```
p = figure(width=400, height=400)
p.border_fill_color = "whitesmoke"
p.min_border_left = 30

p.circle([1,2,3,4,5], [2,5,8,2,7], size=10)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Glyphen

Es ist auch möglich, die visuellen Eigenschaften von Glyphen zu gestalten. Wenn ihr `bokeh.plotting` verwendet, geschieht dies häufig beim Aufruf der Glyph-Methoden:

```
p.circle (line_color = "red", fill_alpha = 0,2, ...)
```

Es ist jedoch auch möglich, diese Eigenschaften direkt an Glyphenobjekten festzulegen. Glyph-Objekte werden in `GlyphRenderer`-Objekten gefunden, die von den Methoden `Plot.add_glyph` und `bokeh.plotting` zurückgegeben werden. Schauen wir uns ein Beispiel an:

[4]:
```
p = figure(width=400, height=400)

# keep a reference to the returned GlyphRenderer
r = p.circle([1,2,3,4,5], [2,5,8,2,7])

r.glyph.size = 50
r.glyph.fill_alpha = 0.2
r.glyph.line_color = "firebrick"
r.glyph.line_dash = [5, 1]
r.glyph.line_width = 2

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

`selection` und `nonselection`-Visuals

Ihr könnt auch angeben, wie die Glyphen aussehen sollen. Die Menge der *ausgewählten* Punkte wird gemäß der optionalen `.selection_glyph`-Eigenschaft eines `GlyphRenderer` gestaltet:

```
r.selection_glyph = Circle(fill_alpha=1, fill_color="firebrick", line_color=None)
```

Die nicht ausgewählten Punkte können mit der optionalen Eigenschaft `.nonselection_glyph` eines `GlyphRenderer` gestaltet werden:

```
r.nonselection_glyph = Circle(fill_alpha=0.2, fill_color="grey", line_color=None)
```

Wenn ihr die Schnittstelle `bokeh.plotting` verwendet, könnt ihr diese visuellen Eigenschaften leichter an die glyph-Methoden übergeben (s.u.). Die glyph-Methode erstellt die `selection`- oder `nonselection`-Glyphen und hängt sie an den Renderer.

[5]:
```
p = figure(width=400, height=400, tools="tap", title="Select a circle")
renderer = p.circle(
    [1, 2, 3, 4, 5],
    [2, 5, 8, 2, 7],
    size=50,
    # set visual properties for selected glyphs
    selection_color="firebrick",
    # set visual properties for non-selected glyphs
    nonselection_fill_alpha=0.2,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    nonselection_fill_color="grey",
    nonselection_line_color="firebrick",
    nonselection_line_alpha=1.0,
)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Es ist auch möglich, das visuelle Erscheinungsbild von Glyphen festzulegen, z.B. durch das Setzen des optionalen `hover_glyph`:

```
r.hover_glyph = Circle(fill_alpha=1, fill_color="firebrick", line_color=None)
```

Oder wenn `bokeh.plotting`-Glyph-Methoden verwendet werden, indem `hover_fill_alpha` usw. an die Glyph-Methode übergeben wird. Schauen wir uns ein Beispiel an, das mit einem `HoverTool` für `hline` konfiguriert ist:

[6]:
```
from bokeh.models.tools import HoverTool
from bokeh.sampledata.glucose import data


subset = data.loc["2010-10-06"]

x, y = subset.index.to_series(), subset["glucose"]

# Basic plot setup
p = figure(
    width=600, height=300, x_axis_type="datetime", title="Hover over points"
)

p.line(x, y, line_dash="4 4", line_width=1, color="gray")

cr = p.circle(
    x,
    y,
    size=20,
    fill_color="grey",
    hover_fill_color="firebrick",
    fill_alpha=0.05,
    hover_alpha=0.3,
    line_color=None,
    hover_line_color="white",
)

p.add_tools(HoverTool(tooltips=None, renderers=[cr], mode="hline"))

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Achsen

Als Nächstes betrachten wir das Styling von Achsen.

Um Achsen zu stylen, müssen zunächst Achsen-Objekte erstellt werden. Die einfachste Möglichkeit ist die Verwendung einiger komfortabler Methoden für `Plot`: axis, xaxis und yaxis. Diese Methoden geben eine Liste von Achsenobjekten zurück:

```
>>> p.xaxis
[<bokeh.models.axes.LinearAxis at 0x106fa2390>]
```

Ihr könnt jedoch Eigenschaften für alle Elemente der Liste festlegen, z.B.:

```
p.xaxis.axis_label = "Temperature"
p.axis.major_label_text_color = "orange"
```

Auch die Tab-Vervollständigung funktioniert: Gebt z.B. `p.xaxis.` in einer Notebook-Zelle ein und drückt anschließend die Tabulator-Taste um eine Liste der Attribute zu sehen, die hier gesetzt werden können:

```
13 p.xaxis.
14 p.xaxis.append
15 p.xaxis.apply_theme
16 p.xaxis.axis_label
17 p.xaxis.axis_label_standoff
18 p.xaxis.axis_label_text_align
19 p.xaxis.axis_label_text_alpha
20 p.xaxis.axis_label_text_baseline
21 p.xaxis.axis_label_text_color
22 p.xaxis.axis_label_text_font
23 p.xaxis.axis_label_text_font_size
```

Achseneigenschaften

Achsenobjekte verfügen über viele konfigurierbare Eigenschaften, mit denen die meisten visuellen Aspekte einer Achse gesteuert werden können. Diese können nach Funktion mit Präfix gruppiert werden:

Line Properties, z.B. * `axis_line_width` * `major_tick_line_dash`, `major_tick_in and major_tick_out` * `minor_tick_line_width`, `minor_tick_in and minor_tick_out`

Text Properties, z.B. * `axis_label`, `axis_label_text_color`, `axis_label_standoff` * `major_label`, `major_label_text_font_size`, `major_label_orientation`

Als einfachen ersten Fall ändern wir die Ausrichtung der wichtigsten Hilfsstrichbeschriftungen auf beiden Achsen einer Zeichnung:

[7]:
```
from math import pi


p = figure(width=400, height=400)
p.x([1, 2, 3, 4, 5], [2, 5, 8, 2, 7], size=10, line_width=2)

p.xaxis.major_label_orientation = pi / 4
p.yaxis.major_label_orientation = "vertical"

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Das nächste Beispiel zeigt Anpassungen an mehreren der verschiedenen Achseneigenschaften gleichzeitig:

[8]:
```
p = figure(width=400, height=400)
p.asterisk([1, 2, 3, 4, 5], [2, 5, 8, 2, 7], size=12, color="olive")

# change just some things about the x-axes
p.xaxis.axis_label = "Temp"
p.xaxis.axis_line_width = 3
p.xaxis.axis_line_color = "red"

# change just some things about the y-axes
p.yaxis.axis_label = "Pressure"
p.yaxis.major_label_text_color = "orange"
p.yaxis.major_label_orientation = "vertical"

# change things on all axes
p.axis.minor_tick_in = -3
p.axis.minor_tick_out = 6

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Markierungen konfigurieren

Alle Bokeh-Achsen verfügen über eine Formatierungseigenschaft, deren Wert ein `TickFormatter`-Objekt ist, mit dem Bokeh die von dieser Achse angezeigten Hilfsstriche formatiert. Bokeh konfiguriert standardmäßige Markierungen für numerische, Datums- oder kategoriale Achsen. Häufig möchten wir jedoch das Erscheinungsbild von Markierungsetiketten anpassen. Dies kann durch Ändern der Eigenschaften des von Bokeh ausgewählten Standardformatierers oder durch das vollständige Ersetzen des Formatierers durch einen neuen Typ erreicht werden.

Zunächst ändern wir nun die Eigenschaften eines Standardformatierers ändern. Das standardmäßige Datumsformat ist so konfiguriert, dass Monat/Tag angezeigt wird, wenn sich die Achse auf der Skala von Tagen befindet. Wenn Sie möchten, dass auch immer das Jahr angezeigt wird, können Sie die Days-Eigenschaft in ein Format ändern, das das Jahr enthält (siehe unten).

[9]:
```
from math import pi

from bokeh.sampledata.glucose import data


week = data.loc["2010-10-01":"2010-10-08"]

p = figure(
    x_axis_type="datetime", title="Glocose Range", height=350, width=800
)
p.xaxis[0].formatter.days = "%d.%m.%Y"
p.xaxis.major_label_orientation = pi / 3

p.line(week.index, week.glucose)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Weitere Informationen, die aktualisiert werden können, findet ihr im Referenzhandbuch für DatetimeTickFormatter.

Zusätzlich zu den Tick-Formatierern, die Bokeh standardmäßig verwendet, gibt es noch andere wie den `NumeralTickFormatter`, der explizit konfiguriert wird. Das folgende Beispiel zeigt, wie ihr einen Formatierer für jede Achse einstellen könnt:

[10]:
```
from bokeh.models import NumeralTickFormatter


p = figure(height=300, width=800)
p.circle([1, 2, 3, 4, 5], [2, 5, 8, 2, 7], size=10)

p.xaxis.formatter = NumeralTickFormatter(format="0.0%")
p.yaxis.formatter = NumeralTickFormatter(format="$0.00")

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Es gibt viele andere Möglichkeiten zur Kontrolle der Markierungsformatierung, einschließlich der Möglichkeit, ein JavaScript-Snippet für die beliebige Formatierung im Browser bereitzustellen. Weitere Informationen findet ihr in Tick Label Formats.

Es ist auch möglich, festzulegen, wo die Markierungen gezeichnet werden. Weitere Informationen findet ihr im Abschnitt Tick Locations des Benutzerhandbuchs.

Grids

Es ist auch möglich, die Gestaltung von Rastern zu steuern.

Rastereigenschaften in Bokeh haben zwei mögliche Präfixe:

- `grid`-Eigenschaften (also line properties) steuern die *Gitternetzlinien*
- `band`-Eigenschaften (also fill properties) steuern die Streifen zwischen den Gitterlinien

Im ersten ersten Beispiel deaktivieren wir die vertikalen Gitterlinien (indem wir die Linienfarbe auf `None`) und das horizontale Gitter hell und gestrichelt setzen.

[11]:
```
p = figure(width=400, height=400)
p.circle([1, 2, 3, 4, 5], [2, 5, 8, 2, 7], size=10)

# change just some things about the x-grid
p.xgrid.grid_line_color = None

# change just some things about the y-grid
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
p.ygrid.grid_line_alpha = 0.5
p.ygrid.grid_line_dash = [6, 4]

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Das nächste Beispiel zeigt, wie die Band-Eigenschaften eines Diagramms angegeben werden können:

[12]:
```
p = figure(width=400, height=400)
p.circle([1, 2, 3, 4, 5], [2, 5, 8, 2, 7], size=10)

# change just some things about the x-grid
p.xgrid.grid_line_color = None

# change just some things about the y-grid
p.ygrid.band_fill_alpha = 0.05
p.ygrid.band_fill_color = "olive"

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

5.2.2 Datenquellen und Transformationen

Überblick

Bokeh kann mit Python-Listen, NumPy-Arrays, pandas-Serien usw. arbeiten. Dabei werden diese Eingaben in eine Bokeh `ColumnDataSource` konvertiert. Obwohl Bokeh dies oft transparent macht, kann es gelegentlich sinnvoll sein, sie explizit zu erstellen.

[1]:
```
from bokeh.io import output_notebook, show
from bokeh.plotting import figure


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Python Dicts

Die `ColumnDataSource` kann aus `bokeh.models` importiert werden:

[2]:
```
from bokeh.models import ColumnDataSource
```

`ColumnDataSource` ist eine Zuordnung von Spaltennamen zu Wertsequenzen. Dabei müssen alle Spalten immer die gleiche Länge haben:

[3]:
```
source = ColumnDataSource(
    data={
        "x": [1, 2, 3, 4, 5],
        "y": [3, 7, 8, 5, 1],
    }
)
```

Bisher haben wir Funktionen wie `p.circle` aufgerufen, indem wir direkt Listen oder Datenarrays übergeben haben. Dann erstellt Bokeh automatisch eine `ColumnDataSource` für uns. Es ist jedoch auch möglich, eine `ColumnDataSource` explizit anzugeben, indem als Quellargument eine Glyph-Methode übergeben wird:

[4]:
```
p = figure(width=400, height=400)
p.circle("x", "y", size=20, source=source)
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

pandas.DataFrame

Es ist auch einfach, `ColumnDataSource`-Objekte direkt aus pandas-DataFrames zu erstellen:

[5]:
```
from bokeh.sampledata.iris import flowers as df


source = ColumnDataSource(df)
p = figure(width=400, height=400)
p.circle("petal_length", "petal_width", source=source)
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Transformationen

Wenn Datenquellen nicht gemeinsam genutzt werden müssen, können Dicts, `pandas.DataFrame`- oder `GroupBy`-Objekte direkt an die Glyph-Methode übergeben werden ohne explizit eine `ColumnDataSource` zu erstellen. In diesem Fall erfolgt die Konvertierung automatisch.

Glyph-Eigenschaften können nicht nur mit Namen von Spalten aus Datenquellen konfiguriert werden, sondern auch mit Transformationsobjekten aus `bokeh.transform`. Dabei ist wichtig zu beachten, dass bei der Verwendung dieser Objekte die Transformationen im Browser und nicht in Python erfolgen.

cumsum

Im Folgenden betrachten wir zunächst eine `cumsum`-Transformation, die aus einer Spalte eine neue Folge von Werten erzeugen kann indem die Werte kumulativ summiert werden. Dies kann für Kreisdiagramme oder Doughnut-Diagramme nützlich sein:

[6]:
```
from math import pi

import pandas as pd

from bokeh.palettes import Category20c
from bokeh.transform import cumsum


x = {
    "United States": 157,
    "United Kingdom": 93,
    "Japan": 89,
    "China": 63,
    "Germany": 44,
    "India": 42,
    "Italy": 40,
    "Australia": 35,
    "Brazil": 32,
    "France": 31,
    "Taiwan": 31,
    "Spain": 29,
}

data = (
    pd.Series(x).reset_index(name="value").rename(columns={"index": "country"})
)
data["color"] = Category20c[len(x)]

# represent each value as an angle = value / total * 2pi
data["angle"] = data["value"] / data["value"].sum() * 2 * pi

p = figure(
    height=350,
    title="Pie Chart",
    toolbar_location=None,
    tools="hover",
    tooltips="@country: @value",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
)

p.wedge(
    x=0,
    y=1,
    radius=0.4,
    # use cumsum to cumulatively sum the values for start and end angles
    start_angle=cumsum("angle", include_zero=True),
    end_angle=cumsum("angle"),
    line_color="white",
    fill_color="color",
    legend_label="country",
    source=data,
)

p.axis.axis_label = None
p.axis.visible = False
p.grid.grid_line_color = None

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

linear_cmap

Mit der `linear_cmap`-Transformation kann eine lineare Farbzuordnung zur Spalte einer Datenquelle eine neue Farbsequenz erzeugen:

[7]:
```
import numpy as np

from bokeh.transform import linear_cmap


N = 4000
data = dict(
    x=np.random.random(size=N) * 100,
    y=np.random.random(size=N) * 100,
    r=np.random.random(size=N) * 1.5,
)

p = figure()

p.circle(
    "x",
    "y",
    radius="r",
    source=data,
    fill_alpha=0.6,
    # color map based on the x-coordinate
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    color=linear_cmap("x", "Viridis256", 0, 100),
)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

5.2.3 Annotations

Überblick

Wir können visuelle Hinweise (Begrenzungslinien, schattierte Bereiche, Beschriftungen und Pfeile usw.) zu unseren Plots hinzufügen, um bestimmte Merkmale hervorzuheben. Bokeh bietet hierfür mehrere Annotationstypen. Um *Annotations* hinzuzufügen, erstellen wir normalerweise ein *Low-Level*-Objekt und fügen es mit `add_layout` unserem Diagramm hinzu. Schauen wir uns einige konkrete Beispiele an:

Spans

Spans sind vertikale oder horizontale Linien, für die die Spannweiten zwischen den Bemaßung angegeben werden kann, z.B.:

[1]:
```
from bokeh.io import output_notebook, show
from bokeh.plotting import figure


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

[2]:
```
import numpy as np

from bokeh.models.annotations import Span


x = np.linspace(0, 20, 200)
y = np.sin(x)

p = figure(y_range=(-2, 2))
p.line(x, y)

upper = Span(location=1, dimension="width", line_color="olive", line_width=4)
p.add_layout(upper)

lower = Span(
    location=-1, dimension="width", line_color="firebrick", line_width=4
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
)
p.add_layout(lower)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Box Annotations

Soll ein Bereich eines Diagramms hervorgehoben werden, so können diese mit `BoxAnnotation`, den Koordinateneigenschaften

- `top`
- `left`
- `bottom`
- `right`

sowie Linien- und Fülleigenschaften das Erscheinungsbild konfiguriert werden.

Infinite Boxen können erstellt werden, indem die Koordinaten nicht angegeben werden. Wenn zum Beispiel `top` nicht angegeben ist, wird die Box immer bis zum oberen Rand des Plotbereichs angezeigt, unabhängig davon, ob ein Verschieben oder Zoomen stattfindet, z.B.:

[3]:
```
import numpy as np

from bokeh.models.annotations import BoxAnnotation


x = np.linspace(0, 20, 200)
y = np.sin(x)

p = figure(y_range=(-2, 2))
p.line(x, y)

# region that always fills the top of the plot
upper = BoxAnnotation(bottom=1, fill_alpha=0.1, fill_color="olive")
p.add_layout(upper)

# region that always fills the bottom of the plot
lower = BoxAnnotation(top=-1, fill_alpha=0.1, fill_color="firebrick")
p.add_layout(lower)

# a finite region
center = BoxAnnotation(
    top=0.6, bottom=-0.3, left=7, right=12, fill_alpha=0.1, fill_color="navy"
)
p.add_layout(center)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Label

Mit der Annotation `Label` können wir einzelne Beschriftungen einfach an Plots anbringen. Die anzuzeigende Position und der Text sind als `x`, `y` und `text` konfiguriert:

```
Label(x=10, y=5, text="Some Label")
```

Standardmäßig befinden sich die Einheiten im *data space*, aber `x_units` und `y_units` können auf `screen` gesetzt werden, um die Beschriftung relativ zur Zeichenfläche zu positionieren, und Labels können mit `x_offset` und `y_offset` positioniert werden.

`Label`-Objekte haben auch Standardtext-, Zeilen- (`border_line`) und Fülleigenschaften (`background_fill`). Die Linien- und Fülleigenschaften gelten für einen Begrenzungsrahmen um den Text:

```
Label(x=10, y=5, text="Some Label", text_font_size="12pt",
      border_line_color="red", background_fill_color="blue")
```

[4]:
```
from bokeh.models.annotations import Label
from bokeh.plotting import figure

p = figure(x_range=(0,10), y_range=(0,10))
p.circle([2, 5, 8], [4, 7, 6], color="olive", size=10)

label = Label(x=5, y=7, x_offset=12, text="Second Point", text_baseline="middle")
p.add_layout(label)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

LabelSet

Mit der Annotation `LabelSet` könnt ihr viele Beschriftungen gleichzeitig erstellen, z.B. wenn ihr einen ganzen Satz von Scatter Markers beschriften möchtet. Sie ähneln `Label`, können aber auch eine `ColumnDataSource` als `source` -Eigenschaft nutzen, wobei sich `x` und `y` auf Spalten in der Datenquelle beziehen.

[5]:
```
from bokeh.models import ColumnDataSource, LabelSet
from bokeh.plotting import figure


source = ColumnDataSource(
    data=dict(
        temp=[166, 171, 172, 168, 174, 162],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        pressure=[165, 189, 220, 141, 260, 174],
        names=["A", "B", "C", "D", "E", "F"],
    )
)

p = figure(x_range=(160, 175))
p.scatter(x="temp", y="pressure", size=8, source=source)
p.xaxis.axis_label = "Temperature (C)"
p.yaxis.axis_label = "Pressure (lbs)"

labels = LabelSet(
    x="temp",
    y="pressure",
    text="names",
    level="glyph",
    x_offset=5,
    y_offset=5,
    source=source,
)


p.add_layout(labels)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Arrows

Mit der Annotation `Arrow` könnt ihr auf verschiedene Elemente in eurer Zeichnung *zeigen+. Dies kann besonders bei Beschriftungen hilfreich sein.

So könnt ihr beispielsweise einen Pfeil erstellen, der von `(0,0)` bis `(1,1)` zeigt:

```
p.add_layout(Arrow(x_start=0, y_start=0, x_end=1, y_end=0))
```

Dieser Pfeil hat die Standardspitze OpenHead. Andere Arten von Pfeilspitzen sind NormalHead und VeeHead. Der Typ des Pfeilkopfs kann durch die Eigenschaften `start` and `end` von `Arrow`-Objekten konfiguriert werden:

```
p.add_layout(Arrow(start=OpenHead(), end=VeeHead(),
             x_start=0, y_start=0, x_end=1, y_end=0))
```

Dies erzeugt einen Doppelpfeil mit `OpenHead` und `VeeHead`. Pfeilspitzen haben darüberhinaus den Standardsatz von Linien- und Fülleigenschaften, um deren Aussehen zu konfigurieren, z.B.:

```
OpenHead(line_color="firebrick", line_width=4)
```

Der Code und das Diagramm unten zeigen mehrere dieser Konfigurationen zusammen.

[6]:
```
from bokeh.models.annotations import Arrow
from bokeh.models.arrow_heads import NormalHead, OpenHead, VeeHead


p = figure(plot_width=600, plot_height=600)

p.circle(
    x=[0, 1, 0.5],
    y=[0, 0, 0.7],
    radius=0.1,
    color=["navy", "yellow", "red"],
    fill_alpha=0.1,
)

p.add_layout(
    Arrow(
        end=OpenHead(line_color="firebrick", line_width=4),
        x_start=0,
        y_start=0,
        x_end=1,
        y_end=0,
    )
)

p.add_layout(
    Arrow(
        end=NormalHead(fill_color="orange"),
        x_start=1,
        y_start=0,
        x_end=0.5,
        y_end=0.7,
    )
)

p.add_layout(
    Arrow(
        end=VeeHead(size=35),
        line_color="red",
        x_start=0.5,
        y_start=0.7,
        x_end=0,
        y_end=0,
    )
)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Legenden

Wenn Diagramme mehrere Glyphen enthalten, ist es wünschenswert, eine Legende hinzuzufügen, um Betrachtern die Interpretation zu erleichtern. Bokeh kann Legenden anhand der hinzugefügten Glyphen leicht generieren.

Einfache Legenden

Im einfachsten Fall könnt ihr einen String als `legend` an eine Glyph-Funktion übergeben:

```
p.circle(x, y, legend="sin(x)")
```

In diesem Fall erstellt Bokeh automatisch eine Legende, die eine Darstellung dieser Glyphe zeigt.

[7]:
```
import numpy as np


x = np.linspace(0, 4 * np.pi, 100)
y = np.sin(x)

p = figure(height=400)

p.circle(x, y, legend_label="sin(x)")
p.line(
    x,
    2 * y,
    legend_label="2*sin(x)",
    line_dash=[4, 4],
    line_color="orange",
    line_width=2,
)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Zusammengesetzte Legenden

Im obigen Beispiel haben wir für jede Glyphenmethode ein anderes Etikett bereitgestellt. Manchmal werden zwei (oder mehr) verschiedene Glyphen mit einer einzigen Datenquelle verwendet. In diesem Fall können zusammengesetzte Legenden erstellt werden. Wenn ihr z.B. eine Sin-Kurve mit einer Linie und einer Markierung plottet, könnt ihr ihnen dieselbe Bezeichnung geben um sie dazu zu bringen, gemeinsam in der Legende zu erscheinen:

```
p.circle(x, y, legend="sin(x)")
p.line(x, y, legend="sin(x)", line_dash=[4, 4], line_color="orange", line_width=2)
```

Farbbalken

Farbbalken sind besonders nützlich, wenn wir die Farbe einer Glyphe entsprechend der Farbzuordnung variiert. Bokeh-Farbbalken werden mit einer Farbzuordnung konfiguriert und mit der `add_layout‘-Methode zu Plots hinzugefügt:

```
color_mapper = LinearColorMapper(palette="Viridis256", low=data_low, high=data_high)
color_bar = ColorBar(color_mapper=color_mapper, location=(0,0))
p.add_layout(color_bar, 'right')
```

Das folgende Beispiel zeigt ein vollständiges Beispiel, in dem auch die Farbzuordnung zur Umwandlung der Glyphenfarbe verwendet wird.

[8]:
```
from bokeh.models import ColorBar, LinearColorMapper
from bokeh.sampledata.autompg import autompg
from bokeh.transform import transform


source = ColumnDataSource(autompg)
color_mapper = LinearColorMapper(
    palette="Viridis256", low=autompg.weight.min(), high=autompg.weight.max()
)

p = figure(
    x_axis_label="Horsepower",
    y_axis_label="MPG",
    tools="",
    toolbar_location=None,
)
p.circle(
    x="hp",
    y="mpg",
    color=transform("weight", color_mapper),
    size=20,
    alpha=0.6,
    source=autompg,
)

color_bar = ColorBar(
    color_mapper=color_mapper,
    label_standoff=12,
    location=(0, 0),
    title="Weight",
)
p.add_layout(color_bar, "right")

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

5.2.4 Interaktionen

[1]:
```
from bokeh.io import output_notebook, show
from bokeh.plotting import figure


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Verknüpfte Interaktionen

Es ist möglich, verschiedenen Bokeh-Plots interaktiv zu verknüpfen. Z.B. können zwei oder mehr Plots verknüpft werden, wobei in einem der Plots verschoben, gezoomt o.ä. wird und die anderen Plots gleichzeitig aktualisiert werden. Es ist auch möglich, Auswahlen zwischen zwei Plots zu verknüpfen, sodass bei der Auswahl von Elementen in einem Plot auch die entsprechenden Elemente im zweiten Plot ausgewählt werden.

Verknüpftes Verschieben

Verknüpftes Verschieben (wenn mehrere Darstellungen Bereiche haben, die synchron bleiben) ist mit Bokeh einfach zu realisieren. Ihr teilt einfach die entsprechenden Bereichsobjekte zwischen zwei (oder mehr) Plots auf. Das folgende Beispiel zeigt, wie ihr dies erreicht, indem ihr die Bereiche der drei Diagramme auf verschiedene Weise verknüpft:

[2]:
```
from bokeh.layouts import gridplot


x = list(range(11))
y0, y1, y2 = x, [10 - i for i in x], [abs(i - 5) for i in x]

plot_options = dict(width=250, height=250, tools="pan,wheel_zoom")

# create a new plot
s1 = figure(**plot_options)
s1.circle(x, y0, size=10, color="navy")

# create a new plot and share both ranges
s2 = figure(x_range=s1.x_range, y_range=s1.y_range, **plot_options)
s2.triangle(x, y1, size=10, color="firebrick")

# create a new plot and share only one range
s3 = figure(x_range=s1.x_range, **plot_options)
s3.square(x, y2, size=10, color="olive")

p = gridplot([[s1, s2, s3]])

# show the results
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Verknüpfte Auswahl

Das Verknüpfen von Auswahlen erfolgt auf ähnliche Weise, indem Datenquellen zwischen Plots gemeinsam genutzt werden. Beachtet, dass normalerweise mit `bokeh.plotting` und `bokeh.charts` die Erstellung einer Standarddatenquelle für einfache Darstellungen automatisch erfolgt. Um jedoch eine Datenquelle gemeinsam zu nutzen, müssen wir sie manuell erstellen und explizit übergeben. Dies wird im folgenden Beispiel veranschaulicht:

[3]:
```
from bokeh.models import ColumnDataSource


x = list(range(-20, 21))
y0, y1 = [abs(xx) for xx in x], [xx**2 for xx in x]

# create a column data source for the plots to share
source = ColumnDataSource(data=dict(x=x, y0=y0, y1=y1))

TOOLS = "box_select,lasso_select,help"

# create a new plot and add a renderer
left = figure(tools=TOOLS, width=300, height=300)
left.circle("x", "y0", source=source)

# create another new plot and add a renderer
right = figure(tools=TOOLS, width=300, height=300)
right.circle("x", "y1", source=source)

p = gridplot([[left, right]])

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Hover Tool

Bokeh verfügt über ein Hover-Tool, mit dem zusätzliche Informationen in einem Popup angezeigt werden können, wenn der Mauszeiger über einer bestimmten Glyphe schwebt. Die grundlegende Konfiguration des Hover-Tools bedeutet, dass eine Liste von `(name, format)`-Tupeln bereitgestellt wird. Die vollständigen Details findet ihr im User's Guide.

Das folgende Beispiel zeigt einige grundlegende Anwendungen des Hover-Werkzeugs mit einer Kreis-Glyphe, wobei die in `utils.py` definierten Hover-Informationen verwendet werden:

[4]:
```
from bokeh.models import HoverTool
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
source = ColumnDataSource(
    data=dict(
        x=[1, 2, 3, 4, 5],
        y=[2, 5, 8, 2, 7],
        desc=["A", "b", "C", "d", "E"],
    )
)

hover = HoverTool(
    tooltips=[
        ("index", "$index"),
        ("(x,y)", "($x, $y)"),
        ("desc", "@desc"),
    ]
)

p = figure(
    width=300, height=300, tools=[hover], title="Mouse over the dots"
)

p.circle("x", "y", size=20, source=source)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Widgets

Bokeh unterstützt die direkte Integration mit einem kleinen grundlegenden Widget-Set. Dies kann in Verbindung mit einem Bokeh-Server oder mit `CustomJS`-Modellen verwendet werden, um eure Dokumente interaktiver zu gestalten. Eine vollständige Liste mit Beispielcode findet ihr im Abschnitt Adding Widgets des Benutzerhandbuchs.

Um die Widgets zu verwenden, fügt sie wie ein Plotobjekt in ein Layout ein:

[5]:
```
from bokeh.models import Column
from bokeh.models.widgets import Slider


slider = Slider(start=0, end=10, value=1, step=0.1, title="foo")

show(Column(slider))
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

CustomJS Callbacks

Damit ein Widget nützlich wird, muss es Aktionen ausführen können. Mit Hilfe des Bokeh-Servers können Widgets echten Python-Code ausführen. Diese Möglichkeit wird unter *Bokeh-Server* erläutert. Hier betrachten wir, wie Widgets mit CustomJS-Callbacks konfiguriert werden können, die JavaScript-Code-Snippets ausführen.

[6]:
```
from bokeh.models import ColumnDataSource, CustomJS, TapTool


callback = CustomJS(code="alert('you tapped a circle!')")
tap = TapTool(callback=callback)

p = figure(width=600, height=300, tools=[tap])

p.circle(x=[1, 2, 3, 4, 5], y=[2, 5, 8, 2, 7], size=20)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Beispiel für ein Slider-Widget

Das folgende Beispiel zeigt eine an einen Slider angehängte Aktion, mit der eine Datenquelle aktualisiert wird.

[7]:
```
from bokeh.layouts import column
from bokeh.models import ColumnDataSource, CustomJS, Slider


x = [x * 0.005 for x in range(0, 201)]

source = ColumnDataSource(data=dict(x=x, y=x))

plot = figure(width=400, height=400)
plot.line("x", "y", source=source, line_width=3, line_alpha=0.6)

slider = Slider(start=0.1, end=6, value=1, step=0.1, title="power")

update_curve = CustomJS(
    args=dict(source=source, slider=slider),
    code="""
    var data = source.data;
    var f = slider.value;
    x = data['x']
    y = data['y']
    for (i = 0; i < x.length; i++) {
        y[i] = Math.pow(x[i], f)
    }

    // necessary becasue we mutated source.data in-place
    source.change.emit();
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    """,
)
slider.js_on_change("value", update_curve)


show(column(slider, plot))
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Beispiel für eine Datenauswahl

Es ist auch möglich, JavaScript-Aktionen auszuführen, wenn sich eine Benutzerauswahl (z.B. Box, Punkt, Lasso) ändert. Dies geschieht durch Anhängen derselben Art von CustomJS-Objekt an die Datenquelle, für die die Auswahl getroffen wird.

Das folgende Beispiel ist etwas komplexer und zeigt die Aktualisierung einer Datenquelle einer Glyphe als Reaktion auf die Auswahl einer anderen Glyphe:

[8]:
```
from random import random

from bokeh.layouts import row
from bokeh.models import ColumnDataSource, CustomJS
from bokeh.plotting import figure, output_file, show


x = [random() for x in range(500)]
y = [random() for y in range(500)]

s1 = ColumnDataSource(data=dict(x=x, y=y))
p1 = figure(width=400, height=400, tools="lasso_select", title="Select Here")
p1.circle("x", "y", source=s1, alpha=0.6)

s2 = ColumnDataSource(data=dict(x=[], y=[]))
p2 = figure(
    width=400,
    height=400,
    x_range=(0, 1),
    y_range=(0, 1),
    tools="",
    title="Watch Here",
)
p2.circle("x", "y", source=s2, alpha=0.6)

s1.selected.js_on_change(
    "indices",
    CustomJS(
        args=dict(s1=s1, s2=s2),
        code="""
        const inds = cb_obj.indices;
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        const d1 = s1.data;
        const d2 = s2.data;
        d2['x'] = []
        d2['y'] = []
        for (let i = 0; i < inds.length; i++) {
            d2['x'].push(d1['x'][inds[i]])
            d2['y'].push(d1['y'][inds[i]])
        }
        s2.change.emit();
    """,
    ),
)

layout = row(p1, p2)

show(layout)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

CustomJS für UI-Events

Alle verfügbaren UI-Ereignisse und ihre Eigenschaften sind im Abschnitt bokeh.events aufgeführt.

[9]:
```
import numpy as np

from bokeh import events
from bokeh.layouts import column, row
from bokeh.models import Button, CustomJS, Div
from bokeh.plotting import figure


x = np.random.random(size=2000) * 100
y = np.random.random(size=2000) * 100

p = figure(tools="box_select")
p.scatter(x, y, radius=1, fill_alpha=0.6, line_color=None)

div = Div(width=400)
button = Button(label="Button")
layout = column(button, row(p, div))

# Events with no attributes
button.js_on_event(
    events.ButtonClick,
    CustomJS(
        args=dict(div=div),
        code="""
div.text = "Button!";
""",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    ),
)

p.js_on_event(
    events.SelectionGeometry,
    CustomJS(
        args=dict(div=div),
        code="""
div.text = "Selection! <p> <p>" + JSON.stringify(cb_obj.geometry, undefined, 2);
""",
    ),
)

show(layout)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Zusätzliche Informationen

Es gibt viele Arten von Interaktionen und Ereignissen, die mit `CustomJS`-Rückrufen verbunden werden können:

- **Widgets**: - Button, Toggle, Dropdown, TextInput, AutocompleteInput, Select, Multiselect, Slider, (DateRangeSlider), DatePicker,
- **Tools**: - TapTool, BoxSelectTool, HoverTool,
- **Selection**: - ColumnDataSource, AjaxDataSource, BlazeDataSource, ServerDataSource
- **Ranges**: - Range1d, DataRange1d, FactorRange

Vollständigere Beispiele findet ihr in JavaScript callbacks.

5.2.5 Visualisierung von Netzwerkgraphen

Bokeh unterstützt nativ die Erstellung von Netzwerkgraphen mit konfigurierbaren Interaktionen zwischen Kanten und Knoten.

Edge- und Node-Renderer

Das Hauptmerkmal von `GraphRenderer` ist, dass es separate GlyphRenderer für Diagrammknoten und Diagrammkanten gibt. Dies ermöglicht das Anpassen der Knoten durch Ändern einer Eigenschaft des `node_renderer`. Es ist möglich, den üblichen Kreisknoten-Glyph durch eine beliebige XYGlyph-Instanz zu ersetzen, z.B. durch einen rechteckige oder elliptische Glyphe. Ebenso können die Stileigenschaften der Kanten über die `edge_renderer`-Eigenschaft geändert werden. Aktuell können Kanten jedoch nur als MultiLine-Zeichen gerendert werden.

Für die Datenquellen, die zu diesen beiden Sub-Renderern gehören, gelten einige Anforderungen:

- Die `ColumnDataSource`, die dem `node_renderer` zugeordnet ist, muss eine Spalte mit dem Namen `"index"` haben, die die eindeutigen Indizes der Knoten enthält.
- Die `ColumnDataSource`, die dem `edge_renderer` zugeordnet ist, enthält zwei erforderliche Spalten: `"start"` und `"end"`. Diese Spalten enthalten die Knotenindizes für den Anfang und das Ende der Kanten.

Es ist möglich, diesen Datenquellen zusätzliche Metadaten hinzuzufügen, um ein vektorisiertes Glyphen-Styling hinzuzufügen oder Daten für Callbacks oder Tooltips verfügbar zu machen.

Im Folgenden zeige ich ein Code-Snippet, das:

- einen Knoten elliptisch darstellt
- `height`- und `width`-Attribute der Ellipse festlegt
- das `fill_color`-Attribut der Ellipse festlegt

[1]:
```
import math

from bokeh.io import output_notebook, show
from bokeh.plotting import figure


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

[2]:
```
import math

from bokeh.models import Ellipse, GraphRenderer
from bokeh.palettes import Spectral8
from bokeh.plotting import figure


N = 8
node_indices = list(range(N))

plot = figure(
    title="Graph Layout Demonstration",
    x_range=(-1.1, 1.1),
    y_range=(-1.1, 1.1),
    tools="",
    toolbar_location=None,
)

graph = GraphRenderer()

graph.node_renderer.glyph = Ellipse(
    height=0.1, width=0.2, fill_color="fill_color"
)
graph.node_renderer.data_source.data = dict(
    index=node_indices, fill_color=Spectral8
)

graph.edge_renderer.data_source.data = dict(start=[0] * N, end=node_indices)
```

Durch Ausführen des obigen Code-Snippets wird kein Diagramm gerendert, da wir nicht angegeben haben, wie das Diagramm im 2D-Raum angeordnet werden soll. Wie das geht, erfahren Sie im folgenden Abschnitt.

Layout Providers

Bokeh verwendet ein separates `LayoutProvider`-Modell, um die Koordinaten eines Graphen im kartesischen Raum anzugeben. Derzeit ist das `StaticLayoutProvider`-Modell der einzige integrierte Provider, der ein Wörterbuch mit (x,y)-Koordinaten für die Knoten enthält.

In unserem Beispiel wird dem obigen Codeausschnitt ein Provider hinzugefügt mit:

[3]:
```
from bokeh.models import Ellipse, GraphRenderer, StaticLayoutProvider
from bokeh.palettes import Spectral8


N = 8
node_indices = list(range(N))

p = figure(
    title="Graph Layout Demonstration",
    x_range=(-1.1, 1.1),
    y_range=(-1.1, 1.1),
    tools="",
    toolbar_location=None,
)

graph = GraphRenderer()

graph.node_renderer.data_source.add(node_indices, "index")
graph.node_renderer.data_source.add(Spectral8, "color")
graph.node_renderer.glyph = Ellipse(height=0.1, width=0.2, fill_color="color")

graph.edge_renderer.data_source.data = dict(start=[0] * N, end=node_indices)

### start of layout code
circ = [i * 2 * math.pi / 8 for i in node_indices]
x = [math.cos(i) for i in circ]
y = [math.sin(i) for i in circ]

graph_layout = dict(zip(node_indices, zip(x, y)))
graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)

p.renderers.append(graph)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Explizite Pfade

Standardmäßig werden vom `StaticLayout`-Provider gerade Pfade zwischen den angegebenen Knoten gezeichnet. Um explizite Kantenpfade bereitzustellen, können jedoch auch Listen mit Pfaden für die `edge_renderer` angegeben werden mit `"xs"` und `"ys"`-Spalten der `ColumnDataSource`. Beachtet jedoch, dass einerseits diese Pfade in derselben Reihenfolge sein sollten wie die Punkte `"start"` und `"end"` und andererseits, dass es keine Validierung gibt, die mit den Knotenpositionen übereinstimmt. Ihr solltet daher sehr vorsichtig sein, wenn ihr explizite Pfade festlegt.

Im folgenden Beispiel erweitern wir das obige Beispiel um quadratische Bezierpfade zwischen den Knoten:

[4]:
```
import math

from bokeh.io import show
from bokeh.models import Ellipse, GraphRenderer, StaticLayoutProvider
from bokeh.palettes import Spectral8
from bokeh.plotting import figure


N = 8
node_indices = list(range(N))

p = figure(
    title="Graph Layout Demonstration",
    x_range=(-1.1, 1.1),
    y_range=(-1.1, 1.1),
    tools="",
    toolbar_location=None,
)

graph = GraphRenderer()

graph.node_renderer.data_source.add(node_indices, "index")
graph.node_renderer.data_source.add(Spectral8, "color")
graph.node_renderer.glyph = Ellipse(height=0.1, width=0.2, fill_color="color")

graph.edge_renderer.data_source.data = dict(start=[0] * N, end=node_indices)

### start of layout code
circ = [i * 2 * math.pi / 8 for i in node_indices]
x = [math.cos(i) for i in circ]
y = [math.sin(i) for i in circ]
graph_layout = dict(zip(node_indices, zip(x, y)))
graph.layout_provider = StaticLayoutProvider(graph_layout=graph_layout)


### Draw quadratic bezier paths
def bezier(start, end, control, steps):
    return [
        (1 - s) ** 2 * start + 2 * (1 - s) * s * control + s**2 * end
        for s in steps
    ]


xs, ys = [], []
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
sx, sy = graph_layout[0]
steps = [i / 100.0 for i in range(100)]
for node_index in node_indices:
    ex, ey = graph_layout[node_index]
    xs.append(bezier(sx, ex, 0, steps))
    ys.append(bezier(sy, ey, 0, steps))
graph.edge_renderer.data_source.data["xs"] = xs
graph.edge_renderer.data_source.data["ys"] = ys

p.renderers.append(graph)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

NetworkX-Integration

Bokeh unterstützt das schnelle Zeichnen eines Netzwerkgraphen mit seiner NetworkX-Integration. `bokeh.models.graphs.from_networkx` akzeptiert ein `networkx.Graph`-Objekt und eine `networkx`-Layout-Methode, um eine konfigurierte `GraphRenderer`-Instanz zurückzugeben.

Hier ist ein Beispiel für die Verwendung der `networkx.spring_layout`-Methode zum Layout des in NetworkX integrierten Datensatzes `karate_club_graph`:

[5]:
```
import networkx as nx

from bokeh.io import show
from bokeh.plotting import figure, from_networkx


G = nx.karate_club_graph()

p = figure(
    title="Networkx Integration Demonstration",
    x_range=(-1.1, 1.1),
    y_range=(-1.1, 1.1),
    tools="",
    toolbar_location=None,
)

graph = from_networkx(G, nx.spring_layout, scale=2, center=(0, 0))
p.renderers.append(graph)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Interaktionen

Ihr könnt das Auswahl- oder Überprüfungsverhalten von Diagrammen konfigurieren, indem ihr die `selection_policy`- und `inspection_policy`-Attribute des `GraphRenderer` festlegt. Diese Attribute akzeptieren eine spezielle `GraphHitTestPolicy`-Instanz.

So kann beispielsweise `selection_policy=NodesAndLinkedEdges()` bewirken, dass mit einem ausgewählten Knoten auch die zugehörigen Kanten ausgewählt werden. In ähnlicher Weise werden durch `inspection_policy=EdgesAndLinkedNodes()` der Start- und Endknoten einer Kante auch beim Bewegen der Maus über eine Kante mit dem HoverTool überprüft.

Mit den `selection_glyph`-, `nonselection_glyph`- und `hover_glyph`-Attributen der Kanten- und Knotenrenderer können die Diagramminteraktionen auch dynamische visuelle Elemente hinzuzufügen.

Hier ist ein Beispiel mit solchen Knoten- und Kanteninteraktionen:

[6]:
```
import networkx as nx

from bokeh.io import show
from bokeh.models import (
    BoxSelectTool,
    Circle,
    HoverTool,
    MultiLine,
    Plot,
    Range1d,
    TapTool,
)
from bokeh.models.graphs import EdgesAndLinkedNodes, NodesAndLinkedEdges
from bokeh.palettes import Spectral4
from bokeh.plotting import from_networkx


G = nx.karate_club_graph()

p = Plot(
    width=400,
    height=400,
    x_range=Range1d(-1.1, 1.1),
    y_range=Range1d(-1.1, 1.1),
)
p.title.text = "Graph Interaction Demonstration"

p.add_tools(HoverTool(tooltips=None), TapTool(), BoxSelectTool())

graph_renderer = from_networkx(G, nx.circular_layout, scale=1, center=(0, 0))

graph_renderer.node_renderer.glyph = Circle(size=15, fill_color=Spectral4[0])
graph_renderer.node_renderer.selection_glyph = Circle(
    size=15, fill_color=Spectral4[2]
)
graph_renderer.node_renderer.hover_glyph = Circle(
    size=15, fill_color=Spectral4[1]
)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
graph_renderer.edge_renderer.glyph = MultiLine(
    line_color="#CCCCCC", line_alpha=0.8, line_width=5
)
graph_renderer.edge_renderer.selection_glyph = MultiLine(
    line_color=Spectral4[2], line_width=5
)
graph_renderer.edge_renderer.hover_glyph = MultiLine(
    line_color=Spectral4[1], line_width=5
)

graph_renderer.selection_policy = NodesAndLinkedEdges()
graph_renderer.inspection_policy = EdgesAndLinkedNodes()

p.renderers.append(graph_renderer)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Knoten- und Kantenattribute

In `from_networkx` werden die Knoten- und Kantenattribute von NetworkX für den `node_renderer`oder `edge_renderer` umgewandelt.

Das Dataset `karate_club_graph` hat beispielsweise ein Knotenattribut mit dem Namen `club`. Mit `from_networkx` ist es möglich, diese Informationen zu verschieben. In ähnlicher Weise können auch weitere Knoten- und Kantenattribute für Farbinformationen verwendet werden:

[7]:
```
import networkx as nx

from bokeh.io import show
from bokeh.models import (
    BoxZoomTool,
    Circle,
    HoverTool,
    MultiLine,
    Plot,
    Range1d,
    ResetTool,
)
from bokeh.palettes import Spectral4
from bokeh.plotting import from_networkx


# Prepare Data
G = nx.karate_club_graph()

SAME_CLUB_COLOR, DIFFERENT_CLUB_COLOR = "black", "red"
edge_attrs = {}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
for start_node, end_node, _ in G.edges(data=True):
    edge_color = (
        SAME_CLUB_COLOR
        if G.nodes[start_node]["club"] == G.nodes[end_node]["club"]
        else DIFFERENT_CLUB_COLOR
    )
    edge_attrs[(start_node, end_node)] = edge_color

nx.set_edge_attributes(G, edge_attrs, "edge_color")

# Show with Bokeh
p = Plot(
    width=400,
    height=400,
    x_range=Range1d(-1.1, 1.1),
    y_range=Range1d(-1.1, 1.1),
)
p.title.text = "Graph Interaction Demonstration"

node_hover_tool = HoverTool(tooltips=[("index", "@index"), ("club", "@club")])
p.add_tools(node_hover_tool, BoxZoomTool(), ResetTool())

graph_renderer = from_networkx(G, nx.spring_layout, scale=1, center=(0, 0))

graph_renderer.node_renderer.glyph = Circle(size=15, fill_color=Spectral4[0])
graph_renderer.edge_renderer.glyph = MultiLine(
    line_color="edge_color", line_alpha=0.8, line_width=1
)
p.renderers.append(graph_renderer)

show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

5.2.6 Geographische Diagramme

[1]:
```
from bokeh.io import output_notebook, show


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

Es ist oft nützlich, Datensätze mit ihrem realen Kontext in Beziehung setzen zu können. Ihr könnt geografische Daten wie jede andere Art von Daten darstellen, z.B. für Texas Unemployment example, aber Bokeh bietet auch mehrere spezialisierte Mechanismen zum Zeichnen von Daten in geografischen Koordinaten:

- TileSource, insbesondere WMTSTileSource: Ermöglicht die Überlagerung von Daten von beliebigen Karten-Tiles-Servern, einschließlich Stamen, OpenStreetMap, ESRI, und benutzerdefinierte Server.
- GeoJSONDataSource: Ermöglicht das Lesen von Daten im GeoJSON-Format zur Verwendung mit Bokeh-Plots und -Glyphen, ähnlich `ColumnDataSource`.

WMTS Tile Source

WTMS ist der gebräuchlichste Webstandard für gekachelte Kartendaten, d.H. Karten, die als Bildabschnitte in Standardgröße bereitgestellt werden, aus denen dann die Gesamtkarte mit einer bestimmten Zoomstufe erstellt werden kann. WTMS verwendet das Web Mercator-Format, wobei Entfernungen von Greenwich, England in nördlicher und westlicher Richtung gemessen werden. Dies ist einfach zu berechnen, verzerrt jedoch die globale Form.

Zuerst erstellen wir ein leeres Bokeh-Diagramm, das die USA abdeckt, wobei die Grenzen in Metern angegeben sind:

[2]:
```
from bokeh.models import WMTSTileSource
from bokeh.plotting import figure
```

[3]:
```
# web mercator coordinates
USA = x_range, y_range = ((-13884029, -7453304), (2698291, 6455972))

p = figure(
    tools="pan, wheel_zoom",
    x_range=x_range,
    y_range=y_range,
    x_axis_type="mercator",
    y_axis_type="mercator",
)
```

[4]:
```
url = "http://a.basemaps.cartocdn.com/rastertiles/voyager/{Z}/{X}/{Y}.png"
attribution = "Tiles by Carto, under CC BY 3.0. Data by OSM, under ODbL"

p.add_tile(WMTSTileSource(url=url, attribution=attribution))
```

[4]:
```
TileRenderer(id='p1046', ...)
```

[5]:
```
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

Ihr könnt jetzt alles hinzufügen, was ihr normalerweise in einem Bokeh-Diagramm verwenden würdet, sofern ihr im Web Mercator-Format Koordinaten dafür erhaltet, z.B.:

[6]:
```
import numpy as np
import pandas as pd


def wgs84_to_web_mercator(df, lon="lon", lat="lat"):
    """Converts decimal longitude/latitude to Web Mercator format"""
    k = 6378137
    df["x"] = df[lon] * (k * np.pi / 180.0)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    df["y"] = np.log(np.tan((90 + df[lat]) * np.pi / 360.0)) * k
    return df


df = pd.DataFrame(
    dict(
        name=["Austin", "NYC"],
        lon=[-97.7431, -74.0059],
        lat=[30.2672, 40.7128],
    )
)
wgs84_to_web_mercator(df)
```

[6]:
```
     name      lon      lat             x             y
0  Austin -97.7431  30.2672 -1.088071e+07  3.537942e+06
1     NYC -74.0059  40.7128 -8.238299e+06  4.970072e+06
```

[7]:
```
p = figure(
    tools="pan, wheel_zoom",
    x_range=x_range,
    y_range=y_range,
    x_axis_type="mercator",
    y_axis_type="mercator",
)

p.add_tile(WMTSTileSource(url=url, attribution=attribution))

p.circle(x=df["x"], y=df["y"], fill_color="orange", size=10)
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

5.2.7 Bokeh-Server

Die Architektur von Bokeh ist so, dass übergeordnete *Modellobjekte*, also Darstellungen wie Plots, Bereiche, Achsen, Glyphen usw.) in Python erstellt und dann in ein JSON-Format konvertiert werden, das von der Client-Bibliothek `BokehJS` verwendet wird. Mit Hilfe des Bokeh-Servers können die *Modellobjekte* in Python und im Browser miteinander synchronisiert werden, wodurch mächtige Funktionen geschaffen werden:

- Browser-Events führen zu serverseitigen Python-Berechnungen oder -Abfragen
- Automatische Push-Aktualisierung des Browser-UI (z.B. Widgets oder Plots)
- Periodische, Timeout- und asynchronen Callbacks für Streaming-Updates

Diese Funktion zur Synchronisierung zwischen serverseitigem Python und dem Browser ist der Hauptzweck des Bokeh-Servers.

Es ist auch möglich, Bokeh-Anwendungen zu definieren, indem ein Standard-Python-Skript erstellt wird. In diesem Fall ist es nicht erforderlich, eine Funktion wie `modify_doc` zu erstellen. Normalerweise erstellt das Skript einfach alle Bokeh-Kontingente und fügt es mit einer Zeile dem Dokument hinzu:

```
curdoc().add_root(layout)
```

Um das Beispiel unten auszuprobieren, kopiert den Code in eine Datei `hello.py` und führt dann Folgendes aus:

```
$ pipenv run python -m bokeh serve --show hello.py
```

[1]:
```
from bokeh.io import curdoc
from bokeh.layouts import column
from bokeh.models.widgets import Button, Paragraph, TextInput


# create some widgets
button = Button(label="Say Hi")
input = TextInput(value="Pythonistas")
output = Paragraph()


# add a callback to a widget
def update():
    output.text = "Hello, " + input.value + "!"


button.on_click(update)

# create a layout for everything
layout = column(button, input, output)

# add the layout to curdoc
curdoc().add_root(layout)
```

5.2.8 Verzeichnisformat für Bokeh-Apps und Templates

Bokeh-Apps können auch mit einem Verzeichnisformat definiert werden. Dieses Format ermöglicht die Verwendung zusätzlicher Module, Daten, Templates, Themes und anderer Funktionen. Das Verzeichnis sollte eine `main.py`-Datei enthalten, das der *Einstiegspunkt* für die App ist, es können jedoch weitere Teile enthalten sein:

```
.
└── myapp
    ├── __init__.py
    ├── main.py
    ├── request_handler.py
    ├── static
    │   └── css
    │       └── custom.min.css
    ├── templates
    │   └── index.html
    └── theme.yaml
```

- `__init__.py`

  initiiert dieses Paket.

Dort können auch Importe relativ zum Paket vorgenommen werden, z.B. `from . import mymod` und `from .mymod import func`.

- `request_handler.py`

  für optionale Funktionen, z.B., um HTTP-Anfragen zu verarbeiten und ein Dict zurückzugeben, das ein Session-Token enthält, wie in Request handler hooks beschrieben.

- `app_hooks.py`

  für optionale Callbacks, die in verschiedenen Phasen der Ausführung ausgelöst werden können, wie in Lifecycle hooks beschrieben.

- `static`

  Unterverzeichnis für statische Ressourcen dieser Anwendung.

- `theme.yaml`

  zum Deklarieren von Standardattributen, die Bokeh auf Modelltypen anwenden soll.

- `templates`

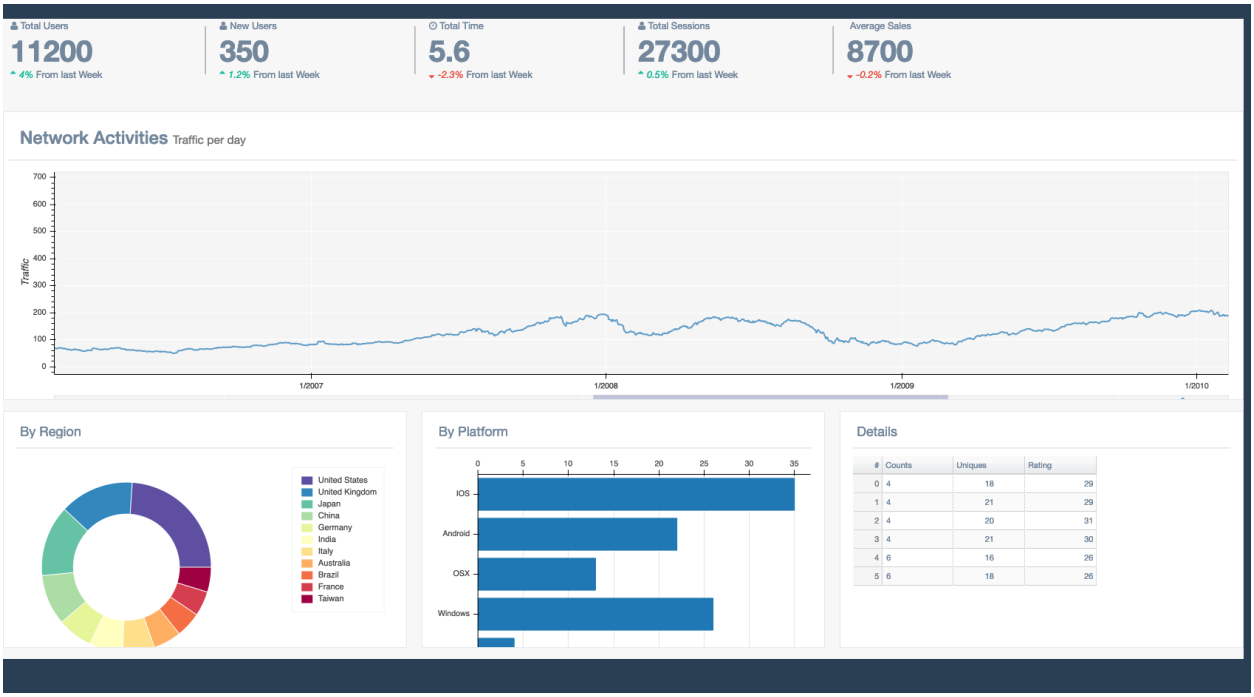
  Unterverzeichnis mit einer Jinja-Vorlage `index.html`. Das Verzeichnis kann weitere Jinja-Vorlagen enthalten, auf die `index.html` verweist.

  Weitere Informationen findet ihr unter Customizing the application's Jinja template.

Ein komplettes Beispiel findet ihr unter: https://github.com/bokeh/bokeh/tree/branch-3.3/examples/server/app/dash. Nachdem ihr den `dash`-Ordner heruntergeladen habt, könnt ihr den Bokeh-Server starten mit

```
$ pipenv run python -m bokeh serve --show dash/

$ pipenv run python -m bokeh serve --show examples/app/dash/
2019-02-08 19:02:42,829 Starting Bokeh server version 1.0.4 (running on Tornado 5.1.1)
2019-02-08 19:02:42,835 Bokeh app running at: http://localhost:5006/dash
```



5.2.9 Einbetten in ein Notebook

Ihr könnt ein Diagramm aktualisieren, ohne es neu zu laden. Übergebt dazu `notebook_handle=True` an `show()`, damit diese ein Handle-Objekt zurückgibt. Anschließend könnt ihr dieses Handle-Objekt mit der Funktion `push_notebook()` verwenden, um das Diagramm aktualisieren.

1. Importe

[1]:
```
from bokeh.io import output_notebook, show


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

- output_notebook konfiguriert den Status bei der Ausgabe von `show()`.
- show zeigt sofort ein Bokeh-Objekt oder eine Anwendung an.

2. Erstellt einige Plots und übergebt diese an `show()`:

[2]:
```
import pandas as pd

from bokeh.plotting import figure
from bokeh.sampledata.stocks import AAPL


df = pd.DataFrame(AAPL)
df["date"] = pd.to_datetime(df["date"])
```

[3]:
```
p = figure(width=616, height=250, x_axis_type="datetime")
p.line(df["date"], df["close"], color="navy", alpha=0.5)

show(p, notebook_handle=True)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

[3]:
```
<bokeh.io.notebook.CommsHandle at 0x12f6824d0>
```

5.2.10 HTML exportieren

Es kann auch eine eigenständige HTML-Datei mit Bokeh-Inhalten generiert werden. Dies wird erreicht durch den Aufruf der Funktion `output_file ("....html")`:

[1]:
```
from bokeh.io import output_notebook, show


output_notebook()
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

[2]:
```
import bokeh.sampledata


bokeh.sampledata.download()
```

```
Using data directory: /Users/veit/.bokeh/data
Skipping 'CGM.csv' (checksum match)
Skipping 'US_Counties.zip' (checksum match)
Skipping 'us_cities.json' (checksum match)
Skipping 'unemployment09.csv' (checksum match)
Skipping 'AAPL.csv' (checksum match)
Skipping 'FB.csv' (checksum match)
Skipping 'GOOG.csv' (checksum match)
Skipping 'IBM.csv' (checksum match)
Skipping 'MSFT.csv' (checksum match)
Skipping 'WPP2012_SA_DB03_POPULATION_QUINQUENNIAL.zip' (checksum match)
Skipping 'gapminder_fertility.csv' (checksum match)
Skipping 'gapminder_population.csv' (checksum match)
Skipping 'gapminder_life_expectancy.csv' (checksum match)
Skipping 'gapminder_regions.csv' (checksum match)
Skipping 'world_cities.zip' (checksum match)
Skipping 'airports.json' (checksum match)
Skipping 'movies.db.zip' (checksum match)
Skipping 'airports.csv' (checksum match)
Skipping 'routes.csv' (checksum match)
Skipping 'haarcascade_frontalface_default.xml' (checksum match)
Skipping 'SampleSuperstore.csv.zip' (checksum match)
```

[3]:
```
import pandas as pd

from bokeh.plotting import figure
from bokeh.sampledata.stocks import AAPL


df = pd.DataFrame(AAPL)
df["date"] = pd.to_datetime(df["date"])
```

[5]:
```
p = figure(width=800, height=250, x_axis_type="datetime")
p.line(df["date"], df["close"], color="navy", alpha=0.5)
```

[5]:
```
GlyphRenderer(id='p1092', ...)
```

Schließlich konfigurieren wir mit output_file die Ausgabe von `show()`:

[6]:
```
from bokeh.io import output_file, show


output_file("plot.html")
show(p)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_exec.v0+json

5.2.11 Export statischer Bilder

Bokeh unterstützt den Export von PNG und SVG.

PNG-Export

Bokeh unterstützt das Exportieren eines Plots oder Layouts in das PNG-Bildformat mit der Funktion `export_png`. Diese Funktion enthält ein zu exportierendes Bokeh-Objekt und einen Dateinamen, in den die PNG-Ausgabe geschrieben werden soll. Häufig handelt es sich bei dem an `export_png` übergebenen Bokeh-Objekt um eine einzelne Darstellung, dies muss jedoch nicht sein. Wenn ein Layout exportiert wird, wird das gesamte Layout in einem PNG-Bild gespeichert.

Die PNG-Exportfunktion erfordert die Installation einiger zusätzlicher Abhängigkeiten, z.B.:

```
$ spack env activate python-38
$ spack install py-selenium@3.141.0%gcc@11.2.0
$ py-pillow@8.0.0%gcc@11.2.0~freetype~imagequant+jpeg~jpeg2000~lcms~tiff~webp~webpmux~
↪xcb+zlib
```

Nach der Installation von Selenium müsst ihr das geckodriver-Binary herunterladen und installieren. Stellt sicher, dass `geckodriver` in `PATH` verfügbar ist. Weitere Informationen findet ihr in der geckodriver-Dokumentation. Schließlich muss auch noch Firefox auf eurem System verfügbar sein.

[1]:
```
import pandas as pd

from bokeh.plotting import figure
from bokeh.sampledata.stocks import AAPL


df = pd.DataFrame(AAPL)
df["date"] = pd.to_datetime(df["date"])
```

[2]:
```
from bokeh.io import export_png


p = figure(width=800, height=250, x_axis_type="datetime")
p.line(df["date"], df["close"], color="navy", alpha=0.5)

export_png(p, filename="plot.png")
```

[2]:
```
'/Users/veit/cusy/trn/pyviz-tutorial/docs/bokeh/embedding-export/plot.png'
```

SVG Export

Bokeh kann auch SVG-Ausgaben im Browser generieren, anstatt HTML-`canvas`-Elemente zu rendern. Dies wird durch Setzen von `output_backend='svg'` für eine `figure` erreicht. Anschließend kann das SVG entweder mit `output_file` in HTML-Dateien oder in mit `components` erstellten Inhalten eingebettet werden. Alternativ können mit `export_svgs` auch `.svg`-Dateien erstellt werden. Beachten Sie, dass für jedes HTML-`canvas`-Element eine SVG-Datei erstellt wird; es ist jedoch nicht möglich, vollständige Layouts oder Widgets in einer SVG-Datei zu erstellen.

[3]:
```
from bokeh.io import export_svgs


p = figure(width=800, height=250, x_axis_type="datetime", output_backend="svg")
p.line(df["date"], df["close"], color="navy", alpha=0.5)

export_svgs(p, filename="plot.svg")
```

[3]:
```
['plot.svg']
```

[4]:
```
from IPython.display import SVG


SVG("plot.svg")
```

[4]:

5.2.12 Bokeh-Plots in Flask einbinden

Exemplarisch betten wir Bokeh-Plots in das Flask-Framework ein.

1. Erstellen der virtuellen Umgebung:

   ```
   $ mkdir embed
   $ cd !$
   $ pipenv install flask bokeh pandas
   ```

2. Einbinden von Bokeh-Plots in Flask:

   1. Dabei wird zunächst in der Datei `flask_embed.py` eine Methode für ein Bokeh-Dokument erstellt:

      ```
      from bokeh.layouts import column
      from bokeh.models import ColumnDataSource, Slider
      from bokeh.plotting import figure
      from bokeh.sampledata.sea_surface_temperature import sea_surface_temperature
      from bokeh.server.server import Server
      from bokeh.themes import Theme


      def modify_doc(doc):
          df = sea_surface_temperature.copy()
          source = ColumnDataSource(data=df)
      ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    plot = figure(
        x_axis_type="datetime",
        y_range=(0, 25),
        y_axis_label="Temperature (Celsius)",
        title="Sea Surface Temperature at 43.18, -70.43",
    )
    plot.line("time", "temperature", source=source)

    def callback(attr, old, new):
        if new == 0:
            data = df
        else:
            data = df.rolling("{0}D".format(new)).mean()
        source.data = ColumnDataSource(data=data).data

    slider = Slider(
        start=0, end=30, value=0, step=1, title="Smoothing by N Days"
    )
    slider.on_change("value", callback)

    doc.add_root(column(slider, plot))

    doc.theme = Theme(filename="theme.yaml")
```

2. **Mit `bokeh.sampledata.sea_surface_temperature` werden Beispieldaten** verwendet, die aufgrund ihrer Größe nicht im Bokeh-Paket enthalten sind. Nach der Installation von Bokeh können diese jedoch mit folgendem Befehl heruntergeladen werden:

   ```
   $ pipenv run bokeh sampledata
   ```

3. Anschließend erstellen wir folgende `theme.yaml`-Datei für die Gestaltung von `Figure` und `Grid`:

   ```
   attrs:
       Figure:
           background_fill_color: "gainsboro"
           outline_line_color: white
           toolbar_location: above
           height: 500
           width: 800
       Grid:
           grid_line_dash: [6, 4]
           grid_line_color: white
   ```

4. Nun fügen wir in `flask_embed.py` eine Route von der Bokeh-App zum Flask-Server-Konfigurationsobjekt hinzu:

   ```
   from bokeh.embed import server_document
   from flask import render_template


   ...
   ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
@app.route("/", methods=["GET"])
def bkapp_page():
    script = server_document("http://localhost:5006/bkapp")
    return render_template("embed.html", script=script, framework="Flask")
```

5. `script` und `framework` werden anschließend in ein Jinja2-Template `templates/embed.html` eingebunden, das den Plot anzeigen soll:

```
<!doctype html>

<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Embedding a Bokeh Server in {{framework}}</title>
</head>

<body>
  <div>
    This Bokeh app below served by a Bokeh server that has been embedded
    in the web app framework {{framework}}. For more information see the section
    <a  target="_blank" href="https://bokeh.pydata.org/en/latest/docs/user_
↪guide/server.html#embedding-bokeh-server-as-a-library">Embedding Bokeh Server␣
↪as a Library</a>
    in the User's Guide.
  </div>
  {{script|safe}}
</body>
</html>
```

6. Nun wird ein Bokeh-Worker in `flask_embed.py` definiert:

```
from flask import Flask
from tornado.ioloop import IOLoop


...


def bk_worker():
    server = Server(
        {"/bkapp": modify_doc},
        io_loop=IOLoop(),
        allow_websocket_origin=["localhost:8000"],
    )
    server.start()
    server.io_loop.start()
```

7. Schließlich wird noch die Flask-App definiert:

```
app = Flask(__name__)
...
if __name__ == "__main__":
    print(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "Opening single process Flask app with embedded Bokeh application on␣
↪http://localhost:8000/"
    )
    print()
    print(
        "Multiple connections may block the Bokeh app in this configuration!"
    )
    print('See "flask_gunicorn_embed.py" for one way to run multi-process')
    app.run(port=8000)
```

3. Falls der Bokeh-Service noch nicht über WebSocket mit Flask kommunizieren kann, sollte dies explizit erlaubt werden mit:

```
$ export BOKEH_ALLOW_WS_ORIGIN=127.0.0.1:5000
```

4. Schließlich kann Flask gestartet werden mit:

```
$ export FLASK_APP=flask_embed.py
$ pipenv run flask run
```

   oder, falls mehrere Bokeh-Worker gestartet werden sollen:

```
$ export FLASK_APP=flask_gunicorn_embed.py
$ pipenv run flask run
```

Siehe auch:

- User Guide/Embedding Plots and Apps/App Sessions
- GnuCash-Expenses-Vis

5.2.13 Bokeh erweitern

Bokeh verfügt über eine Vielzahl von integrierten Typen, mit denen interaktive Visualisierungen und Datenanwendungen im Browser erstellt werden können. Es gibt jedoch spezifische Funktionen, die nicht in die Kernbibliothek aufgenommen wurden. Es gibt jedoch die Möglichkeit, Bokeh zu erweitern

- um das Verhalten vorhandener Bokeh-Modelle zu verändern
- um neue Modelle hinzuzufügen, die JavaScript-Bibliotheken von Drittanbietern mit Python verbinden
- um spezialisierte Modelle für bestimmte Fachdomänen zu erstellen

Solche benutzerdefinierten Erweiterungen können mit Standard-Releases erstellt und verwendet werden. Dabei ist es nicht erforderlich, eine Entwicklungsumgebung einzurichten oder etwas aus den Sourcen zu erstellen.

Struktur von Bokeh-Modellen

Python-Modelle

JavaScript-Modelle und Ansichten

Während die Python-Seite meistens deklarativ ist, ohne viel oder echten Code, erfordert die JavaScript-Seite Code um das Modell zu implementieren. Gegebenenfalls muss auch Code für eine entsprechende Ansicht bereitgestellt werden.

Nachfolgend findet ihr eine kommentierte TypeScript-Implementierung für `Custom` und `CustomView`. Bei integrierten Modellen ist dieser Code direkt in den endgültigen BokehJS-Skripts enthalten. Im nächsten Abschnitt wird gezeigt, wie ihr diesen Code mit benutzerdefinierten Erweiterungen verknüpfen könnt.

Hinweis:

BokehJS wurde ursprünglich in CoffeeScript geschrieben, wird jedoch nach TypeScript portiert. Dementsprechend ist die Anleitung hier in TypeScript. Benutzerdefinierte Erweiterungen können jedoch auch in CoffeeScript oder in reinem JavaScript geschrieben werden.

```
import {UIElement, UIElementView} from "models/ui/ui_element"

import {div} from "core/dom"
import * as p from "core/properties"

export class CustomView extends UIElementView {

  override connect_signals(): void {
    super.connect_signals()

    // Set BokehJS listener so that the program can process new data when
    // the slider has a change event.
    this.connect(this.model.slider.change, () => {
      this.render()
    })
  }

  override render(): void {
    // BokehJS views create <div> elements by default. These are accessible
    // as ``this.el``. Many Bokeh views ignore the default <div> and
    // instead do things like draw to the HTML canvas. In this case though,
    // the program changes the contents of the <div> based on the current
    // slider value.
    super.render()

    this.shadow_el.appendChild(div({
      style: {
        padding: '2px',
        color: '#b88d8e',
        backgroundColor: '#2a3153',
      },
    }, `${this.model.text}: ${this.model.slider.value}`))
  }
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
}

export class Custom extends UIElement {
  slider: {value: string}

  // Generally, the ``__name__`` class attribute should match the name of
  // the corresponding Python class exactly. TypeScript matches the name
  // automatically during compilation, so, barring some special cases, you
  // don't have to do this manually. This helps avoid typos, which stop
  // serialization/deserialization of the model.
  static __name__ = "Surface3d"

  static {
    // If there is an associated view, this is typically boilerplate.
    this.prototype.default_view = CustomView

    // The this.define() block adds corresponding "properties" to the JS
    // model. These should normally line up 1-1 with the Python model
    // class. Most property types have counterparts. For example,
    // bokeh.core.properties.String will correspond to ``String`` in the
    // JS implementation. Where JS lacks a given type, you can use
    // ``p.Any`` as a "wildcard" property type.
    this.define<Custom.Props>(({String, Ref}) => ({
      text:   [ String ],
      slider: [ Ref(Slider) ],
    }))
  }
}
```

Zusammenfügen

Bei integrierten Bokeh-Modellen wird die Implementierung in BokehJS vom Build-Prozess automatisch mit dem entsprechenden Python-Modell abgeglichen. Um JavaScript-Implementierungen mit Python-Modellen zu verbinden, ist ein zusätzlicher Schritt erforderlich. Die Python-Klasse sollte über ein Klassenattribut `__implementation__` verfügen, dessen Name der TypeScript-Code (oder JavaScript- oder CoffeeScript-Code) ist, der das clientseitige Modell sowie optionale Ansichten definiert.

Vorausgesetzt, der obige TypeScript-Code wurde in einer Datei `custom.ts` gespeichert, könnte die vollständige Python-Klasse folgendermaßen aussehen:

[1]:
```
from bokeh.core.properties import Instance, String
from bokeh.models import LayoutDOM, Slider


class Custom(LayoutDOM):
    __implementation__ = "custom.ts"

    text = String(default="Custom text")

    slider = Instance(Slider)
```

Wenn diese Klasse dann in einem Python-Modul `custom.py` definiert ist, kann die benutzerdefinierte Erweiterung

jetzt genau wie jedes integrierte Bokeh-Modell verwendet werden:

[2]:
```
from bokeh.io import output_file, show
from bokeh.layouts import column
from bokeh.models import Slider


slider = Slider(start=0, end=10, step=0.1, value=0, title="value")

custom = Custom(text="Special Slider Display", slider=slider)

layout = column(slider, custom)

show(layout)
```

Integration in Bokeh Server

Es sind keine besonderen Arbeiten oder Modifikationen erforderlich, um benutzerdefinierte Erweiterungen in den Bokeh-Server zu integrieren. Wie bei Standalone-Dokumenten ist die JavaScript-Implementierung automatisch in der gerenderten Anwendung enthalten. Zusätzlich erfolgt die Standardsynchronisation der Bokeh-Modelleigenschaften, die für alle integrierten Modelle gilt, auch transparent für benutzerdefinierte Erweiterungen.

Beispiele

In der Bokeh-Dokumentation stehen einige vollständige Beispiele zur Verfügung, die als Referenz dienen sollen. In vielen Fällen ist es erforderlich, den Quellcode der Basisklassen in bokehjs/src/lib/models zu studieren.

- Specialized Axis Ticking
- A New Custom Tool
- Wrapping A JavaScript Library
- Creating Latex Labels
- Adding A Custom Widget

5.2.14 Bokeh-Integration

Bokeh spielt auch mit anderen Visualisierungsbibliotheken wie Vaex, HoloViews oder *Datashader* zusammen.

hvPlot

Viele Bibliotheken wie *pandas*, xarray, Dask, Streamz und Intake haben eine High-Level-Plot-API, die meist auf *Matplotlib* beruht. Sie bieten jedoch nicht die Vorteile moderner Web-APIs, wie *Bokeh* und HoloViews. hvPlot ist eine auf HoloViews basierende High-Level-Plot-API, die eine allgemeine und konsistente API zum Plotten von allen oben genannten Formaten bietet.

Siehe auch:

- User Guide
- Gallery
- GitHub

hvPlot-Installation

hvPlot kann installiert werden mit

```
$ pipenv install hvplot
Installing hvplot...
Adding hvplot to Pipfile's [packages]...
✓ Installation Succeeded
...
```

hvPlot-Beispiele

Installation

Um die Beispiele ausführen zu können, muss zusätzlich Snappy installiert werden.

Mit Spack könnt ihr Snappy in eurem Kernel bereitstellen, z.B. mit:

```
$ spack env activate python-311
$ spack install snappy
```

Alternativ könnt ihr Snappy auch mit anderen Paketmanagern installieren, z.B.

- Für Debian/Ubuntu:

  ```
  $ sudo apt install libsnappy-dev
  ```

- Für Windows:

  Snappy benötigt Microsoft Visual C++ 14.0. Dies kann installiert werden mit den Microsoft C++ Build Tools.

- Für Mac OS:

  ```
  $ brew install snappy
  ```

Anschließend sollten für euren Kernel noch weitere Pakete installiert werden, z.B. mit:

```
$ pipenv install intake intake-parquet s3fs python-snappy pyviz-comms
...
```

Einführung

Als erstes importieren wir NumPy und pandas um anschließend einen kleinen Satz zufälliger Daten zu erstellen:

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


index = pd.date_range("1/1/2000", periods=1000)
df = pd.DataFrame(
    np.random.randn(1000, 4), index=index, columns=list("ABCD")
).cumsum()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
df.head()
```

[1]:

```
                   A         B         C         D
2000-01-01  1.431985  1.378913  0.539567  0.257977
2000-01-02  1.266573  0.834050  1.176750  1.458363
2000-01-03  1.799283  0.945437  1.792629 -0.220764
2000-01-04  2.131248  0.160377  1.186063 -0.702810
2000-01-05  3.574972  2.274926  1.759324 -1.297244
```

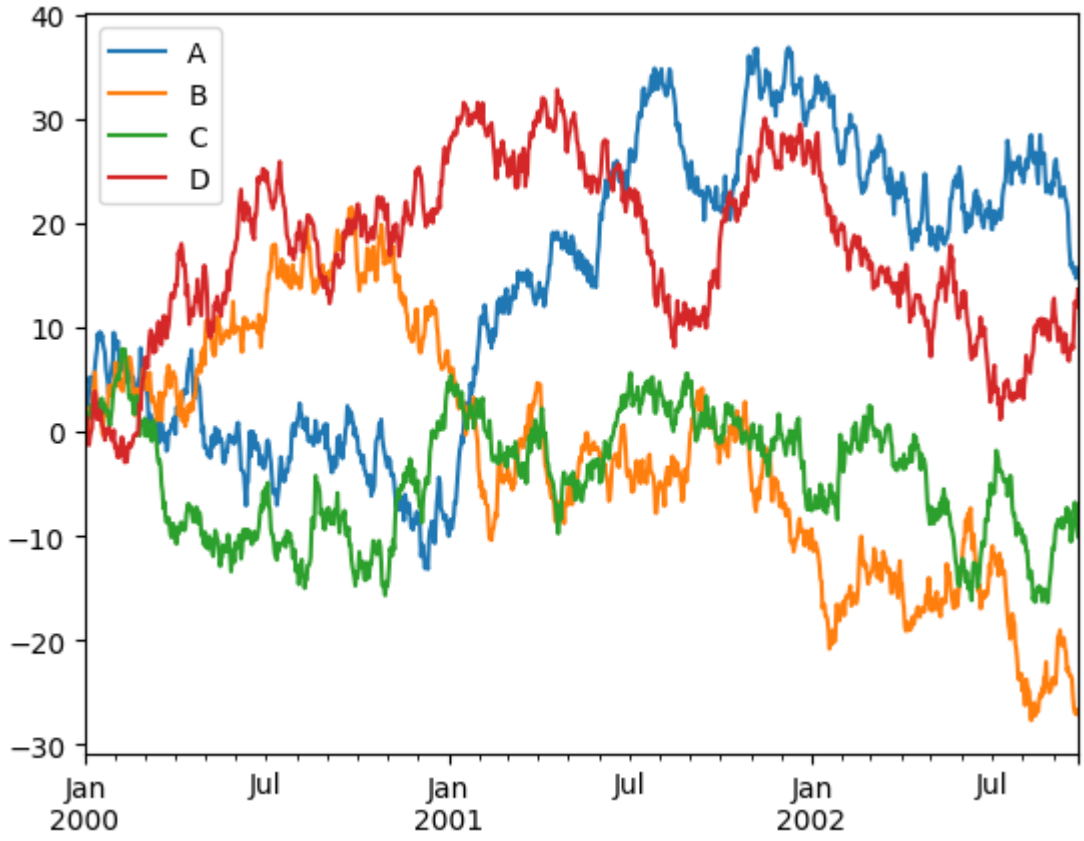
pandas.plot ()-API

pandas bietet standardmäßig Matplotlib-basiertes Plotten mit der `.plot()`-Methode:

[2]:

```
%matplotlib inline

df.plot();
```



Das Ergebnis ist ein PNG-Bild, das leicht angezeigt werden kann, ansonsten aber statisch ist.

> **Hinweis:** In pandas > 0.25.0 kann das Backend ausgetauscht werden, z.B. mit `pd.options.backend.plotting == 'holoviews'`,. Weitere Informationen hierzu findet ihr unter pandas-API.

.hvplot()

Wenn wir statt `%matplotlib inline` zu `import hvplot.pandas` und der `df.hvplot`-Methode wechseln, , wird jetzt ein interaktiv erforschbares Bokeh-Diagramm erzeugt mit Verschieben und Vergrößern/Verkleinern sowie anklickbaren Legenden:

[3]:
```
import hvplot.pandas

df.hvplot()
```

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

[3]:
```
:NdOverlay   [Variable]
   :Curve   [index]   (value)
```

Ein solches interaktive Diagramm erleichtert das Erkunden der der Daten erheblich, ohne dass hierfür zusätzlicher Code geschrieben werden müsste.

Native hvPlot-API

Für das obige Diagramm hat hvPlot die pandas-`.hvplot()`-Methode dynamisch hinzugefügt, sodass ihr dieselbe Syntax wie bei pandas-Plots verwenden könnt. Wenn ihr ein expliziteres Vorgehen bevorzugt, könnt ihr stattdessen direkt mit den hvPlot-Objekten arbeiten:

[4]:
```
import holoviews as hv

from hvplot import hvPlot


hv.extension("bokeh")

plot = hvPlot(df)
plot(y=["A", "B", "C", "D"])
```

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

[4]:
```
:NdOverlay   [Variable]
   :Curve   [index]   (value)
```

Hilfe

Wenn ihr in IPython oder Jupyter-Notebooks arbeitet, vervollständigen die `hvplot`-Methoden automatisch gültige Schlüsselwörter. Wenn ihr beispielsweise nach dem Deklarieren des Plottyps die Tabulatortaste drücken, werden alle gültigen Schlüsselwörter und die Dokumentzeichenfolge angezeigt:

```
df.hvplot.line(TAB
```

Außerhalb einer interaktiven Umgebung werden mit `hvplot.help` alle Informationen für einen Diagrammtyp angezeigt, z.B.:

[ ]:
```
hvplot.help("line")
```

Siehe auch:

Weitere Informationen zu den verfügbaren Optionen erhaltet ihr in Customization.

Plotting

In den folgenden Beispielen wird neben der pandas- auch die dask-hvPlot-API verwendet:

[6]:
```
import hvplot.dask
```

Das `hvplot.sample_data`-Modul erstellt diese Datensätze als Intake-Datenkataloge, die wir mit pandas laden können:

[7]:
```
from hvplot.sample_data import airline_flights, us_crime


crime = us_crime.read()
print(type(crime))
crime.head()
```

```
<class 'pandas.core.frame.DataFrame'>
```

[7]:
```
   Year  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]
```

Alternativ können wir `dask.DataFrame` verwenden:

[8]:
```
flights = airline_flights.to_dask().persist()
print(type(flights))
flights.head()
```

```
<class 'dask.dataframe.core.DataFrame'>
```

[8]:
```
     year  month   day  dayofweek  dep_time  crs_dep_time  arr_time  \
0  2008.0   11.0  15.0        6.0    1411.0        1420.0    1535.0
1  2008.0   11.0  28.0        5.0    1222.0        1230.0    1345.0
2  2008.0   11.0  22.0        6.0    1414.0        1420.0    1540.0
3  2008.0   11.0  15.0        6.0    1304.0        1305.0    1507.0
4  2008.0   11.0  22.0        6.0    1323.0        1305.0    1536.0

   crs_arr_time carrier  flight_num  ...  taxi_in  taxi_out  cancelled  \
0        1546.0   b'OO'      4391.0  ...      5.0      11.0        0.0
1        1356.0   b'OO'      4391.0  ...      5.0      15.0        0.0
2        1546.0   b'OO'      4391.0  ...      5.0      10.0        0.0
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
3        1519.0   b'OO'      4392.0  ...     10.0       9.0        0.0
4        1519.0   b'OO'      4392.0  ...      5.0      21.0        0.0

   cancellation_code  diverted  carrier_delay weather_delay nas_delay  \
0               None       0.0            NaN           NaN       NaN
1               None       0.0            NaN           NaN       NaN
2               None       0.0            NaN           NaN       NaN
3               None       0.0            NaN           NaN       NaN
4               None       0.0            0.0           0.0       0.0

   security_delay  late_aircraft_delay
0             NaN                  NaN
1             NaN                  NaN
2             NaN                  NaN
3             NaN                  NaN
4             0.0                 17.0

[5 rows x 29 columns]
```

Die Plot-API

Die Schnittstellen

- `dask.dataframe.DataFrame.hvplot`
- `pandas.DataFrame.hvplot`
- `intake.DataSource.plot`

und ihre `Series`-Äquivalente bieten eine leistungsfähige High-Level-API um auch komplexe Plots erzeugen zu können. Dabei kann die `.hvplot`-API entweder direkt oder als Namespace verwendet werden, um bestimmte Plottypen zu generieren.

Die expliziteste Methode zur Verwendung der Plot-API besteht darin, die Namen der Spalten anzugeben, die auf der x- bzw. y-Achse geplottet werden sollen:

[9]:
```
crime.hvplot.line(x="Year", y="Violent Crime rate")
```

[9]:
```
:Curve   [Year]   (Violent Crime rate)
```

Zusätzlich kann auch noch der Diagrammtyp mit `kind` angegeben werden:

[10]:
```
crime.hvplot(x="Year", y="Violent Crime rate", kind="scatter")
```

[10]:
```
:Scatter   [Year]   (Violent Crime rate)
```

Mit der `by`-Variable könnt ihr die Daten in einer oder mehreren zusätzlichen Spalten gruppieren. Als Beispiel wird im Folgenden die Abfahrtsverzögerung (`'depdelay'`) als Funktion der `'distance'` dargestellt und die Daten nach `'carrier'` gruppiert:

[11]:
```
flight_subset = flights[flights.carrier.isin([b"OH", b"F9"])]
flight_subset.hvplot(
    x="distance",
    y="depdelay",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    by="carrier",
    kind="scatter",
    alpha=0.2,
    persist=True,
)
```

[11]:
```
:NdOverlay   [carrier]
   :Scatter   [distance]   (depdelay)
```

Im obigen Beispiel haben wir die x- und y-Achsen explizit angegeben.

Andernfalls würde für die x-Achse die pandas-Indexspalte verwendet werden und für die y-Achse alle Nicht-Index-Spalten mit der Standardbezeichnung `'value'`. Wollt ihr nur die y-Achsenbezeichnung explizit angeben, so steht euch die `value_label`-Option zur Verfügung.

[12]:
```
crime.hvplot(
    x="Year",
    y=["Violent Crime rate", "Robbery rate", "Burglary rate"],
    value_label="Rate (per 100k people)",
)
```

[12]:
```
:NdOverlay   [Variable]
   :Curve   [Year]   (Rate (per 100k people))
```

Der `hvplot`-Namespace

Statt des `kind`-Argument können wir auch den `hvplot`-Namespace für den Plotaufruf zu verwenden. Die unterstützten Plottypen lassen sich leicht ermitteln mit der Tab-Vervollständigung, also

```
crime.hvplot.TAB
```

Verfügbare Diagrammtypen sind:

- *area()* zeichnet ein Flächendiagramm ähnlich einem Liniendiagramm, außer dass der Bereich unter der Kurve gefüllt und optional gestapelt wird
- *bar()* zeichnet ein Flächendiagramm ähnlich einem Liniendiagramm, außer dass der Bereich unter der Kurve gefüllt und optional gestapelt wird
- *bivariate()* zeichnet ein Flächendiagramm ähnlich einem Liniendiagramm, außer dass der Bereich unter der Kurve gefüllt und optional gestapelt wird
- *box()* zeichnet ein Box-Whisker-Diagramm, in dem die Verteilung einer oder mehrerer Variablen verglichen wird
- heatmap() zeichnet Hex-Bins
- hexbins() zeichnet die Verteilung eines oder mehrerer Histogramme als Satz von Containern
- histogram() zeichnet die Kernel-Dichteschätzung einer oder mehrerer Variablen
- *kde()* zeichnet die Kernel-Dichteschätzung einer oder mehrerer Variablen
- line() zeichnet ein Liniendiagramm (z.B. für eine Zeitreihe)
- *step()* zeichnet ein Schrittdiagramm, das einem Liniendiagramm ähnelt
- *scatter()* zeichnet ein Streudiagramm, in dem zwei Variablen verglichen werden
- *table()* erzeugt eine SlickGrid-Datentabelle

- violin() zeichnet ein Violinen-Diagramm, in dem die Verteilung einer oder mehrerer Variablen mithilfe der Kernel-Dichteschätzung verglichen wird

area()

Wie die meisten anderen Diagrammtypen unterstützt das `area`-Diagramm die drei oben beschriebenen Möglichkeiten zum Definieren eines Diagramms. Ein Flächendiagramm ist am nützlichsten, wenn mehrere Variablen in einem gestapelten Diagramm dargestellt werden. Dies kann durch Angabe für die `x`-, `y`- und `by`-Spalten erreicht werden oder mit `columns` und `index`/`use_index` als Optionen für die `x`-Achse.

[13]:
```
crime.hvplot.area(x="Year", y=["Robbery", "Aggravated assault"])
```

[13]:
```
:NdOverlay   [Variable]
   :Area   [Year]   (value,Baseline)
```

Wir können auch explizit `stacked` auf `False` setzen und einen `alpha`-Wert definieren und um die Werte direkt vergleichen zu können:

[14]:
```
crime.hvplot.area(
    x="Year", y=["Aggravated assault", "Robbery"], stacked=False, alpha=0.4
)
```

[14]:
```
:NdOverlay   [Variable]
   :Area   [Year]   (value)
```

Eine andere Verwendung für ein Flächendiagramm besteht darin, die Streuung eines Werts zu visualisieren. Wenn wir beispielsweise den Flugdatensatz verwenden, möchten wir möglicherweise die Streuung der mittleren Verspätungswerte zwischen den Fluggesellschaften sehen. Zu diesem Zweck berechnen wir die mittlere Verzögerung nach Tag und Carrier und dann die minimale/maximale mittlere Verzögerung für alle Carrier. Da die Ausgabe von `hvplot` nur ein reguläres Holoviews-Objekt ist, können wir den Overlay-Operator (*) verwenden, um die Diagramme übereinander zu platzieren.

[15]:
```
delay_min_max = (
    flights.groupby(["day", "carrier"])["carrier_delay"]
    .mean()
    .groupby("day")
    .agg([np.min, np.max])
)
delay_mean = flights.groupby("day")["carrier_delay"].mean()

delay_min_max.hvplot.area(
    x="day", y="amin", y2="amax", alpha=0.2
) * delay_mean.hvplot()
```

[15]:
```
:Overlay
   .Area.I              :Area   [day]   (amin,amax)
   .Curve.Carrier_delay :Curve   [day]   (carrier_delay)
```

bar()

Im einfachsten Fall können wir `.hvplot.bar` verwenden. Um die Beschriftung auf der x-Achse um 90° zu drehen, geben wir noch `rot=90` an.

[16]:
```
crime.hvplot.bar(x="Year", y="Violent Crime rate", rot=90)
```

[16]:
```
:Bars   [Year]   (Violent Crime rate)
```

Wenn wir stattdessen mehrere Spalten vergleichen möchten, können wir eine Liste von Spalten festlegen. Mit der `stacked`-Option können wir dann die Spaltenwerte einfacher vergleichen:

[17]:
```
crime.hvplot.bar(
    x="Year",
    y=["Violent crime total", "Property crime total"],
    stacked=True,
    rot=90,
    width=800,
    legend="top_left",
)
```

[17]:
```
:Bars   [Year,Variable]   (value)
```

scatter()

Das Streudiagramm unterstützt viele der Funktionen der obigen Diagrammtypen, kann jedoch mit der `c`-Option auch eingefärbt werden.

[18]:
```
crime.hvplot.scatter(x="Violent Crime rate", y="Burglary rate", c="Year")
```

[18]:
```
:Scatter   [Violent Crime rate]   (Burglary rate,Year)
```

Um Farbe zur Darstellung einer Dimension zu verwenden, kann die `cmap`-Option genutzt werden werden, um die zu verwendende Farbkarte anzugeben. Zusätzlich kann die Farbleiste deaktiviert werden mit `colorbar=False`.

step()

Ein Schrittdiagramm ist einem Liniendiagramm sehr ähnlich, aber anstatt linear zwischen Abtastwerten zu interpolieren, visualisiert das Schrittdiagramm diskrete Schritte. Die Position der Schritte kann mit dem `where`-Schlüsselwort un den Werten `'pre'`, `'mid'` (Standard) und `'post'` gesteuert werden.

[19]:
```
crime.hvplot.step(x="Year", y=["Robbery", "Aggravated assault"])
```

[19]:
```
:NdOverlay   [Variable]
   :Curve   [Year]   (value)
```

hexbin()

Mit der `hexbin`-Methode können Sie hexagonale Bin-Diagramme erstellen. Sie können eine nützliche Alternative zu Streudiagrammen sein, wenn die Daten zu dicht sind, um jeden Punkt einzeln zu zeichnen. Da unsere Flugdaten nicht gleichmäßig auf einer linearen Skala verteilt sind, verwenden wir die `logz`-Option für eine logarithmische Skala.

[20]:
```
flights.hvplot.hexbin(
    x="airtime", y="arrdelay", width=600, height=500, logz=True
)
```

[20]:
```
:HexTiles   [airtime,arrdelay]
```

bivariate()

Mit der `bivariate`-Methode könnt ihr ein 2D-Dichtediagramm erstellen. Bivariate Diagramme sind neben Hexbin-Diagrammen eine weitere Alternative zu Streudiagrammen, wenn die Daten zu dicht sind, um jeden Punkt einzeln zu zeichnen.

[21]:
```
crime.hvplot.bivariate(
    x="Violent Crime rate", y="Burglary rate", width=600, height=500
)
```

[21]:
```
:Bivariate   [Violent Crime rate,Burglary rate]   (Density)
```

heatmap()

`HeatMap` kann die Beziehung zwischen drei Variablen anzeigen und neben den Variablen `'x'` und `'y'` zusätzlich `'C'` anzeigen. Zusätzlich werden mit der `reduce_function` die Werte für jeden Container aus den Stichproben berechnet.

[22]:
```
flights.compute().hvplot.heatmap(
    x="day", y="carrier", C="depdelay", reduce_function=np.mean, colorbar=True
)
```

[22]:
```
:HeatMap   [day,carrier]   (depdelay)
```

table()

Im Gegensatz zu allen anderen Plottypen kann für eine Tabelle nur angegeben werden, ob alle Spalten oder mit `columns` nur eine Teilmenge angezeigt werden soll.

[23]:
```
crime.hvplot.table(
    columns=["Year", "Population", "Violent Crime rate"], width=400
)
```

[23]:
```
:Table   [Year,Population,Violent Crime rate]
```

hist()

Das Zeichnen von Verteilungen unterscheidet sich geringfügig von anderen Plots, da sie im einfachen Fall nur eine Variable darstellen. Daher muss bei diesem Plottyp kein `index` oder `x`-Wert angegeben werden, sondern

- deklariert eine einzelne y-Variable, z.B. `source.plot.hist(variable)` oder
- deklariert eine y-Variable und eine by-Variable, z.B. `source.plot.hist(variable, by='Group')` oder
- deklariert Spalten oder zeichnet alle Spalten, z.B. `source.plot.hist()` oder `source.plot.hist(columns=['A', 'B', 'C'])`

[24]:
```
crime.hvplot.hist(y="Violent Crime rate")
```

[24]:
```
:Histogram   [Violent Crime rate]   (Violent Crime rate_count)
```

Alternativ önnen wir auch die Verteilung mehrerer Spalten darstellen:

[25]:
```
columns = ["Violent Crime rate", "Property crime rate", "Burglary rate"]
crime.hvplot.hist(y=columns, bins=50, alpha=0.5, legend="top", height=400)
```

[25]:
```
:NdOverlay   [Element]
   :Histogram   [Burglary rate]   (Burglary rate_count)
```

Wir können die Daten auch nach anderen Variablen gruppieren und die Carrier in eigene `subplots` aufteilen:

[26]:
```
flight_subset = flights[flights.carrier.isin([b"AA", b"US", b"OH"])]
flight_subset.hvplot.hist(
    "depdelay",
    by="carrier",
    bins=20,
    bin_range=(-20, 100),
    width=300,
    subplots=True,
)
```

[26]:
```
:NdLayout   [carrier]
   :Histogram   [depdelay]   (depdelay_count)
```

kde(), density()

Ihr könnt Dichtediagramme auch mit `hvplot.kde()` oder `hvplot.density()` erstellen:

[27]:
```
crime.hvplot.kde(y="Violent Crime rate")
```

[27]:
```
:Distribution   [Violent Crime rate]   (Density)
```

Der Vergleich der Verteilung mehrerer Spalten ist ebenfalls möglich:

[28]:
```
columns = ["Violent Crime rate", "Property crime rate", "Burglary rate"]
crime.hvplot.kde(y=columns, alpha=0.5, value_label="Rate", legend="top_right")
```

[28]:
```
:NdOverlay   [Variable]
   :Distribution   [Rate]   (Density)
```

`hvplot.kde` unterstützt auch das `by`-Schlüsselwort:

[29]:
```
flight_subset = flights[flights.carrier.isin([b"AA", b"US", b"OH"])]
flight_subset.hvplot.kde(
    "depdelay", by="carrier", xlim=(-20, 70), width=300, subplots=True
)
```

[29]:
```
:NdLayout   [carrier]
   :Distribution   [depdelay]   (Density)
```

box()

Genau wie die anderen verteilungsbasierten Diagrammtypen unterstützt das Box-Whisker-Diagramm das Zeichnen einer einzelnen Spalte:

[30]:
```
crime.hvplot.box(y="Violent Crime rate")
```

[30]:
```
:BoxWhisker   (Violent Crime rate)
```

Es unterstützt auch mehrere Spalten und die gleichen Optionen wie berits oben genannt:`legend`, `invert` und `value_label`:

[31]:
```
columns = [
    "Burglary rate",
    "Larceny-theft rate",
    "Motor vehicle theft rate",
    "Property crime rate",
    "Violent Crime rate",
]
crime.hvplot.box(
    y=columns,
    group_label="Crime",
    legend=False,
    value_label="Rate (per 100k)",
    invert=True,
)
```

[31]:
```
:BoxWhisker   [Crime]   (Rate (per 100k))
```

Auch die Verwendung des by-Schlüsselworts zum Aufteilen der Daten in mehrere Teilmengen wird unterstützt:

[32]:
```
flight_subset = flights[flights.carrier.isin([b"AA", b"US", b"OH"])]
flight_subset.hvplot.box("depdelay", by="carrier", ylim=(-10, 70))
```

[32]:
```
:BoxWhisker   [carrier]   (depdelay)
```

Zusammengesetzte Diagramme

Eine der Hauptstärken von HoloViews ist die einfache Erstellung verschiedener Diagramme. Einzelne Diagramme können mit den Operatoren * und + überlagert bzw. zusammengesetzt werden.

Siehe auch:

- Composing Elements

---

[33]:
```
crime.hvplot(x="Year", y="Violent Crime rate") * crime.hvplot.scatter(
    x="Year", y="Violent Crime rate", c="k"
)
```

[33]:
```
:Overlay
   .Curve.I   :Curve   [Year]   (Violent Crime rate)
   .Scatter.I :Scatter   [Year]   (Violent Crime rate)
```

Wir können auch verschiedene Diagramme und Tabellen zusammen erstellen:

[34]:
```
(
    crime.hvplot.bar(x="Year", y="Violent Crime rate", rot=90, width=550)
    + crime.hvplot.table(
        ["Year", "Population", "Violent Crime rate"], width=420
    )
)
```

[34]:
```
:Layout
   .Bars.I  :Bars   [Year]   (Violent Crime rate)
   .Table.I :Table   [Year,Population,Violent Crime rate]
```

Big Data

In den vorherigen Beispielen fassten wir den relativ große Airline-Datensatz zusammen indem wir Teilmengen für die Darstellung bildeten. Stattdessen können wir die Daten jedoch auch mithilfe von *Datashader* aggregieren, wobei der gesamte verfügbare Rohdatensatz gerendert wird (sofern die Auflösung des Bildschirms dies zulässt).

[35]:
```
flights.hvplot.scatter(x="distance", y="airtime", datashade=True)
```

[35]:
```
:DynamicMap   []
   :RGB   [distance,airtime]   (R,G,B,A)
```

groupby

Dank der Fähigkeit von HoloViews, einen Parameterraum mit einer Reihe von Widgets zu erkunden, können wir eine Gruppe entlang einer bestimmten Spalte oder Dimension anwenden, z.B. die Verteilung der Abflugverzögerungen nach Carriern und Tag gruppiert anzeigen, wobei Benutzer*innen auswählen können, welcher Tag angezeigt werden soll:

[36]:
```
flights.hvplot.violin(
    y="depdelay", by="carrier", groupby="dayofweek", ylim=(-20, 60), height=500
)
```

[36]:
```
:DynamicMap   [dayofweek]
   :Violin   [carrier]   (depdelay)
```

datashader

Die Starke von Bokeh ist, Daten aus Python (oder R) im Webbrowser zu rendern. Aufgrund der Art und Weise, wie Webbrowser entworfen wurden, gibt es Einschrankungen, wie viele Daten auf diese Weise angezeigt werden konnen. Die meisten Webbrowser konnen bis zu 100.000 oder 200.000 Datenpunkte in einem Bokeh-Diagramm verarbeiten, bevor sie langsamer werden oder Speicherprobleme haben.

Die datashader-Bibliothek soll Bokeh insofern erweitern, dass auch die Visualisierung fur sehr große Datensatze moglich werden soll indem die getreue Darstellung der Gesamtverteilung gewahrleistet werden soll, nicht jeodch einzelne Datenpunkte. datashader wird installiert mit

```
$ pipenv install datashader
```

Wann soll Datashader nicht verwendet werden?

- zum Zeichnen von weniger als 100.000 Datenpunkten
- wenn jeder Datenpunkt wichtig ist – die Standard-Bokeh gibt alle Datenpunkte wieder, nicht jedoch zusammen mit `datashader`
- fur volle Interaktivitat (`hover`-Tools) mit jedem Datenpunkt

Wann soll Datashader verwendet werden?

- tatsachlich große Daten; wenn Bokeh/Matplotlib Probleme machen
- wenn die Verteilung bedeutender ist als einzelne Datenpunkte
- wenn im Wesentlichen die Verteilung analysiert werden soll

Wie funktioniert Datashader?

- Tools wie Bokeh ordnen Daten direkt in ein HTML/JavaScript-Diagramm ein
- Datashader stellt Daten in ein Aggregate-Array in Bildschirmgroße dar, aus dem ein Bild erstellt und in ein Bokeh-Diagramm eingebettet werden kann
- nur das Bild mit fester Große muss an den Browser gesendet werden, sodass Millionen oder Milliarden von Datenpunkten verwendet werden konnen
- jeder Schritt passt sich automatisch an die Daten an, kann aber angepasst werden

Vom Datashader unterstutzte Visualisierungen

Datashader unterstutzt derzeit

- Scatterplots/Heatmaps
- Zeitreihen
- Verbundene Punkte (Trajektorien)
- Raster

In jedem Fall kann die Ausgabe problemlos in Bokeh-Diagramme eingebettet werden, wobei interaktives Resampling auf Schwenk- und Zoombereich, in Notebooks oder Apps erfolgt. Legenden und Hover-Informationen konnen aus den Aggregat-Arrays generiert werden um Interaktivitat zu ermoglichen.

Big Data originalgetreu visualisieren

Wenn die Daten so groß sind, dass einzelne Punkte nicht leicht zu erkennen sind, ist es entscheidend, dass die Visualisierung prinzipiell erstellt wird und die zugrunde liegende Verteilung fur Ihr visuelles System getreu aufzeigt. Zum Beispiel zeigen alle diese Diagramme die gleichen Daten, aber ist eine von ihnen auch die tatsachliche Verteilung?

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


np.random.seed(1)
num = 10000

dists = {
    cat: pd.DataFrame(
        dict(
            x=np.random.normal(x, s, num),
            y=np.random.normal(y, s, num),
            val=val,
            cat=cat,
        )
    )
    for x, y, s, val, cat in [
        (2, 2, 0.01, 10, "d1"),
        (2, -2, 0.1, 20, "d2"),
        (-2, -2, 0.5, 30, "d3"),
        (-2, 2, 1.0, 40, "d4"),
        (0, 0, 3, 50, "d5"),
    ]
}

df = pd.concat(dists, ignore_index=True)
df["cat"] = df["cat"].astype("category")
df.tail()
```

[1]:
```
              x         y  val cat
49995 -1.397579  0.610189   50  d5
49996 -2.649610  3.080821   50  d5
49997  1.933360  0.243676   50  d5
```

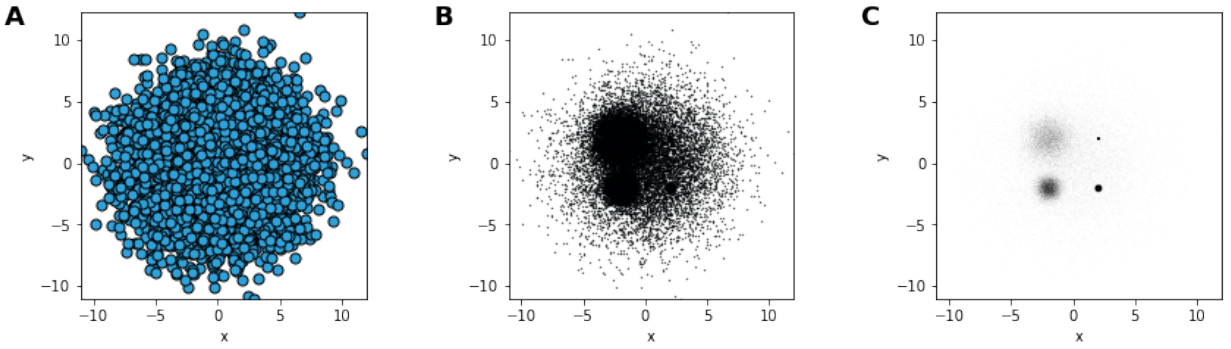
(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
49998  4.306374  1.032139   50  d5
49999 -0.493567 -2.242669   50  d5
```

Hier haben wir 50000 Punkte, 10000 in jeder von funf Kategorien mit zugehorigen numerischen Werten. Diese Datenmenge kann nur langsam mit Bokeh oder ahnlichen Bibliotheken geplottet werden, da die vollstandigen Daten zum Webbrowser uber- tragen werden mussen. Daruber zeigen sich beim Plotten von Daten dieser Große mit Standardansatzen einige Probleme:

- Plot A leidet an *overplotting*, wobei die Verteilung durch spater geplottete Datenpunkte verdeckt wird.
- Plot B verwendet kleinere Punkte, um ein Uberplotten zu vermeiden, leidet jedoch an Ubersattigung, wobei Unter- schiede in der Datenpunktdichte nicht sichtbar sind, da alle Dichten oberhalb eines bestimmten Wertes als die gleiche reine schwarze Farbe angezeigt werden
- Plot C verwendet Transparenz, um Ubersattigung zu vermeiden, leidet jedoch an Untersattigung, wobei die 10.000 Daten- punkte in der großten Kategorie (bei `0,0`) uberhaupt nicht sichtbar sind.
- Bokeh kann 50.000 Punkte verarbeiten, aber wenn die Daten großer waren, wurden diese Darstellungen an *undersampling* leiden, wobei die Verteilung aufgrund zu geringer Datenpunkte in vergroßerten Bereichen nicht sichtbar oder irrefuhrend wird.

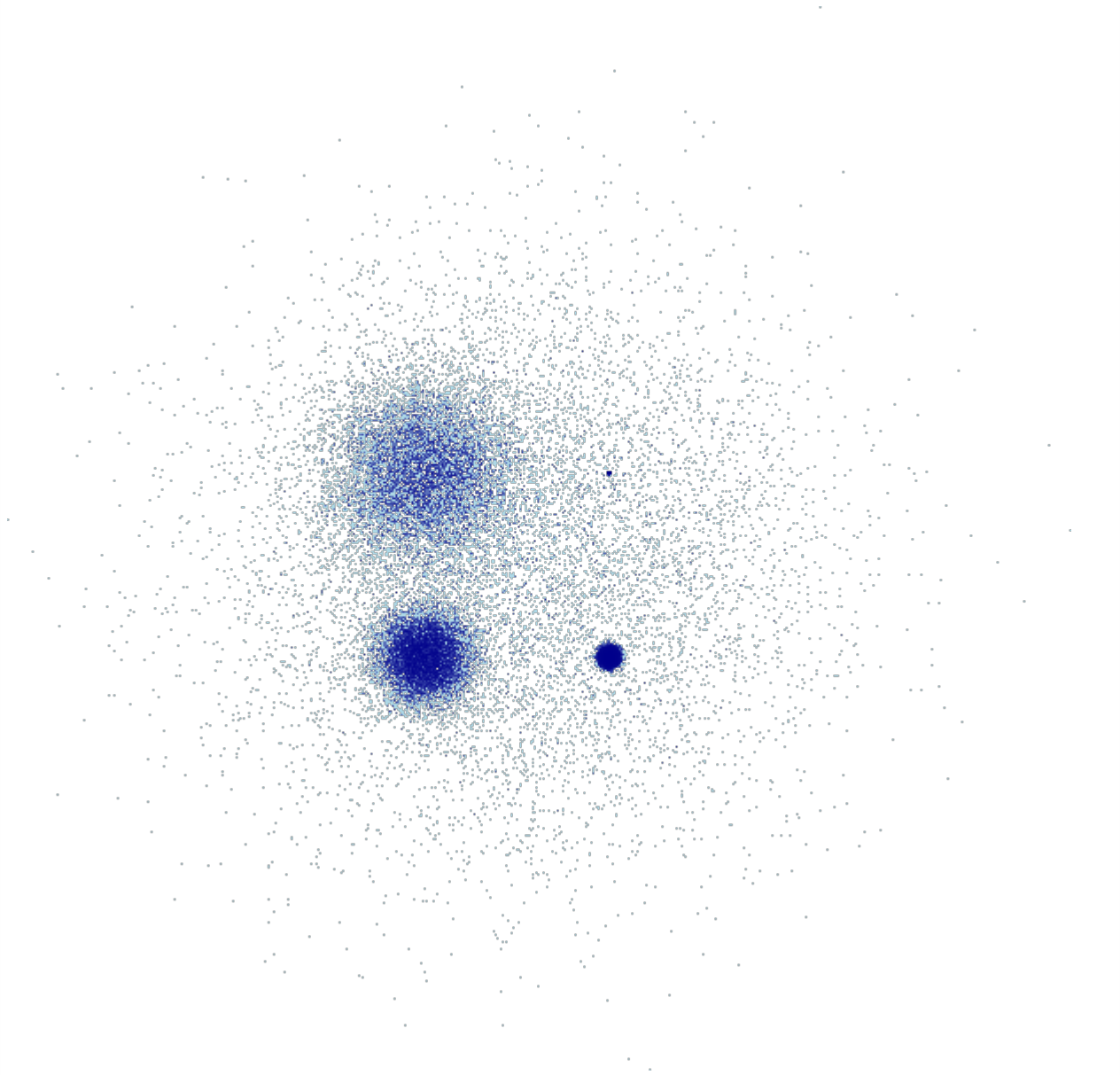


PlotA und PlotB erfordern auch ein zeitaufwandiges und fehleranfalliges manuelles Anpassen der Parameter, was problematisch ist, wenn die Daten so groß sind, dass die Visualisierung maßgeblich fur das Verstandnis der Daten wird. Mit dem Datashader konnen wir all diese Probleme vermeiden, indem wir die Daten in ein Array rendern, das automatisch den Umfang aller Dimensionen ermoglicht und dann die tatsachliche Verteilung ohne Parameteranpassung und mit sehr wenig Code anzeigt:

[2]:
```
import datashader as ds
import datashader.transfer_functions as tf
%time tf.shade(ds.Canvas().points(df,'x','y'))
```

```
CPU times: user 309 ms, sys: 20.8 ms, total: 329 ms
Wall time: 330 ms
```

[2]:



Projektion und Aggregation

In den ersten Schritten der Datashader-Pipeline wird die Auswahl getroffen * welche Variablen auf der X- und Y-Achse dargestellt werden sollen * in welcher Große die Werte zusammengefasst werden sollen * welchen Wertebereich das Array abdecken sollte * welche Funktion zum Aggregieren verwendet werden soll

[3]:
```
canvas = ds.Canvas(
    plot_width=250, plot_height=250, x_range=(-4, 4), y_range=(-4, 4)
)
agg = canvas.points(df, "x", "y", agg=ds.count())
agg
```

[3]:
```
<xarray.DataArray (y: 250, x: 250)>
array([[0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       ...,
       [0, 1, 0, ..., 0, 0, 0],
       [0, 0, 0, ..., 0, 0, 0],
       [1, 0, 0, ..., 0, 0, 0]], dtype=uint32)
Coordinates:
  * x        (x) float64 -3.984 -3.952 -3.92 -3.888 ... 3.888 3.92 3.952 3.984
  * y        (y) float64 -3.984 -3.952 -3.92 -3.888 ... 3.888 3.92 3.952 3.984
Attributes:
    x_range:  (-4, 4)
    y_range:  (-4, 4)
```

Hier legen wir fest, dass die Spalten x und y auf die x- und y-Achsen gemappt und mit `count` aggregiert werden sollen. Dies fuhrt zu einem 2D-xarray der angeforderten Große, das einen Wert fur jedes mogliche Pixel enthalt und die Anzahl der Datenpunkte zahlt, die diesem zugeordnet wurden. Ein xarray ahnelt einer NumPy- oder Pandas-Datenstruktur und unterstutzt ahnliche Operationen, ermoglicht jedoch beliebige mehrdimensionale Daten.

Zu den verfugbaren Reduktionsfunktionen, die zum Aggregieren verwendet werden konnen, gehoren:

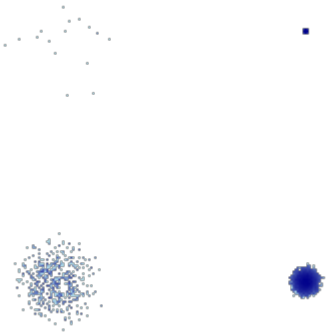
- `count()`: ganzzahlige Anzahl von Datenpunkten fur jedes Pixel (Standardeinstellung)
- `any()`: für jeden Datenpunkt ein Pixel, sonst `0`
- `sum(column)`: Gesamtwert der angegebenen Spalte fur alle Datenpunkte in diesem Pixel
- `count_cat(column)`: Anzahl von Datenpunkten per Kategorie anhand der angegebenen kategorialen Spalte, die mit Pandas Categorical data-Typ deklariert werden muss

Transformation

Sobald Daten in der Xarray-Aggregatform vorliegen, konnen sie auf verschiedene Arten verarbeitet werden, wodurch Datashader noch flexibler und leistungsfahiger wird. Anstatt alle Daten aufzuzeichnen, konnen wir zum Beispiel nur die Anzahl der 99. Perzentile darstellen:

[4]:
```
tf.shade(agg.where(agg>=np.percentile(agg,99)))
```

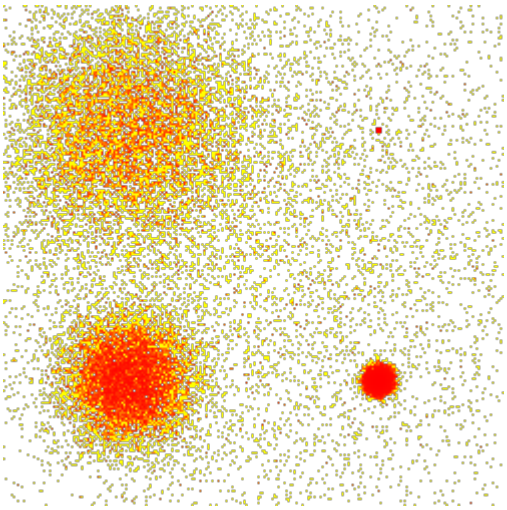
[4]:



Colormapping

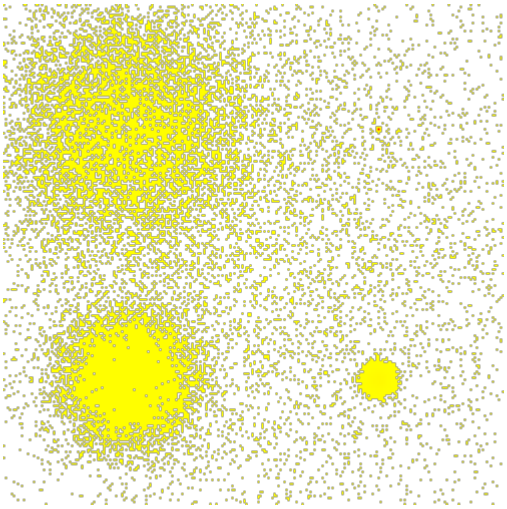
Die Werte in einem Array aggregierter Daten konnen in Pixelfarben konvertiert werden. Dabei unterstutzt Datashader jede Bokeh-Palette oder Liste von Farben:

[5]:
```
tf.shade(agg, cmap=["yellow", "red"])
```

[5]:


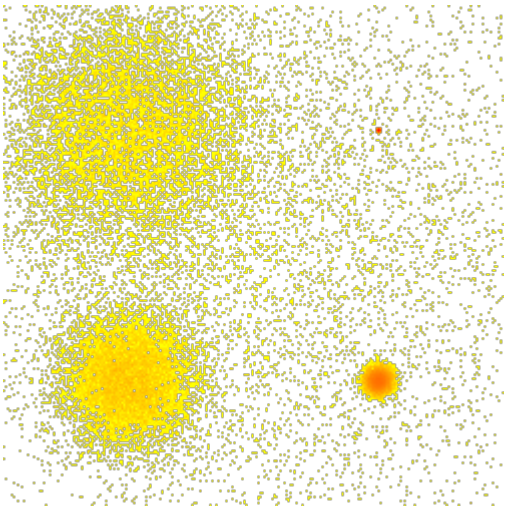
Wir konnen auch wahlen, wie die Datenwerte in Farben abgebildet werden sollen: * `linear` * `log` * `eq_hist`

[6]:
```
tf.shade(agg, cmap=["yellow", "red"], how="linear")
```

[6]:


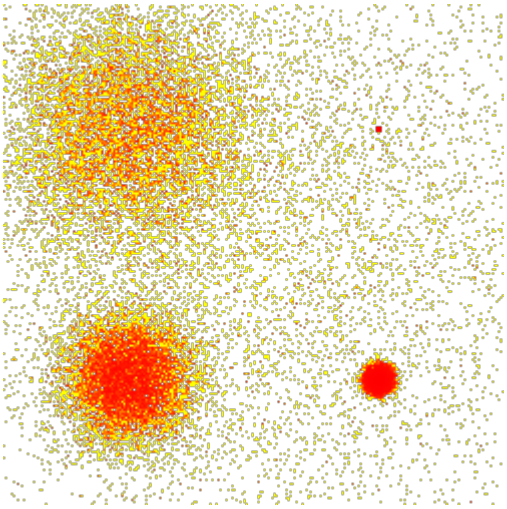
[7]:
```
tf.shade(agg, cmap=["yellow", "red"], how="log")
```

[7]:



[8]:
```
tf.shade(agg, cmap=["yellow", "red"], how="eq_hist")
```

[8]:

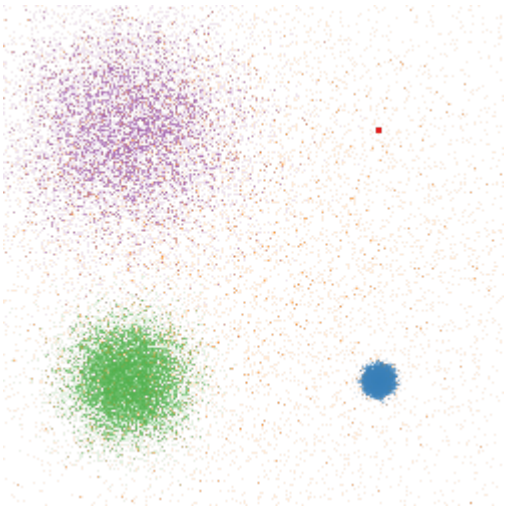


Bei `linear` wird rot nur fur das einzelne Pixel mit der hochsten Dichte verwendet. Das `log`-Mapping weist ahnliche Probleme auf, ist jedoch weniger schwerwiegend, da ein breiter Bereich von Datenwerten gelb abgebildet wird. Die Einstellung `eq_hist` (Standard) vermittelt korrekt die Dichteunterschiede zwischen den verschiedenen Verteilungen, indem das Histogramm der Pixelwerte so abgeglichen wird, dass jede Pixelfarbe gleich haufig verwendet wird.

Bei mehreren Kategorien konnen auch die einzelnen Aggregate eingefarbt werden:

[9]:
```
color_key = dict(d1="blue", d2="green", d3="yellow", d4="orange", d5="red")
aggc = canvas.points(df, "x", "y", ds.count_cat("cat"))
tf.shade(aggc, color_key)
```

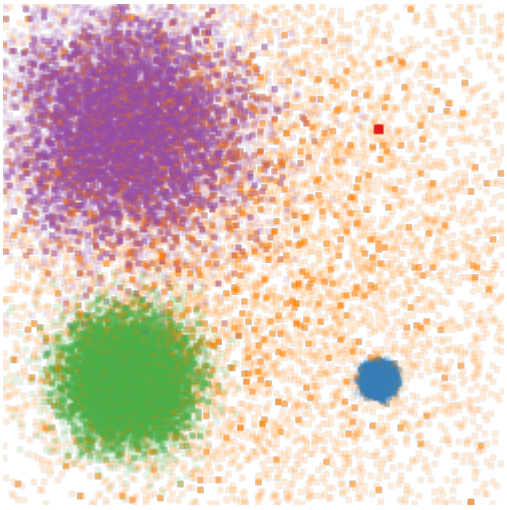
[9]:



Wenn die Punkte zu klein erscheinen, konnt ihr sie mit spreading im endgultigen Bild vergrößern.

[10]:
```
tf.spread(tf.shade(aggc, color_key))
```

[10]:



`tf.spread` verwendet eine feste (wenn auch konfigurierbare) Ausbreitungsgroße, wahrend ein ahnlicher Befehl `tf.dynspread` unterschiedlich verteilt, abhangig von der Plotdichte in dieser Ansicht.

Einbetten

Die von Datashader erzeugten Bilder konnen mit jedem Plot- oder Anzeigeprogramm verwendet werden. Bokeh bietet daruberhinaus interaktives Zoomen und Verschieben, um auch extrem große Datensatze zu untersuchen. Wir mussen nur die obigen Befehle in eine Callback-Funktion einbinden und sie dann einer Bokeh-`figure` hinzufugen:

[11]:
```
import bokeh.plotting as bp

from datashader.bokeh_ext import InteractiveImage


bp.output_notebook()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
p = bp.figure(tools="pan,wheel_zoom,reset", x_range=(-5, 5), y_range=(-5, 5))


def image_callback(x_range, y_range, w, h):
    cvs = ds.Canvas(
        plot_width=w, plot_height=h, x_range=x_range, y_range=y_range
    )
    agg = cvs.points(df, "x", "y", ds.count_cat("cat"))
    img = tf.shade(agg, color_key)
    return tf.dynspread(img, threshold=0.25)


InteractiveImage(p, image_callback)
```

Data type cannot be displayed: application/javascript, application/vnd.bokehjs_load.v0+json

[11]:
```
<datashader.bokeh_ext.InteractiveImage at 0x12825c630>
```

Damit konnt ihr jetzt auch die Achsenwerte sehen, die in bloßen Bildern nicht sichtbar sind. Wenn ihr den Zoom aktiviert, werden beliebige Bereiche des Diagramms vergroßert wobei ein neues Datashader-Bild mit dem Callback gerendert und im Diagramm angezeigt wird.

Ihr konnt auch problemlos andere Bokeh-Daten im selben Plot uberlagern oder `map`-Tiles fur geographische Daten im Web Mercator-Format in den Hintergrund setzen.

Datashader funktioniert ahnlich auch fur line-Plots (z.B. Zeitreihen und Trajektorien). So konnen alle Datenpunkte verwendet werden, ohne selbst eine Unterteilung treffen zu mussen. Es kann auch Rasterdaten (z.B. Satellitenwetterdaten) verwenden, um es in einem angeforderten Grid zu rastern, das dann analysiert oder eingefarbt oder mit anderen Nicht-Rasterdaten kombiniert werden kann. Wenn ihr beispielsweise Hohenangaben in Rasterform und Einkommensdaten als einzelne Punkte habt, konnt ihr leicht alle Pixel zeichnen, bei denen das durchschnittliche Einkommen uber einem bestimmten Schwellenwert liegt und die Hohe unter einem bestimmten Wert liegt. Dies ware mit einem traditionellen Workflow nur sehr schwierig auszudrucken.

Weitere Informationen findet ihr unter anaconda.org/jbednar/notebooks.

GeoViews

Mit GeoViews lassen sich geografische, meteorologische und ozeanografische Datensätze leicht erkunden und visualisieren. GeoViews basiert auf HoloViews und ergänzt deren Visualisierung multidimensionaler Daten um geografischen Plot-Typen, die auf *Cartopy* basieren.

Installation

Wenn *Cartopy* bereits installiert ist, kann GeoViews mit pipenv installiert werden:

```
$ pipenv install geoviews
```

Beispiele

GeoViews arbeitet gut mit Iris- und xarray-Bibliotheken für mehrdimensionale Arrays zusammen, wie sie in netCDF-Dateien gespeichert sind. GeoViews akzeptiert jedoch auch Daten als NumPy-Arrays und Pandas-Datenrahmen. In jedem Fall können die Daten in ihrem ursprünglichen, nativen Format gespeichert und in ein HoloViews- oder GeoViews-Objekt verpackt werden, das interaktive Visualisierungen ermöglicht.

xarray-Beispiel

Für das folgende Beispiel benötigen wir ebenfalls noch xarray. Es kann mit Spack installiert werden mit:

```
$ spack install py-xarray
```

[1]:
```
import geoviews as gv
import geoviews.feature as gf
import xarray as xr

from cartopy import crs


gv.extension("bokeh", "matplotlib")
```

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

[2]:
```
(gf.ocean + gf.land + gf.ocean * gf.land * gf.coastline * gf.borders).opts(
    "Feature", projection=crs.Geostationary(), global_extent=True, height=325
).cols(3)
```

[2]:
```
:Layout
   .Ocean.I   :Feature   [Longitude,Latitude]
   .Land.I    :Feature   [Longitude,Latitude]
   .Overlay.I :Overlay
      .Ocean.I     :Feature   [Longitude,Latitude]
      .Land.I      :Feature   [Longitude,Latitude]
      .Coastline.I :Feature   [Longitude,Latitude]
      .Borders.I   :Feature   [Longitude,Latitude]
```

Als Beispieldaten verwenden wir geoviews-sample-data.zip und entpacken es als `data`-Verzeichnis.

[3]:
```
!curl http://assets.holoviews.org/geoviews-sample-data.zip --output geoviews-sample-data.
↪zip
!unzip geoviews-sample-data.zip
```

```
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100 6246k  100 6246k    0     0  5679k      0  0:00:01  0:00:01 --:--:-- 5694k
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Archive:  geoviews-sample-data.zip
  inflating: ensemble.nc
  inflating: ensembles.nc
  inflating: pre-industrial.nc
```

[4]:
```
dataset = gv.Dataset(xr.open_dataset("./ensemble.nc"))
ensemble = dataset.to(
    gv.Image, ["longitude", "latitude"], "surface_temperature"
)

gv.output(
    ensemble.opts(
        cmap="viridis", colorbar=True, fig_size=200, backend="matplotlib"
    )
    * gf.coastline(),
    backend="matplotlib",
)
```

GeoPandas-Beispiel

GeoViews unterstützt auch nativ *GeoPandas*-Datenstrukturen, so dass wir shapefiles und Choropleth einfach darstellen können:

[4]:
```
import geopandas as gpd


gv.Polygons(
    gpd.read_file(gpd.datasets.get_path("naturalearth_lowres")),
    vdims=["pop_est", ("name", "Country")],
).opts(tools=["hover"], width=600, projection=crs.Robinson())
```

```
/tmp/ipykernel_88986/2104909695.py:5: FutureWarning: The geopandas.dataset module is␣
↪deprecated and will be removed in GeoPandas 1.0. You can get the original
↪'naturalearth_lowres' data from https://www.naturalearthdata.com/downloads/110m-
↪cultural-vectors/.
  gpd.read_file(gpd.datasets.get_path("naturalearth_lowres")),
```

[4]:
```
:Polygons   [Longitude,Latitude]   (pop_est,name)
```

KAPITEL 6

OpenGL

Die Open Graphics Library OpenGL ist eine API-Spezifikation für 2D- und 3D-Grafikanwendungen. Neben Vaex stehen noch weitere Python-Bibliotheken zur Verfügung, die OpenGL nutzen:

Vispy
: Python-Bibliothek für interaktive wissenschaftliche Visualisierungen

Mayavi
: Python-Bibliothek für die 3D-Visualisierung wissenschaftlicher Daten

itkwidgets
: Jupyter-Widgets zur Visualisierung von Bildern, Punktmengen und Netzen in 2D und 3D

vedo
: Python-Modul für die wissenschaftliche Analyse von 3D-Daten

Polyscope
: Ein C++ und Python-Viewer für 3D-Daten wie Netze und Punktwolken

Glumpy
: Schnittstelle zwischen NumPy und OpenGL

Siehe auch:

- Nicolas P. Rougier: Python & OpenGL for Scientific Visualization

KAPITEL 7

D3.js

d3js ist eine Javascript-Bibliothek zur Datenvisualisierung, die U.A. (unter anderem) von den folgenden Python-Bibliotheken genutzt wird:

7.1 bqplot

2-D Plot-Bibliothek für Jupyter-Notebooks

bqplot basiert auf The Grammar of Graphics (Statistics and Computing).

Siehe auch:

- Docs
- GitHub
- Gallery

7.1.1 Ziele

Einheitliches Framework für 2D-Visualisierungen
: mit einer Pythonic-API

Erweiterbare API
: erlaubt das Hinzufügen von User Interactions (Verschieben, Zoomen, Auswahl USW. (und so weiter))

Benutzerdefinierte API
: Benutzer können eigene Visualisierungen mithilfe des internen Objektmodells erstellen, das von The Grammar of Graphics inspiriert ist (figure, marks, axes, scales) inspiriert ist und deren Visualisierungen interaktiv erschlossen werden können

Kontextbasierte API
: ähnlich wie Matplotlibs `pyplot` werden sinnvolle Auswahlmöglichkeiten für die meisten Parameter geboten

bqplot-API und -Objektmodell

bqplot-API

[1]:
```
import numpy as np

from bqplot import pyplot as plt


plt.figure(1)
n = 200
x = np.linspace(0.0, 10.0, n)
y = np.cumsum(np.random.randn(n))
plt.plot(x, y, axes_options={"y": {"grid_lines": "dashed"}})
plt.show()
```

```
VBox(children=(Figure(axes=[Axis(scale=LinearScale()), Axis(grid_lines='dashed',␣
↪orientation='vertical', scale...
```

bqplot-Objektmodell

[2]:
```
import numpy as np

from bqplot import Axis, Bars, Figure, LinearScale, Lines, OrdinalScale
from IPython.display import display


size = 20
np.random.seed(0)

x_data = np.arange(size)

x_ord = OrdinalScale()
y_sc = LinearScale()

bar = Bars(
    x=x_data,
    y=np.random.randn(2, size),
    scales={"x": x_ord, "y": y_sc},
    type="stacked",
)
line = Lines(
    x=x_data,
    y=np.random.randn(size),
    scales={"x": x_ord, "y": y_sc},
    stroke_width=3,
    colors=["red"],
    display_legend=True,
    labels=["Line chart"],
)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
ax_x = Axis(scale=x_ord, grid_lines="solid", label="X")
ax_y = Axis(
    scale=y_sc,
    orientation="vertical",
    tick_format="0.2f",
    grid_lines="solid",
    label="Y",
)

Figure(
    marks=[bar, line],
    axes=[ax_x, ax_y],
    title="API Example",
    legend_location="bottom-right",
)
```

```
Figure(axes=[Axis(label='X', scale=OrdinalScale()), Axis(label='Y', orientation='vertical
↪', scale=LinearScale(...
```

Markieren-Interaktionen

[1]:
```
from __future__ import print_function

import numpy as np
import pandas as pd

from bqplot import *
from ipywidgets import Layout
```

Streudiagramm

Auswahl in Streudiagrammen

Klickt auf einen Punkt im Streudiagramm, um ihn auszuwählen und führt anschließend die Zelle unten aus, um die Auswahl zu überprüfen. Versucht schließlich, die `Strg`-Taste (oder auf dem Mac) gedrückt zu halten und auf einen anderen Punkt zu klicken. Durch Klicken auf den Hintergrund wird die Auswahl zurückgesetzt.

[2]:
```
x_sc = LinearScale()
y_sc = LinearScale()

x_data = np.arange(20)
y_data = np.random.randn(20)

scatter_chart = Scatter(
    x=x_data,
    y=y_data,
    scales={"x": x_sc, "y": y_sc},
    colors=["dodgerblue"],
    interactions={"click": "select"},
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    selected_style={"opacity": 1.0, "fill": "DarkOrange", "stroke": "Red"},
    unselected_style={"opacity": 0.5},
)

ax_x = Axis(scale=x_sc)
ax_y = Axis(scale=y_sc, orientation="vertical", tick_format="0.2f")

Figure(marks=[scatter_chart], axes=[ax_x, ax_y])
```

```
Figure(axes=[Axis(scale=LinearScale()), Axis(orientation='vertical', scale=LinearScale(),
↪ tick_format='0.2f')]...
```

[3]:
```
scatter_chart.selected
```

Alternativ kann das Attribut `selected` direkt in Python festgelegt werden, z.B. mit der folgenden Zeile:

[4]:
```
scatter_chart.selected = [1, 2, 3]
```

Streudiagramm-Interaktionen und -Tooltips

[5]:
```
from ipywidgets import *
```

[6]:
```
x_sc = LinearScale()
y_sc = LinearScale()

x_data = np.arange(20)
y_data = np.random.randn(20)

dd = Dropdown(options=["First", "Second", "Third", "Fourth"])
scatter_chart = Scatter(
    x=x_data,
    y=y_data,
    scales={"x": x_sc, "y": y_sc},
    colors=["dodgerblue"],
    names=np.arange(100, 200),
    names_unique=False,
    display_names=False,
    display_legend=True,
    labels=["Blue"],
)
ins = Button(icon="fa-legal")
scatter_chart.tooltip = ins

scatter_chart2 = Scatter(
    x=x_data,
    y=np.random.randn(20),
    scales={"x": x_sc, "y": y_sc},
    colors=["orangered"],
    tooltip=dd,
    names=np.arange(100, 200),
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    names_unique=False,
    display_names=False,
    display_legend=True,
    labels=["Red"],
)

ax_x = Axis(scale=x_sc)
ax_y = Axis(scale=y_sc, orientation="vertical", tick_format="0.2f")

Figure(marks=[scatter_chart, scatter_chart2], axes=[ax_x, ax_y])
```

```
Figure(axes=[Axis(scale=LinearScale()), Axis(orientation='vertical', scale=LinearScale(),
↪ tick_format='0.2f')]...
```

Call backs und benutzerdefinierte Nachrichten definieren:

[7]:
```
def print_event(self, target):
    print(target)


# Adding call back to scatter events
# print custom mssg on hover and background click of Blue Scatter
scatter_chart.on_hover(print_event)
scatter_chart.on_background_click(print_event)

# print custom mssg on click of an element or legend of Red Scatter
scatter_chart2.on_element_click(print_event)
scatter_chart2.on_legend_click(print_event)
```

`figure` als Tooltip definieren:

[8]:
```
# Adding figure as tooltip
x_sc = LinearScale()
y_sc = LinearScale()

x_data = np.arange(10)
y_data = np.random.randn(10)

lc = Lines(x=x_data, y=y_data, scales={"x": x_sc, "y": y_sc})
ax_x = Axis(scale=x_sc)
ax_y = Axis(scale=y_sc, orientation="vertical", tick_format="0.2f")
tooltip_fig = Figure(
    marks=[lc], axes=[ax_x, ax_y], layout=Layout(min_width="600px")
)

scatter_chart.tooltip = tooltip_fig
```

[9]:
```
# Changing interaction from hover to click for tooltip
scatter_chart.interactions = {"click": "tooltip"}
```

Liniendiagramm

[10]:
```
# Adding default tooltip to Line Chart
x_sc = LinearScale()
y_sc = LinearScale()

x_data = np.arange(100)
y_data = np.random.randn(3, 100)

def_tt = Tooltip(
    fields=["name", "index"], formats=["", ".2f"], labels=["id", "line_num"]
)
line_chart = Lines(
    x=x_data,
    y=y_data,
    scales={"x": x_sc, "y": y_sc},
    tooltip=def_tt,
    display_legend=True,
    labels=["line 1", "line 2", "line 3"],
)

ax_x = Axis(scale=x_sc)
ax_y = Axis(scale=y_sc, orientation="vertical", tick_format="0.2f")

Figure(marks=[line_chart], axes=[ax_x, ax_y])
```

```
Figure(axes=[Axis(scale=LinearScale()), Axis(orientation='vertical', scale=LinearScale(),
↪ tick_format='0.2f')]...
```

[11]:
```
# Adding call back to print event when legend or the line is clicked
line_chart.on_legend_click(print_event)
line_chart.on_element_click(print_event)
```

Balkendiagramm

[12]:
```
# Adding interaction to select bar on click for Bar Chart
x_sc = OrdinalScale()
y_sc = LinearScale()

x_data = np.arange(10)
y_data = np.random.randn(2, 10)

bar_chart = Bars(
    x=x_data,
    y=[y_data[0, :].tolist(), y_data[1, :].tolist()],
    scales={"x": x_sc, "y": y_sc},
    interactions={"click": "select"},
    selected_style={"stroke": "orange", "fill": "red"},
    labels=["Level 1", "Level 2"],
    display_legend=True,
)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
ax_x = Axis(scale=x_sc)
ax_y = Axis(scale=y_sc, orientation="vertical", tick_format="0.2f")

Figure(marks=[bar_chart], axes=[ax_x, ax_y])
```

```
Figure(axes=[Axis(scale=OrdinalScale()), Axis(orientation='vertical',␣
↪scale=LinearScale(), tick_format='0.2f')...
```

[13]:
```
# Adding a tooltip on hover in addition to select on click
def_tt = Tooltip(fields=["x", "y"], formats=["", ".2f"])
bar_chart.tooltip = def_tt
bar_chart.interactions = {
    "legend_hover": "highlight_axes",
    "hover": "tooltip",
    "click": "select",
}
```

[14]:
```
# Changing tooltip to be on click
bar_chart.interactions = {"click": "tooltip"}
```

[15]:
```
# Call back on legend being clicked
bar_chart.type = "grouped"
bar_chart.on_legend_click(print_event)
```

Histogramm

[16]:
```
# Adding tooltip for Histogram
x_sc = LinearScale()
y_sc = LinearScale()

sample_data = np.random.randn(100)

def_tt = Tooltip(formats=["", ".2f"], fields=["count", "midpoint"])
hist = Hist(
    sample=sample_data,
    scales={"sample": x_sc, "count": y_sc},
    tooltip=def_tt,
    display_legend=True,
    labels=["Test Hist"],
    select_bars=True,
)
ax_x = Axis(scale=x_sc, tick_format="0.2f")
ax_y = Axis(scale=y_sc, orientation="vertical", tick_format="0.2f")

Figure(marks=[hist], axes=[ax_x, ax_y])
```

```
Figure(axes=[Axis(scale=LinearScale(), tick_format='0.2f'), Axis(orientation='vertical',␣
↪scale=LinearScale(), ...
```

[17]:
```
# Changing tooltip to be displayed on click
hist.interactions = {"click": "tooltip"}
```

[18]:
```
# Changing tooltip to be on click of legend
hist.interactions = {"legend_click": "tooltip"}
```

Tortendiagramm

[19]:
```
pie_data = np.abs(np.random.randn(10))

sc = ColorScale(scheme="Reds")
tooltip_widget = Tooltip(
    fields=["size", "index", "color"], formats=["0.2f", "", "0.2f"]
)
pie = Pie(
    sizes=pie_data,
    scales={"color": sc},
    color=np.random.randn(10),
    tooltip=tooltip_widget,
    interactions={"click": "tooltip"},
    selected_style={"fill": "red"},
)

pie.selected_style = {"opacity": "1", "stroke": "white", "stroke-width": "2"}
pie.unselected_style = {"opacity": "0.2"}

Figure(marks=[pie])
```

```
Figure(fig_margin={'top': 60, 'bottom': 60, 'left': 60, 'right': 60},␣
↪marks=[Pie(color=array([ 0.53097893, -0....
```

[20]:
```
# Changing interaction to select on click and tooltip on hover
pie.interactions = {"click": "select", "hover": "tooltip"}
```

Selektoren

`bqplot.interacts` bietet die folgenden Selektoren:

Klasse	Beschreibung
`BrushIntervalSelector(**kwargs)`	Auswahl eines Intervalls mit dem Pinsel
`BrushSelector(**kwargs)`	Auswahl mit dem Pinsel
`HandDraw(**kwargs)`	Handgezeichnete Interaktion
`IndexSelector(**kwargs)`	Auswahl eines Index
`FastIntervalSelector(**kwargs)`	Schnelle Auswahl eines Intervalls
`MultiSelector(**kwargs)`	Multiselektor-Interaktion
`OneDSelector(**kwargs)`	Eindimensionale Auswahl
`Interaction(**kwargs)`	Basisklasse für Interaktionen
`PanZoom(**kwargs)`	Interaktionen zum Verschieben und Zoomen von Skalen
`Selector(**kwargs)`	Auswahlinteraktion
`TwoDSelector(**kwargs)`	Zweidimensionale Auswahlinteraktion

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


symbol = "Security 1"
symbol2 = "Security 2"
```

[2]:
```
price_data = pd.DataFrame(
    np.cumsum(np.random.randn(150, 2).dot([[0.5, 0.4], [0.4, 1.0]]), axis=0)
    + 100,
    columns=["Security 1", "Security 2"],
    index=pd.date_range(start="01-01-2007", periods=150),
)

dates_actual = price_data.index.values
prices = price_data[symbol].values
```

[3]:
```
from bqplot import (
    Axis,
    Bars,
    DateScale,
    Figure,
    Hist,
    LinearScale,
    Lines,
    Scatter,
)
from bqplot.interacts import (
    BrushIntervalSelector,
    BrushSelector,
    FastIntervalSelector,
    HandDraw,
    IndexSelector,
    LassoSelector,
    MultiSelector,
    PanZoom,
)
from ipywidgets import HTML, ToggleButtons, VBox
from traitlets import link
```

Liniendiagramm-Selektoren

FastIntervalSelector

[4]:
```
## First we define a Figure
dt_x_fast = DateScale()
lin_y = LinearScale()

x_ax = Axis(label="Index", scale=dt_x_fast)
x_ay = Axis(label=(symbol + " Price"), scale=lin_y, orientation="vertical")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
lc = Lines(
    x=dates_actual,
    y=prices,
    scales={"x": dt_x_fast, "y": lin_y},
    colors=["orange"],
)
lc_2 = Lines(
    x=dates_actual[50:],
    y=prices[50:] + 2,
    scales={"x": dt_x_fast, "y": lin_y},
    colors=["blue"],
)
```

[5]:
```
## Next we define the type of selector we would like
intsel_fast = FastIntervalSelector(scale=dt_x_fast, marks=[lc, lc_2])
```

[6]:
```
## Now, we define a function that will be called when the FastIntervalSelector is␣
↪interacted with
def fast_interval_change_callback(change):
    db_fast.value = "The selected period is " + str(change.new)
```

[7]:
```
## Now we connect the selectors to that function
intsel_fast.observe(fast_interval_change_callback, names=["selected"])
```

[8]:
```
## We use the HTML widget to see the value of what we are selecting and modify it when␣
↪an interaction is performed
## on the selector
db_fast = HTML()
db_fast.value = "The selected period is " + str(intsel_fast.selected)

fig_fast_intsel = Figure(
    marks=[lc, lc_2],
    axes=[x_ax, x_ay],
    title="Fast Interval Selector Example",
    interaction=intsel_fast,
)  # This is where we assign the interaction to this particular Figure

VBox([db_fast, fig_fast_intsel])
```

```
VBox(children=(HTML(value='The selected period is None'), Figure(axes=[Axis(label='Index
↪', scale=DateScale(), ...
```

IndexSelector

[9]:
```
db_index = HTML(value="[]")
```

[10]:
```
## Now we try a selector made to select all the y-values associated with a single x-value
index_sel = IndexSelector(scale=dt_x_fast, marks=[lc, lc_2])
```

[11]:
```
## Now, we define a function that will be called when the selectors are interacted with
def index_change_callback(change):
    db_index.value = "The selected date is " + str(change.new)
```

[12]:
```
index_sel.observe(index_change_callback, names=["selected"])
```

[13]:
```
fig_index_sel = Figure(
    marks=[lc, lc_2],
    axes=[x_ax, x_ay],
    title="Index Selector Example",
    interaction=index_sel,
)
VBox([db_index, fig_index_sel])
```

```
VBox(children=(HTML(value='[]'), Figure(axes=[Axis(label='Index', scale=DateScale(),␣
↪side='bottom'), Axis(labe...
```

Rückgabe von Indizes ausgewählter Werte:

[14]:
```
from datetime import datetime as py_dtime


dt_x_index = DateScale(min=np.datetime64(py_dtime(2006, 6, 1)))
lin_y2 = LinearScale()

lc2_index = Lines(
    x=dates_actual, y=prices, scales={"x": dt_x_index, "y": lin_y2}
)

x_ax1 = Axis(label="Date", scale=dt_x_index)
x_ay2 = Axis(label=(symbol + " Price"), scale=lin_y2, orientation="vertical")
```

[15]:
```
intsel_date = FastIntervalSelector(scale=dt_x_index, marks=[lc2_index])
```

[16]:
```
db_date = HTML()
db_date.value = str(intsel_date.selected)
```

[17]:
```
## Now, we define a function that will be called when the selectors are interacted with -
↪ a callback
def date_interval_change_callback(change):
    db_date.value = str(change.new)
```

[18]:
```
## Notice here that we call the observe on the Mark lc2_index rather than on the␣
↪selector intsel_date
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
lc2_index.observe(date_interval_change_callback, names=["selected"])

fig_date_mark = Figure(
    marks=[lc2_index],
    axes=[x_ax1, x_ay2],
    title="Fast Interval Selector Selected Indices Example",
    interaction=intsel_date,
)

VBox([db_date, fig_date_mark])
```

```
VBox(children=(HTML(value='None'), Figure(axes=[Axis(label='Date',␣
↪scale=DateScale(min=datetime.datetime(2006,...
```

BrushSelector

Wir können dasselbe mit jedem anderen Selektor tun:

[19]:
```
## Defining a new Figure
dt_x_brush = DateScale(min=np.datetime64(py_dtime(2006, 6, 1)))
lin_y2_brush = LinearScale()

lc3_brush = Lines(
    x=dates_actual, y=prices, scales={"x": dt_x_brush, "y": lin_y2_brush}
)

x_ax_brush = Axis(label="Date", scale=dt_x_brush)
x_ay_brush = Axis(
    label=(symbol + " Price"), scale=lin_y2_brush, orientation="vertical"
)
```

[20]:
```
db_brush = HTML(value="[]")
```

[21]:
```
brushsel_date = BrushIntervalSelector(
    scale=dt_x_brush, marks=[lc3_brush], color="FireBrick"
)
```

[22]:
```
## Now, we define a function that will be called when the selectors are interacted with -
↪ a callback
def date_brush_change_callback(change):
    db_brush.value = str(change.new)
```

[23]:
```
lc3_brush.observe(date_brush_change_callback, names=["selected"])
```

[24]:
```
fig_brush_sel = Figure(
    marks=[lc3_brush],
    axes=[x_ax_brush, x_ay_brush],
    title="Brush Selector Selected Indices Example",
    interaction=brushsel_date,
)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
VBox([db_brush, fig_brush_sel])
```

```
VBox(children=(HTML(value='[]'), Figure(axes=[Axis(label='Date',␣
↪scale=DateScale(min=datetime.datetime(2006, 6...
```

Scatter-Chart-Selektoren

BrushSelector

[25]:
```
date_fmt = "%m-%d-%Y"

sec2_data = price_data[symbol2].values
dates = price_data.index.values
```

[26]:
```
sc_x = LinearScale()
sc_y = LinearScale()

scatt = Scatter(x=prices, y=sec2_data, scales={"x": sc_x, "y": sc_y})

sc_xax = Axis(label=(symbol), scale=sc_x)
sc_yax = Axis(label=(symbol2), scale=sc_y, orientation="vertical")
```

[27]:
```
br_sel = BrushSelector(x_scale=sc_x, y_scale=sc_y, marks=[scatt], color="red")

db_scat_brush = HTML(value="[]")
```

[28]:
```
## call back for the selector
def brush_callback(change):
    db_scat_brush.value = str(br_sel.selected)
```

[29]:
```
br_sel.observe(brush_callback, names=["brushing"])
```

[30]:
```
fig_scat_brush = Figure(
    marks=[scatt],
    axes=[sc_xax, sc_yax],
    title="Scatter Chart Brush Selector Example",
    interaction=br_sel,
)
```

[31]:
```
VBox([db_scat_brush, fig_scat_brush])
```

```
VBox(children=(HTML(value='[]'), Figure(axes=[Axis(label='Security 1',␣
↪scale=LinearScale(), side='bottom'), Ax...
```

BrushSelector mit Datumswerten

[32]:
```
sc_brush_dt_x = DateScale(date_format=date_fmt)
sc_brush_dt_y = LinearScale()

scatt2 = Scatter(
    x=dates_actual,
    y=sec2_data,
    scales={"x": sc_brush_dt_x, "y": sc_brush_dt_y},
)
```

[33]:
```
br_sel_dt = BrushSelector(
    x_scale=sc_brush_dt_x, y_scale=sc_brush_dt_y, marks=[scatt2]
)
```

[34]:
```
db_brush_dt = HTML(value=str(br_sel_dt.selected))
```

[35]:
```
## call back for the selector
def brush_dt_callback(change):
    db_brush_dt.value = str(br_sel_dt.selected)
```

[36]:
```
br_sel_dt.observe(brush_dt_callback, names=["brushing"])
```

[37]:
```
sc_xax = Axis(label=(symbol), scale=sc_brush_dt_x)
sc_yax = Axis(label=(symbol2), scale=sc_brush_dt_y, orientation="vertical")
fig_brush_dt = Figure(
    marks=[scatt2],
    axes=[sc_xax, sc_yax],
    title="Brush Selector with Dates Example",
    interaction=br_sel_dt,
)
```

[38]:
```
VBox([db_brush_dt, fig_brush_dt])
```

```
VBox(children=(HTML(value='None'), Figure(axes=[Axis(label='Security 1',␣
↪scale=DateScale(), side='bottom'), Ax...
```

Histogramm-Selektoren

[39]:
```
## call back for selectors
def interval_change_callback(name, value):
    db3.value = str(value)


## call back for the selector
def brush_callback(change):
    if not br_intsel.brushing:
        db3.value = str(br_intsel.selected)
```

[40]:
```
returns = np.log(prices[1:]) - np.log(prices[:-1])
hist_x = LinearScale()
hist_y = LinearScale()
hist = Hist(sample=returns, scales={"sample": hist_x, "count": hist_y})

br_intsel = BrushIntervalSelector(scale=hist_x, marks=[hist])
br_intsel.observe(brush_callback, names=["selected"])
br_intsel.observe(brush_callback, names=["brushing"])

db3 = HTML()
db3.value = str(br_intsel.selected)

h_xax = Axis(
    scale=hist_x,
    label="Returns",
    grids="off",
    set_ticks=True,
    tick_format="0.2%",
)
h_yax = Axis(
    scale=hist_y, label="Freq", orientation="vertical", grid_lines="none"
)

fig_hist = Figure(
    marks=[hist],
    axes=[h_xax, h_yax],
    title="Histogram Selection Example",
    interaction=br_intsel,
)
VBox([db3, fig_hist])
```

```
VBox(children=(HTML(value='None'), Figure(axes=[Axis(label='Returns',␣
↪scale=LinearScale(), tick_format='0.2%')...
```

MultiSelector

- Dieser Selektor bietet die Möglichkeit, mehrere Pinsel-Selektoren in einem Diagramm zu verwenden.
- Der erste Pinsel funktioniert wie ein normaler Pinsel.
- `Ctrl + click` erstellt einen neuen Pinsel, der wie der normale Pinsel funktioniert.
- Der `active`-Pinsel hat einen grünen Rand, während alle `inactive`-Pinsel einen roten Rand haben. `inactive`
- `Shift + click` deaktiviert den aktuellen `active`-Pinsel. Klickt nun auf einen `inactive`-Pinsel um ihn `active` zu machen.
- `Ctrl + Alt + Shift + click` löscht alle Pinsel und setzt sie zurück.

[41]:
```
def multi_sel_callback(change):
    if not multi_sel.brushing:
        db4.value = str(multi_sel.selected)
```

[42]:
```
line_x = LinearScale()
line_y = LinearScale()
line = Lines(
    x=np.arange(100), y=np.random.randn(100), scales={"x": line_x, "y": line_y}
)

multi_sel = MultiSelector(scale=line_x, marks=[line])
multi_sel.observe(multi_sel_callback, names=["selected"])
multi_sel.observe(multi_sel_callback, names=["brushing"])

db4 = HTML()
db4.value = str(multi_sel.selected)

h_xax = Axis(scale=line_x, label="Returns", grid_lines="none")
h_yax = Axis(
    scale=hist_y, label="Freq", orientation="vertical", grid_lines="none"
)

fig_multi = Figure(
    marks=[line],
    axes=[h_xax, h_yax],
    title="Multi-Selector Example",
    interaction=multi_sel,
)
VBox([db4, fig_multi])
```

```
VBox(children=(HTML(value='{}'), Figure(axes=[Axis(grid_lines='none', label='Returns',␣
↪scale=LinearScale()), A...
```

[43]:
```
# changing the names of the intervals.
multi_sel.names = ["int1", "int2", "int3"]
```

Multifunktionswähler mit Datum

[44]:
```
def multi_sel_dt_callback(change):
    if not multi_sel_dt.brushing:
        db_multi_dt.value = str(multi_sel_dt.selected)
```

[45]:
```
line_dt_x = DateScale(min=np.datetime64(py_dtime(2007, 1, 1)))
line_dt_y = LinearScale()
line_dt = Lines(
    x=dates_actual,
    y=sec2_data,
    scales={"x": line_dt_x, "y": line_dt_y},
    colors=["red"],
)

multi_sel_dt = MultiSelector(scale=line_dt_x)
multi_sel_dt.observe(multi_sel_dt_callback, names=["selected"])
multi_sel_dt.observe(multi_sel_dt_callback, names=["brushing"])
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
db_multi_dt = HTML()
db_multi_dt.value = str(multi_sel_dt.selected)

h_xax_dt = Axis(scale=line_dt_x, label="Returns", grid_lines="none")
h_yax_dt = Axis(
    scale=line_dt_y, label="Freq", orientation="vertical", grid_lines="none"
)

fig_multi_dt = Figure(
    marks=[line_dt],
    axes=[h_xax_dt, h_yax_dt],
    title="Multi-Selector with Date Example",
    interaction=multi_sel_dt,
)
VBox([db_multi_dt, fig_multi_dt])
```

```
VBox(children=(HTML(value='{}'), Figure(axes=[Axis(grid_lines='none', label='Returns',␣
↪scale=DateScale(min=dat...
```

LassoSelector

[46]:
```
lasso_sel = LassoSelector()
```

[47]:
```
xs, ys = LinearScale(), LinearScale()
data = np.arange(20)
line_lasso = Lines(x=data, y=data, scales={"x": xs, "y": ys})
scatter_lasso = Scatter(
    x=data, y=data, scales={"x": xs, "y": ys}, colors=["skyblue"]
)
bar_lasso = Bars(x=data, y=data / 2.0, scales={"x": xs, "y": ys})
xax_lasso, yax_lasso = Axis(scale=xs, label="X"), Axis(
    scale=ys, label="Y", orientation="vertical"
)
fig_lasso = Figure(
    marks=[scatter_lasso, line_lasso, bar_lasso],
    axes=[xax_lasso, yax_lasso],
    title="Lasso Selector Example",
    interaction=lasso_sel,
)
lasso_sel.marks = [scatter_lasso, line_lasso]
fig_lasso
```

```
Figure(axes=[Axis(label='X', scale=LinearScale()), Axis(label='Y', orientation='vertical
↪', scale=LinearScale()...
```

[48]:
```
scatter_lasso.selected, line_lasso.selected
```

[48]:
```
(None, None)
```

PanZoom

[49]:
```
xs_pz = DateScale(min=np.datetime64(py_dtime(2007, 1, 1)))
ys_pz = LinearScale()
line_pz = Lines(
    x=dates_actual,
    y=sec2_data,
    scales={"x": xs_pz, "y": ys_pz},
    colors=["red"],
)

panzoom = PanZoom(scales={"x": [xs_pz], "y": [ys_pz]})
xax = Axis(scale=xs_pz, label="Date", grids="off")
yax = Axis(
    scale=ys_pz, label="Price", orientation="vertical", grid_lines="none"
)

Figure(marks=[line_pz], axes=[xax, yax], interaction=panzoom)
```

```
Figure(axes=[Axis(label='Date', scale=DateScale(min=datetime.datetime(2007, 1, 1, 0,␣
↪0))), Axis(grid_lines='no...
```

HandDraw

[50]:
```
xs_hd = DateScale(min=np.datetime64(py_dtime(2007, 1, 1)))
ys_hd = LinearScale()
line_hd = Lines(
    x=dates_actual,
    y=sec2_data,
    scales={"x": xs_hd, "y": ys_hd},
    colors=["red"],
)

handdraw = HandDraw(lines=line_hd)
xax = Axis(scale=xs_hd, label="Date", grid_lines="none")
yax = Axis(
    scale=ys_hd, label="Price", orientation="vertical", grid_lines="none"
)

Figure(marks=[line_hd], axes=[xax, yax], interaction=handdraw)
```

```
Figure(axes=[Axis(grid_lines='none', label='Date', scale=DateScale(min=datetime.
↪datetime(2007, 1, 1, 0, 0))), ...
```

`Figure` mit allen Interaktionen

`Figure` definieren:

[51]:
```
dt_x = DateScale(date_format=date_fmt, min=py_dtime(2007, 1, 1))
lc1_x = LinearScale()
lc2_y = LinearScale()

lc2 = Lines(
    x=np.linspace(0.0, 10.0, len(prices)),
    y=prices * 0.25,
    scales={"x": lc1_x, "y": lc2_y},
    display_legend=True,
    labels=["Security 1"],
)

lc3 = Lines(
    x=dates_actual,
    y=sec2_data,
    scales={"x": dt_x, "y": lc2_y},
    colors=["red"],
    display_legend=True,
    labels=["Security 2"],
)

lc4 = Lines(
    x=np.linspace(0.0, 10.0, len(prices)),
    y=sec2_data * 0.75,
    scales={"x": LinearScale(min=5, max=10), "y": lc2_y},
    colors=["green"],
    display_legend=True,
    labels=["Security 2 squared"],
)

x_ax1 = Axis(label="Date", scale=dt_x)
x_ax2 = Axis(label="Time", scale=lc1_x, side="top", grid_lines="none")
x_ay2 = Axis(label=(symbol + " Price"), scale=lc2_y, orientation="vertical")


fig = Figure(marks=[lc2, lc3, lc4], axes=[x_ax1, x_ax2, x_ay2])
```

Interaktionen definieren:

[52]:
```
multi_sel = MultiSelector(scale=dt_x, marks=[lc2, lc3])
br_intsel = BrushIntervalSelector(scale=lc1_x, marks=[lc2, lc3])
index_sel = IndexSelector(scale=dt_x, marks=[lc2, lc3])
int_sel = FastIntervalSelector(scale=dt_x, marks=[lc3, lc2])

hd = HandDraw(lines=lc2)
hd2 = HandDraw(lines=lc3)
pz = PanZoom(scales={"x": [dt_x], "y": [lc2_y]})

deb = HTML()
deb.value = "[]"
```

Call-Back-Handler für die Interaktionen definieren:

[53]:
```
def test_callback(change):
    deb.value = str(change.new)


multi_sel.observe(test_callback, names=["selected"])
br_intsel.observe(test_callback, names=["selected"])
index_sel.observe(test_callback, names=["selected"])
int_sel.observe(test_callback, names=["selected"])
```

[54]:
```
from collections import OrderedDict


selection_interacts = ToggleButtons(
    options=OrderedDict(
        [
            ("HandDraw1", hd),
            ("HandDraw2", hd2),
            ("PanZoom", pz),
            ("FastIntervalSelector", int_sel),
            ("IndexSelector", index_sel),
            ("BrushIntervalSelector", br_intsel),
            ("MultiSelector", multi_sel),
            ("None", None),
        ]
    )
)

link((selection_interacts, "value"), (fig, "interaction"))
VBox([deb, fig, selection_interacts], align_self="stretch")
```

```
VBox(children=(HTML(value='[]'), Figure(axes=[Axis(label='Date',␣
↪scale=DateScale(min=datetime.datetime(2007, 1...
```

Special-Widgets

Traitlet-Widgets

Widget-Properties sind IPython-Traitlets. Um Änderungen vorzunehmen, kann die Methode `observe` des Widgets zum Registrieren eines callback verwendet werden.

Weitere Informationen erhaltet ihr unter Traitlet events.

[1]:
```
import bqplot as bq
import ipywidgets as widgets
import numpy as np


### Create random data
size = 2000  # number of rows
scale = 1.0
scaleLocal = 20
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
np.random.seed(0)
x_data = np.arange(size)
y_data = (
    np.cumsum(np.random.randn(size) * scale)
    + np.random.randn(size) * scaleLocal
)


x_sc = bq.LinearScale()
y_sc = bq.LinearScale()

ax_x = bq.Axis(label="X", scale=x_sc, grid_lines="solid", tick_format="0f")
ax_y = bq.Axis(
    label="Y", scale=y_sc, orientation="vertical", tick_format="0.2f"
)

line1 = bq.Lines(
    x=x_data,
    y=y_data,
    scales={"x": x_sc, "y": y_sc},
    colors=["blue"],
    display_legend=False,
    labels=["y1"],
    stroke_width=1.0,
)

m_fig = dict(left=100, top=50, bottom=50, right=100)
fig = bq.Figure(axes=[ax_x, ax_y], marks=[line1], fig_margin=m_fig)

x_range = widgets.IntRangeSlider(
    value=[min(x_data), max(x_data)],
    min=min(x_data),
    max=max(x_data),
    step=(max(x_data) - min(x_data)) / 100,
    description="X Axis",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
    readout=True,
)


def updateXAxis(change):
    # Update X-axis min/max value here
    if change["type"] == "change" and change["name"] == "value":
        x_sc.min = change["new"][0]
        x_sc.max = change["new"][1]


x_range.observe(updateXAxis)
widgets.VBox([fig, x_range])
```

```
VBox(children=(Figure(axes=[Axis(label='X', scale=LinearScale(), tick_format='0f'),␣
↪Axis(label='Y', orientatio...
```

Linking Widgets

Beim Synchronisieren von Traitlet-Attributen tritt möglicherweise eine Verzögerung aufgrund der Latenz bei der Kommunikation mit dem Server auf. Ihr könnt die Widget-Attribute jedoch auch im Browser entweder uni- oder bidirektional direkt mit den Link-Widgets verknüpfen.

Diese Javascript-Links bleiben auch bestehen, wenn Widgets in HTML-Webseiten ohne Kernel eingebettet werden.

Weitere Informationen erhaltet ihr unter Linking widgets attributes from the client side.

[2]:
```
# Sliders to change minimum and maximum valuese on x scale of bqplot figure using jslink
x_rangeMin = widgets.IntSlider(
    min=min(x_data),
    max=max(x_data),
    step=(max(x_data) - min(x_data)) / 100,
    description="Min x scale: ",
    value=x_sc.min,
)
x_rangeMax = widgets.IntSlider(
    min=min(x_data),
    max=max(x_data),
    step=(max(x_data) - min(x_data)) / 100,
    description="Max x scale: ",
    value=x_sc.max,
)

widgets.jslink((x_sc, "min"), (x_rangeMin, "value"))
widgets.jslink((x_sc, "max"), (x_rangeMax, "value"))

widgets.VBox([fig, x_rangeMin, x_rangeMax])
```

```
VBox(children=(Figure(axes=[Axis(label='X', scale=LinearScale(), side='bottom', tick_
↪format='0f'), Axis(label=...
```

KAPITEL 8

Javascript

Javascript wird von den folgenden Python-Bibliotheken direkt verwendet, also ohne z.B. *D3.js* zu nutzen:

8.1 pythreejs

Eine Python/ThreeJS-Bridge, die die Jupyter-Widget-Infrastruktur verwendet.

8.1.1 Beispiele

Animation

[1]:
```
import ipywidgets

from IPython.display import display
from pythreejs import *
```

[2]:
```
view_width = 600
view_height = 400
```

[3]:
```
sphere = Mesh(
    SphereBufferGeometry(1, 32, 16), MeshStandardMaterial(color="red")
)
```

[4]:
```
cube = Mesh(
    BoxBufferGeometry(1, 1, 1),
    MeshPhysicalMaterial(color="green"),
    position=[2, 0, 4],
)
```

[5]:
```
camera = PerspectiveCamera(
    position=[10, 6, 10], aspect=view_width / view_height
)
key_light = DirectionalLight(position=[0, 10, 10])
ambient_light = AmbientLight()
```

[6]:
```
positon_track = VectorKeyframeTrack(
    name=".position",
    times=[0, 2, 5],
    values=[
        10,
        6,
        10,
        6.3,
        3.78,
        6.3,
        -2.98,
        0.84,
        9.2,
    ],
)
rotation_track = QuaternionKeyframeTrack(
    name=".quaternion",
    times=[0, 2, 5],
    values=[
        -0.184,
        0.375,
        0.0762,
        0.905,
        -0.184,
        0.375,
        0.0762,
        0.905,
        -0.0430,
        -0.156,
        -0.00681,
        0.987,
    ],
)
```

[7]:
```
camera_clip = AnimationClip(tracks=[positon_track, rotation_track])
camera_action = AnimationAction(AnimationMixer(camera), camera_clip, camera)
```

[8]:
```
scene = Scene(children=[sphere, cube, camera, key_light, ambient_light])
controller = OrbitControls(controlling=camera)
renderer = Renderer(
    camera=camera,
    scene=scene,
    controls=[controller],
    width=view_width,
    height=view_height,
)
```

[9]:
```
renderer
```

```
Renderer(camera=PerspectiveCamera(aspect=1.5, position=(10.0, 6.0, 10.0),␣
↪projectionMatrix=(1.0, 0.0, 0.0, 0.0...
```

[10]:
```
camera_action
```

```
AnimationAction(clip=AnimationClip(tracks=(VectorKeyframeTrack(name='.position',␣
↪times=array([0, 2, 5], dtype=...
```

Siehe auch:

- pythreejs

8.2 ipyvolume

[1]:
```
import ipyvolume as ipv
import ipywidgets as widgets
import numpy as np
```

[2]:
```
x, y, z = np.random.random((3, 10000))
ipv.figure()
scatter = ipv.scatter(x, y, z, size=1, marker="sphere")
ipv.show()
```

```
Container(figure=Figure(box_center=[0.5, 0.5, 0.5], box_size=[1.0, 1.0, 1.0],␣
↪camera=PerspectiveCamera(fov=45....
```

[3]:
```
x, y, z, u, v, w = np.random.random((6, 1000)) * 2 - 1
selected = np.random.randint(0, 1000, 100)
fig = ipv.figure()
quiver = ipv.quiver(
    x, y, z, u, v, w, size=5, size_selected=8, selected=selected
)
ipv.show()
```

```
Container(figure=Figure(box_center=[0.5, 0.5, 0.5], box_size=[1.0, 1.0, 1.0],␣
↪camera=PerspectiveCamera(fov=45....
```

[4]:
```
size = widgets.FloatSlider(min=0, max=30, step=0.1)
size_selected = widgets.FloatSlider(min=0, max=30, step=0.1)
color = widgets.ColorPicker()
color_selected = widgets.ColorPicker()
widgets.jslink((quiver, "size"), (size, "value"))
widgets.jslink((quiver, "size_selected"), (size_selected, "value"))
widgets.jslink((quiver, "color"), (color, "value"))
widgets.jslink((quiver, "color_selected"), (color_selected, "value"))
widgets.VBox([size, size_selected, color, color_selected])
```

```
VBox(children=(FloatSlider(value=0.0, max=30.0), FloatSlider(value=0.0, max=30.0),␣
↪ColorPicker(value='black'),...
```

8.2.1 Animations

[5]:
```
# create 2d grids: x, y, and r
u = np.linspace(-10, 10, 25)
x, y = np.meshgrid(u, u)
r = np.sqrt(x**2 + y**2)
print("x,y and z are of shape", x.shape)
# and turn them into 1d
x = x.flatten()
y = y.flatten()
r = r.flatten()
print("and flattened of shape", x.shape)
```

```
x,y and z are of shape (25, 25)
and flattened of shape (625,)
```

[6]:
```
# create a sequence of 15 time elements
time = np.linspace(0, np.pi * 2, 15)
z = np.array([(np.cos(r + t) * np.exp(-r / 5)) for t in time])
print("z is of shape", z.shape)
```

```
z is of shape (15, 625)
```

[7]:
```
# draw the scatter plot, and add controls with animate_glyphs
ipv.figure()
s = ipv.scatter(x, z, y, marker="sphere")
ipv.animation_control(s, interval=200)
ipv.ylim(-3, 3)
ipv.show()
```

```
Container(children=[HBox(children=(Play(value=0, interval=200, max=14),␣
↪IntSlider(value=0, max=14)))], figure=...
```

[8]:
```
# Now also include, color, which containts rgb values
color = np.array([[np.cos(r + t), 1-np.abs(z[i]), 0.1+z[i]*0] for i, t in␣
↪enumerate(time)])
size = (z+1)
print("color is of shape", color.shape)
```

```
color is of shape (15, 3, 625)
```

[9]:
```
color = np.transpose(color, (0, 2, 1))  # flip the last axes
```

[10]:
```
ipv.figure()
s = ipv.scatter(x, z, y, color=color, size=size, marker="sphere")
ipv.animation_control(s, interval=200)
ipv.ylim(-3, 3)
ipv.show()
```

```
Container(children=[HBox(children=(Play(value=0, interval=200, max=14),␣
↪IntSlider(value=0, max=14)))], figure=...
```

Siehe auch:

- ipyvolume

8.3 ipyleaflet

[1]:
```
from ipyleaflet import Map, basemaps
```

[2]:
```
center = [52.5162, 13.3777]
zoom = 6
```

[3]:
```
Map(basemap=basemaps.OpenStreetMap.Mapnik, center=center, zoom=zoom)
```

```
Map(center=[52.5162, 13.3777], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪ 'zoom_in_title', 'zo...
```

[4]:
```
Map(basemap=basemaps.Strava.All, center=center, zoom=zoom)
```

```
Map(center=[52.5162, 13.3777], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪ 'zoom_in_title', 'zo...
```

[5]:
```
from ipyleaflet import WidgetControl
from ipywidgets import IntSlider, jslink


m = Map(basemap=basemaps.OpenStreetMap.Mapnik, center=center)
zoom_slider = IntSlider(description="Zoom level:", min=0, max=19, value=9)
jslink((zoom_slider, "value"), (m, "zoom"))
widget_control1 = WidgetControl(widget=zoom_slider, position="topright")
m.add_control(widget_control1)

m
```

```
Map(center=[52.5162, 13.3777], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪ 'zoom_in_title', 'zo...
```

[6]:
```
from ipywidgets import Label

label = Label()
display(label)

def handle_interaction(**kwargs):
    if kwargs.get('type') == 'mousemove':
        label.value = str(kwargs.get('coordinates'))
m.on_interaction(handle_interaction)

m
```

```
Label(value='')
```

```
Map(bottom=43188.0, center=[52.5162, 13.3777], controls=(ZoomControl(options=['position',
↪ 'zoom_in_text', 'zoo...
```

[7]:
```
import json
import os

import pandas as pd
import requests

from geojson import GeoJSON
from ipyleaflet import LayersControl


if not os.path.exists("europe_110.geo.json"):
    url = "https://github.com/jupyter-widgets/ipyleaflet/raw/master/examples/europe_110.
↪geo.json"
    r = requests.get(url)
    with open("europe_110.geo.json", "w") as f:
        f.write(r.content.decode("utf-8"))
with open("europe_110.geo.json", "r") as f:
    data = json.load(f)

geo_json = GeoJSON(
    data=data,
    style={"opacity": 1, "dashArray": "9", "fillOpacity": 0.1, "weight": 1},
    hover_style={"color": "white", "dashArray": "0", "fillOpacity": 0.5},
    style_callback={"color": "black", "fillColor": "grey"},
)
m.add_layer(geo_json)
m.add_control(LayersControl())

m
```

```
Map(bottom=43188.0, center=[52.5162, 13.3777], controls=(ZoomControl(options=['position',
↪ 'zoom_in_text', 'zoo...
```

[8]:
```
import ipywidgets


ipywidgets.HBox([m, Map(center=[52.5162, 13.3777], zoom=5)])
```

```
HBox(children=(Map(bottom=43188.0, center=[52.5162, 13.3777],␣
↪controls=(ZoomControl(options=['position', 'zoom...
```

8.3.1 WMS-Layer

Mit ipyleaflet lassen sich auch eigene WMS-Layer definieren. Üblicherweise werden Optionen wie `layers`, `format` und `transparent` in der Request-URL übergeben. Wenn ihr zusätzliche Parameter benötigt, könnt ihr diese jedoch auch in eigenen WMSLayer-Klassen definieren. Im folgenden wird beispielsweise ein Zeitparameter hinzugefügt, indem ein benutzerdefinierter `TimeWMSLayer` definiert wird.

Beispiel für einen eigenen WMS-Layer

[9]:
```
from ipyleaflet import Map, WMSLayer, basemaps
from traitlets import Unicode


class TimeWMSLayer(WMSLayer):
    time = Unicode("").tag(sync=True, o=True)


time_wms = TimeWMSLayer(
    url="https://mesonet.agron.iastate.edu/cgi-bin/wms/nexrad/n0r-t.cgi?",
    layers="nexrad-n0r-wmst",
    time="2005-08-29T13:00:00Z",
    format="image/png",
    transparent=True,
    attribution="Weather data © 2012 IEM Nexrad",
)

m = Map(basemap=basemaps.CartoDB.Positron, center=(30.661, -88.645), zoom=5)

m.add_layer(time_wms)

m
```

```
Map(center=[30.661, -88.645], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪'zoom_in_title', 'zoo...
```

Da es sich um ein Widget handelt, könnt ihr den WMS-Parameter `time_wms.time` auch einfach ändern:

[10]:
```
time_wms.time = "2005-08-29T19:00"
```

Oder ihr könnt einen Slider mit passenden Zeitintervallen definieren:

[11]:
```
from ipywidgets import SelectionSlider


time_options = [
    "13:00",
    "13:30",
    "14:00",
    "14:30",
    "15:00",
    "15:30",
    "16:00",
    "16:30",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
]

slider = SelectionSlider(description="Time:", options=time_options)


def update_wms(change):
    time_wms.time = "2005-08-29T{}".format(slider.value)


slider.observe(update_wms, "value")

slider
```

```
SelectionSlider(description='Time:', options=('13:00', '13:30', '14:00', '14:30', '15:00
↪', '15:30', '16:00', '...
```

WMS-Layer-Attribute

Attribute	Standardwert	Kommentar
url	"https://{s}.tile.openstreetmap.org/{z}/{x}/{y}.png"	
min_zoom	0	
max_zoom	18	
tile_size	256	
attributi	"Map data (c) OpenStreetMap contributors"	
detect_re	False	
opacity	1.0	
visible	True	
service	"WMS"	
request	"GetMap"	
layers	""	Kommagetrennte Liste der anzuzeigenden WMS-Ebenen
styles	""	Kommagetrennte Liste der WMS-Stile
format	"image/jpeg"	Für Ebenen mit Transparenz solltet ihr "image/png" verwenden
transpare	False	
version	"1.1.1"	Version des zu verwendenden WMS-Dienstes
crs	""	

8.4 xarray-leaflet

`xarray-leaflet` ist eine xarray-Erweiterung für das Plotten von gekachelten Karten. Sowohl xarray als auch *Leaflet* können mit Datenfragmenten arbeiten, `xarray` durch Dask Chunks und Leaflet durch *map tiles*. Mit `xarray-leaflet` arbeiten beide zusammen.

8.4.1 Installation

Ihr könnt `xarray-leaflet` in eurem Jupyter-Kernel installieren mit:

```
$ pipenv install xarray-leaflet
Installing xarray-leaflet...
...
```

Standardmäßig generiert `xarray-leaflet` Kacheln in temporären Verzeichnissen. Bei dynamischen Karten wird bei jeder Interaktion mit der Karte ein neues Verzeichnis erstellt, entweder durch Ziehen oder Zoomen. Dies liegt daran, dass eine direkte Zuordnung zwischen dem Kachelverzeichnis und der URL besteht, unter der die Kacheln bereitgestellt werden. Da bei dynamischen Karten Kacheln nicht vom Browser zwischengespeichert werden sollten, muss sich die URL ständig ändern. Diese temporären Verzeichnisse werden derzeit nicht automatisch bereinigt. Ihr solltet dies daher regelmäßig selbst tun. In ix-Systemen sind sie unter `/tmp/xarray_leaflet_*`.

8.4.2 Beispiel

Um das Beispiel ausführen zu können, müsst Ihr zusätzlich die folgenden Pakete in eurem Kernel installieren:

- requests
- tqdm
- xarray-leaflet
- dask

[1]:
```
import os
import warnings
import zipfile

import matplotlib.pyplot as plt
import numpy as np
import requests
import rioxarray
import xarray_leaflet

from ipyleaflet import LayersControl, Map, WidgetControl, basemaps
from ipywidgets import FloatSlider
from tqdm import tqdm
```

Wir werden das DEM (Digital Elevation Model) für Europa aus dem HydroSHEDS-Datensatz anzeigen, das das Terrain darstellt. Laden wir zunächst die Daten herunter:

[2]:
```
url = 'https://edcintl.cr.usgs.gov/downloads/sciweb1/shared/hydrosheds/sa_30s_zip_grid/
↪eu_dem_30s_grid.zip'
filename = os.path.basename(url)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
name = filename[:filename.find('_grid')]
adffile = os.path.join(name, name, 'w001001.adf')

if not os.path.exists(adffile):
    r = requests.get(url, stream=True)
    with open(filename, 'wb') as f:
        total_length = int(r.headers.get('content-length'))
        for chunk in tqdm(r.iter_content(chunk_size=1024), total=(total_length/1024) +␣
↪1):
            if chunk:
                f.write(chunk)
                f.flush()
    zip = zipfile.ZipFile(filename)
    zip.extractall('.')
```

Es ist ein Datensatz, den Rasterio öffnen kann, aber um ein schönes `DataArray` mit allen Metadaten zu erhalten, öffnen wir es mit rioxarray:

[3]:
```
da = rioxarray.open_rasterio(adffile, masked=True)
da
```

[3]:
```
<xarray.DataArray (band: 1, y: 5760, x: 9480)>
[54604800 values with dtype=float32]
Coordinates:
  * band         (band) int64 1
  * x            (x) float64 -14.0 -13.99 -13.98 -13.97 ... 64.98 64.99 65.0
  * y            (y) float64 60.0 59.99 59.98 59.97 ... 12.03 12.02 12.01 12.0
    spatial_ref  int64 0
Attributes:
    scale_factor:  1.0
    add_offset:    0.0
```

Die Projektion ist `EPSG:4326` (auch bekannt als `WGS84`). Dabei entspricht die Koordinate `x` den Längengraden und `y` den Breitengraden (in Grad). Es gibt nur ein Band.

[4]:
```
da = da.sel(band=1)
da.name = "DEM"
```

Der Datensatz kann zu groß sein, um ihn im Speicher zu halten, daher werden wir ihn in kleinere Teile zerlegen. Dies wird auch die Leistung verbessern, da die Erzeugung einer Kachel mit Dask parallel erfolgen kann.

[5]:
```
da = da.chunk((1000, 1000))
warnings.filterwarnings("ignore")
```

Wir müssen lediglich eine Karte erstellen, bevor wir sie an unsere DataArray-Erweiterung übergeben.

[6]:
```
m = Map(
    center=[52.5162, 13.3777],
    zoom=3,
    basemap=basemaps.CartoDB.Positron,
    interpolation="nearest",
)
m
```

```
Map(center=[52.5162, 13.3777], controls=(ZoomControl(options=['position', 'zoom_in_text',
↪ 'zoom_in_title', 'zo...
```

Um unsere Daten auf der Karte darzustellen, rufen wir `leaflet.plot()` mit unserem `DataArray` auf und übergeben die Karte als Parameter. Wir erhalten eine Ebene zurück, die wir z.B. mit einem Schieberegler zur Einstellung der Deckkraft weiter steuern können.

[7]:
```
l = da.leaflet.plot(m, colormap=plt.cm.terrain)
l.interact(opacity=(0., 1.))
```

```
Url()
```

```
Box(children=(FloatSlider(value=1.0, description='opacity', max=1.0),))
```

Nun fügen wir den Schieberegler in die Karte ein. Außerdem werden wir ein Ebenensteuerelement einfügen.

[8]:
```
layers_control = LayersControl(position="topright")
m.add_control(layers_control)

opacity_slider = FloatSlider(description="Opacity:", min=0, max=1, value=1)


def set_opacity(change):
    l.opacity = change["new"]


opacity_slider.observe(set_opacity, names="value")
slider_control = WidgetControl(widget=opacity_slider, position="bottomleft")
m.add_control(slider_control)
```

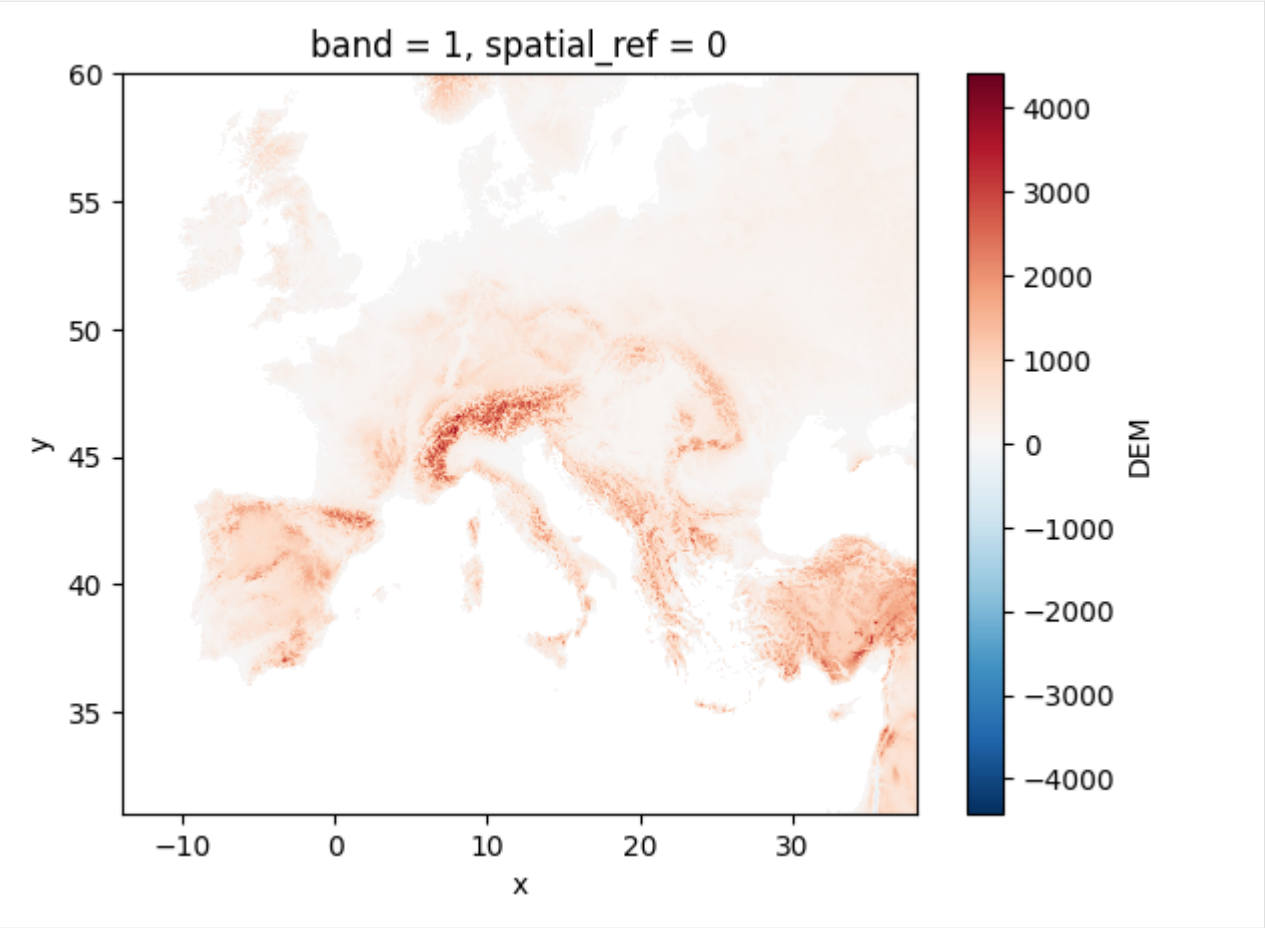
Die `select()`-Methode ermöglicht es, eine Region durch Klicken und Ziehen eines Kästchens auf der Karte auszuwählen Zuvür müsst ihr jedoch auf den -Button klicken.

[9]:
```
da.leaflet.select()
```

Ihr könnt dann das ausgewählte Datenfeld zurückerhalten und es z.B. plotten.

[10]:
```
da_selected = da.leaflet.get_selection()
```

[11]:
```
if da_selected is not None:
    da_selected.plot.imshow()
```



Schließlich können die Auswahl und die Buttons entfernt werden:

[12]:
```
da.leaflet.unselect()
```