Jupyter Tutorial

Veit Schiele

15.11.2026

Inhaltsverzeichnis

1 Einführung **5**
- 1.1 Zielgruppe . . . 5
- 1.2 Warum Jupyter? . . . 5
- 1.3 Jupyter-Infrastruktur . . . 6
- 1.4 Arbeitsbereich . . . 6

2 Was ist neu? **7**

3 Notebook **9**
- 3.1 Jupyter Notebook installieren . . . 9
- 3.2 Notebook erstellen . . . 10
- 3.3 Tastaturkürzel . . . 13
- 3.4 Jupyter-Pfade und -Konfiguration . . . 14
- 3.5 Parametrisierung und Zeitplanung . . . 15
- 3.6 Testen . . . 18

4 JupyterLab **29**
- 4.1 JupyterLab installieren . . . 29
- 4.2 JupyterLab-Erweiterungen . . . 30
- 4.3 JupyterLab auf JupyterHub . . . 32
- 4.4 Real-Time Collaboration . . . 32
- 4.5 Scheduler . . . 33

5 JupyterHub **35**
- 5.1 Installation . . . 35
- 5.2 Konfiguration . . . 36
- 5.3 systemdspawner . . . 39
- 5.4 Service `nbviewer` erstellen . . . 42
- 5.5 `ipyparallel` . . . 42

6 Binder **61**

7 `nbconvert` **63**
- 7.1 Installation . . . 63
- 7.2 Verwenden auf der Kommandozeile . . . 64
- 7.3 nb2xls . . . 65
- 7.4 Eigene Exporter . . . 65

8 nbviewer **67**
8.1 Installation . . . 67
8.2 Erweitern des Notebook-Viewers . . . 67
8.3 Zugriffsbeschränkung . . . 68

9 Kernels **69**
9.1 Kernel installieren, anzeigen und starten . . . 69
9.2 Was ist neu in Python 3.8? . . . 72
9.3 Was ist neu in Python 3.9? . . . 75
9.4 Was ist neu in Python 3.10 . . . 79
9.5 R-Kernel . . . 81

10 ipywidgets **83**
10.1 Beispiele . . . 83
10.2 Widget-Liste . . . 84
10.3 Widget Events . . . 95
10.4 Benutzerdefiniertes Widget . . . 98
10.5 ipywidgets-Bibliotheken . . . 101
10.6 Einbetten von Jupyter-Widgets . . . 126

11 nbextensions **129**
11.1 Installation . . . 129
11.2 Liste der Erweiterungen . . . 131
11.3 Plugin erstellen . . . 134
11.4 setup.ipynb . . . 137
11.5 ipylayout . . . 138

12 Daten visualisieren **141**

13 Dashboards **143**
13.1 Voilà vs. Panel . . . 144
13.2 Voilà . . . 145
13.3 Panel . . . 159

14 Sphinx **209**
14.1 nbsphinx . . . 209
14.2 Executable Books . . . 214

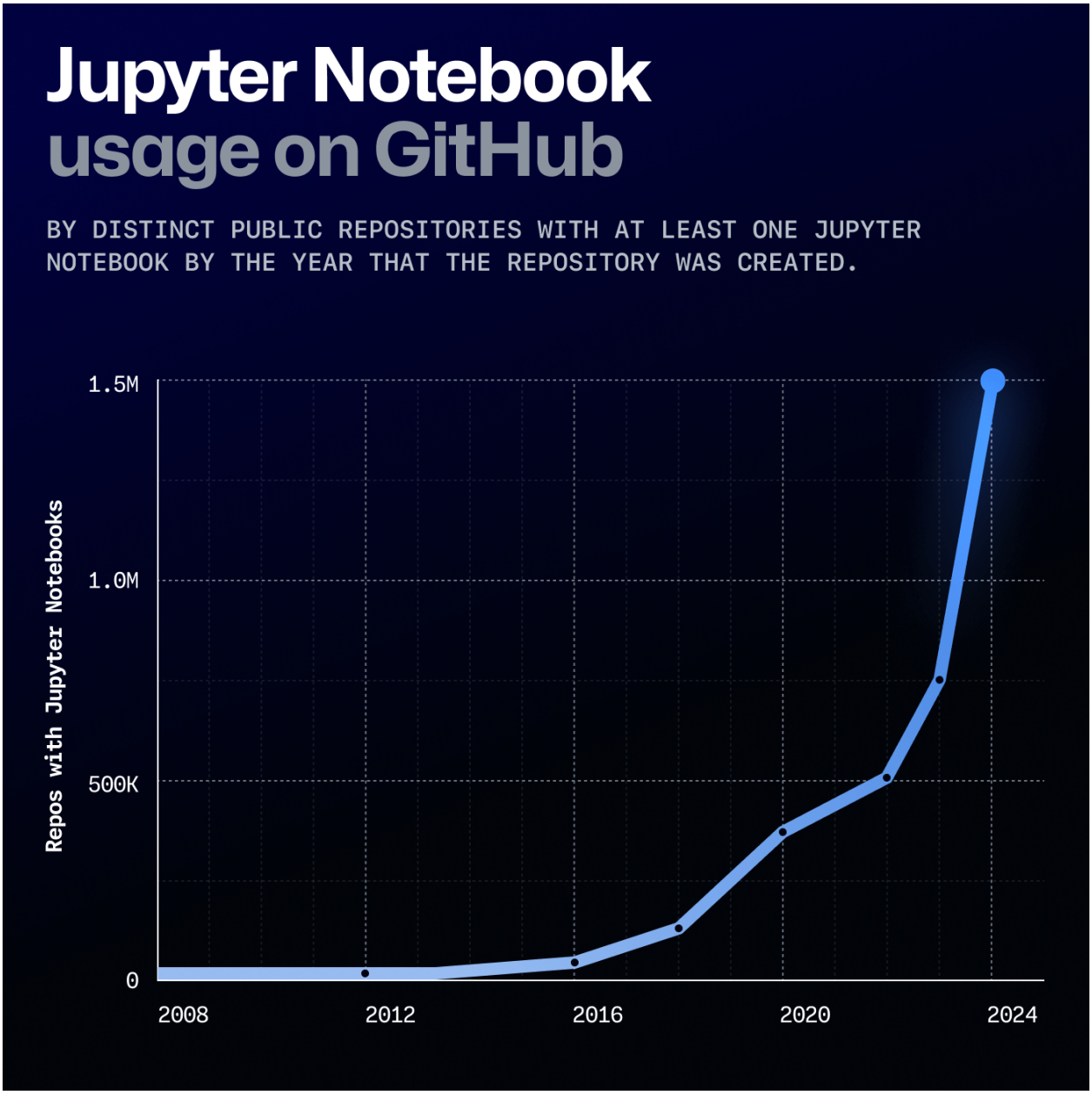
15 Anwendungsbeispiele **217**

16 Stichwortverzeichnis **219**

Stichwortverzeichnis **221**

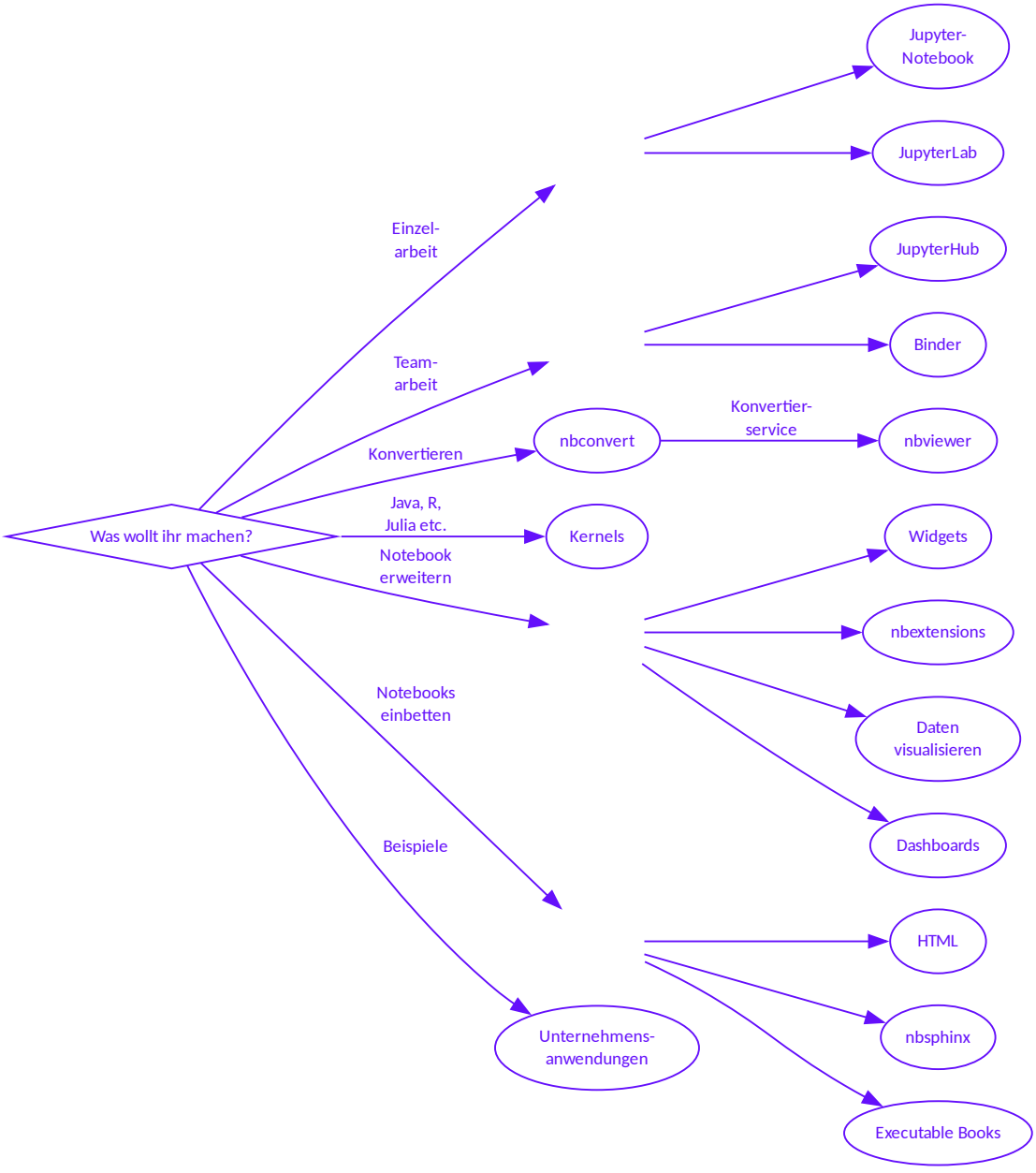
Jupyter-Notebooks erfreuen sich in den Datenwissenschaften wachsender Beliebtheit und wurden zum De-facto-Standard für schnelles Prototyping und explorative Analysen. Sie beflügeln nicht nur Experimente und Innovationen enorm, sie machen auch den gesamten Forschungsprozess schneller und zuverlässiger.

> *„Der sprunghafte Anstieg der Nutzung von Jupyter Notebooks zeigt, dass Open Source eine wachsende Gemeinschaft unterstreicht, insbesondere da Python zur meistgenutzten Sprache … aufsteigt. Seit 2018 hat die Nutzung von Jupyter Notebooks stetig zugenommen – und dieses Wachstum stieg im Jahr 2022 sprunghaft an, als die Forschung und das Experimentieren mit generativer KI und maschinellem Lernen Fahrt aufnahmen. Seit 2022 ist die Nutzung von Jupyter Notebooks auf GitHub um mehr als 170 % angestiegen. Und seit dem letzten Jahr ist die Nutzung um 92 % gestiegen. Datenwissenschaftler*innen und Forschende im Bereich maschinelles Lernen nutzen die Open-Source-Anwendung häufig für maschinelles Lernen, Datenvisualisierung und mehr.“*



– Octoverse: AI leads Python to top language as the number of global developers surges

Zudem entstehen viele zusätzliche Komponenten, die die ursprünglichen Grenzen ihrer Nutzung erweitern und neue Verwendungsmöglichkeiten eröffnen.



Das Jupyter-Tutorial ist jedoch nur ein Teil einer Reihe von Tutorials zur Datenanalyse und -visualisierung:

- Python Basics
- Python für Data Science
- PyViz Tutorial
- cusy Design-System: Datenvisualisierung

Alle Tutorials dienen als Seminarunterlagen für unsere aufeinander abgestimmten Trainings:

Dauer	Titel
3 Tage	Einführung in Python
2 Tage	Fortgeschrittenes Python
2 Tage	Entwurfsmuster in Python
2 Tage	Effizient Testen mit Python
1 Tag	Software-Dokumentation mit Sphinx
2 Tage	Technisches Schreiben
3 Tage	Jupyter-Notebooks für effiziente Data-Science-Workflows
2 Tage	Numerische Berechnungen mit NumPy
2 Tage	Daten analysieren mit pandas
3 Tage	Daten lesen, schreiben und bereitstellen mit Python
2 Tage	Daten bereinigen und validieren mit Python
5 Tage	Daten visualisieren mit Python
1 Tag	Datenvisualisierungen gestalten
2 Tage	Dashboards erstellen
3 Tage	Code und Daten versioniert und reproduzierbar speichern
Abonnement à 2 h im Quartal	Neues aus Python für Data-Science

KAPITEL 1

Einführung

1.1 Zielgruppe

Die Nutzung von Jupyter-Notebooks ist vielfältig und reicht von den Datenwissenschaften über Data-Engineering und Datenanalyse bis hin zu System-Engineering. Dabei sind die Fähigkeiten und Arbeitsabläufe der einzelnen Zielgruppen sehr unterschiedlich. Eine der großen Stärken von Jupyter-Notebooks ist jedoch, dass sie eine enge Zusammenarbeit dieser unterschiedlichen Fachgruppen in funktionsübergreifenden Teams ermöglichen.

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

Data-Engineers
: prüfen die Qualität des Codes und machen ihn robuster, effizienter und skalierbar.

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

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

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

1.2 Warum Jupyter?

Wie können nun diese vielfältigen Aufgaben vereinfacht werden? Es wird sich kaum ein Werkzeug finden, das all diese Aufgaben abdeckt und selbst für einzelne Aufgaben sind häufig mehrere Werkzeuge notwendig. Daher suchen wir auf einer abstrakteren Ebene allgemeinere Muster für Tools und Sprachen, mit denen Daten analysiert und visualisiert sowie ein Projekt dokumentiert und präsentiert werden kann. Genau dies streben wir mit dem Project Jupyter an.

Das Projekt Jupyter startete 2014 mit dem Ziel, ein konsistentes Set von Open-Source-Tools für wissenschaftliche Forschung, reproduzierbare Workflows, Computational Narratives und Datenanalyse zu erstellen. Bereits 2017 wurde

Jupyter dann mit dem ACM Software Systems Award ausgezeichnet - eine prestigeträchtige Auszeichnung, die es u.a. mit Unix und dem Web teilt.

Um zu verstehen, warum Jupyter-Notebooks so erfolgreich sind, schauen wir uns die Kernfunktionen einmal genauer an:

Jupyter Notebook Format
: Jupyter Notebooks sind ein offenes, auf JSON basierendes Dokumentenformat mit vollständigen Aufzeichnungen der Sitzungen des Benutzers und des enthaltenen Codes.

Interactive Computing Protocol
: Das Notebook kommuniziert mit einem Rechenkernel über das *Interactive Computing Protocol*, einem offenen Netzwerkprotokoll basierend auf JSON-Daten über ZMQ und WebSockets.

Kernels
: Rechenkernel sind Prozesse, die interaktiven Code in einer bestimmten Programmiersprache ausführen und die Ausgabe an den Benutzer zurückgeben.

Siehe auch

- Jupyter celebrates 20 years

1.3 Jupyter-Infrastruktur

Eine Plattform für die oben genannten Use Cases erfordert eine umfangreiche Infrastruktur, die nicht nur die Bereitstellung der Kernel sowie die Parametrisierung, Zeitsteuerung und Parallelisierung von Notebooks erlaubt, sondern darüberhinaus auch die gleichmäßige Bereitstellung der Ressourcen.

Mit diesem Tutorial wird eine Plattform bereitgestellt, die über Jupyter Notebooks hinaus schnelle, flexible und umfassende Datenanalysen ermöglicht. Aktuell gehen wir jedoch noch nicht darauf ein, wie sie sich um *Streaming Pipelines* und *Domain Driven Data Stores* erweitern lässt.

Die Beispiele des Jupyter-Tutorials könnt ihr jedoch auch lokal erstellen und ausführen.

1.4 Arbeitsbereich

Die Einrichtung des Arbeitsbereichs umfasst die Installation und Konfiguration von IPython und *Jupyter-Notebooks*, *nbextensions* und *ipywidgets*.

KAPITEL 2

Was ist neu?

24.1.0

- Add matplotlib for social cards
- Use git tag for versioning the docs
- Switch voila example to bqplot vueitfy
- Switch to panel sampledata
- Add sphinx-lint
- Add more alert boxes
- Remove node env
- Remove nbviewer env
- Remove qgrid as it is not being developed further
- Update MacTex install
- Add JupyterHub env
- Add Python 3.11 kernel config

1.1.1

- Jupyter-Tutorial 1.1.1
- Fix PDF structure

1.1.0

- Jupyter-Tutorial 1.1.0
- Add ‘What’s new’ section
- Add Executable Books
- Beautify the Jupyter overview
- Add JupyterLab documentation

1.0.0

- Moving the Data Science content into Python4DataScience
  - /first-steps/index.html -> /notebook/index.html
  - /first-steps/create-notebook.html -> /notebook/create-notebook.html
  - /first-steps/install.html -> /notebook/install.html
  - /workspace/jupyter/$rest -> /
  - /workspace/first-steps/$rest -> /notebook/
  - /workspace/ipython/$rest -> Python4DataScience:/workspace/ipython/
  - /workspace/numpy/$rest -> Python4DataScience:/workspace/numpy/
  - /workspace/pandas/$rest -> Python4DataScience:/workspace/pandas/
  - /data-processing/$rest -> Python4DataScience:/data-processing/
  - /clean-prep/$rest -> Python4DataScience:/clean-prep/
  - /parameterise/$rest -> /notebook/parameterise/
  - /performance/ipyparallel//$rest -> /hub/ipyparallel/
  - /performance/ -> Python4DataScience:/performance/
  - /productive/ -> Python4DataScience:/productive/
  - /testing/$rest -> /notebook/testing/
  - /web/dashboards/$rest -> /dashboards/

KAPITEL 3

Notebook

Jupyter Notebooks erweitern den konsolenbasierten Ansatz zum interaktiven Rechnen um eine webbasierte Anwendung, mit der der gesamte Prozess aufgezeichnet werden kann: von der Entwicklung und Ausführung des Codes bis zur Dokumentation und Präsentation der Ergebnisse.

3.1 Jupyter Notebook installieren

3.1.1 Erstellen einer virtuellen Umgebung mit `jupyter`

Virtuelle Python-Umgebungen ermöglichen die Installation von Python-Paketen an einem isolierten Ort für eine bestimmte Anwendung, anstatt sie global zu installieren. So habt ihr eure eigenen Installationsverzeichnisse und teilt keine Bibliotheken mit anderen virtuellen Umgebungen:

```
$ python3 -m venv myproject
$ cd myproject
$ . bin/activate
$ python -m pip install jupyter
```

3.1.2 Jupyter-Notebook starten

```
$ jupyter notebook
...
[I 12:46:53.852 NotebookApp] The Jupyter Notebook is running at:
[I 12:46:53.852 NotebookApp] http://localhost:8888/?
↪token=53abd45a3002329de77f66886e4ca02539d664c2f5e6072e
[I 12:46:53.852 NotebookApp] Use Control-C to stop this server and shut down all kernels␣
↪(twice to skip confirmation).
[C 12:46:53.858 NotebookApp]
    To access the notebook, open this file in a browser:
        file:///Users/veit/Library/Jupyter/runtime/nbserver-7372-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/?token=53abd45a3002329de77f66886e4ca02539d664c2f5e6072e
```

Euer Standard-Webbrowser wird dann mit dieser URL geöffnet.

Wenn das Notebook in eurem Browser geöffnet wird, wird das Notebook-Dashboard mit einer Liste der Notebooks, Dateien und Unterverzeichnisse in dem Verzeichnis angezeigt, in dem der Notebook-Server gestartet wurde. In den meisten Fällen möchtet ihr einen Notebook-Server in eurem Projektverzeichnis starten.



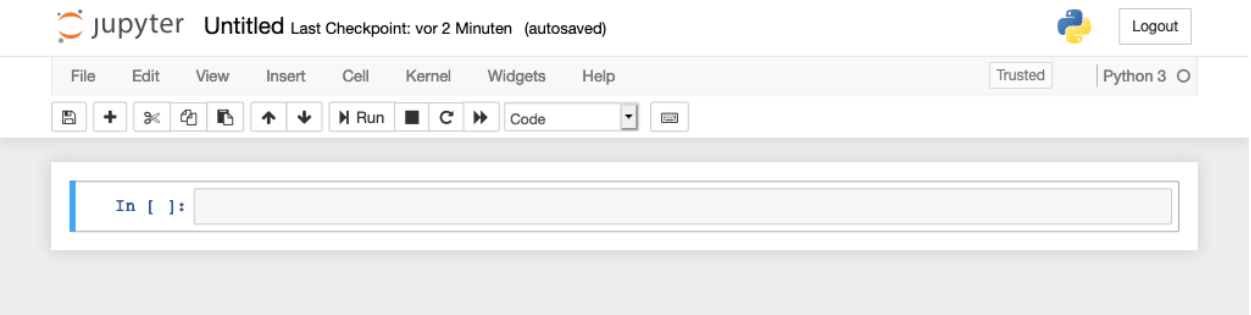
3.2 Notebook erstellen

Nachdem der Notebook-Server gestartet wurde, können wir unser erstes Notebook erstellen.

3.2.1 Erstellen eines Notebooks

In eurem Standard-Browser solltet ihr das Notebook-Dashboard mit dem Menü *New* auf der rechten Seite sehen. In diesem Menü werden alle Notebook-Kernel aufgeführt, initial jedoch vermutlich nur *Python 3*.

Nachdem ihr *New* → *Python 3* ausgewählt habt, wird ein neues Notebook `Untitled.ipynb` erstellt und in einem neuen Reiter angezeigt:



3.2.2 Umbenennen des Notebooks

Als nächstes solltet ihr dieses Notebook umbenennen indem ihr auf den Titel *Untitled* klickt:



3.2.3 Die Notebook-Oberfläche

Es gibt zwei wichtige Begriffe, um Jupyter Notebooks zu beschreiben: *Zelle* und *Kernel*:

Notebook-Kernel
: *Rechenmaschine*, die den in einem Notebook enthaltenen Code ausführt.

Notebook-Zelle
: Container für Text, der in einem Notebook angezeigt werden soll oder für Code, der vom Kernel des Notebooks ausgeführt werden soll.

  Code
  : enthält Code, der im Kernel ausgeführt werden soll und dessen Ausgabe unterhalb angezeigt wird.

    Vor den Code-Zellen sind eckige Klammern, die die Reihenfolge anzeigen, in der der Code ausgeführt wurde.

    `In [ ]:`
    : zeigt an, dass der Code noch nicht ausgeführt wurde.

    `In [*]:`
    : zeigt an, dass die Ausführung noch nicht abgeschlossen ist.

> **Warnung**
>
> Der Output von Zellen kann später in anderen Zellen verwendet werden. Daher ist das Ergebnis von der Reihenfolge abhängig. Wenn ihr eine andere Reihenfolge als die von oben nach unten wählt, erhaltet ihr später möglicherweise andere Ergebnisse, wenn ihr z.B. *Cell* → *Run All* wählt.

  Markdown
  : enthält mit Markdown formatierten Text, der interpretiert wird sobald *Run* gedrückt wird.

3.2.4 Was ist eine `ipynb`-Datei?

Diese Datei beschreibt ein Notebook im JSON-Format. Jede Zelle und ihr Inhalt einschließlich Bildern werden dort zusammen mit einigen Metadaten aufgelistet. Ihr könnt euch diese anschauen wenn ihr im Dashboard das Notebook auswählt und dann auf *edit* klickt. So sieht z.B. (zum Beispiel) die JSON-Datei für `my-first-notebook.ipynb` folgendermaßen aus:

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {},
   "source": [
    "# My first notebook"
   ]
  },
  {
   "cell_type": "code",
   "execution_count": 1,
   "metadata": {},
   "outputs": [
    {
     "name": "stdout",
     "output_type": "stream",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)


```
     "text": [
      "Hello World!\n"
     ]
    }
   ],
   "source": [
    "print('Hello World!')"
   ]
  }
 ],
 "metadata": {
  "kernelspec": {
   "display_name": "Python 3",
   "language": "python",
   "name": "python3"
  },
  "language_info": {
   "codemirror_mode": {
    "name": "ipython",
    "version": 3
   },
   "file_extension": ".py",
   "mimetype": "text/x-python",
   "name": "python",
   "nbconvert_exporter": "python",
   "pygments_lexer": "ipython3",
   "version": "3.7.0"
  }
 },
 "nbformat": 4,
 "nbformat_minor": 2
}
```


3.2.5 Speichern und Checkpoints

Beim Klick auf *Save and Checkpoint* wird eure `ipynb`-Datei gespeichert. Aber was hat es mit dem *Checkpoint* auf sich?

Jedesmal, wenn ihr ein neues Notebook anlegt, wird auch eine Datei angelegt, die üblicherweise alle 120 Sekunden automatisch eure Änderungen speichert. Dieser Checkpoint findet sich üblicherweise in einem versteckten Verzeichnis namens `.ipynb_checkpoints/`. Diese Checkpoint-Datei ermöglicht euch daher, eure nicht gespeicherten Daten im Falle eines unerwarteten Problems wiederherzustellen. Ihr könnt in *File* → *Revert to Checkpoint* zu einer der letzten Checkpoints zurückgehen.

3.2.6 Tipps & Tricks

1. Gebt dem Notebook einen Titel (`# MY TITLE`) und ein aussagekräftiges Vorwort, um den Inhalt und Zweck des Notebooks zu beschreiben.
2. Erstellt Überschriften und Dokumentationen in Markdown-Zellen, um euer Notebook zu strukturieren und eure Workflow-Schritte zu erläutern. Dabei ist es vollkommen egal, ob ihr das für eure Kollegen oder für euch zukünftig selbst macht.
3. Verwendet *Table of Contents (2)* aus der *Liste der Erweiterungen*, um ein Inhaltsverzeichnis zu erstellen.

4. Verwendet die Notebook-Erweiterung *setup*.
5. Verwendet *Snippets* aus der *Liste der Erweiterungen*, um weitere, häufig benötigte Code-Blöcke, z.B. typische Importanweisungen, bequem einfügen zu können.

3.3 Tastaturkürzel

Wenn ihr die Jupyter-Tastaturkürzel kennt, könnt ihr viel effizienter mit Notebooks arbeiten. Jupyter-Notebooks haben zwei verschiedene Tastatureingabemodi:

- Im **Bearbeitungsmodus** könnt ihr Code oder Text in eine Zelle eingeben. Dies wird durch einen grünen Zellrand angezeigt.
- Der **Befehlsmodus** bindet die Tastatur an Befehle auf Notebook-Ebene und wird durch einen grauen Zellrand mit einem blauen linken Rand angezeigt.

Befehlsmodus	
Dieser Modus ist mit verfügbar.	
f	Suchen und Ersetzen
	Bearbeitungsmodus aufrufen
--f, --p, p	die Befehlspalette öffnen
-	Zelle ausführen, unten auswählen
-, -	Ausgewählte Zellen ausführen
-	Zelle ausführen und unten einfügen
y	Zelle in Code ändern
m	Zelle in Markdown ändern
r	Zelle in Rohdaten ändern
1, 2 etc.	Zelle in Überschrift 1, Überschrift2, etc. ändern
k, ↑	Zelle oben auswählen
j, ↓	Zelle unten auswählen
-k, -↑	markierte Zellen oben erweitern
-j, -↓	markierte Zellen nach unten erweitern
-a	alle Zellen auswählen
a	Zelle oben einfügen
b	Zelle unten einfügen
x	ausgewählte Zellen ausschneiden
c	markierte Zellen kopieren
-v	Zellen oben einfügen
v	Zellen unten einfügen
z	Löschen von Zellen rückgängig machen
d d	Ausgewählte Zellen löschen
-m	Rmarkierte Zellen zusammenführen, oder aktuelle Zelle mit darunter liegen
-s, s	Speichern und Checkpoint
l	Zeilennummern ein- und ausschalten
o	Ausgabe der ausgewählten Zellen umschalten
-o	Blättern in der Ausgabe ausgewählter Zellen ein- und ausschalten
h	Tastaturkürzel anzeigen
i i	den Kernel unterbrechen
0 0	den Kernel neu starten (mit Dialog)
-v	Dialog zum Einfügen aus der Systemzwischenablage
, q	den Pager schließen

Bearbeitungsmodus	
Dieser Modus ist mit verfügbar.	
	Code-Vervollständigung oder Einrückung
-	Tooltip
-]	Einrücken
-[	Ausrücken
-a	alles markieren
-z	rückgängig machen
-/	kommentieren
-d	ganze Zeile löschen
-u	Auswahl rückgängig machen
	Überschreibmarkierung umschalten
-↑	zum Zellenanfang gehen
-↓	Gehe zum Zellenende
-←	Gehe ein Wort nach links
-→	gehe ein Wort nach rechts
-	lösche ein Wort vorher
-	lösche ein Wort nachher
--z	wiederholen
--u	Auswahl wiederholen
-k	Zeilen löschen im Emacs-Stil
-	Zeile links vom Cursor löschen
-	Zeile rechts vom Cursor löschen
-m,	Kommandomodus aufrufen
--f, --p	Öffnen der Befehlspalette
-	Zelle ausführen, unten auswählen
- -	Ausgewählte Zellen ausführen
-	Zelle ausführen und unten einfügen
---	Zelle an der/den Schreibmarke(n) teilen
-s	Speichern und Checkpoint
↓	Cursor nach unten bewegen
↑	Cursor nach oben bewegen

3.3.1 Eigene Tastaturkürzel

Ihr könnt auch eure eigenen Tastenkombinationen unter *Help* → *Edit Keyboard Shortcuts* definieren.

Siehe auch

- Keyboard Shortcut Customization

3.4 Jupyter-Pfade und -Konfiguration

Konfigurationsdateien werden üblicherweise im `~/.jupyter`-Verzeichnis gespeichert. Mit der Umgebungsvariablen `JUPYTER_CONFIG_DIR` kann jedoch auch ein anderes Verzeichnis festgelegt werden. Falls Jupyter im `JUPYTER_CONFIG_DIR` keine Konfiguration findet, durchläuft Jupyter den Suchpfad mit */SYS.PREFIX/etc/jupyter/* und anschließend für Unix `/usr/local/etc/jupyter/` und `/etc/jupyter/`, für Windows `%PROGRAMDATA%\jupyter\`.

Ihr könnt Euch die aktuell verwendeten Konfigurationsverzeichnisse aufzulisten lassen mit:

```
$ jupyter --paths
config:
    /Users/veit/.jupyter
    /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/bin/../etc/jupyter
    /usr/local/etc/jupyter
    /etc/jupyter
...
```

3.4.1 Erstellen der Konfigurationsdateien

Ihr könnt eine Standardkonfiguration erstellen mit:

```
$ jupyter notebook --generate-config
Writing default config to: /Users/veit/.jupyter/jupyter_notebook_config.py
```

Allgemeiner lassen sich Konfigurationsdateien für alle Jupyter-Applikationen anlegen mit `jupyter` *`APPLICATION`* `--generate-config`.

3.4.2 Ändern der Konfiguration

… durch Bearbeiten der Konfigurationsdatei

z.B. in `jupyter_notebook_config.py`:

```
c.NotebookApp.port = 8754
```

Sofern die Werte als `list`, `dict` oder `set` gespeichert werden, können diese auch ergänzt werden mit `append`, `extend`, `prepend`, `add` und `update`, z.B.:

```
c.TemplateExporter.template_path.append("./templates")
```

… mit der Befehlszeile

z.B.:

```
$ jupyter notebook --NotebookApp.port=8754
```

Dabei gibt es für häufig verwendete Optionen Aliase wie z.B. für `--port` oder `--no-browser`.

Die Befehlszeilenoptionen überschreiben die in einer Konfigurationsdatei festgelegten Optionen.

Siehe auch

traitlets.config

3.5 Parametrisierung und Zeitplanung

Mit *JupyterLab* könnt ihr den *Jupyter Scheduler* zum Parametrisieren und zeitgesteuertem Ausführen verwenden. Für Jupyter Notebooks steht euch papermill zur Verfügung.

3.5.1 Installieren

```
$ uv add papermill
```

3.5.2 Verwenden

1. Parametrisieren

   Der erste Schritt ist die Parametrisierung des Notebook. Dazu werden die Zellen in *View* → *Cell Toolbar* → *Tags* als Parameter markiert.

2. Überprüfen

   Ihr könnt das Notebook inspizieren, z.B. mit

   ```
   $ uv run papermill --help-notebook docs/refactoring/parameterise/input.ipynb
   Usage: papermill [OPTIONS] NOTEBOOK_PATH [OUTPUT_PATH]

   Parameters inferred for notebook 'docs/refactoring/parameterise/input.ipynb':
     msg: Unknown type (default None)
   ```

3. Ausführen

   Es gibt zwei Möglichkeiten, ein Notebook mit Parametern auszuführen:

   - … via Python API

     Die Funktion `execute_notebook()` kann aufgerufen werden, um ein Notebook mit einem Dict von Parametern auszuführen:

     ```
     execute_notebook(INPUT_NOTEBOOK, OUTPUT_NOTEBOOK, DICTIONARY_OF_PARAMETERS)
     ```

     z.B. für `input.ipynb`:

     ```
     In [1]: import papermill as pm

     In [2]: pm.execute_notebook(
                 "PATH/TO/INPUT_NOTEBOOK.ipynb",
                 "PATH/TO/OUTPUT_NOTEBOOK.ipynb",
                 parameters=dict(salutation="Hello", name="pythonistas"),
             )
     ```

     Das Ergebnis ist `output.ipynb`:

     ```
     In [1]: salutation = None
             name = None

     In [2]: # Parameters
             salutation = "Hello"
             name = "pythonistas"

     In [3]: from datetime import date


             today = date.today()
             print(
                 salutation,
     ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            name,
            "- welcome to our event on this " + today.strftime("%A, %d %B %Y"),
        )

Out[3]: Hello pythonistas - welcome to our event on this Monday, 26 June 2023
```

```
import papermill as pm

pm.execute_notebook(
    "PATH/TO/INPUT_NOTEBOOK.ipynb",
    "PATH/TO/OUTPUT_NOTEBOOK.ipynb",
    parameters=dict(salutation="Hello", name="pythonistas"),
)
```

Siehe auch

– Workflow reference

- … via CLI

```
$ uv run papermill input.ipynb output.ipynb -p salutation 'Hello' -p name
↪'pythonistas'
```

Alternativ kann auch eine YAML-Datei mit den Parametern angegeben werden, z.B. `params.yaml`:

Quellcode 1: params.yaml

```
salutation: "Hello"
name: "Pythonistas"
```

```
$ uv run papermill input.ipynb output.ipynb -f params.yaml
```

Mit `-b` kann ein base64-kodierte YAML-String angegeben werden, die die Parameterwerte enthält:

```
$ uv run papermill input.ipynb output.ipynb -b␣
↪c2FsdXRhdGlvbjogIkhlbGxvIgpuYW1lOiAiUHl0aG9uaXN0YXMi
```

Siehe auch

– CLI reference

Ihr könnt dem Dateinamen auch einen Zeitstempel hinzufügen:

```
$ dt=$(date '+%Y-%m-%d_%H:%M:%S')
$ uv run papermill input.ipynb output_$(date '+%Y-%m-%d_%H:%M:%S').ipynb -f␣
↪params.yaml
```

Dies erzeugt eine Ausgabedatei, deren Dateiname einen Zeitstempel enthält, z.B. `output_2023-06-26_15:57:33.ipynb`.

Schließlich könnt ihr `crontab -e` verwenden, um die beiden Befehle automatisch zu bestimmten Zeiten auszuführen, z.B. am ersten Tag eines jeden Monats:

```
dt=$(date '+%Y-%m-%d_%H:%M:%S')
0 0 1 * * cd ~/jupyter-notebook && uv run papermill input.ipynb output_$(date '+
↪%Y-%m-%d_%H:%M:%S').ipynb -f params.yaml
```

4. Speichern

   Papermill kann Notebooks an einer Reihe von Orten speichern, einschließlich S3, Azure Data Blobs und Azure Data Lakes. Papermill erlaubt auch, neue Datenspeicher hinzuzufügen.

   Siehe auch

   - papermill Storage
   - Extending papermill through entry points

3.6 Testen

3.6.1 Konzepte

Test Case (Testfall)
: testet eine einzelnes Szenario.

Test Fixture (Prüfvorrichtung)
: ist eine konsistente Testumgebung.

Siehe auch

- pytest fixtures

Test Suite
: ist eine Sammlung mehrerer Test Cases.

Test Runner
: durchläuft eine Test Suite und stellt die Ergebnisse dar.

3.6.2 Notebooks

Unittests

[1]:
```
def add(a, b):
    return a + b
```

[2]:
```
import unittest


class TestNotebook(unittest.TestCase):
    def test_add(self):
        self.assertEqual(add(2, 2), 5)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
unittest.main(argv=[""], verbosity=2, exit=False)
```

```
test_add (__main__.TestNotebook.test_add) ... FAIL

======================================================================
FAIL: test_add (__main__.TestNotebook.test_add)
----------------------------------------------------------------------
Traceback (most recent call last):
  File "/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_33771/2216555184.py",␣
↪line 6, in test_add
    self.assertEqual(add(2, 2), 5)
AssertionError: 4 != 5

----------------------------------------------------------------------
Ran 1 test in 0.001s

FAILED (failures=1)
```

[2]:
```
<unittest.main.TestProgram at 0x10323bdd0>
```

Alternativ kann auch ipython-unittest verwendet werden. Dies ermöglicht die Verwendung der folgenden *Cell Magics* in iPython:

- `%%unittest_main` führt Testfälle aus, die in einer Zelle definiert sind
- `%%unittest_testcase` erstellt einen Testfall mit der in einer Zelle definierten Funktion und führt ihn aus
- `%%unittest` konvertiert Python `assert` in Unittest-Funktionen
- `%%external` um externe Unittests durchzuführen
- `%%write {mode}` um externe Dateien zu schreiben

[3]:
```
%reload_ext ipython_unittest
```

[4]:
```
%%unittest_main
class MyTest(unittest.TestCase):
    def test_1_plus_1_equals_2(self):
        sum = 1 + 1
        self.assertEqual(sum, 2)

    def test_2_plus_2_equals_4(self):
        self.assertEqual(2 + 2, 4)
```

```

Success
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

[4]:
```
<unittest.runner.TextTestResult run=2 errors=0 failures=0>
```

[5]:
```
%%unittest_testcase
def test_1_plus_1_equals_2(self):
    sum = 1 + 1
    self.assertEqual(sum, 2)

def test_2_plus_2_equals_4(self):
    self.assertEqual(2 + 2, 4)
```

```
Success
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

[5]:
```
<unittest.runner.TextTestResult run=2 errors=0 failures=0>
```

[6]:
```
%%unittest
"1 plus 1 equals 2"
sum = 1 + 1
assert sum == 2
"2 plus 2 equals 4"
assert 2 + 2 == 4
```

```
Success
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
```

[6]:
```
<unittest.runner.TextTestResult run=2 errors=0 failures=0>
```

Standardmäßig trennt Docstring in dieser Magie die Unittest-Methoden. Wenn jedoch keine Docstrings verwendet werden, erstellen die *Cell Magics* für jede `assert`-Methode.

Diese *Cell Magics* unterstützen optionale Argumente:

- `-p` (`--previous`) `P`

  setzt den Cursor auf P Zellen vor (Standardwert: `-1` entspricht der nächsten Zelle)

  Dies funktioniert jedoch nur, wenn auch jupyter_dojo installiert ist.

- `-s` (`--stream`) `S`

  legt den *Ooutput-Stream* fest (Standardwert: `sys.stdout`)

- `-t` (`--testcase`) `T`

  definiert den Namen des TestCase für `%%unittest` und `%%unittest_testcase`

- `-u` (`--unparse`)

  gibt den Quellcode nach den Transformationen aus

Doctests

[1]:
```
import doctest


def add(a, b):
    """
    This is a test:
    >>> add(7,6)
    13
    """
    return a + b


doctest.testmod(verbose=True)
```

```
Trying:
    add(7,6)
Expecting:
    13
ok
1 items had no tests:
    __main__
1 items passed all tests:
   1 tests in __main__.add
1 tests in 2 items.
1 passed and 0 failed.
Test passed.
```

[1]:
```
TestResults(failed=0, attempted=1)
```

Debugging

[2]:
```
import doctest


doctest.testmod()


def multiply(a, b):
    """
    This is a test:
    >>> multiply(2, 2)
    5
    """
    import pdb

    pdb.set_trace()
    return a * b
```

1. `import pdb` importiert den Python Debugger
2. `pdb.set_trace()` erstellt einen sog. *Breakpoint*, der den Python-Debugger startet.

See also:

- pdb – The Python Debugger

Mock

Mock-Objekte fördern Tests, die auf dem Verhalten von Objekten basieren. Die Python-Bibliothek mock ermöglicht euch, Teile des zu testenden Systems durch Scheinobjekte zu ersetzen und Aussagen über deren Verwendung zu treffen.

Installation

mock ist seit Python 3.3 in der Python-Standardbibliothek enthalten. Für ältere Versionen von Python könnt ihr sie installieren mit:

```
$ bin/python -m pip install mock
```

Beispiel

In unserem Beispiel wollen wir prüfen, ob die Arbeitstage von Montag bis Freitag korrekt ermittelt werden.

1. Zunächst importieren wir `datetime` und `Mock`:

```
[1]: from datetime import datetime
     from unittest.mock import Mock
```

2. Dann definieren wir zwei Testtage:

```
[2]: monday = datetime(year=2021, month=10, day=11)
     saturday = datetime(year=2021, month=10, day=16)
```

3. Nun definieren wir eine Methode zur Überprüfung der Arbeitstage, wobei die `datetime`-Bibliothek von Python Montage als `0` und Sonntage als `6` behandelt:

```
[3]: def is_workingday():
         today = datetime.today()
         return (0 <= today.weekday() < 5)
```

4. Dann mocken wir `datetime`:

```
[4]: datetime = Mock()
```

5. Schließlich testen wir unsere beiden Mock-Objekte:

```
[5]: datetime.today.return_value = monday
     assert is_workingday()
```

```
[6]: datetime.today.return_value = saturday
     assert not is_workingday()
```

```
[7]: datetime.today.return_value = monday
     assert not is_workingday()
```

```
---------------------------------------------------------------------------
AssertionError                            Traceback (most recent call last)
Cell In[7], line 2
      1 datetime.today.return_value = monday
----> 2 assert not is_workingday()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
AssertionError:
```

Siehe auch:

- Introducing time-machine, a New Python Library for Mocking the Current Time

mock.ANY

Mit mock.ANY könnt ihr prüfen, ob ein Wert überhaupt vorhanden ist, ohne einen genauen Wert prüfen zu müssen:

[8]:
```
from unittest.mock import ANY


mock = Mock(return_value=None)
mock("foo", bar=object())
mock.assert_called_once_with("foo", bar=ANY)
```

Hinweis:

In test_report.py des OpenStack Containerdienstes Zun findet ihr weitere praktische Beispiele für `ANY`.

patch-Dekorator

Um Mock-Klassen oder Objekte zu erzeugen, kann der `patch`-Dekorator verwendet werden. In den folgenden Beispielen wird die Ausgabe von `os.listdir` gemockt. Dazu muss die Datei `example.txt` nicht im Verzeichnis vorhanden sein:

[9]:
```
import os

from unittest import mock
```

[10]:
```
@mock.patch("os.listdir", mock.MagicMock(return_value="example.txt"))
def test_listdir():
    assert "example.txt" == os.listdir()


test_listdir()
```

Alternativ kann der Rückgabewert auch separat definiert werden:

[11]:
```
@mock.patch("os.listdir")
def test_listdir(mock_listdir):
    mock_listdir.return_value = "example.txt"
    assert "example.txt" == os.listdir()


test_listdir()
```

Hinweis:

Mit responses könnt ihr Mock-Objekte für die httpx-Bibliothek erstellen.

3.6.3 Tools

ipytest

Setup

[1]:
```
# Set the file name (required)
__file__ = "testing.ipynb"

# Add ipython magics
# Add ipython magics
import ipytest
import pytest


ipytest.autoconfig()
```

Test Case

[2]:
```
%%ipytest

def test_sorted():
    assert sorted([4, 2, 1, 3]) == [1, 2, 3, 4]
```

```
.                                                                        [
↪    [100%]
1 passed in 0.00s
```

Test Fixture

[3]:
```
%%ipytest

@pytest.fixture
def dict_list():
    return [
        dict(a="a", b=3),
        dict(a="c", b=1),
        dict(a="b", b=2),
    ]


def test_sorted__key_example_1(dict_list):
    assert sorted(dict_list, key=lambda d: d["a"]) == [
        dict(a="a", b=3),
        dict(a="b", b=2),
        dict(a="c", b=1),
    ]


def test_sorted__key_example_2(dict_list):
    assert sorted(dict_list, key=lambda d: d["b"]) == [
        dict(a="c", b=1),
        dict(a="b", b=2),
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        dict(a="a", b=3),
    ]
```

```
..                                                                      [
↪    [100%]
2 passed in 0.01s
```

Testparametrisierung

[4]:
```
%%ipytest

@pytest.mark.parametrize(
    "input,expected",
    [
        ([2, 1], [1, 2]),
        ("zasdqw", list("adqswz")),
    ],
)
def test_examples(input, expected):
    actual = sorted(input)
    assert actual == expected
```

```
..                                                                      [
↪    [100%]
2 passed in 0.01s
```

Referenz

%%run_pytest ...

IPython-Magic, die zuerst die Zelle und dann `run_pytest` ausführt. In der Zelle übergebene Argumente werden direkt an pytest weitergeleitet. Zuvor sollten mit `import ipytest.magics` die Magics importiert worden sein.

ipytest.run_pytest(module=None, filename=None, pytest_options=(), pytest_plugins=())

führt die Tests im bestehenden Modul (standardmäßig `main`) mit pytest aus.

Argumente:

- `module`: das Modul, das die Tests enthält. Wenn nicht angegeben wird, wird `__main__` verwendet.
- `filename`: Dateiname der Datei, die die Tests enthält. Wenn nichts angegeben wird, wird das `__file__`-Attribut des übergebenen Moduls verwendet.
- `pytest_options`: zusätzliche Optionen, die an pytest übergeben werden
- `pytest_plugins`: zusätzliche pytest-Plugins

ipytest.run_tests(doctest=False, items=None)

Argumente:

- `doctest`: Wenn als Wert `True` angegeben wird, wird nach Doctests gesucht.
- `items`: Das *globals*-Objekt, das die Tests enthält. Wenn `None` angegeben wird, wird das *globals*-Objekt aus dem Call Stack ermittelt.

ipytest.clean_tests(pattern="test*", items=None)

löscht diejenigen Tests, deren Namen dem angegebenen Muster entsprechen.

In IPython werden die Ergebnisse aller Auswertungen in globalen Variablen gespeichert, sofern sie nicht explizit gelöscht werden. Dieses Verhalten impliziert, dass beim Umbenennen von Tests die vorherigen Definitionen weiterhin gefunden werden, wenn sie nicht gelöscht werden. Diese Methode zielt darauf ab, diesen Prozess zu vereinfachen.

Ein effektive Methode besteht darin, mit `clean_tests` eine Zelle zu beginnen, dann alle Testfälle zu definieren und schließlich `run_tests` aufzurufen. Auf diese Weise funktioniert das Umbenennen von Tests wie erwartet.

Argumente:

- `pattern`: Ein glob-Pattern, das verwendet wird, um die zu löschenden Tests zu finden.
- `items`: Das *globals*-Objekt, das die Tests enthält. Wenn `None` angegeben wird, wird das *globals*-Objekt aus dem Call Stack ermittelt.

ipytest.collect_tests(doctest=False, items=None)

sammelt alle Testfälle und sendet sie an `unittest.TestSuite`.

Die Argumente sind die gleichen wie für `ipytest.run_tests`.

ipytest.assert_equals(a, b, *args, **kwargs)

vergleicht zwei Objekte und wirft eine *Exception*, wenn sie nicht gleich sind.

Die Methode `ipytest.get_assert_function` bestimmt die zu verwendende Assert-Implementierung in Abhängigkeit von den folgenden Argumenten:

- `a, b`: die zwei zu vergleichenden Objekte.
- `args, kwargs`: (Schlüsselwort)-Argumente, die an die zugrundeliegende Testfunktion übergeben werden.

ipytest.get_assert_function(a, b)

bestimmt die zu verwendende Assert-Funktion in Abhängigkeit von den Argumenten.

Wenn eines der Objekte `numpy.ndarray`, `pandas.Series`, `pandas.DataFrame` oder `pandas.Panel` ist, werden die von `numpy` und `pandas` bereitgestellten Assert-Funktionen zurückgegeben.

ipytest.unittest_assert_equals(a, b)

vergleicht zwei Objekte mit der `assertEqual`-Methode von `unittest.TestCase`.

Hypothesis

Hypothesis ist eine Bibliothek, mit der ihr Tests schreiben könnt, die anhand einer Quelle von Beispielen parametrisiert werden. Anschließend werden einfache und nachvollziehbare Beispiele generiert, anhand derer eure Tests fehlschlagen können und ihr mit geringen Aufwänden Fehler finden könnt.

Beispiel

Zum Testen von Listen mit Fließkommazahlen werden viele Beispiele ausprobiert, jedoch im Report nur ein einfaches Beispiel für jeden Bug (eindeutiger exception type und Position) angegeben:

[1]:
```
from hypothesis import given
from hypothesis.strategies import lists, floats
```

[2]:
```
# Add ipython magics
import ipytest
import pytest


ipytest.autoconfig()
```

[3]:
```
%%ipytest

@given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
def test_mean(ls):
    mean = sum(ls) / len(ls)
    assert min(ls) <= mean <= max(ls)
```

```
F                                                                        ␣
↪    [100%]
========================================== FAILURES␣
↪==========================================
_____________________________________________ test_mean ___________________________________
↪____________

    @given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
>   def test_mean(ls):

/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_37502/1742712940.py:2:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
↪ _ _ _ _ _

ls = [1.922532935891866e+307, 1.797693134860272e+308]

    @given(lists(floats(allow_nan=False, allow_infinity=False), min_size=1))
    def test_mean(ls):
        mean = sum(ls) / len(ls)
>       assert min(ls) <= mean <= max(ls)
E       assert inf <= 1.797693134860272e+308
E        +  where 1.797693134860272e+308 = max([1.922532935891866e+307, 1.
↪797693134860272e+308])
E       Falsifying example: test_mean(
E           ls=[1.922532935891866e+307, 1.797693134860272e+308],
E       )

/var/folders/hk/s8m0bblj0g10hw885gld52mc0000gn/T/ipykernel_37502/1742712940.py:4:␣
↪AssertionError
======================================== warnings summary␣
↪========================================
../../../../../../.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/_pytest/config/__init__.py:1204
  /srv/jupyter/.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-packages/
↪_pytest/config/__init__.py:1204: PytestAssertRewriteWarning: Module already imported␣
↪so cannot be rewritten: hypothesis
    self._mark_plugins_for_rewrite(hook)

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
=================================== short test summary info␣
↪====================================
FAILED t_c7729a99e261490d9a679829703295f9.py::test_mean - assert inf <= 1.
↪797693134860272e+308
1 failed, 1 warning in 2.14s
```

Installation

```
$ uv add hypothesis
```

Alternativ kann Hypothesis auch mit Erweiterungen installiert werden, z.B.:

```
$ uv add hypothesis[numpy,pandas]
```

Bemerkung:

Falls ihr uv noch nicht installiert habt, findet ihr eine Anleitung hierzu unter *Jupyter Notebook installieren*.

Siehe auch:

- Hypothesis for the Scientific Stack

KAPITEL 4

JupyterLab

JupyterLab ist ein erweiterbarer, funktionsreicher Editor zum Erstellen und Bearbeiten von *Jupyter Notebooks*:

- Ihr könnt mehrere Dokumente und Aktivitäten in eurem Arbeitsbereich mit Hilfe von Registerkarten nebeneinander anordnen.
- Code-Konsolen bieten temporäre Scratchpads für die interaktive Ausführung von Code, die auch mit einem *Notebook-Kernel* verknüpft werden können.
- Es gibt auch eine Vorschau von CSV- un Vega-Dateien.
- *JupyterLab-Erweiterungen* können jeden Teil von JupyterLab anpassen oder verbessern.

JupyterLab verwendet aktuell das gleiche Notebook-Dokumentenformat wie das klassische *Jupyter Notebook*. Erst Notebook 7 wird das klassische Jupyter Notebook-Format ersetzen.

Siehe auch

Migrating to Notebook 7

4.1 JupyterLab installieren

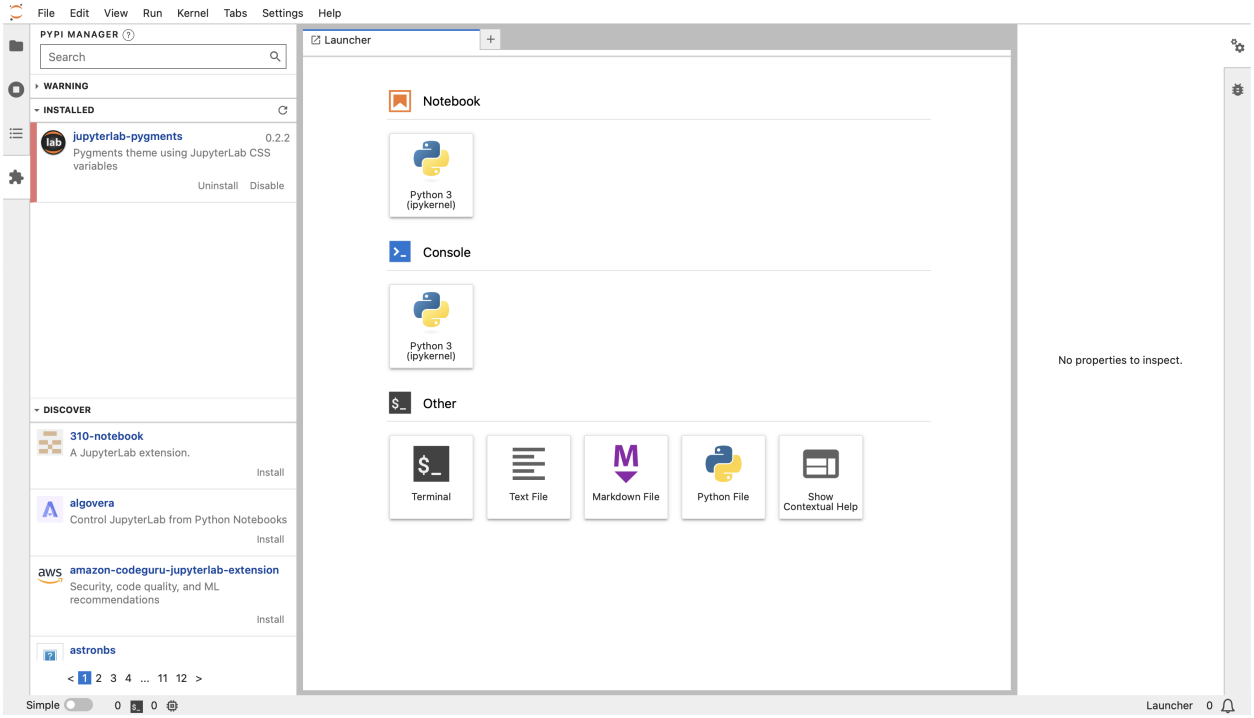
4.1.1 Erstellen einer virtuellen Umgebung mit JupyterLab

```
$ python3 -m venv myproject
$ cd myproject
$ . bin/activate
$ python -m pip install jupyterlab
```

4.1.2 JupyterLab starten

```
$ jupyter lab
[I 2023-06-16 13:01:43.205 ServerApp] Package jupyterlab took 0.0000s to import
...
    To access the server, open this file in a browser:
        file:///Users/veit/Library/Jupyter/runtime/jpserver-48904-open.html
    Or copy and paste one of these URLs:
        http://localhost:8888/lab?token=72d33027f130e602f43ef0cdfbff7471c8406ffafd94f075
        http://127.0.0.1:8888/lab?token=72d33027f130e602f43ef0cdfbff7471c8406ffafd94f075
```

Euer Standard-Webbrowser wird dann mit dieser URL geöffnet.



4.1.3 Lokalisierung

Seit Version 3.0 bietet JupyterLab die Möglichkeit, die Anzeigesprache der Oberfläche einzustellen. Hierfür müssen die entsprechenden Sprachpakete installiert werden, z.B.:

```
$ python -m pip install jupyterlab-language-pack-de-DE
```

Im language-packs-Repository findet ihr eine Liste der verfügbaren Sprachpakete.

Anschließend könnt ihr in *Settings* → *Language* die neu installierte Sprache auswählen.

4.2 JupyterLab-Erweiterungen

JupyterLab ist als erweiterbare Umgebung konzipiert. Dabei können JupyterLab-Erweiterungen jeden Teil von JupyterLab anpassen. Sie können neue Themen, Dateibetrachter und -editoren oder Renderer für umfangreiche Ausgaben in *Notebook* bereitstellen.

Siehe auch

JupyterLab Extensions by Examples

4.2.1 Installieren von Erweiterungen

Eine JupyterLab-Erweiterung enthält JavaScript, das in Jupyterlab installiert und im Browser ausgeführt wird. Die meisten JupyterLab-Erweiterungen können mit pip installiert werden. Diese Pakete können auch serverseitige Komponenten enthalten, die für die Funktion der Erweiterung erforderlich sind.

Seit JupyterLab 4 verwendet der Standard-Extension-Manager PyPI als Quelle für die verfügbaren Erweiterungen und pip, um sie zu installieren. Eine Erweiterung wird aufgelistet, wenn das Python-Paket den Trove-Klassifizierer `Framework :: Jupyter :: JupyterLab :: Extensions :: Prebuilt` hat.

Warnung

Es wird nicht überprüft, ob die Erweiterung mit der aktuellen JupyterLab-Version kompatibel ist.

Gefahr

Die Installation einer Erweiterung ermöglicht die Ausführung von beliebigem Code auf dem Server, dem Kernel und dem Browser. Vermeidet daher die Installation von Erweiterungen, denen ihr nicht vertraut.

4.2.2 Konfigurieren des Extension Manager

Standardmäßig gibt es zwei Erweiterungsmanager, die von JupyterLab bereitgestellt werden:

`pypi`
: Standardeinstellung, die das Installieren von pypi.org erlaubt.

`readonly`
: zeigt die installierten Erweiterungen an mit der Möglichkeit, sie zu deaktivieren oder aktivieren.

Ihr könnt den Manager mit der Kommandozeilenoption `--LabApp.extension_manager` angeben, z.B. `jupyter lab --LabApp.extension_manager=`*`readonly`*.

Bei der Suche nach Erweiterungen im Erweiterungsmanager zeigt JupyterLab üblicherweise alle Suchergebnisse an und jede beliebige Quell-Erweiterung kann installiert werden. Um die Sicherheit zu erhöhen, kann JupyterLab jedoch so konfiguriert sein, dass die Erweiterungen nur anhand der Block- oder Allow-Listen aktiviert werden können.

Das Laden der Listen könnt ihr definieren mit `blocked_extensions_uris` oder `allowed_extensions_uris`, die eine Liste von Komma-getrennten URIs enthalten, z.B. `--LabServerApp.blocked_extensions_uris=http://example.com/blocklist.json` mit folgender `blocklist.json`-Datei:

```
{
  "blocked_extensions": [
    {
      "name": "@jupyterlab-examples/launcher",
      "type": "jupyterlab",
      "reason": "@jupyterlab-examples/launcher is blocklisted for test purpose - Do NOT␣
↪take this for granted!!!",
      "creation_date": "2020-03-11T03:28:56.782Z",
      "last_update_date":  "2020-03-11T03:28:56.782Z"
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    }
  ]
}
```

Ein anderes Beispiel zeigt eine `allowlist.json`-Datei, die alle Erweiterungen der JupyterLab-Organisation erlauben:

```
{
  "allowed_extensions": [
    {
      "name": "@jupyterlab/*",
      "type": "jupyterlab",
      "reason": "All @jupyterlab org extensions are allowed, of course...",
      "creation_date": "2020-03-11T03:28:56.782Z",
      "last_update_date":  "2020-03-11T03:28:56.782Z"
    }
  ]
}
```

4.3 JupyterLab auf JupyterHub

JupyterLab funktioniert standardmäßig mit *JupyterHub* 1.0 und kann sogar neben den klassischen *Notebooks* laufen.

Wenn JupyterLab mit JupyterHub eingesetzt wird, werden im *File*-Menü zusätzliche Menüpunkte angezeigt zum *Log Out* oder zum Aufruf des *Hub Control Panel*.

Falls JupyterLab noch nicht der Standard ist, könnt ihr die Konfiguration in `jupyterhub_config.py` ändern:

```
c.Spawner.default_url = "/lab"
```

4.4 Real-Time Collaboration

Ab JupyterLab 4 ist es möglich, Real-Time Collaboration zu aktivieren, indem die Erweiterung `jupyter_collaboration` installiert wird. Dies ermöglicht die Zusammenarbeit in Echtzeit zwischen mehreren Clients. Außerdem könnt ihr die Cursor anderer sehen.

Siehe auch

- jupyter_collaboration documentation

4.4.1 Installation

```
$ python -m pip install jupyter-collaboration
```

Um ein Dokument für andere freizugeben, könnt ihr die URL kopieren und versenden, oder ihr könnt zusätzlich jupyterlab-link-share installieren, mit der ihr den Link einschließlich des Tokens freigeben könnt.

4.5 Scheduler

Jupyter Scheduler ist eine Sammlung von Erweiterungen zur Programmierung von Jobs, die sofort oder nach einem Zeitplan ausgeführt werden sollen. Er hat eine Lab- (Client-) und eine Server-Erweiterung. Beide werden benötigt, um Notebooks planen und ausführen zu können. Wenn ihr Jupyter Scheduler über den JupyterLab-Extension-Manager installiert, installiert ihr möglicherweise nur die Client-Erweiterung und nicht die Server-Erweiterung. Installiert den Jupyter Scheduler daher mit pip:

```
$ python -m pip install jupyter_scheduler
```

Dadurch werden die Lab- und Servererweiterungen automatisch aktiviert. Ihr könnt dies überprüfen mit

```
$ jupyter server extension list
...
    jupyter_scheduler enabled
    - Validating jupyter_scheduler...
Package jupyter_scheduler took 0.0785s to import
      jupyter_scheduler 1.3.2 OK
...
```

und

```
$ jupyter labextension list
...
        @jupyterlab/scheduler v1.3.2 enabled  X
...
```

1. Um einen Jog aus einem geöffneten Notebook zu erstellen, klickt in der oberen Symbolleiste des geöffneten Notizbuchs auf die Schaltfläche *Create a notebook job*.
2. Gebt eurem Notebook-Job einen Namen, wählt die Ausgabeformate und gebt Parameter an, die bei der Ausführung eures Notebooks als lokale Variablen gesetzt werden. Diese parametrisierte Ausführung ähnelt der in *Papermill* verwendeten.
3. Um einen Job zu erstellen, der einmalig ausgeführt wird, wählt *Run now* und klickt auf *Create*.
4. Um eine Job-Definition zu erstellen, die wiederholt nach einem Zeitplan ausgeführt wird, wählt *Run on a schedule*.

KAPITEL 5

JupyterHub

JupyterHub ist ein Multi-User Server für *Jupyter Notebooks*, der viele verschiedene Instanzen von Jupyter Notebooks erzeugen und verwalten kann und der als Proxy fungiert.

5.1 Installation

1. Python3.6 und pip installieren:

```
$ sudo apt update
$ sudo apt install python3
$ python3 -V
Python 3.10.6
$ sudo apt install python3-pip
```

2. Service-User `jupyter` erstellen:

```
$ sudo useradd -s /bin/bash -rmd /srv/jupyter jupyter
```

3. Zum Service-User `jupyter` wechseln:

```
$ sudo -u jupyter -i
```

4. uv installieren:

```
$  curl -LsSf https://astral.sh/uv/install.sh | sh
```

5. Automatische Shell-Vervollständigung aktivieren:

```
$ echo 'eval "$(uv generate-shell-completion bash)"' >> ~/.bashrc
```

6. Virtuelle Umgebung erstellen und JupyterHub installieren:

```
$ uv init --package jupyterhub_env
$ cd jupyterhub_env
$ uv add jupyterhub
```

7. `nodejs` und `npm` installieren:

```
$ sudo apt install nodejs npm
$ node -v
v23.3.0
$ npm -v
10.9.0
```

8. Installieren des HTTP-Proxy:

```
$ sudo npm install -g configurable-http-proxy
```

9. Wenn JupyterLab und Notebook in derselben Umgebung laufen sollen, müssen diese ebenfalls hier installiert werden:

```
$  uv add jupyterlab notebook
```

10. Testen der Installation:

```
$  uv run jupyterhub -h
$  configurable-http-proxy -h
```

11. Starten des JupyterHub:

```
$  uv run jupyterhub
...
[I 2025-01-10 18:07:29.993 JupyterHub app:3770] JupyterHub is now running at http://
↪:8000
```

Mit `ctrl-c` könnt ihr den Prozess wieder beenden.

5.2 Konfiguration

5.2.1 JupyterHub-Konfiguration

Konfigurationsdatei erstellen:

```
$  uv run jupyterhub --generate-config
Writing default config to: jupyterhub_config.py
```

Siehe auch

- JupyterHub Configuration Basics
- JupyterHub Networking basics

5.2.2 System-Service für JupyterHub

1. Ermitteln des *Python Virtual Environment*:

```
$ cd jupyterhub_env
$ pwd
/srv/jupyter/jupyterhub_env
```

2. Konfigurieren des absoluten Pfades zu `jupyterhub-singleuser` in der `jupyterhub_config.py`-Datei:

```
c.Spawner.cmd = ["/srv/jupyter/jupyterhub_env/.venv/bin/jupyterhub-singleuser"]
```

3. Hinzufügen einer neuen systemd-Unit-Datei `/etc/systemd/system/jupyterhub.service` mit dem Befehl `$ sudo systemctl edit --force --full jupyterhub.service`

   Fügt eure entsprechende Python-Umgebung ein.

```
[Unit]
Description=Jupyterhub

[Service]
User=root
Environment="PATH=/bin:/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/srv/
↪jupyter/.local/share/virtualenvs/jupyterhub-aFv4x91W/bin"
ExecStart=/srv/jupyter//srv/jupyter/jupyterhub_env/.venv/bin/jupyterhub -f /srv/
↪jupyter/jupyterhub_env/jupyterhub_config.py

[Install]
WantedBy=multi-user.target
```

4. Laden der Konfiguration mit `sudo systemctl daemon-reload`.
5. Der JupyterHub lässt sich verwalten mit `sudo systemctl` *START*|*STOP*|*STATUS* `jupyterhub`
6. Um sicherzustellen, dass der Dienst auch bei einem Systemstart mitgeladen wird, wird folgendes aufgerufen:

```
$ sudo systemctl enable jupyterhub.service
Created symlink /etc/systemd/system/multi-user.target.wants/jupyterhub.service → /
↪etc/systemd/system/jupyterhub.service.
```

7. Um den `jupyterhub-singleuser`-Spawner nutzen und einen eigenen Server starten zu können, müssen die ix-User in der Gruppe `jupyter` eingetragen werden, z.B. mit `usermod -aG jupyter` *VEIT*.

5.2.3 TLS-Verschlüsselung

Da JupyterHub eine Authentifizierung beinhaltet und die Ausführung von beliebigem Code erlaubt, sollte es nicht ohne SSL (HTTPS) ausgeführt werden. Dazu muss ein offizielles, vertrauenswürdiges SSL-Zertifikat erstellt werden. Nachdem ihr einen Schlüssel und ein Zertifikat erhalten und installiert habt, konfiguriert ihr jedoch nicht das JupyterHub selbst sondern den vorgeschalteten Apache Webserver.

1. Hierfür werden zunächst die Zusatzmodule aktiviert mit

```
# a2enmod ssl rewrite proxy proxy_http proxy_wstunnel
```

2. Anschließend kann der VirtualHost in `/etc/apache2/sites-available/jupyter.cusy.io.conf` konfiguriert werden mit

```
# redirect HTTP to HTTPS
<VirtualHost 172.31.50.170:80>
    ServerName jupyter.cusy.io
    ServerAdmin webmaster@cusy.io

    ErrorLog ${APACHE_LOG_DIR}/jupyter.cusy.io_error.log
    CustomLog ${APACHE_LOG_DIR}/jupyter.cusy.io_access.log combined

    Redirect / https://jupyter.cusy.io/
</VirtualHost>

<VirtualHost 172.31.50.170:443>
  ServerName jupyter.cusy.io
  ServerAdmin webmaster@cusy.io

  # configure SSL
  SSLEngine On
  SSLCertificateFile /etc/ssl/certs/jupyter.cusy.io_cert.pem
  SSLCertificateKeyFile /etc/ssl/private/jupyter.cusy.io_sec_key.pem
  # for an up-to-date SSL configuration see e.g.
  # https://ssl-config.mozilla.org/

  # Use RewriteEngine to handle websocket connection upgrades
  RewriteEngine On
  RewriteCond %{HTTP:Connection} Upgrade [NC]
  RewriteCond %{HTTP:Upgrade} websocket [NC]
  RewriteRule /(.*) ws://127.0.0.1:8000/$1 [P,L]

  <Location "/">
    # preserve Host header to avoid cross-origin problems
    ProxyPreserveHost on
    # proxy to JupyterHub
    ProxyPass         http://127.0.0.1:8000/
    ProxyPassReverse  http://127.0.0.1:8000/
  </Location>

  ErrorLog ${APACHE_LOG_DIR}/jupyter.cusy.io_error.log
  CustomLog ${APACHE_LOG_DIR}/jupyter.cusy.io_access.log combined
</VirtualHost>
```

3. Dieser VirtualHost wird aktiviert mit `# a2ensite` *`JUPYTER.CUSY.IO`*`.conf`.
4. Schließlich könnt ihr den Status des Apache-Webserver überprüfen mit

```
# systemctl status apache2
 apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset:␣
↪enabled)
   Active: active (running) (Result: exit-code) since Mon 2019-03-25 16:50:26 CET;␣
↪1 day 22h ago
  Process: 31773 ExecReload=/usr/sbin/apachectl graceful (code=exited, status=0/
↪SUCCESS)
 Main PID: 20273 (apache2)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    Tasks: 55 (limit: 4915)
   CGroup: /system.slice/apache2.service
           ├─20273 /usr/sbin/apache2 -k start
           ├─31779 /usr/sbin/apache2 -k start
           └─31780 /usr/sbin/apache2 -k start

Mar 27 06:25:01 jupyter.cusy.io systemd[1]: Reloaded The Apache HTTP Server.
```

5.2.4 Cookie-Secret

Das Cookie secret ist zum Verschlüsseln der Browser-Cookies, die zur Authentifizierung verwendet werden.

1. Das Cookie-Secret kann z.B. erstellt werden mit

```
$ openssl rand -hex 32 > /srv/jupyter/venv/jupyterhub_cookie_secret
```

2. Die Datei sollte weder für `group` noch für `anonymous` lesbar sein:

```
$ chmod 600 /srv/jupyter/venv/jupyterhub_cookie_secret
```

3. Schließlich wird es in die `jupyterhub_config.py`-Datei eingetragen:

```
c.JupyterHub.cookie_secret_file = "jupyterhub_cookie_secret"
```

5.2.5 Proxy authentication token

Der Hub authentifiziert seine Anforderungen an den Proxy unter Verwendung eines geheimen Tokens, auf das sich der Hub und der Proxy einigen. Üblicherweise muss der Proxy authentication token nicht festgelegt werden, da der Hub selbst einen zufälligen Schlüssel generiert. Dies bedeutet, dass der Proxy jedes Mal neu gestartet werden muss sofern der Proxy nicht ein Unterprozess des Hubs ist.

1. Alternativ kann Der Wert z.B. generiert werden mit

```
$ openssl rand -hex 32
```

2. Anschließend kann er in der Konfigurationsdatei eingetragen werde, z.B. mit

```
c.JupyterHub.proxy_auth_token = (
    "18a0335b7c2e7edeaf7466894a32bea8d1c3cff4b07860298dbe353ecb179fc6"
)
```

5.3 systemdspawner

Der systemdspawner ermöglicht es JupyterHub, Einzelbenutzer-Notebookserver mit systemd zu erzeugen. Ihr erhaltet Isolation und Sicherheit ohne Docker, rkt o.ä. (oder ähnliches) verwenden zu müssen. Zudem bietet systemdspawner weitere Features:

- der maximal zulässige Speicher und die maximal verfügbare CPU pro Person kann über cgroups begrenzt und mit `systemd-cgtop` überprüft werden.
- alle erhalten ein eigenes `/tmp`-Verzeichnis um die Isolation zu erhöhen
- Notebook-Server können als bestimmte lokale User auf dem System gestartet werden
- die Nutzung von sudo in Notebooks kann eingeschränkt werden

- die Pfade, in die gelesen und geschrieben werden können, lassen sich einschränken
- Protokolle für jedes einzelne Notebook können verwaltet werden

5.3.1 Anforderungen

systemdspawner setzt systemd v211 voraus; die sicherheitsrelevanten Funktionen erfordern systemd v228. Ihr könnt überprüfen, welche Version von systemd bei euch verfügbar ist mit

```
$ systemctl --version | head -1
systemd 249 (249.11-0ubuntu3.7)
```

Zum Limitieren der Speicher- und CPU-Zuweisungen müssen zudem bestimmte Kernel-Optionen zur Verfügung stehen. Dies können mit dem check-kernel.bash überprüft werden.

Wenn die Standardeinstellung `c.SystemdSpawner.dynamic_users = False` verwendet wird, wird der Server mit dem lokalen Unix-User-Account gestartet. Daher erfordert dieser Spawner, dass alle User, bereits ein lokales Konto auf der Maschine haben. Mit `c.SystemdSpawner.dynamic_users = True` sind hingegen keine lokalen User-Accounts erforderlich; sie werden durch systemd bei Bedarf dynamisch erstellt.

5.3.2 Installation und Konfiguration

Ihr könnt systemdspawner installieren mit

```
$ uv add jupyterhub-systemdspawner
```

Anschließend kann er in der `jupyterhub_config.py` aktiviert werden mit

```
c.JupyterHub.spawner_class = "systemdspawner.SystemdSpawner"
```

Es stehen euch viele weitere Konfigurationsmöglichkeiten offen, z.B.

`c.SystemdSpawner.mem_limit = '4G'`
: gibt den maximalen Speicherplatz an, der von jedem einzelnen User verwendet werden kann. Die Einstellung `None` deaktiviert die Speicherbegrenzung.

  Auch wenn einzelne User so viel Speicher wie möglich verwenden können sollen, ist es dennoch sinnvoll, eine Speicherbegrenzung von 80–90 % des gesamten physischen Speichers festzulegen. Dadurch wird verhindert, dass ein User den Rechner versehentlich im Alleingang lahmlegen kann.

`c.SystemdSpawner.cpu_limit = 4.0`
: Eine Fließkommazahl, die die Anzahl der CPU-Kerne angibt, die jeder User verwenden kann.

`c.SystemdSpawner.user_workingdir = '/home/USERNAME'`
: Das Verzeichnis, in dem der Notebook-Server eines jeden User gestartet wird. Dieses Verzeichnis sehen auch die User, wenn sie ihre Notebook-Server öffnen. Normalerweise ist dies das Heimatverzeichnis des Benutzers.

`c.SystemdSpawner.username_template = 'jupyter-USERNAME'`
: Vorlage für den Unix-User-Namen, unter dem jeder User angelegt werden soll.

`c.SystemdSpawner.default_shell = '/bin/bash'`
: Die Standard-Shell, die für das Terminal im Notebook verwendet wird. Setzt die Umgebungsvariable `SHELL` auf diesen Wert.

`c.SystemdSpawner.extra_paths = ['/home/USERNAME/conda/bin']`
: Liste der Pfade, die der Umgebungsvariablen `PATH` für den gespawnten Notebook-Server vorangestellt werden sollen. Dies ist einfacher als das Setzen der `env`-Eigenschaft, da ihr `PATH` hinzufügen und nicht komplett ersetzen wollt. Dies ist sehr nützlich, wenn ihr eine virtualenv- oder conda-Installation standardmäßig in `PATH` des Users aufnehmen wollt.

c.SystemdSpawner.unit_name_template = 'jupyter-*USERNAME*-singleuser'
Namensvorlage der Systemd-Service-Einheit für jeden User-Notebook-Server. Dies ermöglicht die Unterscheidung zwischen mehreren JupyterHubs mit systemd-Spawner auf derselben Maschine. Sollte nur `[a-zA-Z0-9_-]` enthalten.

c.SystemdSpawner.unit_extra_properties = *'LimitNOFILE': '16384'*
Dict von Schlüssel-Wert-Paaren, die verwendet werden, um beliebige Eigenschaften zu den gespawnten Jupyerhub-Units hinzuzufügen – s.a. (siehe auch) `man systemd-run` für Details zu den Eigenschaften.

c.SystemdSpawner.isolate_tmp = True
Wenn dieser Wert auf `True` gesetzt wird, wird für jeden User ein separates, privates `/tmp`-Verzeichnis angelegt. Dies ist sehr nützlich, um sich gegen das versehentliche Durchsickern von ansonsten privaten Informationen zu schützen.

Dies erfordert systemd Version > 227. Wenn ihr dies in früheren Versionen aktiviert, wird das Spawnen fehlschlagen.

c.SystemdSpawner.isolate_devices = True
Wenn ihr diese Option auf `True` setzt, wird für jeden User ein separates, privates `/dev`-Verzeichnis eingerichtet. Dadurch wird verhindert, dass User direkt auf Hardware-Devices zugreifen, was eine potenzielle Quelle für Sicherheitsprobleme sein könnte. `/dev/null`, `/dev/zero`, `/dev/random` und die `ttyp`-Pseudo-Geräte sind bereits gemountet, so dass die meisten User keine Veränderung bemerken sollten, wenn dies aktiviert ist.

c.SystemdSpawner.disable_user_sudo = True
Wenn Ihr diese Option auf `True` setzt, wird verhindert, dass User `sudo` oder andere Mittel verwenden können, um andere User zu werden. Dies hilft dabei, den Schaden einzudämmen, der durch die Kompromittierung der Anmeldeinformationen eines Benutzers entsteht, wenn dieser auch `sudo`-Rechte auf dem Rechner hat – ein webbasierter Exploit kann nun nur noch die eigenen Daten des Users beschädigen.

Dies erfordert systemd Version > 228. Wenn ihr dies in früheren Versionen aktiviert, wird das Spawnen fehlschlagen.

c.SystemdSpawner.readonly_paths = ['/']
Liste der Dateisystempfade, die für den Notebook-Server des Users schreibgeschützt eingehängt werden sollen. Damit werden eventuell vorhandene Dateisystemberechtigungen außer Kraft gesetzt. Unterpfade von Pfaden, die `readonly` gemountet sind, können mit `readwrite_paths` als `readwrite` markiert werden. Dies ist nützlich, um `/` als schreibgeschützt zu markieren und nur die Pfade aufzulisten, in die Notebook-User schreiben dürfen. Wenn die hier aufgeführten Pfade nicht existieren, erhaltet ihr eine Fehlermeldung.

Dies erfordert systemd Version > 228. Wenn ihr diese Funktion in früheren Versionen aktiviert, wird das Spawnen fehlschlagen. Bis zur systemd-Version 231 kann es auch nur Verzeichnisse und keine Dateien enthalten.

c.SystemdSpawner.readwrite_paths = ['/home/*USERNAME*']
Liste der Dateisystempfade, die für den Notebook-Server des Users schreibgeschützt eingehängt werden sollen. Dies macht nur Sinn, wenn `readonly_paths` verwendet wird, um einige Pfade schreibgeschützt zu machen. Dies setzt die Dateisystemberechtigungen nicht außer Kraft – der User muss über die entsprechenden Rechte verfügen, um auf diese Pfade zu schreiben.

Dies erfordert systemd Version > 228. Wenn ihr diese Funktion in früheren Versionen aktiviert, wird das Spawnen fehlschlagen. Bis systemd Version 231 kann es auch nur Verzeichnisse und keine Dateien enthalten.

Siehe auch

- systemdspawner

5.4 Service `nbviewer` erstellen

1. Das Konfigurieren des Notebook-Viewer als JupyterHub-Service hat den Vorteil, dass nur Benutzer, die sich zuvor beim JupyterHub angemeldet haben, die `nbviewer`-Instanz aufrufen können. Damit kann der Zugriff auf Notebooks geschützt werden, als JupyterHub-Service in `/srv/jupyter/jupyter-tutorial/jupyterhub_config.py`:

```
c.JupyterHub.services = [
    {
        "name": "nbviewer",
        "url": "http://127.0.0.1:9000",
        "cwd": "/srv/jupyterhub/nbviewer-repo",
        "command": [
            "/srv/jupyter/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/bin/
↪python",
            "-m",
            "nbviewer",
        ],
    }
]
```

`name`
: Der Pfadname unter dem der Notebook-Viewer erreichbar ist `/services/`*`NAME`*

`url`
: Protokoll, Adresse und Port, die `nbviewer` verwendet

`cwd`
: Der Pfad zum `nbviewer`-Repository

`command`
: Kommando um `nbviewer` zu starten

5.5 `ipyparallel`

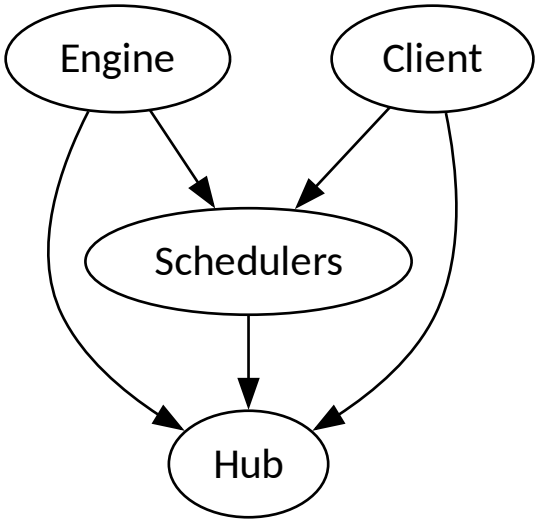
Dieser Abschnitt gibt einen Überblick über ipyparallel, das verschiedene Arten von Parallelisierung unterstützt, U.A. (unter anderem):

- Single Program, Multiple Data (SPMD)
- Multiple program, multiple data (MPMD)
- Message Passing Interface (MPI)

5.5.1 Überblick

Architektur

Die `IPython.parallel`-Architektur besteht aus vier Komponenten:



IPython-Engine

Die IPython-Engine ist eine Erweiterung des IPython-Kernels für Jupyter. Das Modul wartet auf Anfragen aus dem Netzwerk, führt Code aus und gibt die Ergebnisse zurück. IPython parallel erweitert das Jupyter-Messaging-Protokoll um native Python-Objektserialisierung und fügt einige zusätzliche Befehle hinzu. Zum parallelen und verteilten Rechnen werden mehrere Engines gestartet.

IPython-Hub

Die Hauptaufgabe des Hubs besteht darin, Verbindungen zu Clients und Engines herzustellen und zu überwachen.

IPython-Schedulers

Alle Aktionen, die an der Engine ausgeführt werden können, durchlaufen einen Scheduler. Während die Engines selbst blockieren, wenn Benutzercode ausgeführt wird, verbergen die Scheduler dies vor dem Benutzer, um eine vollständig asynchrone Schnittstelle für eine Reihe von Engines bereitzustellen.

IPython-Client

Es gibt ein Hauptobjekt `Client` um eine Verbindung zum Cluster herzustellen. Für jedes Ausführungsmodell gibt es dann einen entsprechenden `View`. Mit diesen `Views` können Benutzer mit einer Reihe von Engines interagieren. Dabei sind die beiden Standardansichten:

- `ipyparallel.DirectView`-Klasse für die explizite Adressierung
- `ipyparallel.LoadBalancedView`-Klasse für zielunabhängiges Scheduling

Installation

```
$ uv add ipyparallel
```

Starten

1. Starten des IPython-Hub:

   ```
   $ uv run ipcontroller
   [IPControllerApp] Hub listening on tcp://127.0.0.1:53847 for registration.
   [IPControllerApp] Hub using DB backend: 'DictDB'
   [IPControllerApp] hub::created hub
   [IPControllerApp] writing connection info to /Users/veit/.ipython/profile_default/
   ↪security/ipcontroller-client.json
   [IPControllerApp] writing connection info to /Users/veit/.ipython/profile_default/
   ↪security/ipcontroller-engine.json
   [IPControllerApp] task::using Python leastload Task scheduler
   ...
   ```

   DB-Backend
   : Die Datenbank, in der die IPython-Tasks verwaltet werden. Neben der In-Memory-Datenbank `DictDB` sind `MongoDB` und `SQLite` die weiteren Optionen.

   `ipcontroller-client.json`
   : Konfigurationsdatei für den IPython-Client

   `ipcontroller-engine.json`
   : Konfigurationsdatei für die IPython-Engine

   Task-Schedulers
   : Das mögliche Routing-Schema. `leastload` weist Aufgaben immer derjenigen Engine zu, die die wenigsten offenen Aufgaben hat. Alternativ lasst sich `lru` (Least Recently Used), `plainrandom`, `twobin` und `weighted` auswählen, wobei die beiden letztgenannten zusätzlich Numpy benötigen.

     Dies kann konfiguriert werden in `ipcontroller_config.py`, z.B. mit `c.TaskScheduler.scheme_name = 'leastload'` oder mit

     ```
     $ uv run ipcontroller --scheme=pure
     ```

2. Starten des IPython-Controller und der -Engines:

   ```
   $ uv run ipcluster start
   [IPClusterStart] Starting ipcluster with [daemon=False]
   [IPClusterStart] Creating pid file: /Users/veit/.ipython/profile_default/pid/
   ↪ipcluster.pid
   [IPClusterStart] Starting Controller with LocalControllerLauncher
   [IPClusterStart] Starting 4 Engines with LocalEngineSetLauncher
   ```

   Batch-Systeme
   : Neben der Möglichkeit, `ipcontroller` und `ipengine` lokal zu starten, siehe Starting the controller and engine on your local machine, die `LocalControllerLauncher` und `LocalEngineSetLauncher` starten, gibt es auch noch die Profile `MPI`, `PBS`, `SGE`, `LSF`, `HTCondor`, `Slurm`, `SSH` und `WindowsHPC`.

     Dies kann konfiguriert werden in `ipcluster_config.py` z.B. mit `c.IPClusterEngines.engine_launcher_class = 'SSH'` oder mit

```
$ uv run ipcluster start --engines=MPI
```

> **Siehe auch**
>
> *MPI*

3. Starten des Jupyter Notebook und Laden der IPython-Parallel-Extension:

```
$ uv run jupyter notebook
[I NotebookApp] Loading IPython parallel extension
[I NotebookApp] [jupyter_nbextensions_configurator] enabled 0.4.1
[I NotebookApp] Serving notebooks from local directory: /Users/veit//jupyter-
↪tutorial
[I NotebookApp] The Jupyter Notebook is running at:
[I NotebookApp] http://localhost:8888/?
↪token=4e9acb8993758c2e7f3bda3b1957614c6f3528ee5e3343b3
```

4. Im Browser kann anschließend unter der Adresse `http://localhost:8888/tree/docs/parallel/ipyparallel#ipyclusters` der Cluster mit dem `default`-Profil gestartet werden.

5.5.2 Überprüfen der Installation

```
[1]: import ipyparallel as ipp


     c = ipp.Client()
     c.ids
```

```
[1]: [0, 1, 2, 3]
```

```
[2]: c[:].apply_sync(lambda : "Hello, World")
```

```
[2]: ['Hello, World', 'Hello, World', 'Hello, World', 'Hello, World']
```

5.5.3 Konfiguration

Für die Konfiguration wird beim Starten des IPython-Hub für Client und Engine jeweils eine Konfigurationsdatei angelegt, üblicherweise in `~/.ipython/profile_default/security/`.

1. Falls wir nicht das `default`-Profil verwenden wollen, sollten wir zunächst ein neues IPython-Profil erstellen mit:

```
$ uv run ipython profile create --parallel --profile=local
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_
↪parallel/ipython_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_
↪parallel/ipython_kernel_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_
↪parallel/ipcontroller_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_
↪parallel/ipengine_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_
↪parallel/ipcluster_config.py
```

`--parallel`
: schließt die Konfigurationsdateien für *Parallel Computing* (`ipengine`, `ipcontroller` ETC. (et cetera)) ein.

1. Beim Starten des IPython-Controller und der -Engines werden die Dateien `ipcontroller-engine.json` und `ipcontroller-client.json` dann in `~/.ipython/profile_default/security/` erzeugt.

`ipcluster` in `mpiexec/mpirun`-Modus

1. Erstellen des Profils:

```
$ uv run ipython profile create --parallel --profile=mpi
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_mpi/
↪ipython_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_mpi/
↪ipython_kernel_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_mpi/
↪ipcontroller_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_mpi/
↪ipengine_config.py'
[ProfileCreate] Generating default config file: '/Users/veit/.ipython/profile_mpi/
↪ipcluster_config.py'
```

2. Editieren von `ipcluster_config.py`:

    1. Damit die MPI-Launcher verwendet werden:

```
c.IPClusterEngines.engine_launcher_class = "MPIEngineSetLauncher"
```

3. Anschließend kann der Cluster gestartet werden mit:

```
$ uv run ipcluster start -n 4 --profile=mpi
[IPClusterStart] Starting ipcluster with [daemon=False]
[IPClusterStart] Creating pid file: /Users/veit/.ipython/profile_mpi/pid/ipcluster.
↪pid
[IPClusterStart] Starting Controller with LocalControllerLauncher
[IPClusterStart] Starting 4 Engines with LocalEngineSetLauncher
[IPClusterStart] Engines appear to have started successfully
```

5.5.4 IPython's `Direct`-Interface

Erstellen eines `DirectView`

[1]:
```
import ipyparallel as ipp


rc = ipp.Client()
```

[2]:
```
rc = ipp.Client(profile="default")
```

[3]:
```
rc.ids
```

[3]:
```
[0, 1, 2, 3]
```

Verwenden aller *Engines*:

[4]:
```
dview = rc[:]
```

map()-Funktion

Python's builtin `map()`-Funktion kann auf eine Sequenz von Elementen angewendet werden und üblicherweise einfach zu parallelisieren.

Beachtet, dass die `DirectView`-Version von `map()` kein automatisches Load-Balancing macht. Hierfür müsst ihr ggf. `LoadBalancedView` verwenden.

[5]:
```
serial_result = list(map(lambda x:x**10, range(32)))
```

[6]:
```
parallel_result = dview.map_sync(lambda x: x**10, range(32))
```

[7]:
```
serial_result == parallel_result
```

[7]:
```
True
```

5.5.5 ipyparallel-Magic

[1]:
```
import ipyparallel as ipp


rc = ipp.Client()
```

[2]:
```
with rc[:].sync_imports():
    import numpy
```

```
importing numpy on engine(s)
got unknown result: 05d7cec8-912387fc506b2f40c1471286
got unknown result: fa6fddcf-779a7f98c12df80a46c1e51f
```

[3]:
```
%px a = numpy.random.rand(2,2)
```

[4]:
```
%px numpy.linalg.eigvals(a)
```

```
Out[0:2]: array([0.65462266, 1.13932216])
Out[1:2]: array([ 0.83473139, -0.29322661])
Out[2:2]: array([0.05028953, 1.30003731])
Out[3:2]: array([0.99612347, 0.42574895])
```

[5]:
```
%px print("hi")
```

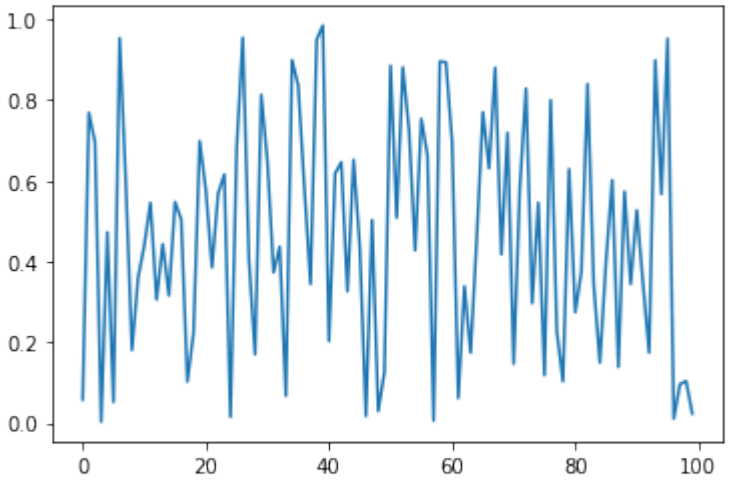
```
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

[6]:
```
%px %pylab inline
```

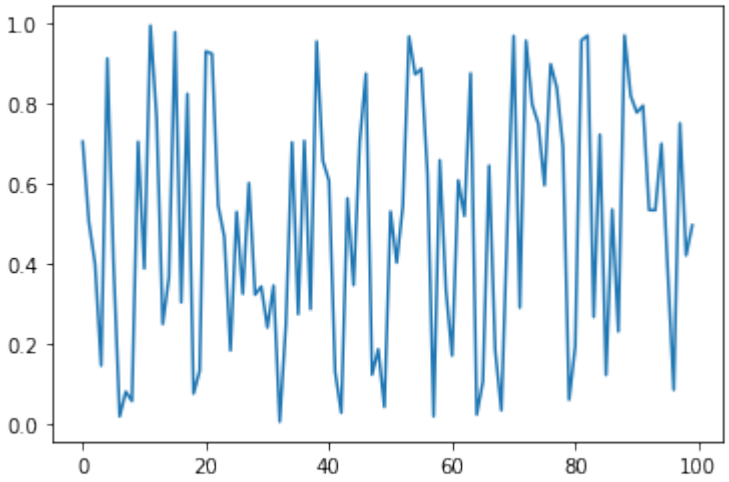
```
[stdout:0] Populating the interactive namespace from numpy and matplotlib
[stdout:1] Populating the interactive namespace from numpy and matplotlib
[stdout:2] Populating the interactive namespace from numpy and matplotlib
[stdout:3] Populating the interactive namespace from numpy and matplotlib
```

[7]:
```
%px plot(rand(100))
```

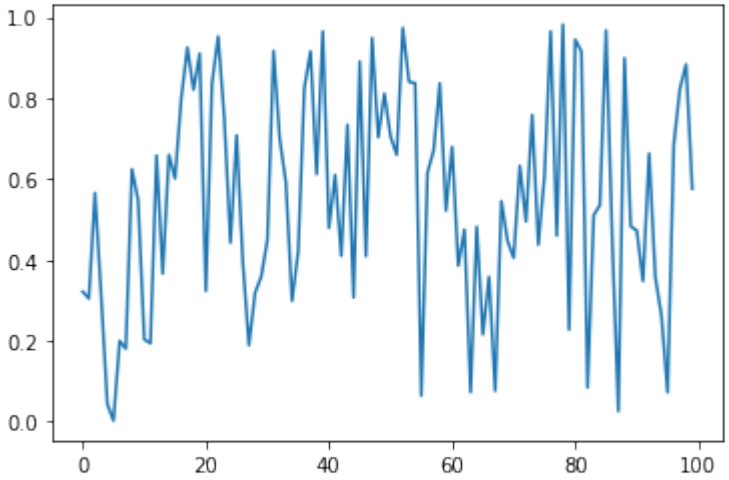
[output:0]



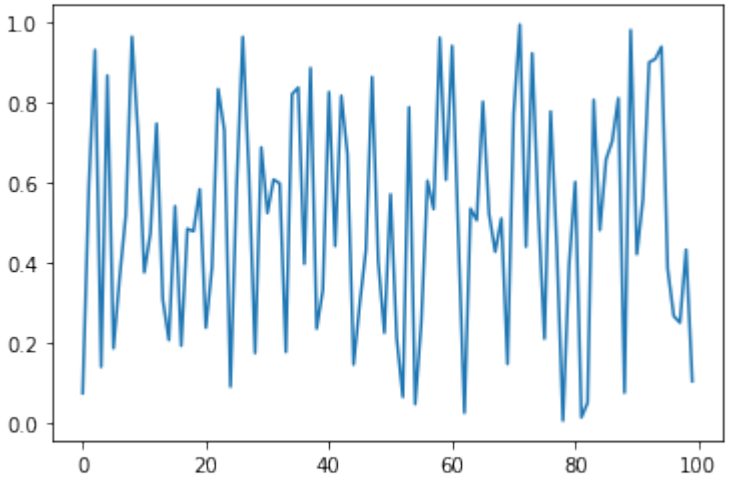
[output:1]



[output:2]



[output:3]



Out[0:5]: [<matplotlib.lines.Line2D at 0x1102e2b10>]

Out[1:5]: [<matplotlib.lines.Line2D at 0x10ea7f110>]

Out[2:5]: [<matplotlib.lines.Line2D at 0x113d70d50>]

Out[3:5]: [<matplotlib.lines.Line2D at 0x10fee7fd0>]

%%px Cell Magic

`--targets`, `--block` und `--noblock`

[8]:
```
%%px --targets ::2
print("I am even")
```

```
[stdout:0] I am even
[stdout:2] I am even
```

[9]:
```
%%px --targets 1
print("I am number 1")
```

```
I am number 1
```

[10]:
```
%%px
print("still all by default")
```

```
[stdout:0] still all by default
[stdout:1] still all by default
[stdout:2] still all by default
[stdout:3] still all by default
```

[11]:
```
%%px --noblock
import time


time.sleep(1)
time.time()
```

[11]:
```
<AsyncResult: execute>
```

[12]:
```
%pxresult
```

```
Out[0:8]: 1570269717.543805
```

```
Out[1:8]: 1570269717.54598
```

```
Out[2:8]: 1570269717.5485692
```

```
Out[3:7]: 1570269717.548548
```

[13]:
```
%%px --block --group-outputs=engine
import numpy as np


A = np.random.random((2,2))
ev = numpy.linalg.eigvals(A)
print(ev)
ev.max()
```

```
[stdout:0] [-0.68780518  0.74951568]
```

```
[output:0]
```

```
Out[0:9]: 0.7495156792444352
```

```
[stdout:1] [0.75691299 1.08526474]
```

```
[output:1]
```

```
Out[1:9]: 1.0852647415435284
```

```
[stdout:2] [-0.2153414   1.46736661]
```

```
[output:2]
```

```
Out[2:9]: 1.467366609797983
```

```
[stdout:3] [0.55698336 0.13540831]
```

```
[output:3]
```

```
Out[3:8]: 0.5569833627025539
```

%pxresult

```
[14]: dview = rc[:]
```

```
[15]: dview.block = False
%px print("hi")
%pxresult
```

```
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

%autopx

```
[16]: dview.block=True
```

```
[17]: %autopx
```

```
%autopx enabled
```

```
[18]: max_evals = []
for i in range(100):
    a = numpy.random.rand(10, 10)
    a = a + a.transpose()
    evals = numpy.linalg.eigvals(a)
    max_evals.append(evals[0].real)
```

```
[19]: print("Average max eigenvalue is: %f" % (sum(max_evals)/len(max_evals)))
```

```
[stdout:0] Average max eigenvalue is: 10.209420
[stdout:1] Average max eigenvalue is: 10.214184
[stdout:2] Average max eigenvalue is: 10.071618
[stdout:3] Average max eigenvalue is: 10.132674
```

%pxconfig

```
[3]: %pxconfig --block
%px print("hi")
```

```
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

[4]:
```
%pxconfig --targets ::2
%px print("hi")
```

```
[stdout:0] hi
[stdout:2] hi
```

[5]:
```
%pxconfig --noblock
%px print("hi")
```

[5]:
```
<AsyncResult: execute>
```

[6]:
```
%pxresult
```

```
[stdout:0] hi
[stdout:2] hi
```

Mehrere aktive Views

Magics von `ipyparallel` sind bestimmten `DirectView`-Objekten zugeordnet. Der aktive View kann jedoch geändert werden mit dem Aufruf der `activate()`-Methode auf einem View.

[3]:
```
even = rc[::2]
even.activate()
%px print("hi")
```

[3]:
```
<AsyncResult: execute>
```

[4]:
```
even.block = True
%px print("hi")
```

```
got unknown result: 98a3ad5c-00fa7f40e334c8a4186d5be5
[stdout:0] hi
[stdout:2] hi
```

Wenn ihr die Anischt aktiviert, könnt ihr auch einen Suffix angeben, so dass dieser bei einer ganzen Reihe von Magics zugeordnet werden kann ohne die bereits vorhandenen zu ersetzen.

[5]:
```
rc.activate()
```

[5]:
```
<DirectView all>
```

[6]:
```
even.activate("_even")
%px print("hi")
```

```
[stdout:0] hi
[stdout:1] hi
[stdout:2] hi
[stdout:3] hi
```

[7]:
```
%px_even print("We aren't odd!")
```

```
[stdout:0] We aren't odd!
[stdout:2] We aren't odd!
```

Dieses Suffix wird am Ende aller Magics angewendet, also z.B.`%autopx_even`, `%pxresult_even` usw.

Der Einfachheit halber hat auch `Client` eine `activate()`-Methode, die einen `DirectView` mit `block = True` erstellt, aktiviert und die neue View zurückgibt.

Die anfänglichen Magics, die beim Erstellen eines Clients registriert werden, sind das Ergebnis des Aufrufs `rc.activate()` mit Standardargumenten.

Engines als Kernel

Engines sind eigentlich dasselbe Objekt wie IPython-Kernels, mit der einzigen Ausnahme, dass Engines eine Verbindung zu einem Controller herstellen, während reguläre Kernels ihre Sockets direkt an Verbindungen zu ihrem Frontend binden.

Manchmal werdet ihr zum Debuggen oder Analysieren euer Frontend direkt mit einer Engine verbinden wollen, um eine direktere Interaktion zu ermöglichen. Auch dies könnt ihr tun, indem ihr die Engine anweist, ihren Kernel auch an euer Frontend bindet:

[ ]:
```
%px import ipyparallel as ipp; ipp.bind_kernel()


%px %qtconsole
```

> **Hinweis:** Seid vorsichtig mit dieser Anweisung, da sie so viele QtConsole startet, wie Engines zur Verfügung stehen.

Alternativ könnet ihr euch die Verbindungsinformationen auch anzeigen lassen und so ermitteln, wie ihr eine Verbindung zu den Engines herstellen könnt, je nachdem, wo sie leben und wo Sie sich befinden:

[ ]:
```
%px %connect_info
```

5.5.6 Task-Interface

Die *Task*-Interface zum Cluster präsentiert die *Engines* als fehlertolerantes, dynamisches LoadBalancing für *Workers*. Im Gegensatz zur *Direct*-Interface gibt es bei der *Task*-Interface keinen direkten Zugriff auf einzelne *Engines*. Indem der IPython-Scheduler die *Worker* zuweist, wird die Schnittstelle einfacher und zugleich leistungsfähiger.

Das Beste ist jedoch, dass beide Schnittstellen gleichzeitig verwendet werden können, um die jeweiligen Stärken zu nutzen. Wenn Berechnungen nicht von früheren Ergebnissen abhängen, ist, ist das *Task*-Interface ideal:

Erstellen einer `LoadBalancedView`-Instanz

[1]:
```
import ipyparallel as ipp
```

[2]:
```
rc = ipp.Client()
```

[3]:
```
rc = ipp.Client(url_file="/srv/jupyter/.ipython/profile_mpi/security/ipcontroller-client.
↪json")
```

[4]:
```
rc = ipp.Client(profile="mpi")
```

[5]:
```
lview = rc.load_balanced_view()
```

`load_balanced_view` ist die Standardansicht.

Siehe auch:

- Views

Schnelle und einfache Parallelität

map()-LoadBalancedView

[6]:
```
lview.block = True
serial_result = map(lambda x: x**10, range(32))
parallel_result = lview.map(lambda x: x**10, range(32))
serial_result == parallel_result
```

[6]:
```
True
```

@lview.parallel()-Decorator

[7]:
```
@lview.parallel()
def f(x):
    return 10.0 * x**4


f.map(range(32))
```

[7]:
```
[0.0,10.0,160.0,...]
```

Abhängigkeiten

Hinweis:

Beachtet, dass der reine ZeroMQ-Scheduler keine Abhängigkeiten unterstützt.

Funktionsabhängigkeiten

`UnmetDependency`

@ipp.require-Decorator

@ipp.depend-Decorator

dependent-Objekt

Dependency

[ ]:
```
client.block=False

ar = lview.apply(f, args, kwargs)
ar2 = lview.apply(f2)

with lview.temp_flags(after=[ar,ar2]):
    ar3 = lview.apply(f3)

with lview.temp_flags(follow=[ar], timeout=2.5):
    ar4 = lview.apply(f3)
```

Hinweis:

Manche parallele Workloads können als Directed acyclic graph (DAG) beschrieben werden. In DAG Dependencies wird anhand eines Beispiels beschrieben, wie NetworkX zur Darstellung der Task-Abhängigkeiten als DAGs dargestellt werden.

ImpossibleDependency

retries und resubmit

Schedulers

[ ]:
```
ipcontroller --scheme=lru
```

Schema	Beschreibung
lru	**Least Recently Used**: Weist die Worker immer der zuletzt verwendeten *Engine* zu. Ähnlich *Round-Robin* berücksichtigt es jedoch nicht die Laufzeit jeder einzelnen Aufgabe.
plainra	**Plain Random**: Wählt zufällig die *Engine* aus, die ausgeführt werden soll.
twobin	**Two-Bin Random**: Benötigt numpy. Wählt zwei *Engines* zufällig aus und verwendt die lru der beiden. Dies ist häufig besser als die rein zufällige Verteilung, erfordert jedoch einen höheren Rechenaufwand.
leastlc	**Least Load**: Standardschema, das die *Engine* immer Aufgaben mit den wenigsten ausstehenden Aufgaben zuweist.
weighte	**Weighted Two-Bin Random**: Gewichtetes **Two-Bin Random**-Schema.

5.5.7 AsyncResult-Objekt

apply() gibt im noblock-Modus ein AsyncResult-Objekt zurück. Dieses erlaubt zu einem späteren Zeitpunkt Anfragen mit der get()-Methode zu den Ergebnissen. Überdies werden in diesem Objekt auch bei der Ausführung anfallende Metadaten gesammelt.

Das AsyncResult-Objekt bietet eine Reihe praktischer Funktionen für die Parallelisierung, die über Pythons multiprocessing.pool.AsyncResult. hinausgehen:

get_dict

AsyncResult.get_dict()

[1]:
```
import os

import ipyparallel as ipp


rc = ipp.Client()
ar = rc[:].apply_async(os.getpid)
pids = ar.get_dict()
rc[:]["pid_map"] = pids
```

Metadaten

Client.metadata

Timing

Iterierbare Map-Ergebnisse

[2]:
```
from __future__ import print_function

import time
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
import ipyparallel as ipp


# create client & view
rc = ipp.Client()
dv = rc[:]
v = rc.load_balanced_view()

# scatter 'id', so id=0,1,2 on engines 0,1,2
dv.scatter("id", rc.ids, flatten=True)
print("Engine IDs: ", dv["id"])

# create a Reference to `id`. This will be a different value on each engine
ref = ipp.Reference("id")
print("sleeping for `id` seconds on each engine")
tic = time.time()
ar = dv.apply(time.sleep, ref)
for i, r in enumerate(ar):
    print("%i: %.3f" % (i, time.time() - tic))


def sleep_here(t):
    import time

    time.sleep(t)
    return id, t


# one call per task
print("running with one call per task")
amr = v.map(sleep_here, [0.01 * t for t in range(100)])
tic = time.time()
for i, r in enumerate(amr):
    print("task %i on engine %i: %.3f" % (i, r[0], time.time() - tic))

print("running with four calls per task")
# with chunksize, we can have four calls per task
amr = v.map(sleep_here, [0.01 * t for t in range(100)], chunksize=4)
tic = time.time()
for i, r in enumerate(amr):
    print("task %i on engine %i: %.3f" % (i, r[0], time.time() - tic))

print("running with two calls per task, with unordered results")
# We can even iterate through faster results first, with ordered=False
amr = v.map(
    sleep_here,
    [0.01 * t for t in range(100, 0, -1)],
    ordered=False,
    chunksize=2,
)
tic = time.time()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
for i, r in enumerate(amr):
    print("slept %.2fs on engine %i: %.3f" % (r[1], r[0], time.time() - tic))
```

```
Engine IDs:  [0, 1, 2]
sleeping for `id` seconds on each engine
0: 0.027
1: 1.022
2: 2.024
running with one call per task
task 0 on engine 2: 0.000
task 1 on engine 1: 0.001
task 2 on engine 0: 0.001
task 3 on engine 2: 0.001
task 4 on engine 1: 0.001
...
slept 0.12s on engine 2: 16.868
slept 0.11s on engine 2: 16.868
slept 0.14s on engine 0: 16.873
slept 0.13s on engine 0: 16.873
slept 0.16s on engine 1: 16.893
...
```

[4]:
```
from functools import reduce
from math import sqrt

import numpy as np


X = np.linspace(0, 100)
add = lambda a, b: a + b
sq = lambda x: x * x
sqrt(reduce(add, map(sq, X)) / len(X))
```

[4]:
```
58.028845747399714
```

1. `map(sq, X)` berechnet das Quadrat jedes Elements in der Liste.
2. `reduce(add, sqX) / len(X)` berechnet den Mittelwert indem die Liste von `AsyncMapResult` summiert und durch die Anzahl dividiert wird.
3. Quadratwurzel der resultierenden Zahl.

   Siehe auch: Wenn ihr die Ergebnisse von `AsyncResult` oder `AsyncMapResult` noch erweitern wollt, könnt ihr dies mit dem `msg_ids`-Attribut. Ein Beispiel hierfür findet ihr unter ipyparallel/docs/source/examples/customresults.py.

5.5.8 MPI

Oft erfordert ein paralleler Algorithmus das Verschieben von Daten zwischen den *Engines*. Eine Möglichkeit besteht darin, Push und Pull über die `DirectView`. Dies ist jedoch langsam, da alle Daten über den *Controller* zum *Client* und dann wieder zurück zum endgültigen Ziel gelangen müssen.

Eine viel bessere Möglichkeit ist die Verwendung des Message Passing Interface (MPI). Die Parallel-Computing-Architektur von IPython wurde von Grund auf für die Integration mit MPI entwickelt. Dieses Notebook gibt eine kurze Einführung in die Verwendung von MPI mit IPython.

Anforderungen

- Eine Standard-MPI-Implementierung wie OpenMPI oder MPICH.

  Für Debian/Ubuntu können diese installiert werden mit

  ```
  $ sudo apt install openmpi-bin
  ```

  oder

  ```
  $ sudo apt install mpich
  ```

  Alternativ können OpenMPI oder MPICH auch mit Spack installiert werden: die Pakete sind `openmpi` oder `mpich`.

- mpi4py

Starten der Engines bei aktiviertem MPI

Automatisches Starten mit `mpiexec` und `ipcluster`

Dies kann z.B. erfolgen mit

```
$ uv run ipcluster start -n 4 --profile=mpi
```

Hierfür muss jedoch zuvor ein entsprechendes Profil angelegt werden; siehe hierfür *Konfiguration*.

Automatisches Starten mit PBS und `ipcluster`

Der `ipcluster`-Befehl bietet auch eine Integration in PBS. Weitere Informationen hierzu erhaltet ihr in Starting IPython Parallel on a traditional cluster.

Beispiel

Die folgende Notebook-Zelle ruft `psum.py` mit folgendem Inhalt auf:

```
import numpy as np

from mpi4py import MPI


def psum(a):
    locsum = np.sum(a)
    rcvBuf = np.array(0.0, "d")
    MPI.COMM_WORLD.Allreduce(
        [locsum, MPI.DOUBLE], [rcvBuf, MPI.DOUBLE], op=MPI.SUM
    )
    return rcvBuf
```

[1]:
```
import ipyparallel as ipp


c = ipp.Client(profile="mpi")
view = c[:]
view.activate()
view.run("psum.py")
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
view.scatter("a", np.arange(16, dtype="float"))
view["a"]
```

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

[2]:
```
%px totalsum = psum(a)
```

[2]:
```
Parallel execution on engines: [0,1,2,3]
```

[3]:
```
view["totalsum"]
```

[3]:
```
[120.0, 120.0, 120.0, 120.0]
```

KAPITEL 6

Binder

Binder bietet eine einfache Möglichkeit, um Computerumgebungen für alle freizugeben. Binder wird verwendet für

Lehre und Ausbildung
: Mit Binder können Links zu interaktiven Datenanalyseumgebungen geteilt werden. Dies eignet sich hervorragend für Workshops, Tutorien und Kurse und ermöglicht euch, Studierende viel schneller mit dem Code vertraut zu machen.

Technische Dokumentation
: Binder-Tools können verwendet werden, um die Dokumentation und Demonstrationen von Tools interaktiv zu gestalten.

Offene Bildungsressourcen
: Binder kann öffentlich zugängliche interaktive Bildungsmaterialien bieten und damit reichhaltigere Erfahrung ermöglichen.

Reproduzierbare wissenschaftliche Analysen
: Binder ermöglicht euch, eine interaktive Umgebung zusammen mit eurem Code und euren Analysen zu teilen. Ihr könnt einen Link freigeben, über den andere eure Arbeit reproduzieren und mit ihr interagieren können. Das Neurolibre-Projekt nutzt z.B. Binder, um neurowissenschaftliche Analysen zu reproduzieren.

Binder bietet einen vollständigen Open-Source-Infrastruktur-Stack. Die wichtigsten Tools sind

`BinderHub`
: stellt den Binder-Dienst in der Cloud bereit

> **Siehe auch**
>
> - Repository
> - Docs
> - Examples

`repo2docker`
: erzeugt reproduzierbare Docker-Images aus einem Git-Repository

Siehe auch

- Repository
- Docs

mybinder.org
: öffentliches BinderHub-Deployment

KAPITEL 7

nbconvert

nbconvert
: konvertiert Notebooks in andere Formate

7.1 Installation

```
$ uv add nbconvert
```

Wichtig

Um alle Funktionen von `nbconvert` nutzen zu können, sind Pandoc und TeX (insbesondere XeLaTeX) erforderlich. Diese müssen separat installiert werden.

7.1.1 Pandoc installieren

`nbconvert` verwendet Pandoc um Markdown in andere Formate als HTML zu konvertieren.

```
$ sudo apt install pandoc
```

```
$ brew install pandoc
```

7.1.2 Tex installieren

Für die Konvertierung in PDF verwendet `nbconvert` das Tex-Ökosystem zur Vorbereitung: Es wird eine `.tex`-Datei erstellt, die von der XeTeX-Engine in ein PDF konvertiert wird.

```
$ sudo apt install texlive-xetex
```

```
$ eval "$(/usr/libexec/path_helper)"
$ brew install --cask mactex
```

Siehe auch

MacTeX

7.2 Verwenden auf der Kommandozeile

```
$ jupyter nbconvert --to FORMAT mynotebook.ipynb
```

latex
: erzeugt eine Datei *NOTEBOOK_NAME*`.tex` und ggf. Bilder als PNG-Dateien in einem Ordner. Mit `--template` kann zwischen einem von zwei Vorlagen ausgewählt werden:

  --template article
  : Standard

    Latex-Artikel, abgeleitet aus dem How-To von Sphinx

  --template report
  : Latex-Bericht mit Inhaltsverzeichnis und Kapiteln

pdf
: erzeugt ein PDF über Latex. Unterstützt die gleichen Vorlagen wie `latex`.

slides
: erstellt Reveal.js-Slides.

script
: konvertiert das Notebook in ein ausführbares Skript. Dies ist der einfachste Weg, ein Python-Skript oder ein Skript in einer anderen Sprache zu erzeugen.

  Bemerkung

  Enthält ein Notebook *Magics*, so können dies möglicherweise nur in einer Jupyter-Session ausgeführt werden.

  Wir können z.B. Python4DataScience/docs/workspace/ipython/mypackage/foo.ipynb in ein Python-Skript verwandeln mit:

```
$ uv run jupyter nbconvert --to script docs/basics/ipython/mypackage/foo.ipynb
```

  Das Ergebnis ist dann `foo.py` mit:

```
#!/usr/bin/env python
# coding: utf-8

# # `foo.ipynb`


# In[1]:
def bar():
    return "bar"


# In[2]:
def has_ip_syntax():
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    listing = get_ipython().getoutput("ls")
    return listing


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

Bemerkung

Um eine Zuordnung von Notebook-Cells zu Slides festzulegen, solltet ihr in *View* → *Cell Toolbar* → *Slideshow* auswählen. Daraufhin wird in jeder Zelle oben rechts ein Menü angezeigt mit den Optionen: *Slide, Sub-Slide, Fragment, Skip, Notes*.

Bemerkung

Für Vortragsnotizen ist eine lokale Kopie von `reveal.js`

erforderlich. Damit `nbconvert` diese findet, kann folgende Option
angegeben werden: `--reveal-prefix` */PATH/TO/REVEAL.JS*.

Weitere Angaben für `FORMAT` sind `asciidoc`, `custom`, `html`, `markdown`, `notebook`, und `rst`.

7.3 nb2xls

nb2xls konvertiert Jupyter-Notebooks in Excel-Dateien `.xlsx`) unter Berücksichtigung von pandas DataFrames und Matplotlib-Ausgaben. Die Eingabezellen werden jedoch nicht konvertiert und Markdown nur zum Teil.

7.4 Eigene Exporter

Siehe auch

Customizing exporters erlaubt euch, eigene Exporter zu schreiben.

KAPITEL 8

nbviewer

nbviewer
: *nbconvert* als Web-Service: Rendert Jupyter Notebooks als statische Webseiten.

8.1 Installation

1. Der Notebook Viewer benötigt mehrere Binärpakete, die auf unserem System installiert werden müssen:

```
$ sudo apt install libmemcached-dev libcurl4-openssl-dev pandoc libevent-dev
```

```
$ brew install libmemcached openssl pandoc libevent
```

1. Anschließend kann der Jupyter Notebook Viewer in einer neuen virtuellen Umgebung installiert werden mit:

   ```
   $ mkdir nbviewer
   $ cd !$
   cd nbviewer
   ```

   Nun kann dann auch `nbviewer` installiert werden:

   ```
   $ uv add nbviewer
   ```

2. Zum Testen kann der Server gestartet werden mit:

   ```
   $ uv run python -m nbviewer --debug --no-cache
   ```

8.2 Erweitern des Notebook-Viewers

Der Notebook-Viewer lässt sich um Provider erweitern, s. (siehe) Extending the Notebook Viewer.

8.3 Zugriffsbeschränkung

Wenn der Viewer als *Service nbviewer erstellen* ausgeführt wird, können nur Benutzer, die sich am JupyterHub authentifiziert haben, auf die Notebooks des `nbviewer` zugreifen.

KAPITEL 9

Kernels

Das Jupyter-Team verwaltet den IPython-Kernel. Zusätzlich zu Python können in Notebooks auch viele andere Sprachen verwendet werden. Dabei sind die folgenden Jupyter-Kernels weit verbreitet:

- R
  - IRKernel: Docs | GitHub
  - IRdisplay: GitHub
  - Repr: GitHub
- Julia
  - IJulia: GitHub
  - Interact.jl: GitHub

Eine Liste verfügbarer Kernel findet ihr unter Jupyter kernels.

Siehe auch

- Using Wolfram Language in Jupyter: A free alternative to Mathematica

9.1 Kernel installieren, anzeigen und starten

9.1.1 Kernel installieren

Kernel werden z.B. in folgenden Verzeichnissen gesucht:

- `/srv/jupyter/.local/share/jupyter/kernels`
- `/usr/local/share/jupyter/kernels`
- `/usr/share/jupyter/kernels`
- `/srv/jupyter/.ipython/kernels`

Um eure neue Umgebung in einem der Verzeichnisse als Jupyter Kernel verfügbar zu machen, solltet ihr ipykernel installieren:

```
$ uv add --dev ipykernel
```

Anschließend könnt ihr euren Kernel registrieren, z.B. mit

```
$ uv run ipython kernel install --user --env VIRTUAL_ENV $(pwd)/.venv --prefix /srv/
↪jupyter/.ipython/kernels --name python311 --display-name 'Python 3.11 Kernel'
```

--user
: installiert den Kernel für den aktuellen Nutzer und nicht systemweit.

--env *ENV VALUE*
: detzt die Umgebungsvariable für den Kernel.

--prefix=*/PATH/TO/KERNEL*
: gibt den Pfad an, in dem der Jupyter-Kernel installiert werden soll.

name *NAME*
: gibt einen Namen für die `kernelspec` an. Dieser wird benötigt, um mehrere IPython-Kernel gleichzeitig verwenden zu können.

display-name *DISPLAY_NAME*
: gibt den Anzeigenamen für die `kernelspec` an.

Mit `ipython kernel install` wird eine `kernelspec`-Datei im JSON-Format für die aktuelle Python-Umgebung erstellt, z.B.:

```
{
 "argv": [
  "/srv/jupyter/.ipython/kernels/python311/.venv/bin/python",
  "-Xfrozen_modules=off",
  "-m",
  "ipykernel_launcher",
  "-f",
  "{connection_file}"
 ],
 "display_name": "project",
 "language": "python",
 "metadata": {
  "debugger": true
 },
 "env": {
  "VIRTUAL_ENV": "/srv/jupyter/.ipython/kernels/python311/.venv"
 }
}
```

argv
: Eine Liste von Befehlszeilenargumenten, die zum Starten des Kernels verwendet werden.

  `connection_file` verweist auf eine Datei, die die IP-Adresse, die Ports und den Authentifizierungsschlüssel enthält, die für die Verbindung benötigt werden. Üblicherweise wird diese JSON-Datei an einem sicheren Ort des aktuellen Profils gespeichert:

  ```
  {
    "shell_port": 61656,
  ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
  "iopub_port": 61657,
  "stdin_port": 61658,
  "control_port": 61659,
  "hb_port": 61660,
  "ip": "127.0.0.1",
  "key": "a0436f6c-1916-498b-8eb9-e81ab9368e84"
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": ""
}
```

display_name
: Der Name des Kernels, wie er im Browser angezeigt werden soll. Im Gegensatz zum in der API verwendeten Kernelnamen kann dieser beliebige Unicode-Zeichen enthalten.

language
: Der Name der Sprache des Kernels. Wenn beim Laden von Notebooks kein passender `kernelspec`-Schlüssel gefunden wird, wird ein Kernel mit einer passenden Sprache verwendet. Auf diese Weise kann ein für ein Python- oder Julia-Kernel geschriebenes Notebook mit dem Python- oder Julia-Kernel des Benutzers verknüpft werden, auch wenn dieser nicht demselben Namen wie der des Autors hat.

interrupt_mode
: Kann entweder `signal` oder `message` sein und gibt an, wie ein Client die Ausführung einer Zelle auf diesem Kernel unterbrechen soll.

  signal
  : sendet ein Interrupt, z.B. `SIGINT` auf *POSIX*-Systemen

  message
  : sendet einen `interrupt_request`, s.a. Kernel Interrupt.

env
: `dict` mit Umgebungsvariablen, die für den Kernel festgelegt werden sollen. Diese werden zu den aktuellen Umgebungsvariablen hinzugefügt, bevor der Kernel gestartet wird.

metadata
: `dict` mit zusätzlichen Attributen zu diesem Kernel. Wird von Clients zur Unterstützung der Kernelauswahl verwendet. Hier hinzugefügte Metadaten sollten einen Namensraum für das Tool zum Lesen und Schreiben dieser Metadaten haben.

9.1.2 Verfügbare Kernel anzeigen

```
$ uv run jupyter kernelspec list
Available kernels:
  mykernel   /Users/veit/Library/Jupyter/kernels/mykernel
  python311  /Users/veit/Library/Jupyter/kernels/python311
  python313  /Users/veit/Library/Jupyter/kernels/python313
```

9.1.3 Kernel starten

```
$ uv run --with jupyter jupyter console --kernel mykernel
Jupyter console 6.6.3

Python 3.13.0 (main, Oct  7 2024, 23:47:22) [Clang 18.1.8 ]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
Type 'copyright', 'credits' or 'license' for more information
IPython 8.29.0 -- An enhanced Interactive Python. Type '?' for help.

In [1]:
```

Mit `ctrl-d` könnt ihr den Kernel wieder beenden.

9.1.4 Kernel löschen

```
$ uv run jupyter kernelspec uninstall mykernel
```

9.1.5 Starndard-Kernel deinstallieren

Sofern noch nicht geschehen, kann eine Konfigurationsdatei erstellt werden, z.B. mit

```
$ uv run jupyter lab --generate-config
```

Anschließend könnt ihr in dieser Konfigurationsdatei folgende Zeile einfügen:

```
c.KernelSpecManager.ensure_native_kernel = False
```

9.2 Was ist neu in Python 3.8?

In Python 3.8 vereinfacht sich die Syntax und auch die Unterstützung für C-Bibliotheken verbessert sich. Im Folgenden erhaltet ihr ein kurzer Überblick über einige der neuen Features. Einen vollständigen Überblick erhaltet ihr in What's New In Python 3.8.

9.2.1 Installation

Überprüfen

[1]:
```
!python3 -V
```

```
Python 3.8.0
```

or

[2]:
```
import sys


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

9.2.2 Assignment Expressions: Walrus operator :=

Bisher wurde z.B. von pip `env_base` folgendermaßen ermittelt:

[3]:
```
import os
```

[4]:
```
def _getuserbase():
    env_base = os.environ.get("PYTHONUSERBASE", None)
    if env_base:
        return env_base
```

Dies kann nun vereinfacht werden mit:

[5]:
```
def _getuserbase():
    if env_base := os.environ.get("PYTHONUSERBASE", None):
        return env_base
```

Auch lassen sich mehrfach verschachtelte `if` vermeiden, wie z.B. in cpython/Lib/copy.py. Aus

[6]:
```
from copyreg import dispatch_table
```

[7]:
```
def copy(x):
    cls = type(x)
    reductor = dispatch_table.get(cls)
    if reductor:
        rv = reductor(x)
    else:
        reductor = getattr(x, "__reduce_ex__", None)
        if reductor:
            rv = reductor(4)
        else:
            reductor = getattr(x, "__reduce__", None)
            if reductor:
                rv = reductor()
            else:
                raise Error(
                    "un(deep)copyable object of type %s" % cls)
```

wird dann:

[8]:
```
def copy(x):
    cls = type(x)
    reductor = dispatch_table.get(cls)
    if reductor := dispatch_table.get(cls):
        rv = reductor(x)
    elif reductor := getattr(x, "__reduce_ex__", None):
        rv = reductor(4)
    elif reductor := getattr(x, "__reduce__", None):
        rv = reductor()
    else:
        raise Error("un(deep)copyable object of type %s" % cls)
```

9.2.3 *Positional-only*-Parameter

In Python 3.8 kann mit / ein Funktionsparameter positionsbezogen angegeben werden. Etliche Python-Funktionen, die in C implementiert sind, erlauben keine Keyword-Argumente. Dieses Verhalten kann nun in Python selbst emuliert werden, z.B. für die pow()-Funktion:

[9]:
```
def pow(x, y, z=None, /):
    "Emulate the built in pow() function"
    r = x ** y
    return r if z is None else r%z
```

9.2.4 `f-strings` unterstützen = für sich selbst dokumentierende Ausdrücke und Debugging

[9]:
```
user = "veit"
member_since = date(2012, 1, 30)
f"{user=} {member_since=}"
```

```
user='veit' member_since=datetime.date(2012, 1, 30)
```

9.2.5 Debug- und Release-Build verwenden dasselbe ABI

Bisher sollte durch Spack ein konsistentes Application Binary Interface (ABI) gewährleistet werden. Dies schloss jedoch nicht die Verwendung von Python im Debug-Build ein. Python 3.8 unterstützt nun auch für Debug Builds die ABI-Kompatibilität. Das `Py_TRACE_REFS`-Makro kann nun gesetzt werden mit der `./configure --with-trace-refs`-Build-Option.

9.2.6 Neue C-API

Mit PEP 587 kommt eine neue C-API zum Konfigurieren der Python-Initialisierung hinzu, die eine genauere Steuerung der gesamten Konfiguration und bessere Fehlerreports bietet.

9.2.7 Vectorcall – ein schnelles Protokoll für CPython

Momentan ist das Protokoll noch nicht vollständig implementiert; dies wird wohl erst mit Python 3.9 kommen. EIne vollständige Beschreibung erhaltet ihr jedoch jetzt bereits in PEP 590.

9.2.8 Update – oder nicht?

Im Folgenden findet ihr eine kurze Übersicht über Probleme, die beim Wechsel zu Python 3.8 auftreten können:

Fehlende Pakete

- opencv-python

Bugs

- Python 3.7.1 wurde 4 Monate nach dem ersten Major Release mit einer langen Liste von Bugfixes veröffentlicht. Ähnliches ist auch bei Python 3.8 zu erwarten.

Syntax

- Die wenigsten Code-Analyse-Werkzeuge und Autoformatter können bereits die Syntax-Änderungen von Python 3.8

Warum dennoch updaten?

Da das Upgrade einige Zeit in Anspruch nehmen wird, kann es verlockend sein, den Wechsel auf unbestimmte Zeit zu verschieben. Warum sollten Sie sich mit Inkompatibilitäten neuer Versionen beschäftigen, wenn eure aktuelle Version zuverlässig funktioniert?

Das Problem ist, dass eure Python nicht auf unbestimmte Zeit unterstützt wird und auch die von euch verwendeten Bibliotheken nicht alle älteren Python-Versionen auf unbestimmte Zeit unterstützen wird. Und je länger ihr ein Update hinauszögert, umso größer und risikoreicher wird es. Daher empfiehlt sich das Update auf die neue Major Version von Python üblicherweise einige Monate nach dem ersten Release.

9.2.9 Portierung

Siehe auch:

- Porting to Python 3.8

9.3 Was ist neu in Python 3.9?

Mit Python 3.9 wird erstmals ein neuer Release-Zyklus verwendet: zukünftig sollen jährlich neue Releases erscheinen (s.a. PEP 602). Die Entwickler*innen erhoffen sich hiervon u.a. schnellere Rückmeldungen zu neuen Features.

Mit dem ersten veröffentlichten Release-Kandidaten soll Python auch eine stabile Binärschnittstelle (engl. *application binary interface,* ABI) erhalten: es soll nun keine ABI-Änderungen mehr in der 3.9-Reihe geben womit Erweiterungsmodule nicht mehr für jede Version neu kompiliert werden müssen.

Weitere Informationen erhaltet ihr in What's New In Python 3.9.

Im Folgenden gebe ich Euch einen kurzen Überblick über einige der neuen Features.

9.3.1 Installation

Überprüfen

[1]:
```
!python3 -V
```

```
Python 3.9.0rc1
```

or

[2]:
```
import sys


assert sys.version_info[:2] >= (3, 9)
```

9.3.2 PEP 584: Dictionary Merge- und Update-Operatoren

Für die built-in dict-Klasse gibt es nun ähnliche Operatoren wie zum Verketten von Listen: Merge (`|`) und Update (`|=`). Damit werden verschiedene Nachteile der bisherigen Methoden `dict.update`, `{**d1, **d2}` und `collections.ChainMap` beseitigt.

Beispiel ipykernel/ipykernel/kernelapp.py

[3]:
```
from IPython.core.application import (  # type:ignore[attr-defined]
    BaseIPythonApplication,
    base_aliases,
    base_flags,
    catch_config_error,
)


kernel_aliases = dict(base_aliases)
kernel_aliases.update(
    {
        "ip": "IPKernelApp.ip",
        "hb": "IPKernelApp.hb_port",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "shell": "IPKernelApp.shell_port",
        "iopub": "IPKernelApp.iopub_port",
        "stdin": "IPKernelApp.stdin_port",
        "control": "IPKernelApp.control_port",
        "f": "IPKernelApp.connection_file",
        "transport": "IPKernelApp.transport",
    }
)

kernel_flags = dict(base_flags)
kernel_flags.update(
    {
        "no-stdout": (
            {"IPKernelApp": {"no_stdout": True}},
            "redirect stdout to the null device",
        ),
        "no-stderr": (
            {"IPKernelApp": {"no_stderr": True}},
            "redirect stderr to the null device",
        ),
        "pylab": (
            {"IPKernelApp": {"pylab": "auto"}},
            """Pre-load matplotlib and numpy for interactive use with
        the default matplotlib backend.""",
        ),
        "trio-loop": (
            {"InteractiveShell": {"trio_loop": False}},
            "Enable Trio as main event loop.",
        ),
    }
)
```

kann vereinfacht werden mit:

[4]:
```
kernel_aliases = base_aliases | {
    "ip": "KernelApp.ip",
    "hb": "KernelApp.hb_port",
    "shell": "KernelApp.shell_port",
    "iopub": "KernelApp.iopub_port",
    "stdin": "KernelApp.stdin_port",
    "parent": "KernelApp.parent",
}
if sys.platform.startswith("win"):
    kernel_aliases["interrupt"] = "KernelApp.interrupt"

kernel_flags = base_flags | {
    "no-stdout": (
        {"KernelApp": {"no_stdout": True}},
        "stdout auf das Nullgerät umleiten",
    ),
    "no-stderr": (
        {"KernelApp": {"no_stderr": True}},
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        "stderr auf das Nullgerät umleiten",
    ),
}
```

Beispiel matplotlib/legend.py

```
[ ]: hm = default_handler_map.copy()
     hm.update(self._custom_handler_map)
     return hm
```

kann vereinfacht werden mit:

```
[ ]: return default_handler_map | self._handler_map
```

9.3.3 PEP 616: `removeprefix()` und `removesuffix()` für String-Methoden

Mit str.removeprefix(prefix) und str.removesuffix(suffix) lassen sich nun einfach Präfixe und Suffixe entfernen. Auch für `bytes`, `bytearray`-Objekte und `collections.UserString`-Zeichenketten wurden ähnliche Methoden hinzugefügt. Insgesamt soll dies zu weniger zerbrechlichem, performanterem und besser lesbarem Code führen.

Beispiel find_recursionlimit.py

```
[ ]: if test_func_name.startswith("test_"):
         print(test_func_name[5:])
     else:
         print(test_func_name)
```

kann vereinfacht werden mit:

```
[ ]: print (test_func_name.removeprefix ("test_"))
```

Beispiel deccheck.py

```
[ ]: if funcname.startswith("context."):
         self.funcname = funcname.replace("context.", "")
         self.contextfunc = True
     else:
         self.funcname = funcname
```

kann vereinfacht werden mit:

```
[ ]: self.contextfunc = funcname.startswith ("context.")
     self.funcname = funcname.removeprefix ("context.")
```

9.3.4 PEP 585: Zusätzliche generische Typen

In *Type Annotations* können nun z.B. `list` oder `dict` als generische Typen direkt verwendet werden – sie müssen nicht mehr extra aus `typing` importiert werden. Das Importieren von `typing` ist damit *deprecated*.

Beispiel

[ ]:
```
def greet_all(names: list[str]) -> None:
    for name in names:
        print("Hello", name)
```

9.3.5 PEP 617: Neuer PEG-Parser

Python 3.9 verwendet nun einen PEG (Parsing Expression Grammar)-Parser anstelle des bisherigen LL-Parser. Dies hat u.a. folgende Vorteile:

- das Parsen abstrakter Syntaxbäume (engl. Abstract Syntax Trees, AST) vereinfacht sich erheblich
- Left recursion wird möglich
- Das Erstellen konkreter Syntaxbäume (engl. Concrete Syntax Trees, CST) wird möglich

Der neue Parser ist damit flexibler und soll vor allem beim Entwerfen neuer Sprachfunktionen genutzt werden. Das ast-Modul verwendet den neuen Parser schon jetzt, ohne dass sich an der Ausgabe etwas geändert hätte.

In Python 3.10 wird der alte Parser und alle davon abhängigen Funktionen – hauptsächlich das veraltete parser-Modul – gelöscht. Nur in Python 3.9 könnt ihr auf der Kommandozeile mit `-X oldparser` oder mit der Umgebungsvariable `PYTHONOLDPARSER=1` zum LL-Parser zurückkehren.

9.3.6 PEP 615: Unterstützung für die IANA-Zeitzonendatenbank in der Standardbibliothek

Das neue zoneinfo-Standardmodul bringt Unterstützung für die IANA-Zeitzonendatenbank in die Standardbibliothek.

[7]:
```
from datetime import datetime, timedelta
from zoneinfo import ZoneInfo
```

Pacific Daylight Time:

[8]:
```
dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
print(dt)
```

```
2020-10-31 12:00:00-07:00
```

[9]:
```
dt.tzname()
```

[9]:
```
'PDT'
```

Pacific Standard Time:

[10]:
```
dt += timedelta(days=7)
print(dt)
```

```
2020-11-07 12:00:00-08:00
```

[11]:
```
print(dt.tzname())
```

```
PST
```

9.4 Was ist neu in Python 3.10

Siehe auch:

- What's New In Python 3.10

[1]:
```
import sys


assert sys.version_info[:2] >= (3, 10)
```

9.4.1 Bessere Fehlermeldungen

Syntaxfehler

- Beim Parsen von Code, der nicht geschlossene Klammern enthält, schließt der Interpreter jetzt die Position der nicht geschlossenen Klammer oder Klammern ein, anstatt `SyntaxError: unexpected EOF` anzuzeigen.
- Vom Interpreter ausgelöste `SyntaxError`-Ausnahmen heben nun den gesamten Fehlerbereich des Ausdrucks hervor, in dem der Syntaxfehler besteht, anstatt nur die Stelle, an der das Problem erkannt wird.
- Es wurden spezialisierte Meldungen für `SyntaxError`-Ausnahmen hinzugefügt, z.B. für
  - fehlende `:` vor Blöcken
  - nicht eingeklammerte Tupel in Comprehensions
  - fehlende Kommas in Auflistungsliteralen und zwischen Ausdrücken
  - fehlende `:` und Werte in Dictionary-Literalen
  - Verwendung von = anstelle von == in Vergleichen
  - Verwendung von * in f-strings

Einrückungsfehler

- Viele `IndentationError` haben jetzt mehr Kontext.

Attribut-Fehler

- `AttributeError` bieten nun Vorschläge für ähnliche Attributnamen in dem Objekt, von dem die Ausnahme ausgelöst wurde.

Name-Fehler

- `NameError` bietet Vorschläge für ähnliche Variablennamen in der Funktion, von der aus die Ausnahme ausgelöst wurde.

9.4.2 Strukturelles Pattern-Matching

Viele funktionale Sprachen haben einen `match`-Ausdruck, z.B. Scala, Rust, F#.

Eine `match`-Anweisung nimmt einen Ausdruck und vergleicht ihn mit aufeinanderfolgenden Mustern, die als ein oder mehrere Fälle angegeben sind. Dies ist oberflächlich gesehen ähnlich wie eine switch-Anweisung in C, Java oder JavaScript, aber viel mächtiger.

match

Die einfachste Form vergleicht einen Wert mit einem oder mehreren Literalen:

[2]:
```
def http_error(status):
    match status:
        case 400:
            return "Bad request"
        case 401:
            return "Unauthorized"
        case 403:
            return "Forbidden"
        case 404:
            return "Not found"
        case 418:
            return "I'm a teapot"
        case _:
            return "Something else"
```

Bemerkung:

Nur in diesem Fall fungiert _ als Platzhalter, der nie versagt, und **nicht** als Variablenname.

Die Fälle prüfen nicht nur auf Gleichheit, sondern binden Variablen neu, die dem angegebenen Muster entsprechen. Zum Beispiel:

[3]:
```
NOT_FOUND = 404
retcode = 200

match retcode:
    case NOT_FOUND:
        print("not found")

print(f"Current value of {NOT_FOUND=}")
```

```
not found
Current value of NOT_FOUND=200
```

> »Wenn diese schlecht durchdachte Funktion wirklich zu Python hinzugefügt wird, verlieren wir ein Prinzip, das ich meinen Studenten immer beigebracht habe: ›Wenn du eine undokumentierte Konstante siehst, kannst du sie immer benennen, ohne die Bedeutung des Codes zu verändern.‹ Das algebraische Substitutionsprinzip? Es gilt dann nicht mehr.« – Brandon Rhodes
>
> «… die Semantik kann ganz anders sein als bei switch. Die Cases prüfen nicht einfach die Gleichheit, sondern binden Variablen neu, die mit dem angegebenen Muster übereinstimmen.» – Jake VanderPlas

Symbolische Konstanten

Muster können benannte Konstanten verwenden. Diese müssen Dotted Names sein, damit sie nicht als Capture-Variable interpretiert werden können:

[4]:
```
from enum import Enum


class Color(Enum):
    RED = 0
    GREEN = 1
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    BLUE = 2


color = Color(2)

match color:
    case color.RED:
        print("I see red!")
    case color.GREEN:
        print("Grass is green")
    case color.BLUE:
        print("I'm feeling the blues :(")
```

```
I'm feeling the blues :(
```

»… „case CONSTANT“ passt eigentlich immer und wird einer Variablen namens CONSTANT zugewiesen« – Armin Ronacher

Siehe auch:

- Structural pattern matching for Python
- PEP 622 – Structural Pattern Matching wurde ersetzt durch
  - PEP 634: Specification
  - PEP 635: Motivation and Rationale
  - PEP 636: Tutorial
- github.com/gvanrossum/patma/
- playground-622.ipynb on binder
- Tobias Kohn: On the Syntax of Pattern Matching in Python

9.5 R-Kernel

1. ZMQ

   Für Ubuntu & Debian:

   ```
   $ sudo apt install libzmq3-dev libcurl4-openssl-dev libssl-dev jupyter-core jupyter-
   ↪client
   ```

2. R-Pakete

   ```
   $ R
   > install.packages(c('crayon', 'pbdZMQ', 'devtools'))
   ...
   --- Please select a CRAN mirror for use in this session ---
   ...
   33: Germany (Münster) [https]
   ...
   Selection: 33
   > devtools::install_github(paste0('IRkernel/', c('repr', 'IRdisplay', 'IRkernel')))
   Downloading GitHub repo IRkernel/repr@master
   ```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
from URL https://api.github.com/repos/IRkernel/repr/zipball/master
...
```

3. Kernel bereitstellen

```
> IRkernel::installspec()
...
[InstallKernelSpec] Installed kernelspec ir in /Users/veit/Library/Jupyter/kernels/
↪ir3.3.3/share/jupyter/kernels/ir
```

Ihr könnt den Kernel auch systemweit bereitstellen:

```
> IRkernel::installspec(user = FALSE)
```

Siehe auch

- IRkernel Installation

KAPITEL 10

ipywidgets

ipywidgets sind interaktive Widgets für Jupyter Notebooks. Sie erweitern Notebooks um die Möglichkeit, dass Nutzer selbst Daten eingeben, Daten manipulieren und die veränderten Ergebnisse sehen können.

10.1 Beispiele

IPython enthält eine Architektur für interaktive Widgets, die Python-Code, der im Kernel ausgeführt wird, und JavaScript/HTML/CSS, die im Browser ausgeführt werden, zusammenfügt. Mit diesen Widgets können Benutzer ihren Code und ihre Daten interaktiv untersuchen.

10.1.1 Interact-Funktion

`ipywidgets.interact` erstellt automatisch User-Interface(UI)-Controls, um Code und Daten interaktiv zu erkunden.

[1]:
```
from __future__ import print_function

import ipywidgets as widgets

from ipywidgets import fixed, interact, interact_manual, interactive
```

Im einfachsten Fall generiert `interact` automatisch Steuerelemente für Funktionsargumente und ruft dann die Funktion mit diesen Argumenten auf, wenn Sie die Steuerelemente interaktiv bearbeiten. Im folgenden eine Funktion, die ihr einziges Argument `x` ausgibt.

[2]:
```
def f(x):
    return x
```

Slider

Wenn ihr eine Funktion mit einem ganzzahligen *keyword argument* (`x=10`) angebt, wird ein Schieberegler generiert und an den Funktionsparameter gebunden:

[3]:
```
interact(f, x=10);
```

```
interactive(children=(IntSlider(value=10, description='x', max=30, min=-10), Output()), _
↪dom_classes=('widget-...
```

Checkbox

Wenn ihr `True` oder `False` angebt, generiert `interact` eine Checkbox:

[4]:
```
interact(f, x=True);
```

```
interactive(children=(Checkbox(value=True, description='x'), Output()), _dom_classes=(
↪'widget-interact',))
```

Textbereich

Wenn ihr einen String übergebt, generiert `interact` einen Textbereich:

[5]:
```
interact(f, x="Hi Pythonistas!")
```

```
interactive(children=(Text(value='Hi Pythonistas!', description='x'), Output()), _dom_
↪classes=('widget-interac...
```

[5]:
```
<function __main__.f(x)>
```

10.1.2 Decorator

`interact` kann auch als Decorator verwendet werden. Auf diese Weise könnt ihr eine Funktion definieren und in einer einzigen Einstellung damit interagieren. Wie das folgende Beispiel zeigt, funktioniert `interact` auch mit Funktionen, die mehrere Argumente haben:

[6]:
```
@interact(x=True, y=1.0)
def g(x, y):
    return (x, y)
```

```
interactive(children=(Checkbox(value=True, description='x'), FloatSlider(value=1.0,␣
↪description='y', max=3.0, ...
```

10.2 Widget-Liste

[1]:
```
import ipywidgets as widgets
```

10.2.1 Numerische Widgets

Es gibt eine Vielzahl von IPython-Widgets, die numerische Werte anzeigen sollen. Dabei haben die Ganzzahl-Widgets ein ähnliches Benennungsschema wie ihre Gegenstücke mit Gleitkommazahlen. Durch Ersetzen von `Float` durch `Int` im Widget-Namen könnt ihr das jeweilige Integer-Äquivalent finden.

IntSlider

[2]:
```
widgets.IntSlider(
    value=7,
    min=0,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    max=10,
    step=1,
    description="Test:",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
    readout=True,
    readout_format="d",
)
```

```
IntSlider(value=7, continuous_update=False, description='Test:', max=10)
```

FloatSlider

[3]:
```
widgets.FloatSlider(
    value=7.5,
    min=0,
    max=10.0,
    step=0.1,
    description="Test:",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
    readout=True,
    readout_format=".1f",
)
```

```
FloatSlider(value=7.5, continuous_update=False, description='Test:', max=10.0, readout_
↪format='.1f')
```

Sliders can also be **displayed vertically**.

[4]:
```
widgets.FloatSlider(
    value=7.5,
    min=0,
    max=10.0,
    step=0.1,
    description="Test:",
    disabled=False,
    continuous_update=False,
    orientation="vertical",
    readout=True,
    readout_format=".1f",
)
```

```
FloatSlider(value=7.5, continuous_update=False, description='Test:', max=10.0,␣
↪orientation='vertical', readout...
```

FloatLogSlider

Der `FloatLogSlider` verfügt über eine Skala, die es einfach macht, einen Schieberegler für einen großen Bereich positiver Zahlen zu verwenden. `min` und `max` beziehen sich auf die minimalen und maximalen Exponenten der Basis und der `value` bezieht sich auf den tatsächlichen Wert des Schiebereglers.

[5]:
```
widgets.FloatLogSlider(
    value=10,
    base=10,
    min=-10,  # max exponent of base
    max=10,  # min exponent of base
    step=0.2,  # exponent step
    description="Log Slider",
)
```

```
FloatLogSlider(value=10.0, description='Log Slider', max=10.0, min=-10.0, step=0.2)
```

IntRangeSlider

[6]:
```
widgets.IntRangeSlider(
    value=[5, 7],
    min=0,
    max=10,
    step=1,
    description="Test:",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
    readout=True,
    readout_format="d",
)
```

```
IntRangeSlider(value=(5, 7), continuous_update=False, description='Test:', max=10)
```

FloatRangeSlider

[7]:
```
widgets.FloatRangeSlider(
    value=[5, 7.5],
    min=0,
    max=10.0,
    step=0.1,
    description="Test:",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
    readout=True,
    readout_format=".1f",
)
```

```
FloatRangeSlider(value=(5.0, 7.5), continuous_update=False, description='Test:', max=10.
↪0, readout_format='.1f...
```

IntProgress

[8]:
```
widgets.IntProgress(
    value=7,
    min=0,
    max=10,
    step=1,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    description="Loading:",
    bar_style="",  # 'success', 'info', 'warning', 'danger' or ''
    orientation="horizontal",
)
```

```
IntProgress(value=7, description='Loading:', max=10)
```

FloatProgress

[9]:
```
widgets.FloatProgress(
    value=7.5,
    min=0,
    max=10.0,
    step=0.1,
    description="Loading:",
    bar_style="info",
    orientation="horizontal",
)
```

```
FloatProgress(value=7.5, bar_style='info', description='Loading:', max=10.0)
```

The numerical text boxes that impose some limit on the data (range, integer-only) impose that restriction when the user presses enter.

BoundedIntText

[10]:
```
widgets.BoundedIntText(
    value=7,
    min=0,
    max=10,
    step=1,
    description="Text:",
    disabled=False
)
```

```
BoundedIntText(value=7, description='Text:', max=10)
```

BoundedFloatText

[11]:
```
widgets.BoundedFloatText(
    value=7.5,
    min=0,
    max=10.0,
    step=0.1,
    description="Text:",
    disabled=False
)
```

```
BoundedFloatText(value=7.5, description='Text:', max=10.0, step=0.1)
```

IntText

[12]:
```
widgets.IntText(
    value=7,
    description="Any:",
    disabled=False
)
```

```
IntText(value=7, description='Any:')
```

FloatText

[13]:
```
widgets.FloatText(
    value=7.5,
    description="Any:",
    disabled=False
)
```

```
FloatText(value=7.5, description='Any:')
```

10.2.2 Boolesche Widgets

Es gibt drei Widgets, die boolesche Wert anzeigen sollen.

ToggleButton

[14]:
```
widgets.ToggleButton(
    value=False,
    description="Click me",
    disabled=False,
    button_style="",  # "success", "info", "warning", "danger" or ""
    tooltip="Description",
    icon="check",
)
```

```
ToggleButton(value=False, description='Click me', icon='check', tooltip='Description')
```

Checkbox

[15]:
```
widgets.Checkbox(
    value=False,
    description="Check me",
    disabled=False
)
```

```
Checkbox(value=False, description='Check me')
```

Valid

Das `Valid`-Widget bietet eine read-only-Anzeige.

[16]:
```
widgets.Valid(
    value=False,
    description="Valid!",
)
```

```
Valid(value=False, description='Valid!')
```

10.2.3 Auswahl-Widgets

Es gibt mehrere Widgets zum Auswählen einzelner Werte und zwei für mehrere Werte. Alle erben von derselben Basisklasse.

Dropdown

[17]:
```
widgets.Dropdown(
    options=["1", "2", "3"],
    value="2",
    description="Number:",
    disabled=False,
)
```

```
Dropdown(description='Number:', index=1, options=('1', '2', '3'), value='2')
```

RadioButtons

[18]:
```
widgets.RadioButtons(
    options=["pepperoni", "pineapple", "anchovies"],
    value='pineapple',
    description="Pizza topping:",
    disabled=False,
)
```

```
RadioButtons(description='Pizza topping:', index=1, options=('pepperoni', 'pineapple',
↪'anchovies'), value='pi...
```

Select

[19]:
```
widgets.Select(
    options=["Linux", "Windows", "OSX"],
    value="OSX",
    rows=3,
    description="OS:",
    disabled=False,
)
```

```
Select(description='OS:', index=2, options=('Linux', 'Windows', 'OSX'), rows=3, value=
↪'OSX')
```

SelectionSlider

[20]:
```
widgets.SelectionSlider(
    options=["scrambled", "sunny side up", "poached", "over easy"],
    value="sunny side up",
    description="I like my eggs ...",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    readout=True,
)
```

```
SelectionSlider(continuous_update=False, description='I like my eggs ...', index=1,␣
↪options=('scrambled', 'sunny...
```

SelectionRangeSlider

`index` ist ein Tupel der Mindest- und Höchstwerte.

[21]:
```
import datetime


dates = [datetime.date(2015, i, 1) for i in range(1, 13)]
options = [(i.strftime("%b"), i) for i in dates]
widgets.SelectionRangeSlider(
    options=options, index=(0, 11),
    description="Months (2015)",
    disabled=False
)
```

```
SelectionRangeSlider(description='Months (2015)', index=(0, 11), options=(('Jan',␣
↪datetime.date(2015, 1, 1)), ...
```

ToggleButtons

[22]:
```
widgets.ToggleButtons(
    options=["Slow", "Regular", "Fast"],
    description="Speed:",
    disabled=False,
    button_style="",  # "success", "info", "warning", "danger" or ""
    tooltips=[
        "Description of slow",
        "Description of regular",
        "Description of fast",
    ],
    # icons=['check'] * 2
)
```

```
ToggleButtons(description='Speed:', options=('Slow', 'Regular', 'Fast'), tooltips=(
↪'Description of slow', 'Des...
```

SelectMultiple

Mehrere Werte können ausgewählt werden bei den gedrückten Tasten shift und/oder ctrl (oder command) und Mausklicks oder Pfeiltasten.

[23]:
```
widgets.SelectMultiple(
    options=["Apples", "Oranges", "Pears"],
    value=["Oranges"],
    rows=3,
    description="Fruits",
    disabled=False,
)
```

```
SelectMultiple(description='Fruits', index=(1,), options=('Apples', 'Oranges', 'Pears'),␣
↪rows=3, value=('Orang...
```

10.2.4 String-Widgets

Es gibt mehrere Widgets, die zum Anzeigen von Strings verwendet werden können. Die Widgets `Text` und `Textarea` akzeptieren Eingaben; die Widgets `HTML` und `HTMLMath` zeigen einen String als HTML an (`HTMLMath` rendert auch mathematische Formeln).

Text

[24]:
```
widgets.Text(
    value="Hello World",
    placeholder="Type something",
    description="String:",
    disabled=False,
)
```

```
Text(value='Hello World', description='String:', placeholder='Type something')
```

Textarea

[25]:
```
widgets.Textarea(
    value="Hello World",
    placeholder="Type something",
    description="String:",
    disabled=False,
)
```

```
Textarea(value='Hello World', description='String:', placeholder='Type something')
```

Label

Das `Label`-Widget ist nützlich für benutzerdefinierte Beschreibungen, die einen ähnlichen Stil wie die integrierten Beschreibungen haben.

[26]:
```
widgets.HBox(
    [widgets.Label(value="The $m$ in $E=mc^2$:"), widgets.FloatSlider()]
)
```

```
HBox(children=(Label(value='The $m$ in $E=mc^2$:'), FloatSlider(value=0.0)))
```

HTML

[27]:
```
widgets.HTML(
    value="Hello <b>World</b>",
    placeholder="Some HTML",
    description="Some HTML",
)
```

```
HTML(value='Hello <b>World</b>', description='Some HTML', placeholder='Some HTML')
```

HTML Math

[28]:
```
widgets.HTMLMath(
    value=r"Some math and <i>HTML</i>: \(x^2\) and $$\frac{x+1}{x-1}$$",
    placeholder="Some HTML",
    description="Some HTML",
)
```

```
HTMLMath(value='Some math and <i>HTML</i>: \\(x^2\\) and $$\\frac{x+1}{x-1}$$',␣
↪description='Some HTML', place...
```

10.2.5 Image

[29]:
```
file = open("smiley.gif", "rb")
image = file.read()
widgets.Image(
    value=image,
    format="gif",
    width=128,
    height=128,
)
```

```
Image(value=b'GIF89a\x1e\x01\x1e\x01\xc4\x1f\x00c\x8d\xff\xea\xea\xea\xc7\xc7\xc7\x00\
↪x00\x00\xa0H\x00\xa6\xa6...
```

10.2.6 Button

[30]:
```
widgets.Button(
    description="Click me",
    disabled=False,
    button_style="",  # "success", "info", "warning", "danger" or ""
    tooltip="Click me",
    icon="check",
)
```

```
Button(description='Click me', icon='check', style=ButtonStyle(), tooltip='Click me')
```

10.2.7 Output

Das `Output`-Widget kann `stdout`, `stderr` und IPython.display erfassen und anzeigen.

Ihr könnt die Ausgabe sowohl an ein Output-Widget anhängen oder programmatisch löschen.

[31]:
```
out = widgets.Output(layout={"border": "1px solid black"})
```

[32]:
```
with out:
    for i in range(5):
        print(i, "Hello world!")
```

[33]:
```
from IPython.display import YouTubeVideo


with out:
    display(YouTubeVideo("eWzY2nGfkXk"))
```

[34]:
```
out
```

```
Output(layout=Layout(border='1px solid black'))
```

[35]:
```
out.clear_output()
```

Wir können Ausgaben auch direkt anhängen mit den Methoden `append_stdout`, `append_stderr` oder `append_display_data`.

[36]:
```
out = widgets.Output(layout={"border": "1px solid black"})
out.append_stdout("Output appended with append_stdout")
out.append_display_data(YouTubeVideo("eWzY2nGfkXk"))
out
```

```
Output(layout=Layout(border='1px solid black'), outputs=({'output_type': 'stream', 'name
↪': 'stdout', 'text': '...
```

Eine ausführliche Dokumentation findet ihr in Output widgets.

10.2.8 Play/Animation-Widget

Das `Play`-Widget ist nützlich, um Animationen auszuführen, die in einer bestimmten Geschwindigkeit durchlaufen werden sollen. Im folgenden Beispiel ist ein Slider mit dem Player verknüpft.

[37]:
```
play = widgets.Play(
    interval=10,
    value=50,
    min=0,
    max=100,
    step=1,
    description="Press play",
    disabled=False,
)
slider = widgets.IntSlider()
widgets.jslink((play, "value"), (slider, "value"))
widgets.HBox([play, slider])
```

```
HBox(children=(Play(value=50, description='Press play', interval=10),␣
↪IntSlider(value=0)))
```

10.2.9 DatePicker

Das Datumsauswahl-Widget funktioniert in Chrome, Firefox und IE Edge, derzeit jedoch nicht in Safari, da dieser `input type="date"` nicht unterstützt.

[38]:
```
widgets.DatePicker(
    description="Pick a Date",
    disabled=False
)
```

```
DatePicker(value=None, description='Pick a Date')
```

10.2.10 Color picker

[39]:
```
widgets.ColorPicker(
    concise=False,
    description="Pick a color",
    value="blue",
    disabled=False
)
```

```
ColorPicker(value='blue', description='Pick a color')
```

10.2.11 Controller

`Controller` ermöglicht die Verwendung eines Game-Controller als Eingabegerät.

[40]:
```
widgets.Controller(
    index=0,
)
```

```
Controller()
```

10.2.12 Container/Layout-Widgets

Diese Widgets werden zum Speichern anderer Widgets verwendet, die als `children` bezeichnet werden.

Box

[41]:
```
items = [widgets.Label(str(i)) for i in range(4)]
widgets.Box(items)
```

```
Box(children=(Label(value='0'), Label(value='1'), Label(value='2'), Label(value='3')))
```

HBox

[42]:
```
items = [widgets.Label(str(i)) for i in range(4)]
widgets.HBox(items)
```

```
HBox(children=(Label(value='0'), Label(value='1'), Label(value='2'), Label(value='3')))
```

VBox

[43]:
```
items = [widgets.Label(str(i)) for i in range(4)]
left_box = widgets.VBox([items[0], items[1]])
right_box = widgets.VBox([items[2], items[3]])
widgets.HBox([left_box, right_box])
```

```
HBox(children=(VBox(children=(Label(value='0'), Label(value='1'))),␣
↪VBox(children=(Label(value='2'), Label(val...
```

Accordion

[44]:
```
accordion = widgets.Accordion(children=[widgets.IntSlider(), widgets.Text()])
accordion.set_title(0, "Slider")
accordion.set_title(1, "Text")
accordion
```

```
Accordion(children=(IntSlider(value=0), Text(value='')), _titles={'0': 'Slider', '1':
↪'Text'})
```

Tabs

In this example the children are set after the tab is created. Titles for the tabes are set in the same way they are for `Accordion`.

[45]:
```
tab_contents = ["P0", "P1", "P2", "P3", "P4"]
children = [widgets.Text(description=name) for name in tab_contents]
tab = widgets.Tab()
tab.children = children
for i in range(len(children)):
    tab.set_title(i, str(i))
tab
```

```
Tab(children=(Text(value='', description='P0'), Text(value='', description='P1'),␣
↪Text(value='', description='...
```

`Accordion` und `Tab`

Im Gegensatz zu den anderen Widgets, die zuvor beschrieben wurden, aktualisieren die Container-Widgets `Accordion` und `Tab` ihr Attribut `selected_index`, wenn der Benutzer das Akkordeon oder den Tab ändert; neben der Nutzereingabe kann der `selected_index` auch programmatisch festgelegt werden.

Wenn `selected_index = None` gewählt wird, werden alle Akkordeons geschlossen oder die Auswahl aller Tabs aufgehoben.

In den folgenden *Notebook Cells* wird der Wert von `selected_index“ des Tab und/oder Akkordeon angezeigt.

[46]:
```
tab.selected_index = 3
```

[47]:
```
accordion.selected_index = None
```

Verschachtelung von Tabs und Akkordeons

Tabs und Akkordeons können so tief verschachtelt werden, wie ihr möchtet. Das folgende Beispiel zeigt einige Tabs mit einem Akkordeon als `children`.

[48]:
```
tab_nest = widgets.Tab()
tab_nest.children = [accordion, accordion]
tab_nest.set_title(0, "An accordion")
tab_nest.set_title(1, "Copy of the accordion")
tab_nest
```

```
Tab(children=(Accordion(children=(IntSlider(value=0), Text(value='')), selected_
↪index=None, _titles={'0': 'Sli...
```

10.3 Widget Events

10.3.1 Special Events

[1]:
```
from __future__ import print_function
```

`Button` kann nicht zur Darstellung eines Datentyps verwendet werden, sondern nur für `on_click`. Mit der `print`-Funktion kann der Docstring von `on_click` ausgegeben werden.

[2]:
```
import ipywidgets as widgets


print(widgets.Button.on_click.__doc__)
```

```
Register a callback to execute when the button is clicked.

        The callback will be called with one argument, the clicked button
        widget instance.

        Parameters
        ----------
        remove: bool (optional)
            Set to true to remove the callback from the list of callbacks.
```

Beispiele

Button-Klicks sind *stateless*, d.h. sie übertragen Nachrichten vom Frontend zum Backend. Wenn ihr die `on_click`-Methode verwendet, wird ein Button angezeigt, der die Nachricht ausgibt, sobald auf sie geklickt wurde.

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


button = widgets.Button(description="Click Me!")
display(button)


def on_button_clicked(b):
    print("Button clicked.")


button.on_click(on_button_clicked)
```

```
Button(description='Click Me!', style=ButtonStyle())
```

10.3.2 Traitlet events

Widget-Properties sind IPython-Traitlets. Um Änderungen vorzunehmen, kann die Methode `observe` des Widgets zum Registrieren eines callback verwendet werden. Den Docstring für `observe` seht ihr unten.

Weitere Informationen erhaltet ihr unter Traitlet events.

[4]:
```
print(widgets.Widget.observe.__doc__)
```

```
Setup a handler to be called when a trait changes.

        This is used to setup dynamic notifications of trait changes.

        Parameters
        ----------
        handler : callable
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            A callable that is called when a trait changes. Its
            signature should be ``handler(change)``, where ``change`` is a
            dictionary. The change dictionary at least holds a 'type' key.
            * ``type``: the type of notification.
            Other keys may be passed depending on the value of 'type'. In the
            case where type is 'change', we also have the following keys:
            * ``owner`` : the HasTraits instance
            * ``old`` : the old value of the modified trait attribute
            * ``new`` : the new value of the modified trait attribute
            * ``name`` : the name of the modified trait attribute.
        names : list, str, All
            If names is All, the handler will apply to all traits.  If a list
            of str, handler will apply to all names in the list.  If a
            str, the handler will apply just to that name.
        type : str, All (default: 'change')
            The type of notification to filter by. If equal to All, then all
            notifications are passed to the observe handler.
```

10.3.3 Linking Widgets

Für die Verknüpfung von Widget-Attributen könnt ihr diese einfach miteinander verknüpfen.

Linking Traitlet-Attribute im Kernel

[5]:
```
caption = widgets.Label(
    value="The values of slider1 and slider2 are synchronized"
)
sliders1, slider2 = widgets.IntSlider(description="Slider 1"),\
                    widgets.IntSlider(description="Slider 2")
l = widgets.link((sliders1, "value"), (slider2, "value"))
display(caption, sliders1, slider2)
```

```
Label(value='The values of slider1 and slider2 are synchronized')
```

```
IntSlider(value=0, description='Slider 1')
```

```
IntSlider(value=0, description='Slider 2')
```

Linking Widgets-Attribute auf der Client-Seite

Beim Synchronisieren von Traitlet-Attributen tritt möglicherweise eine Verzögerung aufgrund der Kommunikation mit dem Server auf. Ihr könnt jedoch die Widget-Attribute auch direkt im Browser mit den Link-Widgets verknüpfen. Dabei bleiben die Javascript-Links mit `jslink` auch bestehen, wenn Widgets in HTML-Webseiten ohne Kernel eingebettet werden.

[6]:
```
caption = widgets.Label(
    value="The values of range1 and range2 are synchronized"
)
range1, range2 = widgets.IntSlider(description="Range 1"),\
                 widgets.IntSlider(description="Range 2")
l = widgets.jslink((range1, "value"), (range2, "value"))
display(caption, range1, range2)
```

```
Label(value='The values of range1 and range2 are synchronized')
```

```
IntSlider(value=0, description='Range 1')
```

```
IntSlider(value=0, description='Range 2')
```

[7]:
```
caption = widgets.Label(
    value="Changes in source_range values are reflected in target_range1"
)
source_range, target_range1 = widgets.IntSlider(description="Source range"),\
                              widgets.IntSlider(description="Target range 1")
dl = widgets.jsdlink((source_range, "value"), (target_range1, "value"))
display(caption, source_range, target_range1)
```

```
Label(value='Changes in source_range values are reflected in target_range1')
```

```
IntSlider(value=0, description='Source range')
```

```
IntSlider(value=0, description='Target range 1')
```

10.3.4 Kontinuierliche Updates

Einige Widgets bieten ein `continuous_update`-Attribut mit der Möglichkeit, Werte kontinuierlich zu aktualisieren. Im folgenden Beispiel sehen wir, dass die *delayed Controls* ihren Wert erst übertragen, nachdem der Benutzer den Schieberegler gezogen oder das Textfeld gesendet hat. Die *continuous slides* übertragen ihre Werte kontinuierlich, sobald sie geändert werden.

[8]:
```
a = widgets.IntSlider(description="Delayed", continuous_update=False)
b = widgets.IntText(description="Delayed", continuous_update=False)
c = widgets.IntSlider(description="Continuous", continuous_update=True)
d = widgets.IntText(description="Continuous", continuous_update=True)

widgets.link((a, "value"), (b, "value"))
widgets.link((a, "value"), (c, "value"))
widgets.link((a, "value"), (d, "value"))
widgets.VBox([a, b, c, d])
```

```
VBox(children=(IntSlider(value=0, continuous_update=False, description='Delayed'),␣
↪IntText(value=0, descriptio...
```

10.4 Benutzerdefiniertes Widget

Das Widget-Framework basiert auf dem Comms-Framework, das dem Kernel ermöglicht, JSON an das Frontend zu senden und zu empfangen. Um nun ein benutzerdefiniertes Widget zu erstellen, muss das Widget sowohl im Browser als auch im Python-Kernel definiert werden.

Weitere Informationen zum Comms-Framework erhaltet ihr im Low Level Widget Explanation.

10.4.1 Python-Kernel

DOMWidget

Um ein Widget zu definieren, muss es von der Basisklasse `Widget` oder `DOMWidget` erben. Wenn das Widget im Jupyter-Notebook angezeigt werden soll, sollte euer Widget von `DOMWidget` erben. Dabei erbt Die `DOMWidget`-Klasse selbst von der `Widget`-Klasse.

`_view_name`

Durch die Übernahme von `DOMWidget` wird dem Widget-Framework **nicht** mitgeteilt, welches Frontend-Widget mit dem Backend-Widget verknüpft werden soll.

Stattdessen müsst ihr dies selbst angeben durch eines der folgenden Attribute:

- `_view_name`
- `_view_module`
- `_view_module_version`

und gegebenenfalls

- `_model_name`
- `_model_module`

[1]:
```
import ipywidgets as widgets

from traitlets import Unicode, validate


class HelloWidget(widgets.DOMWidget):
    _view_name = Unicode("HelloView").tag(sync=True)
    _view_module = Unicode("hello").tag(sync=True)
    _view_module_version = Unicode("0.1.0").tag(sync=True)
```

`sync=True`-Traitlets

Traitlets ist ein Framework, mit dem Python-Klassen Attribute mit Typprüfung, dynamisch berechneten Standardwerten und Callbacks bei Änderung haben können. Das `sync=True`- Keyword-Argument weist das Widget-Framework an, den Wert mit dem Browser zu synchronisieren; ohne würde der Browser nichts von `_view_name` oder `_view_module` erfahren.

10.4.2 Frontend (JavaScript)

Models und *Views*

Das Frontend des IPython-Widget-Frameworks hängt stark von Backbone.js ab. *Backbone.js* ist ein MVC-Framework (Model View Controller), das im Backend definierte Widgets automatisch mit generischen *Backbone.js*-Modellen im Frontend synchronisiert: das vorher definierte `_view_name`-Merkmal wird vom Widget-Framework verwendet, um die entsprechende *Backbone.js-View* zu erstellen und diese mit dem *Model* zu verknüpfen.

`@jupyter-widgets/base` importieren

Ihr müsst zuerst das `@jupyter-widgets/base`-Modul mit der `define`-Methode von RequireJS importieren:

[2]:
```
%%javascript
define('hello', ["@jupyter-widgets/base"], function(widgets) {

});
```

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

View definieren

Als nächstes definieren wir die Widget-*View*-Klasse wobei wir von `DOMWidgetView` mit der `.extend`-Methode erben.

[3]:
```
%%javascript
require.undef('hello');

define('hello', ["@jupyter-widgets/base"], function(widgets) {
    // Define the HelloView
    var HelloView = widgets.DOMWidgetView.extend({});
    return {
        HelloView: HelloView
    }
});
```

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

`render`-Methode

Zum Schluss müssen wir noch die Basismethode `render` überschreiben um eine benutzerdefinierte Rendering-Logik zu definieren. Ein Handle auf das Standard-DOM-Element des Widgets kann mit `this.el` aufgerufen werden. Die `el`-Eigenschaft ist das DOM-Element, das der Ansicht zugeordnet ist.

[4]:
```
%%javascript
require.undef('hello');

define('hello', ["@jupyter-widgets/base"], function(widgets) {
    var HelloView = widgets.DOMWidgetView.extend({
        // Render the view.
        render: function() {
            this.el.textContent = 'Hello World!';
        },
    });
    return {
        HelloView: HelloView
    };
});
```

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

10.4.3 Test

Das Widget lässt sich jetzt wie jedes andere Widget anzeigen mit

[5]:
```
HelloWidget()
```

```
HelloWidget()
```

10.4.4 *Stateful* Widget

Mit dem obigen Beispiel könnt ihr noch nicht viel tun. Um dies zu ändern, müsst ihr das Widget *stateful* machen. Anstelle einer statischen *Hello World!*-Meldung soll eine vom Backend festgelegter *String* angezeigt werden. Hierzu wird zunächst ein neues Traitlet hinzugefügt. Verwendet hierbei den Namen von `value`, um mit dem Rest des Widget-Frameworks konsistent zu bleiben und die Verwendung eures Widgets mit Interaktion zu ermöglichen.

10.4.5 Jupyter Widgets aus einem Template erstellen

Mit widget-cookiecutter ist ein Cookiecutter-Template verfügbar. Es enthält eine Implementierung für ein Platzhalter-Widget *Hello World*. Darüberhinaus erleichtert es euch das Packen und Verteilen eurer Jupyter Widgets.

10.5 ipywidgets-Bibliotheken

Beliebte Widget-Bibliotheken sind

qplot
: 2-D Plotting-Bibliothek für Jupyter-Notebooks

  - bqplot

ipycanvas
: Interaktive Canvas-Elemente in Jupyter-Notebooks

10.5.1 ipycanvas

stellt die Web-Canvas-API zur Verfügung. Es gibt jedoch einige Unterschiede:

- Das Canvas-Widget macht die CanvasRenderingContext2D-API direkt verfügbar
- Die gesamte API ist in `snake_case` anstatt in `camelCase` geschrieben, sodass beispielsweise das in Python geschriebene `canvas.fillStyle = "red"` in JavaScript zu `canvas.fill_style = 'red'` wird.

Installation

```
$ uv add ipycanvas
```

Erstellen von Canvas-Elementen

Bevor wir mit dem Erstellen von Canvas-Elementen beginnen können, zunächst ein Hinweis zum Canvas-Raster. Der Ursprung eines Gitters befindet sich in der oberen linken Ecke bei der Koordinate (0,0). Alle Elemente werden relativ zu diesem Ursprung platziert.

Es gibt vier Methoden zum Zeichnen von Rechtecken:

- `fill_rect(x, y, width, height=None)` zeichnet ein gefülltes Rechteck
- `stroke_rect(x, y, width, height=None)` zeichnet einen rechteckigen Umriss
- `fill_rects(x, y, width, height=None)` zeichnet gefüllte Rechtecke
- `stroke_rects(x, y, width, height=None)` zeichnet rechteckige Umrisse

Mit `height=None` wird derselbe Wert verwendet wie bei `width`.

Bei `*_rects` sind `x`, `y`, `width` und `height` ganze Zahlen, Listen von ganzen Zahlen oder NumPy-Arrays.

[1]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(120, 100))
canvas.fill_style = "lime"
canvas.stroke_style = "green"

canvas.fill_rect(10, 20, 100, 50)
canvas.stroke_rect(10, 20, 100, 50)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
canvas
```

```
Canvas(layout=Layout(height='100px', width='120px'), size=(120, 100))
```

[2]:
```
from ipycanvas import MultiCanvas


# Create a multi-layer canvas with 2 layers
multi_canvas = MultiCanvas(2, size=(165, 115))
multi_canvas[0]  #  Access first layer (background)
multi_canvas[0].fill_style = "lime"
multi_canvas[0].stroke_style = "green"
multi_canvas[0].fill_rect(10, 20, 100, 50)
multi_canvas[0].stroke_rect(10, 20, 100, 50)

multi_canvas[1]  #  Access last layer
multi_canvas[1].fill_style = "red"
multi_canvas[1].stroke_style = "brown"
multi_canvas[1].fill_rect(55, 45, 100, 50)
multi_canvas[1].stroke_rect(55, 45, 100, 50)

multi_canvas
```

```
MultiCanvas(layout=Layout(height='115px', width='165px'))
```

[3]:
```
import numpy as np

from ipycanvas import Canvas


n_particles = 75_000

x = np.array(np.random.rayleigh(350, n_particles), dtype=np.int32)
y = np.array(np.random.rayleigh(150, n_particles), dtype=np.int32)
size = np.random.randint(1, 3, n_particles)

canvas = Canvas(size=(1000, 500))

canvas.fill_style = "green"
canvas.fill_rects(x, y, size)

canvas
```

```
Canvas(layout=Layout(height='500px', width='1000px'), size=(1000, 500))
```

Da `Canvas` ein ipywidget ist, kann es

- mehrfach in einem Notebook vorkommen
- die Attribute ändern
- geänderte Attribute auf andere Widget-Attribute verlinken

Canvas löschen

[4]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(120, 100))
# Perform some drawings...
canvas.clear()
```

[5]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(165, 115))

canvas.fill_style = "lime"
canvas.stroke_style = "brown"

canvas.fill_rect(10, 20, 100, 50)
canvas.clear_rect(52, 42, 100, 50)
canvas.stroke_rect(55, 45, 100, 50)

canvas
```

```
Canvas(layout=Layout(height='115px', width='165px'), size=(165, 115))
```

Formen

Die verfügbaren Zeichenkommandos sind:

- `move_to(x, y)`:
- `line_to(x, y)`:
- `arc(x, y, radius, start_angle, end_angle, anticlockwise=False)`:
- `arc_to(x1, y1, x2, y2, radius)`:
- `quadratic_curve_to(cp1x, cp1y, x, y)`:
- `bezier_curve_to(cp1x, cp1y, cp2x, cp2y, x, y)`:
- `rect(x, y, width, height)`:

Kreise zeichnen

Es gibt vier verschiedene Arten, Kreise zu zeichnen:

- `fill_arc(x, y, radius, start_angle, end_angle, anticlockwise=False)`
- `stroke_arc(x, y, radius, start_angle, end_angle, anticlockwise=False)`
- `fill_arcs(x, y, radius, start_angle, end_angle, anticlockwise=False)`
- `stroke_arcs(x, y, radius, start_angle, end_angle, anticlockwise=False)`

Bei `*_arcs` sind `x`, `y` und `radius` NumPy-Arrays, Listen oder skalare Werte.

[6]:
```
from math import pi
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
from ipycanvas import Canvas


canvas = Canvas(size=(200, 200))

canvas.fill_style = "red"
canvas.stroke_style = "green"

canvas.fill_arc(60, 60, 50, 0, pi)
canvas.stroke_arc(60, 60, 40, 0, 2 * pi)

canvas
```

```
Canvas(layout=Layout(height='200px', width='200px'), size=(200, 200))
```

Zeichenpfade

Ein Pfad ist eine Liste von Punkten, die durch Liniensegmente verbunden sind, die unterschiedliche Formen, gerade oder gekrümmt, geschlossen oder offen, unterschiedlich breit und farbig sein können. Dabei stehen die folgenden Funktionen zur Verfügung:

- `begin_path()` erstellt einen neuen Pfad
- `close_path()` fügt dem Pfad eine gerade Linie hinzu, die zum Anfang des aktuellen Pfads führt
- `stroke()` zeichnet die Form entlang der Kontur
- `fill(rule)` zeichnet die Form durch eine Füllung innerhalb des Pfades

[7]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(100, 100))

# Draw simple triangle shape
canvas.begin_path()
canvas.move_to(75, 50)
canvas.line_to(100, 75)
canvas.line_to(100, 25)
canvas.fill()

canvas
```

```
Canvas(layout=Layout(height='100px', width='100px'), size=(100, 100))
```

Beispiele

arc

[8]:
```
from math import pi

from ipycanvas import Canvas


```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
canvas = Canvas(size=(200, 200))

# Draw smiley face
canvas.begin_path()
canvas.arc(75, 75, 50, 0, pi * 2, True)  # Outer circle
canvas.move_to(110, 75)
canvas.arc(75, 75, 35, 0, pi, False)  # Mouth (clockwise)
canvas.move_to(65, 65)
canvas.arc(60, 65, 5, 0, pi * 2, True)  # Left eye
canvas.move_to(95, 65)
canvas.arc(90, 65, 5, 0, pi * 2, True)  # Right eye
canvas.stroke()

canvas
```

```
Canvas(layout=Layout(height='200px', width='200px'), size=(200, 200))
```

bezier_curve_to

[9]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(200, 200))

# Cubic curves example
canvas.begin_path()
canvas.move_to(75, 40)
canvas.bezier_curve_to(75, 37, 70, 25, 50, 25)
canvas.bezier_curve_to(20, 25, 20, 62.5, 20, 62.5)
canvas.bezier_curve_to(20, 80, 40, 102, 75, 120)
canvas.bezier_curve_to(110, 102, 130, 80, 130, 62.5)
canvas.bezier_curve_to(130, 62.5, 130, 25, 100, 25)
canvas.bezier_curve_to(85, 25, 75, 37, 75, 40)
canvas.fill()

canvas
```

```
Canvas(layout=Layout(height='200px', width='200px'), size=(200, 200))
```

Stile und Farben

Farben

`Canvas` hat zwei Farbattribute, eines für Striche und eines für Flächen; zudem kann die Transparenz geändert werden.

- `stroke_style` erwartet eine gültige HTML-Farbe. Der Standardwert ist schwarz.
- `fill_style` erwartet eine gültige HTML-Farbe. Der Standardwert ist schwarz.
- `global_alpha` gibt die Transparenz an. Der Standardwert ist `1.0`.

Linien

Linienart

Linien lassen sich durch folgende Attribute beschreiben:

- `line_width`
- `line_cap`
- `line_join`
- `miter_limit`
- `get_line_dash()`
- `set_line_dash(segments)`
- `line_dash_offset`

Linienbreite

[10]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(400, 280))
canvas.scale(2)

for i in range(10):
    width = 1 + i
    x = 5 + i * 20
    canvas.line_width = width

    canvas.fill_text(str(width), x - 5, 15)

    canvas.begin_path()
    canvas.move_to(x, 20)
    canvas.line_to(x, 140)
    canvas.stroke()
canvas
```

```
Canvas(layout=Layout(height='280px', width='400px'), size=(400, 280))
```

Linienende

[11]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(320, 360))

# Possible line_cap values
line_caps = ["butt", "round", "square"]

canvas.scale(2)

# Draw guides
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
canvas.stroke_style = "#09f"
canvas.begin_path()
canvas.move_to(10, 30)
canvas.line_to(140, 30)
canvas.move_to(10, 140)
canvas.line_to(140, 140)
canvas.stroke()

# Draw lines
canvas.stroke_style = "black"
canvas.font = "15px serif"

for i in range(len(line_caps)):
    line_cap = line_caps[i]
    x = 25 + i * 50

    canvas.fill_text(line_cap, x - 15, 15)
    canvas.line_width = 15
    canvas.line_cap = line_cap
    canvas.begin_path()
    canvas.move_to(x, 30)
    canvas.line_to(x, 140)
    canvas.stroke()

canvas
```

```
Canvas(layout=Layout(height='360px', width='320px'), size=(320, 360))
```

Linienverbindung

legt das Erscheinungsbild der *Ecken* fest, an denen sich Linien treffen.

[12]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(320, 360))

# Possible line_join values
line_joins = ["round", "bevel", "miter"]

min_y = 40
max_y = 80
spacing = 45

canvas.line_width = 10
canvas.scale(2)
for i in range(len(line_joins)):
    line_join = line_joins[i]

    y1 = min_y + i * spacing
    y2 = max_y + i * spacing
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    canvas.line_join = line_join

    canvas.fill_text(line_join, 60, y1 - 10)

    canvas.begin_path()
    canvas.move_to(-5, y1)
    canvas.line_to(35, y2)
    canvas.line_to(75, y1)
    canvas.line_to(115, y2)
    canvas.line_to(155, y1)
    canvas.stroke()

canvas
```

```
Canvas(layout=Layout(height='360px', width='320px'), size=(320, 360))
```

Strichmuster

[13]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(400, 280))
canvas.scale(2)

line_dashes = [[5, 10], [10, 5], [5, 10, 20], [10, 20], [20, 10], [20, 20]]

canvas.line_width = 2

for i in range(len(line_dashes)):
    x = 5 + i * 20

    canvas.set_line_dash(line_dashes[i])
    canvas.begin_path()
    canvas.move_to(x, 0)
    canvas.line_to(x, 140)
    canvas.stroke()
canvas
```

```
Canvas(layout=Layout(height='280px', width='400px'), size=(400, 280))
```

Text

Es gibt zwei Methoden zur Gestaltung von Text:

- `fill_text(text, x, y, max_width=None)`
- `stroke_text(text, x, y, max_width=None)`

[14]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(400, 50))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
canvas.font = "32px serif"
canvas.fill_text("Drawing from Python is cool!", 10, 32)
canvas
```

```
Canvas(layout=Layout(height='50px', width='400px'), size=(400, 50))
```

[15]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(400, 50))

canvas.font = "32px serif"
canvas.stroke_text("Hello There!", 10, 32)
canvas
```

```
Canvas(layout=Layout(height='50px', width='400px'), size=(400, 50))
```

- `font` gibt den aktuellen Textstil an. Der Standardwert ist `"12px sans-serif"`.
- `text_align` gibt die Textausrichtung an. Mögliche Werte sind `"start"`, `"end"`, `"left"`, `"right"` oder `"center"` .
- `text_baseline` gibt die Ausrichtung an der Grundlinie an. Mögliche Werte sind `"top"`, `"hanging"`, `"middle"`, `"alphabetic"`, `"ideographic"` und `"bottom"`. Der Standardwert ist `"alphabetic"`.
- `direction` gibt die Textrichtung an. Mögliche Werte sind `"ltr"`, `"rtl"`, `"inherit"`. Der Standardwert ist `"inherit"`.

Selbstverständlich kann auch `fill_style` und `stroke_style` zum EInfürben der Text verwendet werden.

Bilder

aus einem NumPy-Array

Mit `put_image_data(image_data, dx, dy)` lässt sich ein Bild darstellen wobei `image_data` ein NumPy-Array angibt und `dx` und `dy` das obere linke Eck des Bildes.

[16]:
```
import numpy as np

from ipycanvas import Canvas


x = np.linspace(-1, 1, 600)
y = np.linspace(-1, 1, 600)

x_grid, y_grid = np.meshgrid(x, y)

blue_channel = np.array(
    np.sin(x_grid**2 + y_grid**2) * 255, dtype=np.int32
)
red_channel = np.zeros_like(blue_channel) + 200
green_channel = np.zeros_like(blue_channel) + 50

image_data = np.stack((red_channel, blue_channel, green_channel), axis=2)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
canvas = Canvas(size=(image_data.shape[0], image_data.shape[1]))
canvas.put_image_data(image_data, 0, 0)

canvas
```

```
Canvas(layout=Layout(height='600px', width='600px'), size=(600, 600))
```

Status

Der Status kann mit zwei Werten angegeben werden:

- `save()` speichert den Status des Canvas-Elements
- `restore()` stellt den zuletzt gespeicherten Status des Canvas-Elements wieder her wobei diese Methode beliebig oft aufgerufen werden kann.

Transformationen

- `translate(x, y)` verschiebt das Canvas-Element
- `rotate(angle)` rotiert das Canvas-Element im Uhrzeigersinn
- `scale(x, y=None)` skaliert das Canvas-Element

Siehe auch:

- API-Referenz

`pythreejs`
Jupyter-Three.js-Bridge

- pythreejs

`ipyvolume`
IPyvolume ist eine Python-Bibliothek zur Visualisierung von 3D-Volumen und Glyphen (z.B. 3D-Scatter-Plots).

- ipyvolume

`ipyleaflet`
Jupyter-Leaflet.js-Bridge

- ipyleaflet

`ipywebrtc`
WebRTC- und MediaStream-API für Jupyter-Notebooks

10.5.2 `ipywebrtc`

ipywebrtc stellt WebRTC- und die MediaStream-API in Jupyter Notebooks zur Verfügung. Damit lassen sich z.B. Screenshots aus einem MediaStream erstellen und mit skimage weiter analysieren. Ihr könnt mit `ipywebrtc` jedoch nicht nur Video-, Bild-, Audio- und Widget-Daten lesen sondern auch Stream-Objekte aufzeichnen. Es stellt sogar eine einfache Chat-Funktion zur Verfügung.

Installation

`ipywebrtc` wird sowohl im Kernel- wie auch im Jupyter-Environment installiert mit

```
$ uv add ipywebrtc
```

Beispiele

Beispiel `VideoStream`

[1]:
```
from ipywebrtc import VideoStream, VideoRecorder
```

[2]:
```
!cd ../../../../data/
!dvc  import-url https://github.com/maartenbreddels/ipywebrtc/raw/master/docs/
↪source/Big.Buck.Bunny.mp4
```

```
Importing 'https://github.com/maartenbreddels/ipywebrtc/raw/master/docs/source/Big.
↪Buck.Bunny.mp4' -> 'Big.Buck.Bunny.mp4'
Saving information to 'Big.Buck.Bunny.mp4.dvc'.

To track the changes with git, run:

        git add Big.Buck.Bunny.mp4.dvc
```

[3]:
```
video = VideoStream.from_url("../../../../data/Big.Buck.Bunny.mp4")
video
```

```
VideoStream(video=Video(value=b'../../../../data/Big.Buck.Bunny.mp4', format='url'))
```

Aufnehmen

Eine Aufnahmetaste lässt sich mit `MediaRecorder.record` erstellen, für Videos mit:

[4]:
```
recorder = VideoRecorder(stream=video)
recorder
```

```
VideoRecorder(stream=VideoStream(video=Video(value=b'../../../../data/Big.Buck.
↪Bunny.mp4', format='url')), vid...
```

Speichern

Der Stream kann entweder über die Download-Taste erfolgen oder programmatisch, z.B. mit:

[5]:
```
recorder.save("../../../../data/example.webm")
```

Beispiel `WidgetStream`

Ein WidgetStream erstellt einen MediaStream aus einem beliebigen Widget.

[6]:
```
from ipywebrtc import VideoStream, WidgetStream
```

[7]:
```
from pythreejs import (
    AmbientLight,
    DirectionalLight,
    Mesh,
    MeshLambertMaterial,
    OrbitControls,
    PerspectiveCamera,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    Renderer,
    Scene,
    SphereGeometry,
)


ball = Mesh(
    geometry=SphereGeometry(radius=1),
    material=MeshLambertMaterial(color="red"),
    position=[2, 1, 0],
)

c = PerspectiveCamera(
    position=[0, 5, 5],
    up=[0, 1, 0],
    children=[
        DirectionalLight(color="white", position=[3, 5, 1], intensity=0.5)
    ],
)

scene = Scene(children=[ball, c, AmbientLight(color="#777777")])

renderer = Renderer(
    camera=c,
    scene=scene,
    controls=[OrbitControls(controlling=c)]
)

renderer
```

```
Renderer(camera=PerspectiveCamera(children=(DirectionalLight(color='white',␣
↪intensity=0.5, position=(3.0, 5.0,...
```

Der folgende `webgl_stream` wird aktualisiert, nachdem sich in der obigen Szene etwas ändert. Dies könnt ihr erreichen indem ihr den Ball mit der Maus verschiebt.

[8]:
```
webgl_stream = WidgetStream(widget=renderer)
webgl_stream
```

```
WidgetStream(widget=Renderer(camera=PerspectiveCamera(children=(DirectionalLight(color=
↪'white', intensity=0.5,...
```

Alternativ könnt ihr auch einen Slider verwenden:

[9]:
```
from ipywidgets import FloatSlider


slider = FloatSlider(
    value=7.5,
    step=0.1,
    description="Test:",
    disabled=False,
    continuous_update=False,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    orientation="horizontal",
    readout=True,
    readout_format=".1f",
)

slider
```

```
FloatSlider(value=7.5, continuous_update=False, description='Test:', readout_format=
↪'.1f')
```

ipysheet
: Interaktive Tabellen um IPython Widgets in Tabellen von Jupyter Notebooks zu verwenden.

10.5.3 ipysheet

ipysheet verbindet ipywidgets mit tabellarischen Daten. Es fügt im Wesentlichen zwei Widgets hinzu: ein *Cell widget* und ein *Sheet widget*. Darüberhinaus gibt es noch Hilfsfunktionen zum Erstellen von Tabellenzeilen und -spalten sowie zur Formatierung und Gestaltung von Zellen.

Siehe auch:

- Interactive spreadsheets in Jupyter
- GitHub
- Docs

Installation

`ipysheet` lässt sich einfach mit Pipenv installieren:

```
$ uv add ipysheet
```

Import

[1]:
```
import ipysheet
```

Zellformatierung

[2]:
```
sheet1 = ipysheet.sheet()
cell0 = ipysheet.cell(0, 0, 0, numeric_format="0.0", type="numeric")
cell1 = ipysheet.cell(1, 0, "Hello", type="text")
cell2 = ipysheet.cell(0, 1, 0.1, numeric_format="0.000", type="numeric")
cell3 = ipysheet.cell(1, 1, 15.9, numeric_format="0.00", type="numeric")
cell4 = ipysheet.cell(2, 2, "14-02-2019", date_format="DD-MM-YYYY", type="date")

sheet1
```

```
Sheet(cells=(Cell(column_end=0, column_start=0, numeric_format='0.0', row_end=0,␣
↪row_start=0, type='numeric', ...
```

Beispiele

Interaktive Tabelle

[3]:
```
from ipywidgets import FloatSlider, Image, IntSlider


slider = FloatSlider()
sheet2 = ipysheet.sheet()
cell1 = ipysheet.cell(0, 0, slider, style={"min-width": "122px"})
cell3 = ipysheet.cell(1, 0, 42.0, numeric_format="0.00")
cell_sum = ipysheet.cell(2, 0, 42.0, numeric_format="0.00")


@ipysheet.calculation(inputs=[(cell1, "value"), cell3], output=cell_sum)
def calculate(a, b):
    return a + b


sheet2
```

```
Sheet(cells=(Cell(column_end=0, column_start=0, row_end=0, row_start=0, style={'min-
↪width': '122px'}, type='wi...
```

Numpy

[4]:
```
import numpy as np

from ipysheet import from_array, to_array


arr = np.random.randn(6, 10)

sheet = from_array(arr)
sheet
```

```
Sheet(cells=(Cell(column_end=9, column_start=0, row_end=5, row_start=0, squeeze_
↪column=False, squeeze_row=Fals...
```

[5]:
```
arr = np.array([True, False, True])

sheet = from_array(arr)
sheet
```

```
Sheet(cells=(Cell(column_end=0, column_start=0, numeric_format=None, row_end=2, row_
↪start=0, squeeze_row=False...
```

[6]:
```
to_array(sheet)
```

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

Tabellensuche

[7]:
```
import numpy as np
import pandas as pd

from ipysheet import from_dataframe
from ipywidgets import Text, VBox, link


df = pd.DataFrame(
    {
        "A": 1.0,
        "B": pd.Timestamp("20130102"),
        "C": pd.Series(1, index=list(range(4)), dtype="float32"),
        "D": np.array([False, True, False, False], dtype="bool"),
        "E": pd.Categorical(["test", "train", "test", "train"]),
        "F": "foo",
    }
)

df.loc[[0, 2], ["B"]] = np.nan


s = from_dataframe(df)

search_box = Text(description="Search:")
link((search_box, "value"), (s, "search_token"))

VBox((search_box, s))
```

```
VBox(children=(Text(value='', description='Search:'), Sheet(cells=(Cell(choice=[],␣
↪column_end=0, column_start=...
```

Plotten editierbarer Tabellen

[8]:
```
import bqplot.pyplot as plt
import numpy as np

from ipysheet import cell, column, sheet
from ipywidgets import HBox
from traitlets import link


size = 18
scale = 100.0
np.random.seed(0)
x_data = np.arange(size)
y_data = np.cumsum(np.random.randn(size) * scale)

fig = plt.figure()
axes_options = {
    "x": {"label": "Date", "tick_format": "%m/%d"},
    "y": {"label": "Price", "tick_format": "0.0f"},
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
}

scatt = plt.scatter(x_data, y_data, colors=["red"], stroke="black")
fig.layout.width = "70%"

sheet1 = sheet(rows=size, columns=2)
x_column = column(0, x_data)
y_column = column(1, y_data)

link((scatt, "x"), (x_column, "value"))
link((scatt, "y"), (y_column, "value"))

HBox((sheet1, fig))
```

```
HBox(children=(Sheet(cells=(Cell(column_end=0, column_start=0, row_end=17, row_
↪start=0, squeeze_row=False, typ...
```

ipydatagrid
: ipydatagrid ist ein schnelles und vielfältiges Datagrid-Widget.

ipyvuetify
: Vuetify-Widgets in Jupyter Notebooks

10.5.4 ipyvuetify

ipyvuetify liefert Jupyter-Widgets, die auf vuetify-UI-Komponenten basieren und das Material Design von Google mit dem Vue.js-Framework implementieren.

Installation

```
$ uv add ipyvuetify
```

Beispiele

Importe

[1]:
```
from threading import Timer

import ipyvuetify as v
import ipywidgets

from traitlets import Any, List, Unicode
```

Menü

[2]:
```
def on_menu_click(widget, event, data):
    if len(layout.children) == 1:
        layout.children = layout.children + [info]
    info.children = [f"Item {items.index(widget)+1} clicked"]


items = [
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    v.ListItem(children=[v.ListItemTitle(children=[f"Item {i}"])])
    for i in range(1, 5)
]

for item in items:
    item.on_event("click", on_menu_click)

menu = v.Menu(
    offset_y=True,
    v_slots=[
        {
            "name": "activator",
            "variable": "menuData",
            "children": v.Btn(
                v_on="menuData.on",
                class_="ma-2",
                color="primary",
                children=[
                    "menu",
                    v.Icon(right=True, children=["arrow_drop_down"]),
                ],
            ),
        }
    ],
    children=[v.List(children=items)],
)

info = v.Chip(class_="ma-2")

layout = v.Layout(children=[menu])
layout
```

```
Layout(children=[Menu(children=[List(children=[ListItem(children=[ListItemTitle(children=[
↪'Item 1'])]), ListIt...
```

Buttons

[3]:
```
v.Layout(
    children=[
        v.Btn(color="primary", children=["primary"]),
        v.Btn(color="error", children=["error"]),
        v.Btn(disabled=True, children=["disabled"]),
        v.Btn(children=["reset"]),
    ]
)v.Layout(
    children=[
        v.Btn(color="primary", flat=True, children=["flat"]),
        v.Btn(color="primary", round=True, children=["round"]),
        v.Btn(
            color="primary",
            flat=True,
            icon=True,
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
            children=[v.Icon(children=["thumb_up"])],
        ),
        v.Btn(color="primary", outline=True, children=["outline"]),
        v.Btn(
            color="primary",
            fab=True,
            large=True,
            children=[v.Icon(children=["edit"])],
        ),
    ]
)
```

```
Layout(children=[Btn(children=['primary'], color='primary'), Btn(children=['error'],
↪ color='error'), Btn(child...
```

[4]:
```
v.Layout(
    children=[
        v.Btn(color="primary", flat=True, children=["flat"]),
        v.Btn(color="primary", round=True, children=["round"]),
        v.Btn(
            color="primary",
            flat=True,
            icon=True,
            children=[v.Icon(children=["thumb_up"])],
        ),
        v.Btn(color="primary", outline=True, children=["outline"]),
        v.Btn(
            color="primary",
            fab=True,
            large=True,
            children=[v.Icon(children=["edit"])],
        ),
    ]
)
```

```
Layout(children=[Btn(children=['flat'], color='primary'), Btn(children=['round'],␣
↪color='primary'), Btn(childr...
```

[5]:
```
def toggleLoading():
    button2.loading = not button2.loading
    button2.disabled = button2.loading


def on_loader_click(*args):
    toggleLoading()
    Timer(2.0, toggleLoading).start()


button2 = v.Btn(loading=False, children=["loader"])
button2.on_event("click", on_loader_click)

v.Layout(children=[button2])
```

```
Layout(children=[Btn(children=['loader'], loading=False)])
```

[6]:
```
toggle_single = v.BtnToggle(
    v_model=2,
    class_="mr-3",
    children=[
        v.Btn(flat=True, children=[v.Icon(children=["format_align_left"])]),
        v.Btn(flat=True, children=[v.Icon(children=["format_align_center"])]),
        v.Btn(flat=True, children=[v.Icon(children=["format_align_right"])]),
        v.Btn(flat=True, children=[v.Icon(children=["format_align_justify"])]),
    ],
)

toggle_multi = v.BtnToggle(
    v_model=[0, 2],
    multiple=True,
    children=[
        v.Btn(flat=True, children=[v.Icon(children=["format_bold"])]),
        v.Btn(flat=True, children=[v.Icon(children=["format_italic"])]),
        v.Btn(flat=True, children=[v.Icon(children=["format_underline"])]),
        v.Btn(flat=True, children=[v.Icon(children=["format_color_fill"])]),
    ],
)

v.Layout(
    children=[
        toggle_single,
        toggle_multi,
    ]
)
```

```
Layout(children=[BtnToggle(children=[Btn(children=[Icon(children=['format_align_left
↪'])]), Btn(children=[Icon(...
```

[7]:
```
v.Layout(
    children=[
        v.Btn(
            color="primary",
            children=[
                v.Icon(left=True, children=["fingerprint"]),
                "Icon left",
            ],
        ),
        v.Btn(
            color="primary",
            children=[
                "Icon right",
                v.Icon(right=True, children=["fingerprint"]),
            ],
        ),
        v.Tooltip(
            bottom=True,
            children=[
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                v.Btn(slot="activator", color="primary", children=["tooltip"]),
                "Insert tooltip text here",
            ],
        ),
    ]
```

```
Layout(children=[Btn(children=[Icon(children=['fingerprint'], left=True), 'Icon left
↪'], color='primary'), Btn(...
```

[8]:
```
lorum_ipsum = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do␣
↪eiusmod tempor incididunt ut labore et dolore magna aliqua."

v.Layout(
    children=[
        v.Dialog(
            width="500",
            v_slots=[
                {
                    "name": "activator",
                    "variable": "x",
                    "children": v.Btn(
                        v_on="x.on",
                        color="success",
                        dark=True,
                        children=["Open dialog"],
                    ),
                }
            ],
            children=[
                v.Card(
                    children=[
                        v.CardTitle(
                            class_="headline gray lighten-2",
                            primary_title=True,
                            children=["Lorem ipsum"],
                        ),
                        v.CardText(children=[lorum_ipsum]),
                    ]
                )
            ],
        )
    ]
)
```

```
Layout(children=[Dialog(children=[Card(children=[CardTitle(children=['Lorem ipsum'],
↪ class_='headline gray lig...
```

Slider

[9]:
```
slider = v.Slider(v_model=25)
slider2 = v.Slider(thumb_label=True, v_model=25)
slider3 = v.Slider(thumb_label="always", v_model=25)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
ipywidgets.jslink((slider, "v_model"), (slider2, "v_model"))

v.Container(
    children=[
        slider,
        slider2,
    ]
)
```

```
Container(children=[Slider(v_model=25), Slider(thumb_label=True, v_model=25)])
```

Reiter

[10]:
```
tab_list = [v.Tab(children=["Tab " + str(i)]) for i in range(1, 4)]
content_list = [v.TabItem(children=[lorum_ipsum]) for i in range(1, 4)]
tabs = v.Tabs(v_model=1, children=tab_list + content_list)
tabs
```

```
Tabs(children=[Tab(children=['Tab 1']), Tab(children=['Tab 2']), Tab(children=['Tab␣
↪3']), TabItem(children=['L...
```

Akkordeon

[11]:
```
vepc1 = v.ExpansionPanel(
    children=[
        v.ExpansionPanelHeader(children=["item1"]),
        v.ExpansionPanelContent(children=["First Text"]),
    ]
)

vepc2 = v.ExpansionPanel(
    children=[
        v.ExpansionPanelHeader(children=["item2"]),
        v.ExpansionPanelContent(children=["Second Text"]),
    ]
)

vep = v.ExpansionPanels(children=[vepc1, vepc2])
vl = v.Layout(class_="pa-4", children=[vep])
vl
```

```
Layout(children=[ExpansionPanels(children=[ExpansionPanel(children=[ExpansionPanelHeader(children=[
↪'item1']), ...
```

In der Vuetify-Dokumentation könnt ihr nach allen verfügbaren Komponenten und Attributen suchen. Dabei orientiert sich Ipyvuetify an der Syntax von Vue.js- und Vuetify, aber es gibt auch einige Unterschiede:

Beschreibung	Vuetify	ipyvuetify
Komponentennamen werden in CamelCase geschrieben und der v-Präfix entfernt	`<v-list-tile .../>`	`ListTile(...)`
Untergeordnete Komponenten und Text werden im Traitlet für untergeordnete Elemente definiert	`<v-btn>text <v-icon .../></v-btn>`	`Btn(children=['text', Icon(...)])`
Flag-Attribute erfordern einen booleschen Wert	`<v-btn round ...`	`Btn(round=True ...`
Attribute sind snake_case	`<v-menu offset-y ...`	`Menu(offset_y=True ...`
Das Attribut `v_model` (Wert in ipywidgets) erhält den Wert direkt	`<v-slider v-model= "some_property" ...`	`Slider(v_model=25...`
Der Geltungsbereich von `slot` kann aktuell noch nicht angegeben werden	`<v-menu><template slot:activator="{ on }"><v-btn v-on=on>`	`Menu(children=[Btn(slot='activator', ...), ...]`
Event-Listener werden mit `on_event` definiert	`<v-btn @click='someMethod()' ...`	`def some_method (widget, event, data):` mit `button.on_event('click', some_method)`
Reguläre HTML-Tags können mit der Klasse `Html` erstellt werden	`<div>...</div>`	`Html(tag='div', children=[...])`
Den Attributen `class` und `style` muss ein Unterstrich angefügt werden	`<v-btn class="mr-3" style="..." >`	`Btn(class_='mr-3', style_='...')`

VuetifyTemplate

Eine engere Übereinstimmung mit der Vue/Vuetify-API erhaltet ihr mit `VuetifyTemplate`. Hierfür erstellt ihr eine Unterklasse von `VuetifyTemplate` und definiert eigene Traitlets. Auf die Traitlets kann über das Template zugegriffen werden, so als befänden sie sich in einem Vue-Modell. Methoden können mit dem Präfix `vue_` definiert werden, z.B. `def vue_button_click(self, data)`, das dann mit `@click="button_click(e)"` aufgerufen werden kann. Im Folgenden zeige ich Euch eine Tabelle mit Suche, Sortierung und Zeilenanzahl:

[12]:
```
import json

import ipyvuetify as v
import pandas as pd
import traitlets


class PandasDataFrame(v.VuetifyTemplate):
    """
    Vuetify DataTable rendering of a pandas DataFrame

    Args:
        data (DataFrame) - the data to render
        title (str) - optional title
    """

    headers = traitlets.List([]).tag(sync=True, allow_null=True)
    items = traitlets.List([]).tag(sync=True, allow_null=True)
    search = traitlets.Unicode("").tag(sync=True)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    title = traitlets.Unicode("DataFrame").tag(sync=True)
    index_col = traitlets.Unicode("").tag(sync=True)
    template = traitlets.Unicode(
        """
        <template>
          <v-card>
            <v-card-title>
              <span class="title font-weight-bold">{{ title }}</span>
              <v-spacer></v-spacer>
                <v-text-field
                    v-model="search"
                    append-icon="search"
                    label="Search ..."
                    single-line
                    hide-details
                ></v-text-field>
            </v-card-title>
            <v-data-table
                :headers="headers"
                :items="items"
                :search="search"
                :item-key="index_col"
                :rows-per-page-items="[25, 50, 250, 500]"
            >
                <template v-slot:no-data>
                  <v-alert :value="true" color="error" icon="warning">
                    Sorry, nothing to display here :(
                  </v-alert>
                </template>
                <template v-slot:no-results>
                    <v-alert :value="true" color="error" icon="warning">
                      Your search for "{{ search }}" found no results.
                    </v-alert>
                </template>
                <template v-slot:items="rows">
                  <td v-for="(element, label, index) in rows.item"
                      @click=cell_click(element)
                      >
                    {{ element }}
                  </td>
                </template>
            </v-data-table>
          </v-card>
        </template>
        """
    ).tag(sync=True)

    def __init__(self, *args, data=pd.DataFrame(), title=None, **kwargs):
        super().__init__(*args, **kwargs)
        data = data.reset_index()
        self.index_col = data.columns[0]
        headers = [{"text": col, "value": col} for col in data.columns]
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        headers[0].update({"align": "left", "sortable": True})
        self.headers = headers
        self.items = json.loads(data.to_json(orient="records"))
        if title is not None:
            self.title = title


iris = pd.read_csv(
    "https://raw.githubusercontent.com/mwaskom/seaborn-data/master/iris.csv"
)
test = PandasDataFrame(data=iris, title="Iris")
test
```

```
PandasDataFrame(headers=[{'text': 'index', 'value': 'index', 'align': 'left',
↪'sortable': True}, {'text': 'sep...
```

[13]:
```
v.Banner(
    single_line=True,
    v_slots=[
        {"name": "icon", "children": v.Icon(children=["thumb_up"])},
        {
            "name": "actions",
            "children": v.Btn(
                text=True, color="deep-purple accent-4", children=["Action"]
            ),
        },
    ],
    children=[
        "One line message text string with two actions on tablet / Desktop"
    ],
)
```

```
Banner(children=['One line message text string with two actions on tablet / Desktop
↪'], single_line=True, v_slo...
```

ipympl
: `ipympl` oder jupyter-matplotlib bieten interaktive Widgets für Matplotlib.

10.5.5 ipympl

Da sich das Jupyter-Widget-Ökosystem zu schnell entwickelt, haben die Matplotlib-Entwickler beschlossen, die Unterstützung in ein eigenes Modul auszulagern: ipympl.

Installation

`ipympl` wird sowohl im Kernel- wie auch im Jupyter-Environment installiert mit

```
$ uv add ipympl
```

Anschließend könnt ihr das Jupyter-Backend in Notebooks aktivieren, indem ihr die folgende *Matplotlib-Magic* verwendet:

[1]:
```
%matplotlib widget
```

Beispiele

Einfache Matplotlib-Interaktion

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


plt.figure(1)
plt.plot(np.sin(np.linspace(0, 20, 100)))
plt.show()
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), (
↪'Back', 'Back to previous ...
```

3D-Plot: subplot3d_demo.py

[3]:
```
from mpl_toolkits.mplot3d import axes3d


fig = plt.figure()
ax = fig.add_subplot(111, projection="3d")

# Grab some test data.
X, Y, Z = axes3d.get_test_data(0.05)

# Plot a basic wireframe.
ax.plot_wireframe(X, Y, Z, rstride=10, cstride=10)

fig.canvas.layout.max_width = "800px"

plt.show()
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), (
↪'Back', 'Back to previous ...
```

Komplexeres Beispiel aus der Matplotlib-Galerie

[4]:
```
import matplotlib.pyplot as plt
import numpy as np


np.random.seed(0)

n_bins = 10
x = np.random.randn(1000, 3)

fig, axes = plt.subplots(nrows=2, ncols=2)
ax0, ax1, ax2, ax3 = axes.flatten()

colors = ["red", "tan", "lime"]
ax0.hist(x, n_bins, density=1, histtype="bar", color=colors, label=colors)
ax0.legend(prop={"size": 10})
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
ax0.set_title("bars with legend")

ax1.hist(x, n_bins, density=1, histtype="bar", stacked=True)
ax1.set_title("stacked bar")

ax2.hist(x, n_bins, histtype="step", stacked=True, fill=False)
ax2.set_title("stack step (unfilled)")

# Make a multiple-histogram of data-sets with different length.
x_multi = [np.random.randn(n) for n in [10000, 5000, 2000]]
ax3.hist(x_multi, n_bins, histtype="bar")
ax3.set_title("different sample sizes")

fig.tight_layout()
plt.show()
```

```
Canvas(toolbar=Toolbar(toolitems=[('Home', 'Reset original view', 'home', 'home'), (
↪'Back', 'Back to previous ...
```

10.6 Einbetten von Jupyter-Widgets

Jupyter-Widgets können serialisiert und dann in andere Kontexte eingebettet werden:

- statische Webseiten
- Sphinx-Dokumentation
- HTML-konvertierte Notebooks auf Nbviewer

Dabei erlaubt das npm-Paket `@jupyter-widgets/html-manager` das Einbetten auf zwei unterschiedliche Weisen:

- das Einbetten der Standardelemente, die auf jeder Website verwendet werden können
- das Einbetten mit RequireJS auch für benutzerdefinierte Widgets.

10.6.1 Einbetten von Widgets in HTML-Seiten

Hierfür stellt das *Widgets*-Menü mehrere Optionen zur Verfügung:

Save Notebook Widget State
: Eine Notebook-Datei wird mit dem aktuellen Widget-Status als Metadaten gespeichert. Dadurch kann sie mit den Widgets im Browser gerendert werden.

Clear Notebook Widget State
: Die Metadaten des Widget-Status werden aus der Notebook-Datei gelöscht.

Embed widgets
: Der Menüpunkt bietet ein Dialogfeld mit einer HTML-Seite, auf der die aktuellen Widgets eingebettet sind. Um benutzerdefinierte Widgets zu unterstützen, wird der RequireJS-Embedder verwendet.

> **Bemerkung**
>
> Das erste Skript-Tag lädt RequireJS von einem CDN. RequireJS sollte jedoch auf der Site selbst zur Verfügung gestellt und dieser Skript-Tag gelöscht werden.

> **Bemerkung**
>
> Das zweite Skript-Tag lädt den RequireJS-Widget-Embedder. Dadurch werden geeignete Module definiert und anschließend eine Funktion zum Rendern aller auf der Seite enthaltenen Widget-Ansichten eingerichtet.
>
> Wenn ihr nur Standard-Widgets einbettet und RequireJS nicht verwendet, könnt ihr die ersten beiden Skript-Tags durch ein Skript-Tag ersetzen, das das Standard-Skript lädt.

Download Widget State
: Die Option lädt eine JSON-Datei herunter, die den serialisierten Status aller derzeit verwendeten Widget-Modelle im Format `application/vnd.jupyter.widget-state+json` enthält, das im npm-Paket `@jupyter-widgets/schema` spezifiziert ist.

10.6.2 Sphinx-Integration

Jupyter Sphinx

jupyter_sphinx ermöglicht Jupyter-spezifische Funktionen in Sphinx. Es kann mit `pip` installiert werden.

Konfiguration

Fügt in der `conf.py`-Datei `jupyter_sphinx.embed_widgets` in der Liste der Erweiterungen hinzu.

Anschließend könnt ihr in reStructuredText folgende Direktiven verwenden:

ipywidgets-setup

```
from ipywidgets import VBox, jsdlink, IntSlider, Button
```

ipywidgets-display

```
s1, s2 = IntSlider(max=200, value=100), IntSlider(value=40)
b = Button(icon="legal")
jsdlink((s1, "value"), (s2, "max"))
VBox([s1, s2, b])
```

Beispiel

```
.. ipywidgets-setup::

   from ipywidgets import VBox, jsdlink, IntSlider, Button

.. ipywidgets-display::
   :hide-code:

   s1, s2 = IntSlider(max=200, value=100), IntSlider(value=40)
   b = Button(icon='legal')
   jsdlink((s1, 'value'), (s2, 'max'))
   VBox([s1, s2, b])
```

Optionen

Die Direktiven `ipywidgets-setup` und `ipywidgets-display` haben die folgenden Optionen:

`ipywidgets-setup`
: mit der Option `:show:` um den Setup-Code als Code-Block darzustellen

`ipywidgets-display`
: mit den folgenden Optionen:

  `:hide-code:`
  : zeigt den Code nicht an sondern nur das Widget

  `:code-below:`
  : zeigt den Code nach dem Widget an

  `:alt:`
  : Alternativer Text, wenn das Widget nicht gerendert werden kann

Siehe auch

Options

KAPITEL 11

nbextensions

Jupyter Notebook Extensions enthält eine Sammlung von Erweiterungen. Diese sind größtenteils in Javascript geschrieben und werden lokal in eurem Browser geladen.

Siehe auch

- Docs
- Github

11.1 Installation

1. Installation mit uv:

```
$ uv add jupyter_contrib_nbextensions
```

2. Installation der zugehörigen Javascript- und CSS-Dateien:

```
$ uv run jupyter contrib nbextension install --user
[I 20:57:19 InstallContribNbextensionsApp] jupyter contrib nbextension install --
↪user
[I 20:57:19 InstallContribNbextensionsApp] Installing jupyter_contrib_nbextensions␣
↪nbextension files to jupyter data directory
...
[I 20:57:20 InstallContribNbextensionsApp] - Writing config: /Users/veit/.jupyter/
↪jupyter_nbconvert_config.json
[I 20:57:20 InstallContribNbextensionsApp] --  Writing updated config file /Users/
↪veit/.jupyter/jupyter_nbconvert_config.json
```

3. Überprüfen der Installation:

```
$ uv run jupyter nbextension list
Known nbextensions:
  config dir: /Users/veit/.jupyter/nbconfig
    notebook section
      nbextensions_configurator/config_menu/main  enabled
      - Validating: problems found:
        - require?  X nbextensions_configurator/config_menu/main
      contrib_nbextensions_help_item/main  enabled
      - Validating: OK
    tree section
      nbextensions_configurator/tree_tab/main  enabled
      - Validating: problems found:
        - require?  X nbextensions_configurator/tree_tab/main
  config dir: /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/bin/../
↪etc/jupyter/nbconfig
    notebook section
      jupyter-js-widgets/extension  enabled
      - Validating: OK
```

4. Latex environments

```
$ uv run --with jupyter jupyter nbextension install --py latex_envs --user
Installing /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/lib/
↪python3.7/site-packages/latex_envs/static -> latex_envs
...
- Validating: OK
    To initialize this nbextension in the browser every time the notebook (or other␣
↪app) loads:
          jupyter nbextension enable latex_envs --user --py
...
$ uv run --with jupyter jupyter nbextension enable --py latex_envs --user
Enabling notebook extension latex_envs/latex_envs...
      - Validating: OK
```

5. yapf Code Prettyfier

   Für Python:

```
$ uv add yapf
```

   Für Javascript:

```
$ npm install js-beautify
...
+ js-beautify@1.10.0
added 29 packages from 21 contributors and audited 32 packages in 2.632s
found 0 vulnerabilities
```

   Für R:

```
$ Rscript -e 'install.packages(c("formatR", "jsonlite"), repos="http://cran.rstudio.
↪com")'
Installiere Pakete nach ‘/usr/local/lib/R/3.6/site-library’
...
```

6. Highlighter

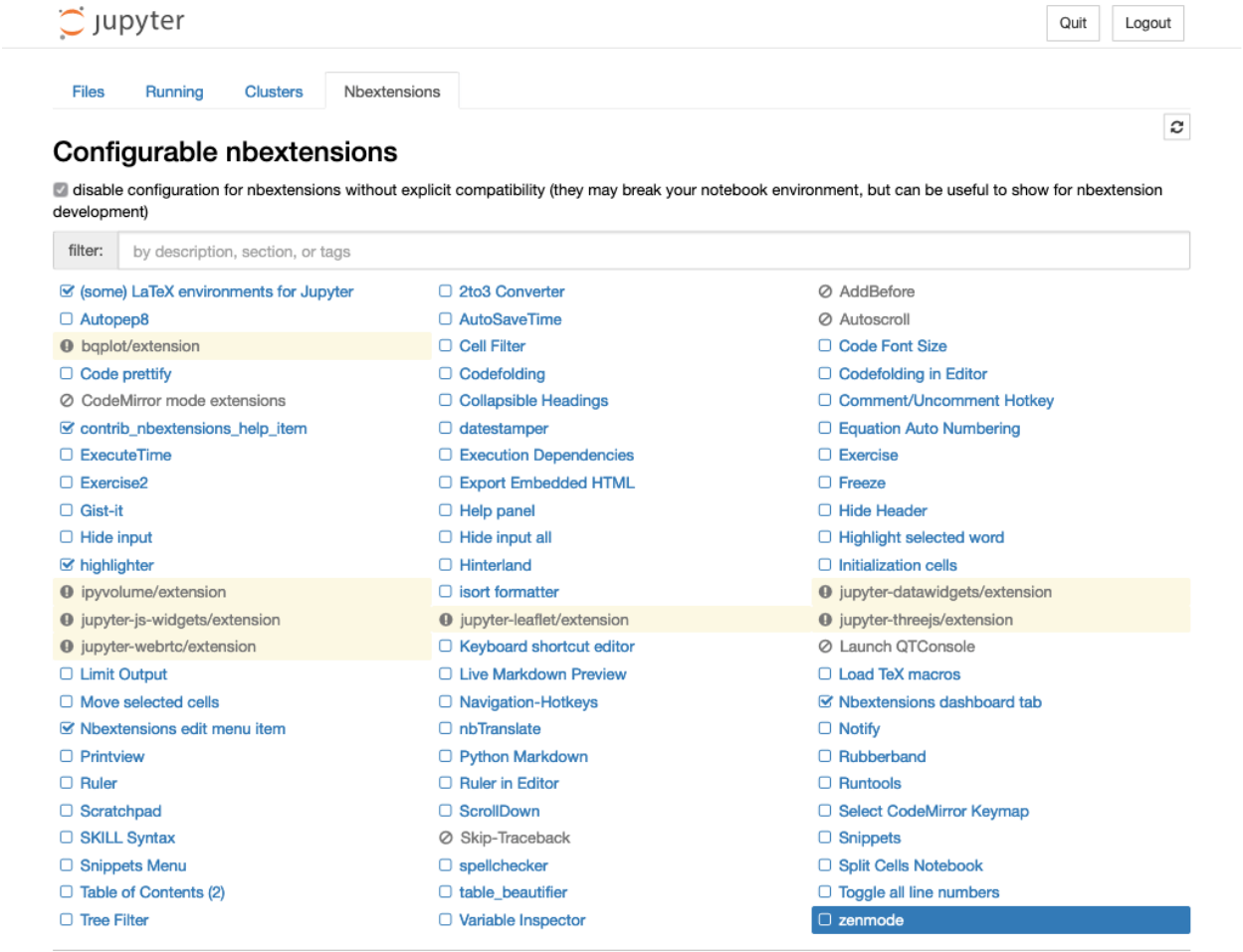
```
$ uv run jupyter nbextension install https://rawgit.com/jfbercher/small_
↪nbextensions/master/highlighter.zip  --user
$ uv run jupyter nbextension enable highlighter/highlighter
```

7. nbTranslate

```
$ uv add jupyter_latex_envs --upgrade --user
$ uv run jupyter nbextension install --py latex_envs --user
$ uv run jupyter nbextension enable --py latex_envs
```

11.2 Liste der Erweiterungen

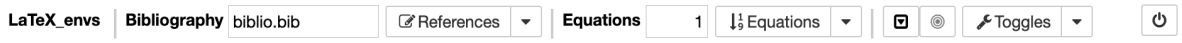
Ihr könnt die Notebook-Erweiterungen aktivieren und konfigurieren, indem ihr auf den Reiter *Nbextensions* klickt. Dort habt ihr Zugriff auf die Erweiterungen, die über Kontrollkästchen aktiviert/deaktiviert werden können. Zusätzlich werden für jede Erweiterung eine Dokumentation und Konfigurationsoptionen angezeigt.



Im Folgenden gebe ich einen kurzen Überblick über einige der Notebook-Erweiterungen.

(some) LaTeX environments for Jupyter notebook
: ermöglicht die Verwendung von Markdown-Zellen für LaTeX-Befehle und -Umgebungen. Zudem werden zwei

Menüs hinzugefügt: *LaTeX_envs* für die schnelle Auswahl der passenden LaTeX-Umgebung und *Some configuration options* weiteren Optionen:



Das Notebook kann anschließend als HTML oder LaTeX-Dokument exportiert werden.

Die Konfiguration der LaTeX-Umgebungen erfolgt in `user_envs.json` und für die Stile in `latex_env.css`. Weitere Umgebungen können in `user_envs.json` oder in `thmsInNb4.js` hinzugefügt werden (→ LaTeX-Environments doc).

jupyter-autopep8
formatiert/verschönert Code in Python-Code-Zellen. Die Erweiterung verwendet autopep8 und ist daher nur mit Python-Kernel zu verwenden.

A Code Prettifier
formatiert/verschönert Code in Notebook-Code-Zellen. Dabei wird der aktuelle Notebook-Kernel verwendet, weswegen das verwendete Prettifier-Paket in diesem Kernel verfügbar sein muss. Beispiel-Implementierungen werden für IPython-, R- und Javascript-Kernel bereitgestellt.

Limit Output
begrenzt die Anzahl der Zeichen, die eine Codezelle als Text oder HTML ausgibt. Dies unterbricht auch Endlosschleifen. Ihr könnt die Anzahl der Zeichen mit dem `ConfigManager` festlegen:

```
from notebook.services.config import ConfigManager

cm = ConfigManager().update("notebook", {"limit_output": 1000})
```

Nbextensions edit menu item
fügt ein Bearbeitungsmenü hinzu, um die Konfigurationsseite für `nbextensions` zu öffnen.

Printview
fügt eine Symbol hinzu, um die Druckansicht des aktuellen Notizbuchs in einem neuen Browser-Reiter anzuzeigen.

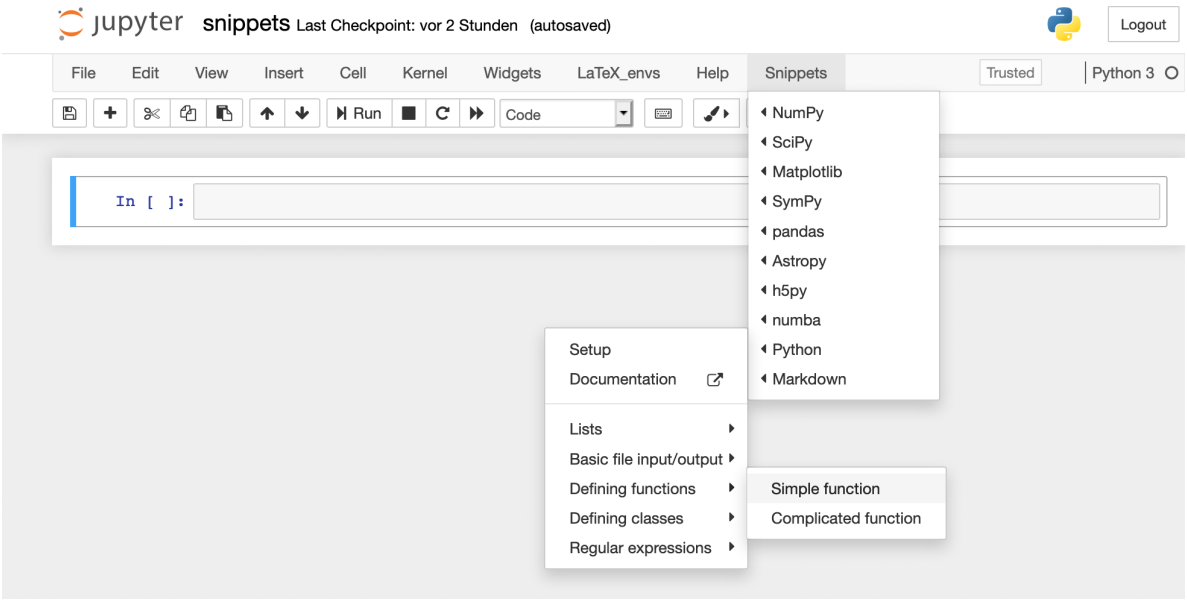
Ruler
fügt ein Lineal nach einer bestimmten Anzahl von Zeichen hinzu. Die Anzahl der Zeichen lässt sich mit dem `ConfigManager` festlegen:

```
from notebook.services.config import ConfigManager

ip = get_ipython()
cm = ConfigManager(parent=ip)
cm.update("notebook", {"ruler_column": [80]})
```

Scratchpad notebook extension
fügt dem Notizbuch eine Notizzelle hinzu. In dieser Zelle könnt ihr Code des aktuellen Kernel ausführen, ohne das Dokument zu ändern.

Snippets
fügt Notebooks ein konfigurierbares Menüelement hinzu um Snippets, Boilerplate und Codebeispiele einzufügen.



Ihr könnt auch eigene Menüeinträge definieren, s. Snippets.

Table of Contents (2)

ermöglicht es, alle Überschriften zu sammeln und in einem schwebenden Fenster, als Sidebar oder in einem Navigationsmenü anzuzeigen.

Falls Überschriften nicht im Inhaltsverzeichnis angezeigt werden sollen, geht dies mit:

```
## My title <a class="tocSkip">
```

Das Inhaltsverzeichnis lässt sich auch exportieren indem ein entsprechendes Template angegeben wird, also z.B.

```
$ jupyter nbconvert mynotebook.ipynb --template toc2
```

Eine allgemeine Dokumentation zu Vorlagen findet ihr in `Customizing exporters`.

Tree-filter

filtert im Jupyter-Dashboard nach Dateinamen.

A 2to3 converter

konvertiert in einer Code-Zelle Python2- in Python3-Code unter Verwendung der lib2to3-Bibliothek

Codefolding

ermöglicht Codefolding in Code-Zellen.

```
In [ ]:
class MyClass(object):
    """
    This is a test class
    """
    def afun(param1):
    """something gets computed here"""
        return param1ram1**2
```

Üblicherweise wird das Codefolding beim Export mit *nbconvert* beibehalten. Dies kann entweder in `jupyter_nbconvert_config.py` geändert werden mit:

```
c.CodeFoldingPreprocessor.remove_folded_code = True
```

oder auf der Kommandozeile mit

```
$ jupyter nbconvert --to html --CodeFoldingPreprocessor.remove_folded_code=True␣
↪mynotebook.ipynb
```

Collapsible Headings
: ermöglicht Notebooks, zusammenklappbare Abschnitte zu haben, die durch Überschriften getrennt werden.

Datestamper
: fügt die aktuelle Zeit und das aktuelle Datum in eine Zelle ein.

Hinterland
: ermöglicht Autovervollständigung.

Variable Inspector
: sammelt alle definierten Variablen und zeigt sie in einem schwebenden Fenster an.

Purpose
: lädt automatisch eine Reihe von Latex-Befehlen aus der Datei `latexdefs.tex` wenn ein Notizbuch geöffnet wird.

11.3 Plugin erstellen

Neben den bestehenden Notebook Extensions können auch weitere Plugins hinzugefügt werden. Das Verzeichnis, in dem `jupyter_contrib_nbextensions/nbextensions` liegt, bekommt ihr mit `pip show` heraus:

```
$ uv pip show jupyter_contrib_nbextensions
Name: jupyter-contrib-nbextensions
Version: 0.7.0
Location: /Users/veit/cusy/trn/jupyter-tutorial/.venv/lib/python3.13/site-packages
Requires: ipython-genutils, jupyter-contrib-core, jupyter-core, jupyter-highlight-
↪selected-word, jupyter-nbextensions-configurator, lxml, nbconvert, notebook, tornado,␣
↪traitlets
Required-by: jupyter-tutorial
```

In diesem Verzeichnis liegen die einzelnen Notebook-Erweiterungen, z.B. mit folgender Struktur:

```
$ tree
.
├── main.js
├── main.yaml
└── readme.md
```

`main.js`
: enthält die eigentliche Logik der Erweiterung, z.B.:

```
define([
    'require',
    'base/js/namespace',
], function (
    requirejs
    $,
    Jupyter,
) {
    "use strict";
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    // define default values for config parameters
    var params = {
        my_config_value : 100
    };

    var initialize = function () {
        $.extend(true, params, Jupyter.notebook.config.myextension);

        $('<link/>')
            .attr({
                rel: 'stylesheet',
                type: 'text/css',
                href: requirejs.toUrl('./myextension.css')
            })
            .appendTo('head');
    };

    var load_ipython_extension = function () {
        return Jupyter.notebook.config.loaded.then(initialize);
    };

    return {
        load_ipython_extension : load_ipython_extension
    };
});
```

main.yaml

yaml-Datei, die die Erweiterung für den Jupyter Extensions Configurator beschreibt.

```
Type: Jupyter Notebook Extension
Compatibility: 3.x, 4.x, 5.x, 6.x
Name: My notebook extensions
Main: main.js
Link: README.md
Description: |
  My notebook extension helps with the use of Jupyter notebooks.
Parameters:
- none
```

Weitere Informationen zu den vom *Configurator* unterstützten Optionen findet ihr auf GitHub: jupyter_nbextensions_configurator.

readme.md

Markdown-Datei, die die Erweiterung beschreibt und angibt, wie sie verwendet werden kann. Dies wird auch im Reiter *Nbextensions* angezeigt.

Siehe auch

- Notebook extension structure

11.3.1 Setup Jupyter Notebook Extension

Dies ist eine Erweiterung, die einige Probleme beim Arbeiten mit Notebooks behebt, die Joel Grus auf der JupyterCon 2018 vorgetragen hat: I Don't Like Notebooks:

- sie fordert euch auf, das Notebook zu benennen
- sie erstellt eine Vorlage, um die Dokumentation zu verbessern
- sie importiert und konfiguriert häufig verwendete Bibliotheken

Installation

1. Findet heraus, wo die Notebook-Extensions installiert sind:

```
$ uv pip show jupyter_contrib_nbextensions
Name: jupyter-contrib-nbextensions
Version: 0.7.0
Location: /Users/veit/cusy/trn/jupyter-tutorial/.venv/lib/python3.13/site-packages
Requires: ipython-genutils, jupyter-contrib-core, jupyter-core, jupyter-highlight-
↪selected-word, jupyter-nbextensions-configurator, lxml, nbconvert, notebook,␣
↪tornado, traitlets
Required-by: jupyter-tutorial
```

2. Ladet das Setup-Verzeichnis in `jupyter_contrib_nbextensions/nbextensions/` herunter.
3. Installiert die Erweiterung mit

```
$ uv run --with jupyter jupyter contrib nbextensions install --user
...
[I 10:54:46 InstallContribNbextensionsApp] Installing /Users/veit/.local/share/
↪virtualenvs/jupyter-tutorial--q5BvmfG/lib/python3.7/site-packages/jupyter_contrib_
↪nbextensions/nbextensions/setup -> setup
[I 10:54:46 InstallContribNbextensionsApp] Making directory: /Users/veit/Library/
↪Jupyter/nbextensions/setup/
[I 10:54:46 InstallContribNbextensionsApp] Copying: /Users/veit/.local/share/
↪virtualenvs/jupyter-tutorial--q5BvmfG/lib/python3.7/site-packages/jupyter_contrib_
↪nbextensions/nbextensions/setup/setup.yaml -> /Users/veit/Library/Jupyter/
↪nbextensions/setup/setup.yaml
[I 10:54:46 InstallContribNbextensionsApp] Copying: /Users/veit/.local/share/
↪virtualenvs/jupyter-tutorial--q5BvmfG/lib/python3.7/site-packages/jupyter_contrib_
↪nbextensions/nbextensions/setup/README.md -> /Users/veit/Library/Jupyter/
↪nbextensions/setup/README.md
[I 10:54:46 InstallContribNbextensionsApp] Copying: /Users/veit/.local/share/
↪virtualenvs/jupyter-tutorial--q5BvmfG/lib/python3.7/site-packages/jupyter_contrib_
↪nbextensions/nbextensions/setup/main.js -> /Users/veit/Library/Jupyter/
↪nbextensions/setup/main.js
[I 10:54:46 InstallContribNbextensionsApp] - Validating: OK
...
```

4. Aktiviert die *Setup*-Extension in *Nbextensions*.

Schließlich könnt ihr ein neues Notebook erstellen, das dann folgende Struktur aufweist: *setup.ipynb*.

Siehe auch

- Set Your Jupyter Notebook up Right with this Extension

- GitHub

11.4 setup.ipynb

11.4.1 Introduction

State notebook purpose here

Imports

Import libraries and write settings here.

[1]:
```
# Data manipulation
import pandas as pd
import numpy as np

# Options for pandas
pd.options.display.max_columns = 50
pd.options.display.max_rows = 30

# Display all cell outputs
from IPython.core.interactiveshell import InteractiveShell

InteractiveShell.ast_node_interactivity = "all"

from IPython import get_ipython

ipython = get_ipython()

# autoreload extension
if "autoreload" not in ipython.extension_manager.loaded:
    %load_ext autoreload

%autoreload 2

# Visualizations
import chart_studio.plotly as py
import plotly.graph_objs as go
from plotly.offline import iplot, init_notebook_mode

init_notebook_mode(connected=True)

import cufflinks as cf

cf.go_offline(connected=True)
cf.set_config_file(theme="white")
```

11.4.2 Analysis/Modeling

Do work here

11.4.3 Results

Show graphs and stats here

11.4.4 Conclusions and Next Steps

Summarize findings here

11.5 ipylayout

`ipylayout` basiert auf GoldenLayout, einem Multi-Screen-Layout-Manager für Webanwendungen.

11.5.1 Installation

`ipylayout` kann einfach mit uv installiert werden:

```
$ uv add ipylayout
Installing ipylayout...
...
```

Sofern noch nicht geschehen, wird auch `ipywidgets` mitinstalliert.

11.5.2 Beispiel

Für das folgende Beispiel benötigt ihr zusätzlich noch die Python-Pakete `ipyleaflet` und `ipympl`.

[1]:
```
%matplotlib widget
import ipylayout
import ipyleaflet
import ipywidgets
import matplotlib.pyplot as plt
import numpy as np


plt.ioff()
```

[1]:
```
<contextlib.ExitStack at 0x1085db1d0>
```

[2]:
```
# create a plot

fig = plt.figure()
fig.canvas.header_visible = False
fig.canvas.layout.min_height = "300px"
fig.canvas.layout.width = "100%"
plt.title("Plotting: y=sin(x)")

x = np.linspace(0, 20, 500)
lines = plt.plot(x, np.sin(x))
```

[3]:
```
# create a slider

slider = ipywidgets.FloatSlider()
```

[4]:
```
# create a map

m = ipyleaflet.Map(
    center=(52.204793, 360.121558),
    zoom=4
)
```

[5]:
```
# create a layout

l = ipylayout.Layout(layout=ipywidgets.Layout(width="100%", height="800px"))
l.theme = "light"  # light or dark
l.config = {
    "content": [
        {
            "type": "row",
            "content": [
                {
                    "type": "component",
                    "componentName": "name0",
                    "componentState": {"label": "A"},
                },
                {
                    "type": "column",
                    "content": [
                        {
                            "type": "component",
                            "componentName": "name1",
                            "componentState": {"label": "B"},
                        },
                        {
                            "type": "component",
                            "componentName": "name2",
                            "componentState": {"label": "C"},
                        },
                    ],
                },
            ],
        }
    ]
}
l.components = {"name0": slider, "name1": m, "name2": fig.canvas}
```

[6]:
```
l
```

```
Layout(config={'content': [{'type': 'row', 'content': [{'type': 'component',
↪'componentName': 'name0', 'compon...
```

ipylayout

ipylayout basiert auf GoldenLayout, einem Multi-Screen-Layout-Manager für Webanwendung

Installation

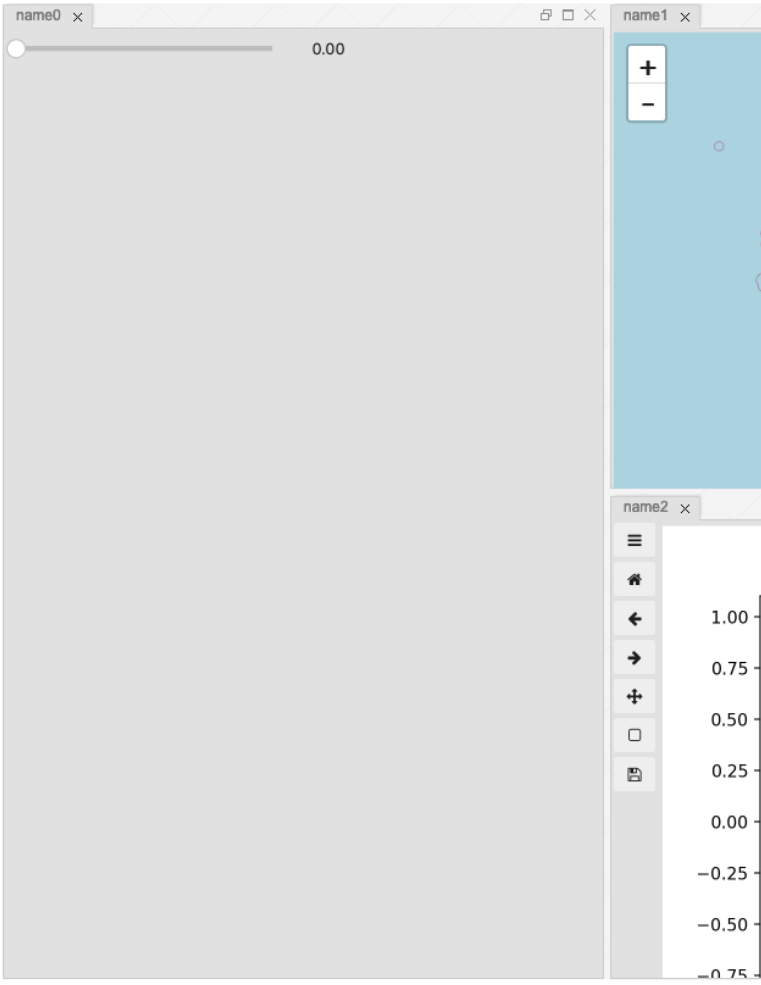
ipylayout kann einfach mit pipenv installiert werden:

```
$ pipenv install ipylayout
Installing ipylayout…
…
```

Sofern noch nicht geschehen, wird auch ipywidgets mitinstalliert.

Beispiel

Für das folgende Beispiel benötigt ihr zusätzlich noch die Python-Pakete ipyleaflet und ip



ipylayout kann auch zusammen mit *Voilà* verwendet werden:

KAPITEL 12

Daten visualisieren

Wir haben die Visualisierung von Daten in ein eigenes Tutorial ausgelagert: PyViz Tutorial.

KAPITEL 13

Dashboards

Voilà
: wurde von QuantStack entwickelt als Dashboard-Lösung auf Basis von Jupyter Notebooks und *ipywidgets*. Es zeigt die Ausgabe aller Notebook-Zellen an.

Panel
: wurde auf Basis von Bokeh und Param entwickelt und bietet ein Python-Toolkit speziell zur Erstellung von Apps und Dashboards, das jedoch nicht nur Bokeh-Plots unterstützt. Es erlaubt mehrstufige und mehrseitige Workflows.

Mit diesem tabellarischen Überblick könnt ihr schnell die Aktivitäten und Lizenzen der verschiedenen Bibliotheken vergleichen.

Tab. 1: GitHub-Insights: Dashboards

Name	Stars	Mitwirkende	Commit-Aktivität	Lizenz
Voilà	Stars 6k	contributors 73	commit activity 12/year	license not identifiable by github
Panel	Stars 5.8k	contributors 246	commit activity 583/year	license BSD-3-Clause

Tipp

cusy Seminare:

- Daten visualisieren mit Python
- Datenvisualisierungen gestalten
- Dashboards erstellen

13.1 Voilà vs. Panel

Ein großer Unterschied zwischen Panel und Voilà besteht in der Verarbeitung der Notebooks: Voilà baut direkt auf dem Notebookformat auf und übernimmt die gesamte Ausgabe in das Voilà-Dashboard, während in Panel die Ausgabe einer Notebook-Zelle explizit als Panel-Objekt deklariert werden muss. Voilà hat daher den Vorteil, dass bestehende Notizbücher unverändert verwendet werden können. Wenn jedoch nicht alle Notebook-Zellen in das Dashboard übernommen werden sollen, müssen zwei ähnliche Notebooks gepflegt werden – eines für Literate Programming und eines für das Dashboard. Soll in einem Dashboard jedoch die umgekehrte Pyramide für das Storytelling verwendet werden, sind in beiden Fällen zwei Notebooks erforderlich.

13.1.1 Skalierbarkeit

Voilà und Panel basieren auf Tornado, aber sie unterscheiden sich dadurch, dass Voilà für jede Person einen neuen Jupyter-Kernel startet, während der Bokeh-Server mehrere Personen mit demselben Python-Prozess bedient. Dieser Unterschied hat im Wesentlichen zwei Auswirkungen:

- Der Overhead pro Person für ein Dashboard ist beim Bokeh-Server viel geringer als bei Voilà: Nachdem die Bibliotheken importiert sind, fällt nur noch ein winziger Overhead für die Erstellung jeder neuen Session an. Für eine Session, die pandas und Matplotlib importiert, beträgt der Overhead pro Benutzer ca. 75 MB, und die Anzahl der Personen, die ein Voilà-Server für eine bestimmte Anwendung verarbeiten kann, verringert. Auch sind die Start- und Datenzugriffszeiten meist langsamer.
- Da sich ein Bokeh-Server einen einzigen Prozess für mehrere Sessions teilt, können Daten oder Verarbeitungen gegebenenfalls auch zwischen den verschiedenen Sessions geteilt werden.

13.1.2 Multi-Page-App

Voilà ist nicht für mehrseitige Anwendungen konzipiert während Panel mehrere Optionen bietet für die Erstellung von mehrseitigen Apps bietet, einschließlich *Pipelines* und Übersichtsseite mit einer Sammlung unabhängiger Dashboards und Apps.

13.1.3 Autorisierung und Authentifizierung

Üblicherweise sollte das Dashboard nicht für die Authentifizierung und Autorisierung verwendet werden, sondern an einen Dienst delegiert werden. ContainDS Dashboards ist ein Beispiel für eine *JupyterHub*-Erweiterung, die dies Dashboard-unabhängig tut. Authentifizierung und Autorisierung kann mit einem der folgenden Tools auch über einen Webserver erfolgen:

- Traefik Forward Auth
- OAuth2 Proxy
- PyCasbin

Wenn ihr dennoch die Dashboarding-Bibliothek die Authentifizierung machen lassen wollt, gibt es einige Optionen unterschiedlicher Reife:

- Panel basiert auf Bokeh, das Authentifizierung bietet, und Panel wird mit einer Reihe von OAuth-Anbietern ausgeliefert, z.B. GitHub, GitLab und Azure.

> **Siehe auch**
>
> Configuring Authentication

- Voilà kann die Authentifizierung von *JupyterHub* wiederverwenden.

13.1.4 BI-Tool

Sowohl Voilà wie auch Panel setzen Programmierkenntnisse voraus, um Dashboards erstellen zu können. Das auf Panel aufbauende Lumen bietet jedoch eine vielversprechend BI-ähnliche Benutzeroberfläche.

13.2 Voilà

Voilà wurde von QuantStack entwickelt als Dashboard-Lösung auf Basis von Jupyter Notebooks und *ipywidgets*. Es zeigt die Ausgabe aller Notebook-Zellen an.

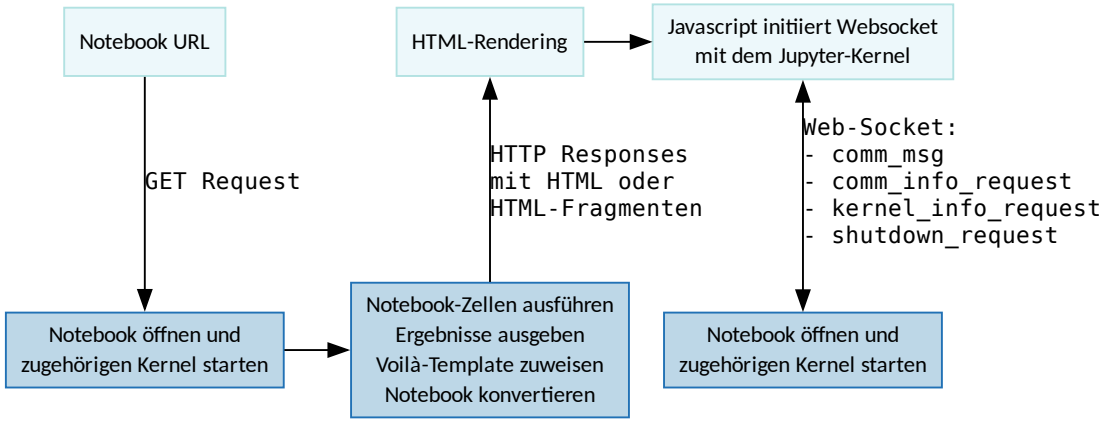
Siehe auch

And voilà!

13.2.1 Features

- Voilà unterstützt *interaktive Jupyter-Widgets*, einschließlich der Roundtrips zum Kernel. Auch benutzerdefinierte Widgets wie bqplot, ipyleaflet, ipyvolume, ipympl, *ipysheet*, plotly, ipywebrtc usw. werden unterstützt.
- Voilà erlaubt keine willkürliche Ausführung von Code durch Nutzer von Dashboards.
- Voilà basiert auf Jupyter-Standardprotokollen und -Dateiformaten und funktioniert mit jedem *Jupyter-Kernel*: C++, Python, Julia. Dies macht es zu einem sprachunabhängigen Dashboard-System.
- Voilà ist erweiterbar. Es enthält ein flexibles *Template*-System zur Erstellung umfangreicher Layouts.

13.2.2 Ausführungsmodell



Ein wichtiger Aspekt dieses Ausführungsmodells ist, dass vom Frontend nicht angegeben werden kann, welcher Code vom Backend ausgeführt wird. Sofern mit der Option `--strip-sources=False` nicht anders angegeben, gelangt der Quellcode des gerenderten Notizbuchs noch nicht einmal an das Frontend. Die Voilà-Instanz des `jupyter_server` erlaubt standardmäßig keine Ausführungsanforderungen.

Siehe auch

- Voilà Gallery
- And voilà!

Installation und Nutzung

Installation

voilà kann installiert werden mit:

```
$ uv add voila
```

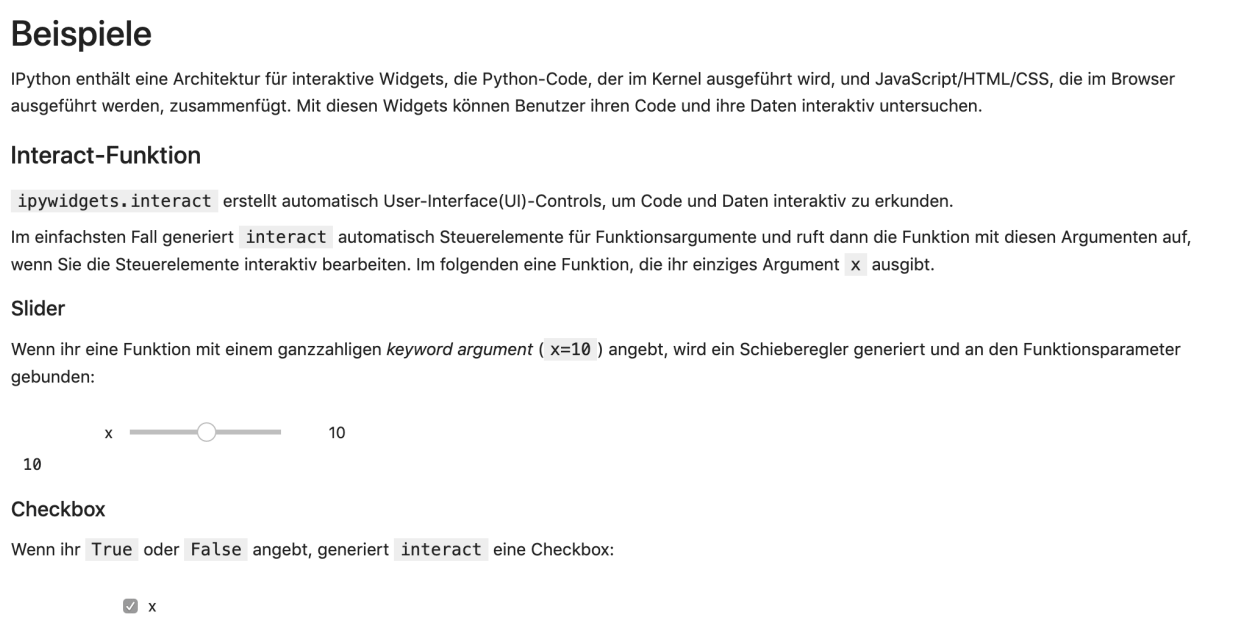
Starten

... als eigenständige Anwendung

Ihr könnt die Installation überprüfen, indem ihr mit voilà z.B. `bqplot_vuetify_example.ipynb` aufruft. Für dieses Notebook müsst ihr jedoch zunächst noch `bqplot` und `ipyvuetify` installieren.

```
$ uv run voila 'docs/dashboards/voila/bqplot_vuetify_example.ipynb'
...
[Voila] Voilà is running at:
http://localhost:8866/
```

Hierbei sollte sich euer Standardbrowser öffnen und die `voila`-Beispiele aus unserem Tutorial anzeigen:



Alternativ könnt ihr euch auch ein Verzeichnis anzeigen lassen mit allen darin enthaltenen Notebooks:

```
$ uv run voila 'docs/dashboards/voila/'
```

Select items to open with voila.
libs
examples.ipynb
networkx.ipynb
custom-widget.ipynb
widget-list.ipynb
widget-events.ipynb

Es ist auch möglich, sich den Quellcode anzeigen zu lassen mit:

```
$ uv run voila --strip_sources=False 'docs/dashboards/voila/bqplot_vuetify_example.ipynb'
```

Bemerkung

Beachtet, dass der Code nur angezeigt wird. Voilà erlaubt Benutzern nicht, den Code zu bearbeiten oder auszuführen.

Beispiele

IPython enthält eine Architektur für interaktive Widgets, die Python-Code, der im Kernel ausgeführt wird, und JavaScript/HTML/CSS, die im Browser ausgeführt werden, zusammenfügt. Mit diesen Widgets können Benutzer ihren Code und ihre Daten interaktiv untersuchen.

Interact-Funktion

`ipywidgets.interact` erstellt automatisch User-Interface(UI)-Controls, um Code und Daten interaktiv zu erkunden.

In [1]:
```
from __future__ import print_function
from ipywidgets import interact, interactive, fixed, interact_manual
import ipywidgets as widgets
```

Im einfachsten Fall generiert `interact` automatisch Steuerelemente für Funktionsargumente und ruft dann die Funktion mit diesen Argumenten auf, wenn Sie die Steuerelemente interaktiv bearbeiten. Im folgenden eine Funktion, die ihr einziges Argument `x` ausgibt.

In [2]:
```
def f(x):
    return x
```

Slider

Wenn ihr eine Funktion mit einem ganzzahligen *keyword argument* (`x=10`) angebt, wird ein Schieberegler generiert und an den Funktionsparameter gebunden:

In [3]:
```
interact(f, x=10);
```

x 10

10

Checkbox

Wenn ihr `True` oder `False` angebt, generiert `interact` eine Checkbox:

In [4]:
```
interact(f, x=True);
```

x

True

Textbereich

Üblicherweise wird das `light`-Theme verwendet; ihr könnt jedoch auch das `dark`-Theme auswählen:

```
$ uv run voila --theme=dark 'docs/dashboards/voila/bqplot_vuetify_example.ipynb'
```

... als Erweiterung des Jupyter-Server

Alternativ könnt ihr voilà auch als Erweiterung des Jupyter-Server starten:

```
$ uv run jupyter notebook
```

Anschließend könnt ihr voilà aufrufen, z.B. unter der URL `http://localhost:8888/voila`.

Templating

Voila-Gridstack

gridstack.js ist ein jQuery-Plugin für Widget-Layouts. Dies ermöglicht mehrspaltige Drag & Drop-Raster und anpassbare, für Bootstrap v3 geeignete Layouts. Zudem funktioniert es mit knockout.js und Touch-Geräten.

Das Gridstack-Voilà-Template verwendet die Metadaten der Notebook-Zellen, um das Layout des Notebooks zu gestalten. Es soll die gesamte Spezifikation für die veralteten Jupyter Dashboards Layout Extension unterstützen.



voila-material

voila-material ist ein Template zur Verwendung von Voilà mit dem Material Design Component Framework.

Installation

```
$ uv add voila-material
```

Verwendung

`voila-material` könnt ihr verwenden mit `voila MY_NOTEBOOK.ipynb --template=material` oder für das Dark-Theme mit `voila MY_NOTEBOOK.ipynb --template=material --theme=dark`.

voila-vuetify

voila-vuetify ist ein Template zur Verwendung von Voilà mit Vuetify.js.

Installation

```
$ uv add voila-vuetify
```

Verwendung

Um `voila-vuetify` in einem Notebook zu verwenden, müsst ihr zunächst `ipyvuetify` importieren:

```
import ipyvuetify as v
```

Anschließend könnt ihr ein Layout erstellen z.B. mit:

```
v.Tabs(
    _metadata={"mount_id": "content-main"},
    children=[
        v.Tab(children=["Tab1"]),
        v.Tab(children=["Tab2"]),
        v.TabItem(
            children=[
                v.Layout(
                    row=True,
                    wrap=True,
                    align_center=True,
                    children=[
                        v.Flex(
                            xs12=True,
                            lg6=True,
                            xl4=True,
                            children=[fig, slider],
                        ),
                        v.Flex(
                            xs12=True,
                            lg6=True,
                            xl4=True,
                            children=[figHist2, sliderHist2],
                        ),
                        v.Flex(xs12=True, xl4=True, children=[fig2]),
                    ],
                )
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

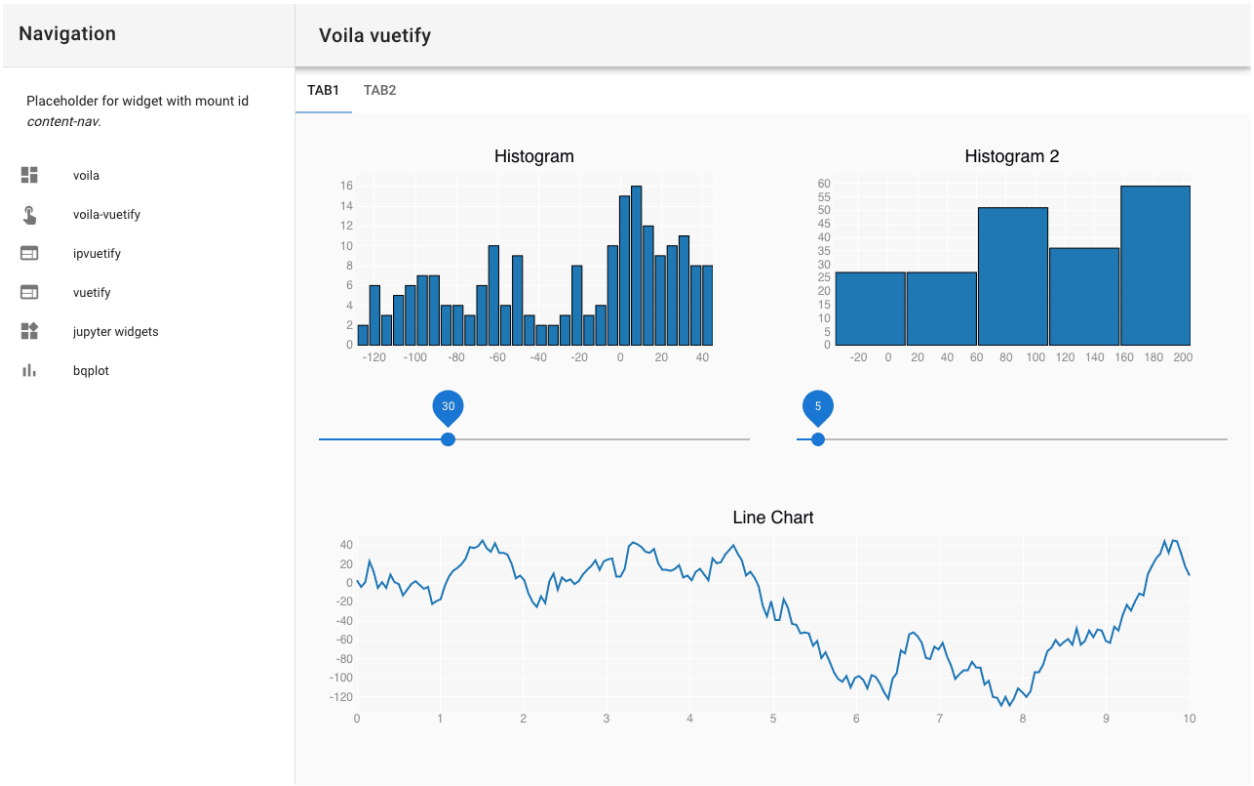
```
            ]
        ),
        v.TabItem(children=[v.Container(children=["Lorum ipsum"])]),
    ],
)
```

bqplot_vuetify_example.ipynb. könnt ihr nutzen mit:

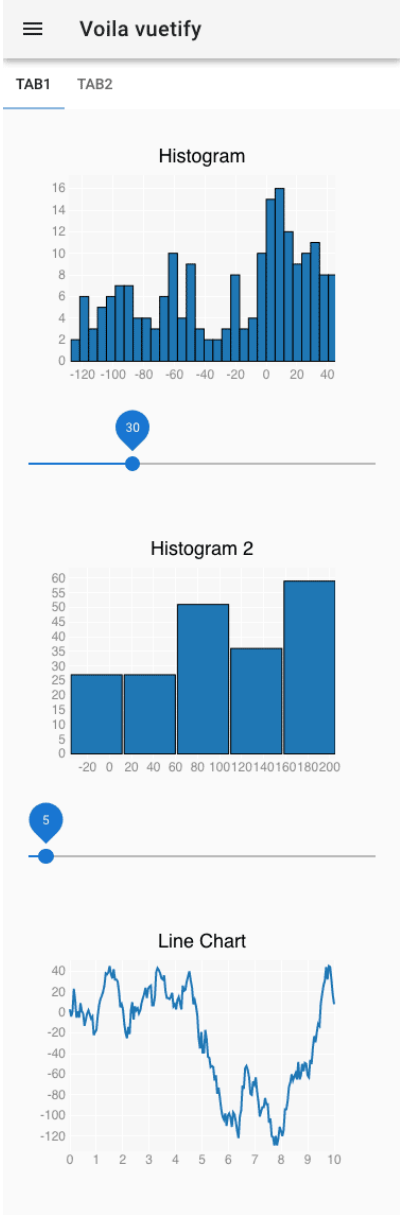
```
$ uv run voila --template vuetify-default 'bqplot_vuetify_example.ipynb'
```

Anschließend öffnet sich euer Standardbrowser mit der URL `http://localhost:8866/` und zeigt euch die Plots im Responsive Material Design.

Beispiel für Voilà-vuetify mit der Monitorauflösung eines Laptop MDPI-Screen:



Beispiel für Voilà-vuetify mit der Monitorauflösung eine iPhone X:



voila-reveal

voila-reveal ist ein Template für Slideshows basierend auf RevealJS. Es wird bereits mit voilà installiert.

Verwendung

Ihr könnt das Template nutzen mit:

```
$ uv run voila --template=reveal 'reveal.ipynb'
```

Durch zusätzliche Optionen können die Standardeinstellungen überschrieben werden, z.B. um den Standardwert für den Übergang `Fade` mit `Zoom` zu überschrieben mit:

```
$ uv run voila --template=reveal --VoilaConfiguration.resources="{'reveal': {'transition
↪': 'zoom'}}" 'reveal.ipynb'
```

Sollen Konfigurationsoptionen dauerhaft gespeichert werden, so kann eine Datei `conf.json` in `./.venv/share/jupyter/nbconvert/templates/reveal/` angelegt werden:

```
{
  "traitlet_configuration": {
    "resources": {
      "reveal": {
        "scroll": false,
        "theme": "simple",
        "transition": "zoom"
      }
    }
  }
}
```

Ihr könnt euer Notebook dann in eine Slideshow verwandeln in *View* → *Cell Toolbar* → *Slideshow*. In der Werkzeugleiste einer könnt ihr auswählen zwischen

Slide
: von links nach rechts

Sub-Slide
: von oben nach unten

Fragment
: Stop innerhalb einer Folie

Notes
: Anmerkungen für Sprecher*innen, die beim Drücken der `t`-Taste in einem neuen Fenster geöffnet werden

Wenn Ihr eure Vortragsfolien auf binder veröffentlichen wollt, müsst Ihr den folgenden Tag in die Metadaten eures Notebooks schreiben in *Edit* → *Edit Notebook Metadata*:

```
"rise": {
    "autolaunch": true
}
```

Ihr könnt ebenfalls das chalkboard reveal-Plugin verwenden wenn Ihr die Metadaten des Notebooks erweitert um:

```
"rise": {
  "enable_chalkboard": true
}
```

Eigene Templates erstellen

Ein Voilà-Template ist ein Ordner, der sich im Virtual-Environment unter `.venv/share/jupyter/voila/templates` befindet und z.B. Folgendes enthält:

```
.venv/share/jupyter/voila/templates/mytheme
├── conf.json
├── nbconvert_templates
│   └── voila.tpl
├── static
│   ├── mytheme.js
│   └── mytheme.css
└── templates
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    ├── 404.html
    ├── browser-open.html
    ├── error.html
    ├── page.html
    └── tree.html
```

conf.json
Konfigurationsdatei, die z.B. auf das Basis-Template verweist:

```
{"base_template": "default"}
```

nbconvert_templates
Benutzerdefinierte Templates für *nbconvert*.

static
Verzeichnis für statische Dateien.

templates
Benutzerdefinierte Tornado-Templates.

Cookiecutter-Template für Voilà

voila-template-cookiecutter ist eine Vorlage für Cookiecutter, das euch den Start erleichtern kann.

Anpassen von Voila mit Hooks

Voilà bietet Hooks, die euch ermöglichen, das Verhalten an eure speziellen Bedürfnisse anzupassen. Diese Hooks ermöglichen, benutzerdefinierte Funktionen an bestimmten Punkten während der Ausführung von Voilà einzubinden, wodurch ihr die Kontrolle über die Ausführung von Notebooks und die Konfiguration des Frontends erhaltet.

Derzeit unterstützt Voilà die folgenden Hooks:

prelaunch_hook
Zugriff und Änderung der Tornado-Anfrage und des Notizbuchs vor der Ausführung, um nach Authentifizierungs-Cookies zu suchen, auf Details der Request-Header zuzugreifen oder das Notebook vor dem Rendering zu ändern z.B.:

```
def prelaunch_hook(req: tornado.web.RequestHandler,
         notebook: nbformat.NotebookNode,
         cwd: str) -> Optional[nbformat.NotebookNode]:
```

req
verweist auf den Tornado RequestHandler, mit dem ihr Parameter, Kopfzeilen usw. (und so weiter) prüfen könnt.

notebook
verweist auf `NotebookNode`, den ihr verändern könnt, um z.B. Zellen zu injizieren oder andere Änderungen auf Notebookebene vorzunehmen.

cwd
ist das aktuelle Arbeitsverzeichnis, falls ihr etwas auf der Festplatte ändern wollt.

Der Rückgabewert eurer Hook-Funktion kann entweder `None` oder ein `NotebookNode` sein.

page_config_hook
passt das `page_config`-Objekt an, das die Konfiguration des Voilà-Frontends steuert. Frontend-Einstellungen wie die URLs für statische Assets oder andere Konfigurationsparameter können so geändert werden.

Der Standard-`page_config_hook` sieht so aus:

```
page_config = {
  "appVersion": __version__,
  "appUrl": "voila/",
  "themesUrl": "/voila/api/themes",
  "baseUrl": base_url,
  "terminalsAvailable": False,
  "fullStaticUrl": url_path_join(base_url, "voila/static"),
  "fullLabextensionsUrl": url_path_join(base_url, "voila/labextensions"),
  "extensionConfig": voila_configuration.extension_config,
}
```

Es gibt zwei Möglichkeiten, die Hook-Funktion zu Voilà hinzuzufügen:

- eine `voila.py`-Konfigurationsdatei in dem Verzeichnis, in dem ihr Voilà startert

```
def prelaunch_hook_function(req, notebook, cwd):
    """Add your prelaunch hook heere"""
    return notebook


def page_config_hook_function(current_page_config, **kwargs):
    """Modify the current_page_config"""
    return new_page_config


c.VoilaConfiguration.prelaunch_hook = hook_function
c.VoilaConfiguration.page_config_hook = page_config_hook
```

- ein Python-Skript, das Voilà startet, z.B.

```
def parameterize_with_papermill(req, notebook, cwd):
    import tornado

    # Grab parameters
    parameters = req.get_argument("parameters", {})

    # try to convert to dict if not e.g. string/unicode
    if not isinstance(parameters, dict):
        try:
            parameters = tornado.escape.json_decode(parameters)
        except ValueError:
            parameters = None

    # if passed and a dict, use papermill to inject parameters
    if parameters and isinstance(parameters, dict):
        from papermill.parameterize import parameterize_notebook

        # setup for papermill
        # these two blocks are done to avoid triggering errors in
        # papermill's notebook loading logic
        for cell in notebook.cells:
            if "tags" not in cell.metadata:
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                cell.metadata.tags = []
            if "papermill" not in notebook.metadata:
                notebook.metadata.papermill = {}

        # Parameterize with papermill
        return parameterize_notebook(notebook, parameters)


def page_config_hook(
    current_page_config: Dict[str, Any],
    base_url: str,
    settings: Dict[str, Any],
    log: Logger,
    voila_configuration: VoilaConfiguration,
    notebook_path: str,
):
    page_config["fullLabextensionsUrl"] = "/custom/labextensions_url"
    return page_config
```

Ihr könnt beide Hooks gleichzeitig verwenden, indem ihr eurer Voilà-App folgendes hinzufügt:

```
from voila.app import Voila
from voila.config import VoilaConfiguration

config = VoilaConfiguration()
config.prelaunch_hook = parameterize_with_papermill
config.page_config_hook = page_config_hook
app = Voila()
app.voila_configuration = config

app.start()
```

bqplot_vuetify_example.ipynb

Import

[1]:
```
import ipyvuetify as v
```

Erster Histogrammplot

[2]:
```
import bqplot
import ipywidgets as widgets
import numpy as np

from bqplot import pyplot as plt


n = 200

x = np.linspace(0.0, 10.0, n)
y = np.cumsum(np.random.randn(n) * 10).astype(int)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
fig = plt.figure(title="Histogram")
np.random.seed(0)
hist = plt.hist(y, bins=25)
hist.scales["sample"].min = float(y.min())
hist.scales["sample"].max = float(y.max())
fig.layout.width = "auto"
fig.layout.height = "auto"
fig.layout.min_height = "300px"  # so it shows nicely in the notebook
fig
```

```
Figure(axes=[Axis(orientation='vertical', scale=LinearScale()),␣
↪Axis(scale=LinearScale(max=140.0, min=-39.0))]...
```

Slider

[3]:
```
slider = v.Slider(thumb_label="always", class_="px-4", v_model=30)
widgets.link((slider, "v_model"), (hist, "bins"))
slider
```

```
Slider(class_='px-4', layout=None, thumb_label='always', v_model=30)
```

Liniendiagramm

[4]:
```
fig2 = plt.figure(title="Line Chart")
np.random.seed(0)
p = plt.plot(x, y)

fig2.layout.width = "auto"
fig2.layout.height = "auto"
fig2.layout.min_height = "300px"  # so it shows nicely in the notebook

fig2
```

```
Figure(axes=[Axis(scale=LinearScale()), Axis(orientation='vertical',␣
↪scale=LinearScale())], fig_margin={'top':...
```

`BrushIntervalSelector` hinzufügen

[5]:
```
brushintsel = bqplot.interacts.BrushIntervalSelector(scale=p.scales["x"])


def update_range(*args):
    if brushintsel.selected is not None and brushintsel.selected.shape == (2,):
        mask = (x > brushintsel.selected[0]) & (x < brushintsel.selected[1])
        hist.sample = y[mask]


brushintsel.observe(update_range, "selected")
fig2.interaction = brushintsel
```

Zweiter Histogrammplot

[6]:
```
n2 = 200

x2 = np.linspace(0.0, 10.0, n)
y2 = np.cumsum(np.random.randn(n) * 10).astype(int)

figHist2 = plt.figure(title="Histogram 2")
np.random.seed(0)
hist2 = plt.hist(y2, bins=25)
hist2.scales["sample"].min = float(y2.min())
hist2.scales["sample"].max = float(y2.max())
figHist2.layout.width = "auto"
figHist2.layout.height = "auto"
figHist2.layout.min_height = "300px"  # so it shows nicely in the notebook

sliderHist2 = v.Slider(
    _metadata={"mount_id": "histogram_bins2"},
    thumb_label="always",
    class_="px-4",
    v_model=5,
)
from traitlets import link


link((sliderHist2, "v_model"), (hist2, "bins"))


display(figHist2)
display(sliderHist2)
```

```
Figure(axes=[Axis(orientation='vertical', scale=LinearScale()),␣
↪Axis(scale=LinearScale(max=205.0, min=-37.0))]...
```

```
Slider(class_='px-4', layout=None, thumb_label='always', v_model=5)
```

Voila vuetify-Layout einrichten

Das Voila vuetify-Template zeigt nicht die Ausgabe des Jupyter Notebook an, sondern nur das Widget mit den `mount_id`-Metadaten.

[7]:
```
v.Tabs(
    _metadata={"mount_id": "content-main"},
    children=[
        v.Tab(children=["Tab1"]),
        v.Tab(children=["Tab2"]),
        v.TabItem(
            children=[
                v.Layout(
                    row=True,
                    wrap=True,
                    align_center=True,
                    children=[
                        v.Flex(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
                            xs12=True,
                            lg6=True,
                            xl4=True,
                            children=[fig, slider],
                        ),
                        v.Flex(
                            xs12=True,
                            lg6=True,
                            xl4=True,
                            children=[figHist2, sliderHist2],
                        ),
                        v.Flex(xs12=True, xl4=True, children=[fig2]),
                    ],
                )
            ]
        ),
        v.TabItem(children=[v.Container(children=["Lorum ipsum"])]),
    ],
)
```

```
Tabs(children=[Tab(children=['Tab1'], layout=None), Tab(children=['Tab2'], layout=None),␣
↪TabItem(children=[Lay...
```

debug.ipynb

[1]:
```
import ipywidgets as widgets


slider = widgets.FloatSlider(description="x")
text = widgets.FloatText(disabled=True, description="$x^2$")


def compute(*ignore):
    text.value = str(slider.value**2)


slider.observe(compute, "value")
slider.value = 14
widgets.VBox([slider, text])
```

```
VBox(children=(FloatSlider(value=14.0, description='x'), FloatText(value=196.0,␣
↪description='$x^2$', disabled=...
```

[2]:
```
import os


def kill_kernel(change):
    os._exit(0)


button = widgets.Button(description="Kill Kernel")
button.on_click(kill_kernel)
button
```

```
Button(description='Kill Kernel', style=ButtonStyle())
```

13.3 Panel

Panel wurde auf Basis von Bokeh und Param entwickelt und bietet ein Python-Toolkit speziell zur Erstellung von Apps und Dashboards, das jedoch nicht nur Bokeh-Plots unterstützt. Es erlaubt mehrstufige und mehrseitige Workflows.

Siehe auch

- Panel Announcement
- Panel: A high-level app and dashboarding solution for the PyData ecosystem.

13.3.1 Installation

Ihr könnt Panel in der virtuellen Umgebung eurer Jupyter-Kernels installieren mit:

```
$ uv add panel
```

Tipp

watchfiles unterstützt die Autoreload-Funktionen von Panel, wenn der `--dev`-Modus aktiviert ist:

```
$ uv add --dev watchfiles
```

Tipp

Für die Syntax-Hervorhebung sollte auch pygments installiert werden:

```
$ uv add pygments
```

Siehe auch

Wenn Panel z.B. in VSCode oder Google Colab verwendet werden soll, schaut euch Develop in other notebook environments an.

13.3.2 Überblick

In einem Panel könnt ihr interaktive Steuerelemente hinzufügen. Dies erlaubt euch, einfache interaktive Apps, aber auch komplexe mehrseitige Dashboards zu erstellen.Wir beginnen zunächst mit einem einfachen Beispiel einer Funktion zum Zeichnen einer Sinuswelle mit Matplotlib:

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

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

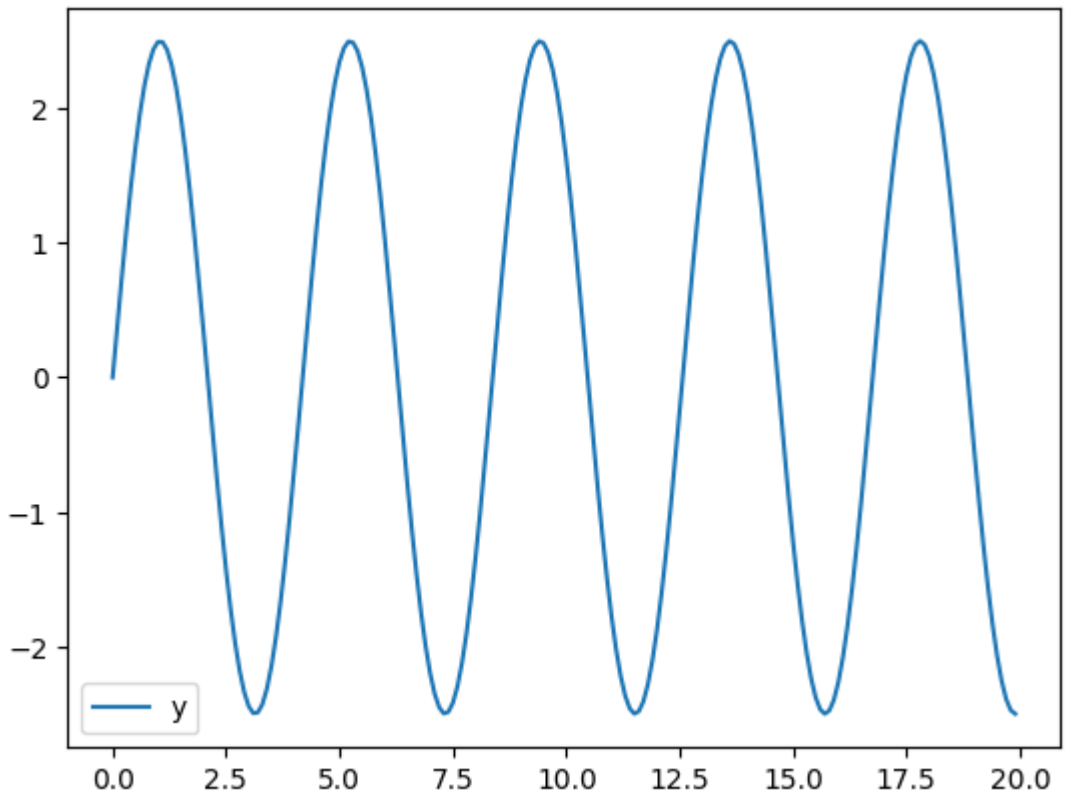
```
%matplotlib inline


def mplplot(df, **kwargs):
    fig = df.plot().get_figure()
    plt.close(fig)
    return fig


def sine(frequency=1.0, amplitude=1.0, n=200, view_fn=mplplot):
    xs = np.arange(n) / n * 20.0
    ys = amplitude * np.sin(frequency * xs)
    df = pd.DataFrame(dict(y=ys), index=xs)
    return view_fn(df, frequency=frequency, amplitude=amplitude, n=n)


sine(1.5, 2.5)
```

[1]:



Interaktive Panels

Wenn wir viele Kombinationen dieser Werte ausprobieren möchten, um zu verstehen, wie sich Frequenz und Amplitude auf dieses Diagramm auswirken, könnten wir die oben genannte Zelle viele Male neu bewerten. Dies wäre jedoch ein langsamer und aufwändiger Prozess. Stattdessen die Werte im Code jedesmal neu angeben zu müssen, empfiehlt sich, die Werte mithilfe von Schiebereglern interaktiv anzupassen. Mit einer solchen Panel-App könnt ihr einfach die Parameter einer Funktion untersuchen. Dabei ähnelt `pn.interact` der Funktion von ipywidgets interact:

[2]:
```
import panel as pn


pn.extension()

pn.interact(sine)
```

> ⚠ Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

> ⚠ Data type cannot be displayed: application/javascript, application/vnd.holoviews_load.v0+json

> ⚠ Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

[2]:
```
Column
    [0] Column
        [0] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [1] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
        [2] IntSlider(end=600, name='n', start=-200, value=200)
    [1] Row
        [0] Matplotlib(Figure, name='interactive00113')
```

Solange ein Live-Python-Prozess ausgeführt wird, wird durch Ziehen dieser Widgets die `sine`-Callback-Funktion aufgerufen und auf die von euch ausgewählte Kombination von Parameter-Werten ausgewertet und die Ergebnisse angezeigt. Mit einem solchen Panel könnt ihr einfach alle Funktionen untersuchen, die ein visuelles Ergebnis eines unterstützten Objekttyps (s. Supported object types and libraries liefern, z.B. Matplotlib, Bokeh, Plotly, Altair oder verschiedene Text- und Bildtypen.

Komponenten von Panels

`interact` ist praktisch, aber was ist, wenn Sie mehr Kontrolle darüber wünschen, wie es aussieht oder funktioniert? Lassen Sie uns zunächst sehen, was `interact` tatsächlich erstellt wird, indem Sie das Objekt greifen und seine Darstellung anzeigen:

[3]:
```
i = pn.interact(sine, n=(5, 100))
print(i)
```

```
Column
    [0] Column
        [0] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [1] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
        [2] IntSlider(end=100, name='n', start=5, value=200)
    [1] Row
        [0] Matplotlib(Figure, name='interactive00154')
```

Wir sehen hier, dass der `interact`-Aufruf ein `pn.Column`-Objekt erstellt hat, das aus einer WidgetBox (mit 3 Widgets) und einer `pn.Row`-Matplotlib-Figure besteht. Das Bedienfeld ist kompositorisch, sodass ihr diese Komponenten beliebig mischen und zuordnen könnt, indem ihr bei Bedarf weitere Objekte hinzufügt:

[4]:
```
pn.Row(i[1][0], pn.Column("<br>\n# Sine waves", i[0][0], i[0][1]))
```

[4]:
```
Row
    [0] Matplotlib(Figure, name='interactive00154')
    [1] Column
        [0] Markdown(str)
        [1] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [2] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
```

Beachtet, dass die Widgets mit ihrem Plot verknüpft bleiben, auch wenn sie sich in einer anderen Notebook-Zelle befinden:

[5]:
```
i[0][2]
```

[5]:
```
IntSlider(end=100, name='n', start=5, value=200)
```

Neue Panels

Mit diesem kompositorischen Ansatz könnt ihr verschiedene Komponenten wie Widgets, Diagramme, Text und andere Elemente, die für eine App oder ein Dashboard benötigt werden, auf beliebige Weise kombinieren. Das `interact`-Beispiel baut auf einem reaktiven Programmiermodell auf, bei dem sich eine Eingabe für die Funktion ändert und das Bedienfeld die Ausgabe der Funktion reaktiv aktualisiert. `interact` ist eine praktische Möglichkeit, Widgets aus den Argumenten für eure Funktion automatisch zu erstellen. Panel bietet jedoch auch eine explizitere reaktive API, mit der ihr Verbindungen zwischen Widgets und Funktionsargumenten definieren und anschließend das resultierende Dashboard manuell von Grund auf neu erstellen können.

Im folgenden Beispiel deklarieren wir explizit jede Komponente einer App:

1. Widgets
2. eine Funktion zum Berechnen von Sinuswerten
3. Spalten- und Zeilencontainer
4. die fertige `sine_panel`-App.

Widget-Objekte haben mehrere Parameter (aktueller Wert, zulässige Bereiche usw.), und hier verwenden wir den `depends`-Decorator von Panel, um zu deklarieren, dass die Eingabewerte der Funktion von den `value`-Parametern der Widgets stammen sollen. Wenn nun die Funktion und die Widgets angezeigt werden, aktualisiert das Panel die angezeigte Ausgabe automatisch, wenn sich eine der Eingaben ändert:

[6]:
```
import panel.widgets as pnw


frequency = pnw.FloatSlider(name="frequency", value=1, start=1.0, end=5)
amplitude = pnw.FloatSlider(name="amplitude", value=1, start=0.1, end=10)


@pn.depends(frequency.param.value, amplitude.param.value)
def reactive_sine(frequency, amplitude):
    return sine(frequency, amplitude)
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
widgets = pn.Column("<br>\n# Sine waves", frequency, amplitude)
sine_panel = pn.Row(reactive_sine, widgets)

sine_panel
```

[6]:
```
Row
    [0] ParamFunction(function, _pane=Matplotlib, defer_load=False)
    [1] Column
        [0] Markdown(str)
        [1] FloatSlider(end=5, name='frequency', start=1.0, value=1)
        [2] FloatSlider(end=10, name='amplitude', start=0.1, value=1)
```

Deploy-Panels

Die obigen Panels funktionieren alle in einer Notebook-Zelle, aber im Gegensatz zu `ipywidgets` und anderen Ansätzen funktionieren `Panel`-Apps auch auf eigenständigen Servern. Die obige App kann beispielsweise als eigener Webserver gestartet werden, mit:

[7]:
```
sine_panel.show()
```

```
Launching server at http://localhost:61964
```

[7]:
```
<panel.io.server.Server at 0x167db3990>
```

Dies startet den Bokeh-Server und öffnet ein Browser-Fenster mit der Anwendung.

Oder ihr könnt einfach angeben, was ihr auf der Webseite sehen möchtet. `servable()`, und dann den Shell-Befehl `uv run panel serve --show example.ipynb`, um einen Server mit diesem Objekt zu starten:

[8]:
```
sine_panel.servable();
```

Das Semikolon vermeidet, dass hier im Notizbuch eine weitere Kopie des Sinusfelds angezeigt wird.

Deklarative Panels

Der obige Kompositionsansatz ist sehr flexibel, verknüpft jedoch domänenspezifischen Code (die Teile über Sinuswellen) mit dem Widget-Anzeigecode. Das ist üblich in prototypischen Projekten, aber in Projekten, bei denen der Code in vielen verschiedenen Kontexten verwendet werden soll, sollen Teile des Codes, die sich auf die zugrunde liegende Domänen (d.h. die Anwendung oder den Forschungsbereich) beziehen, von denen getrennt werden, die an bestimmte Anzeigetechnologien gebunden sind (wie Jupyter-Notebooks oder Webserver).

Für solche Verwendungen unterstützt Panel Objekte, die mit der separaten Param-Bibliothek deklariert wurden. Dies bietet eine Möglichkeit, die Parameter eurer Objekte (Code, Parameter, Anwendung und Dashboard-Technologie) unabhängig zu erfassen und zu deklarieren. Der obige Code kann zum Beispiel in einem Objekt erfasst werden, das die Bereiche und Werte aller Parameter sowie die Generierung des Diagramms unabhängig von der Panel-Bibliothek oder einer anderen Art der Interaktion mit dem Objekt deklariert:

[9]:
```
import param


class Sine(param.Parameterized):
    amplitude = param.Number(default=1, bounds=(0, None), softbounds=(0, 5))
    frequency = param.Number(default=2, bounds=(0, 10))
    n = param.Integer(default=200, bounds=(1, 200))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    def view(self):
        return sine(self.frequency, self.amplitude, self.n)


sine_obj = Sine()
```

Die `Sine`-Klasse und die `sine_obj`-Instanz sind nicht abhängig von Panel, Jupyter oder einem anderen GUI- oder Web-Toolkit – sie deklarieren einfach Fakten über eine bestimmte Domäne (z.B., dass Sinuswellen Frequenz- und Amplitudenparameter annehmen und dass die Amplitude eine Zahl größer oder gleich Null ist). Diese Informationen reichen dann für Panel aus, um eine bearbeitbare und anzeigbare Darstellung für dieses Objekt zu erstellen, ohne dass etwas angegeben werden muss, das von den domänenspezifischen Details abhängt, die in Die `Sine`-Klasse und die `sine_obj`-Instanz sind nicht abhängig von Panel, Jupyter oder einem anderen GUI- oder Web-Toolkit. Sie deklarieren einfach Fakten über einen bestimmten Bereich (z. B., dass Sinuswellen Frequenz- und Amplitudenparameter annehmen und dass die Amplitude eine Zahl größer oder gleich Null ist). Diese Informationen reichen dann für Panel aus, um eine bearbeitbare und anzeigbare Darstellung für dieses Objekt zu erstellen, ohne dass etwas angegeben werden muss, das von den domänenspezifischen Details abhängt, die außerhalb von `sine_obj` enthalten sind:

[10]:
```
pn.Row(sine_obj.param, sine_obj.view)
```

[10]:
```
Row
    [0] Column(margin=(5, 10), name='Sine')
        [0] StaticText(value='<b>Sine</b>')
        [1] FloatSlider(end=5, name='Amplitude', value=1)
        [2] FloatSlider(end=10, name='Frequency', value=2)
        [3] IntSlider(end=200, name='N', start=1, value=200)
    [1] ParamMethod(method, _pane=Matplotlib, defer_load=False)
```

Um eine bestimmte Domäne zu unterstützen, könnt ihr Hierarchien solcher Klassen erstellen, in denen alle Parameter und Funktionen zusammengefasst sind, die ihr für verschiedene Objektfamilien benötigt. Dabei werden sowohl Parameter als auch Code in den Klassen übernommen, und zwar unabhängig von einer bestimmten GUI-Bibliothek oder sogar das Vorhandensein einer GUI überhaupt. Dieser Ansatz macht es praktisch, eine große Codebasis beizubehalten, die mit Panel vollständig angezeigt und bearbeitet werden kann, und zwar auf eine Weise, die im Laufe der Zeit beibehalten und angepasst werden kann.

Verknüpfen von Plots und Aktionen zwischen Panels

Die oben genannten Ansätze arbeiten jeweils mit einer Vielzahl von anzeigbaren Objekten, einschließlich Bildern, Gleichungen, Tabellen und Diagrammen. In jedem Fall bietet das Panel interaktive Funktionen mithilfe von Widgets und aktualisiert die angezeigten Objekte entsprechend, wobei nur sehr wenige Annahmen darüber getroffen werden, was tatsächlich angezeigt wird. Panel unterstützt auch eine umfassendere und dynamischere Interaktivität, bei der das angezeigte Objekt selbst interaktiv ist, z.B. JavaScript-basierte Diagramme von Bokeh und Plotly.

Wenn wir beispielsweise den mit Pandas gelieferten Matplotlib-Wrapper durch denBokeh-Wrapper hvPlot ersetzen, erhalten wir automatisch interaktive Plots, die *zooming*, *panning* und *hovering* ermöglichen:

[11]:
```
import hvplot.pandas


def hvplot(df, **kwargs):
    return df.hvplot()
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
pn.interact(sine, view_fn=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

[11]:
```
Column(sizing_mode='fixed')
    [0] Column
        [0] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [1] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
        [2] IntSlider(end=600, name='n', start=-200, value=200)
    [1] Row(sizing_mode='fixed')
        [0] HoloViews(Curve, height=300, name='interactive00996', sizing_mode='fixed',␣
↪width=700)
```

Diese interaktiven Aktionen können mit komplexeren Interaktionen in einem Plot (z. B. `tap`, `hover`) kombiniert werden um das Erkunden von Daten und das Aufdecken von Zusammenhängen zu vereinfachen. Beispielsweise können wir HoloViews verwenden, um eine umfassendere Version des hvPlot-Beispiels zu erstellen, das dynamisch aktualisiert wird um die Position auf dem Kreis anzuzeigen, wenn wir mit der Maus über die Sinuskurve fahren:

[12]:
```
import holoviews as hv


tap = hv.streams.PointerX(x=0)


def hvplot2(df, frequency, **kwargs):
    plot = df.hvplot(width=500, padding=(0, 0.1))
    tap.source = plot

    def unit_circle(x):
        cx = np.cos(x * frequency)
        sx = np.sin(x * frequency)
        circle = hv.Path(
            [hv.Ellipse(0, 0, 2), [(-1, 0), (1, 0)], [(0, -1), (0, 1)]]
        ).opts(color="black")
        triangle = hv.Path(
            [[(0, 0), (cx, sx)], [(0, 0), (cx, 0)], [(cx, 0), (cx, sx)]]
        ).opts(color="red", line_width=2)
        labels = hv.Labels(
            [(cx / 2, 0, "%.2f" % cx), (cx, sx / 2.0, "%.2f" % sx)]
        )
        labels = labels.opts(
            padding=0.1, xaxis=None, yaxis=None, text_baseline="bottom"
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        )
        return circle * triangle * labels

    vline = hv.DynamicMap(hv.VLine, streams=[tap]).opts(color="black")

    return (plot * vline).opts(toolbar="right")


unit_curve = pn.interact(
    sine, view_fn=hvplot2, n=(1, 200), frequency=(0, 10.0)
)

pn.Column(
    pn.Row(
        "# The Unit Circle",
        pn.Spacer(width=45),
        unit_curve[0][0],
        unit_curve[0][2],
    ),
    unit_curve[1],
)
```

[12]:
```
Column
    [0] Row
        [0] Markdown(str)
        [1] Spacer(width=45)
        [2] FloatSlider(end=10.0, name='frequency', value=1.0)
        [3] IntSlider(end=200, name='n', start=1, value=200)
    [1] Row(sizing_mode='fixed')
        [0] HoloViews(DynamicMap, height=300, name='interactive01145', sizing_mode='fixed
↪', width=500)
```

13.3.3 Interaktionen

Die `interact`-Funktion (`panel.interact`) erstellt automatisch Steuerelemente zum interaktiven Durchsuchen von Code und Daten.

[1]:
```
import panel as pn

from panel import widgets
from panel.interact import fixed, interact, interact_manual, interactive


pn.extension()
```



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

(Fortsetzung der vorherigen Seite)

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

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

interact

Auf der einfachsten Ebene werden `interact`-Steuerelemente für Funktionsargumente automatisch generiert und die Funktion dann mit diesen Argumenten aufgerufen, wenn ihr die Steuerelemente interaktiv bearbeitet. Zur Verwendung von `interact` müsst ihr eine Funktion definieren, die ihr untersuchen möchtet. Hier ist eine Funktion, die das einzige Argument ausgibt: `x`.

[2]:
```
def f(x):
    return x
```

Wenn ihr diese Funktion als erstes Argument zusammen mit einem ganzzahligen Schlüsselwortargument `x=10` an `interact` übergebt, wird ein Schieberegler generiert und an den Funktionsparameter gebunden.

[3]:
```
interact(f, x=10)
```

[3]:
```
Column
    [0] Column
        [0] IntSlider(end=30, name='x', start=-10, value=10)
    [1] Row
        [0] Str(int, name='interactive00113')
```

Wenn ihr den Schieberegler bewegt, wird die Funktion aufgerufen, die den aktuellen Wert von `x` ausgibt.

Wenn ihr `True` oder `False` übergebt, generiert `interact` ein Kontrollkästchen:

[4]:
```
interact(f, x=True)
```

[4]:
```
Column
    [0] Column
        [0] Checkbox(name='x', value=True)
    [1] Row
        [0] Str(bool, name='interactive00142')
```

Wenn ihr eine Zeichenfolge übergebt, generiert `interact` ein Textbereich.

[5]:
```
interact(f, x="Hi Pythonistas!")
```

[5]:
```
Column
    [0] Column
        [0] TextInput(name='x', value='Hi Pythonistas!')
    [1] Row
        [0] Markdown(str, name='interactive00171')
```

`interact` kann auch als *Decorator* verwendet werden. Auf diese Weise könnt ihr sowohl eine Funktion definieren als auch die Art der Interaktion festlegen. Wie das folgende Beispiel zeigt, funktioniert `interact` auch mit Funktionen, die mehrere Argumente haben.

[6]:
```
@interact(x=True, y=1.0)
def g(x, y):
    return (x, y)


g
```

[6]:
```
Column
    [0] Column
        [0] Checkbox(name='x', value=True)
        [1] FloatSlider(end=3.0, name='y', start=-1.0, value=1.0)
    [1] Row
        [0] Str(tuple, name='interactive00200')
```

Layout interaktiver Widgets

Die `interact`-Funktion gibt ein Panel zurück, das die Widgets und die Anzeigeausgabe enthält. Durch Indizieren dieser Panel können wir das Layout für die Objekte genau so festlegen, wie wir wollen:

[7]:
```
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd


%matplotlib inline


def mplplot(df, **kwargs):
    fig = df.plot().get_figure()
    plt.close(fig)
    return fig


def sine(frequency=1.0, amplitude=1.0, n=200, view_fn=mplplot):
    xs = np.arange(n) / n * 20.0
    ys = amplitude * np.sin(frequency * xs)
    df = pd.DataFrame(dict(y=ys), index=xs)
    return view_fn(df, frequency=frequency, amplitude=amplitude, n=n)
```

[8]:
```
i = pn.interact(sine, n=(5, 100))
pn.Row(i[1][0], pn.Column("<br>\n### Sine waves", i[0][0], i[0][1]))
```

[8]:
```
Row
    [0] Matplotlib(Figure, name='interactive00234')
    [1] Column
        [0] Markdown(str)
        [1] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [2] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
```

[9]:
```
layout = interact(f, x=10)

pn.Column('**A custom interact layout**', pn.Row(layout[0], layout[1]))
```

[9]:
```
Column
    [0] Markdown(str)
    [1] Row
        [0] Column
            [0] IntSlider(end=30, name='x', start=-10, value=10)
        [1] Row
            [0] Str(int, name='interactive00286')
```

Argumente festlegen mit `fixed`

Es kann vorkommen, dass ihr eine Funktion mithilfe von `interact` untersuchen möchtet, aber eines oder mehrere ihrer Argumente auf bestimmte Werte festlegen möchtet. Dies kann durch die `fixed`-Funktion erreicht werden:

[10]:
```
def h(p, q):
    return (p, q)
```

[11]:
```
interact(h, p=5, q=fixed(20))
```

[11]:
```
Column
    [0] Column
        [0] IntSlider(end=15, name='p', start=-5, value=5)
    [1] Row
        [0] Str(tuple, name='interactive00330')
```

Widget Abbreviations

Wenn ihr bestimmte Werte übergebt, verwendet `interact` automatisch das passende Widget, z.B. eine Checkbox für `True` oder den `IntSlider` für ganzzahlige Werte. Ihr müsst also nicht explizit das passende Widget angeben:

[12]:
```
interact(f, x=widgets.FloatSlider(start=0.0,end=10.0,step=0.01,value=3.0))
```

[12]:
```
Column
    [0] Column
        [0] FloatSlider(end=10.0, step=0.01, value=3.0)
    [1] Row
        [0] Str(float, name='interactive00362')
```

[13]:
```
interact(f, x=(0.0,10.0,0.01,3.0))
```

[13]:
```
Column
    [0] Column
        [0] FloatSlider(end=10.0, name='x', step=0.01, value=3.0)
    [1] Row
        [0] Str(float, name='interactive00388')
```

Dieses Beispiel verdeutlicht, wie `interact` die Schlüsselwortargumente verarbeitet werden:

1. Wenn das Schlüsselwortargument eine `Widget`-Instanz mit einem `value`-Attribut ist, wird dieses Widget verwendet. Dabei kann jedes Widget mit einem `value`-Attribut verwendet werden, auch benutzerdefinierte.

2. Andernfalls wird der Wert als *Widget Abbreviation* behandelt, die vor der Verwendung in ein Widget konvertiert wird.

Die folgende Tabelle gibt einen Überblick über verschiedene *Widget Abbreviations*:

Keyword argument	Widget
`True` oder `False`	`Checkbox`
`"Hi Pythonistas!"`	`Text`
Ganzzahliger Wert als `min`, `max`,`step`, `value`	`IntSlider`
Gleitkommazahl als `min`, `max`,`step`, `value`	`FloatSlider`
`["apple", "pear"]` oder `{"one": 1, "two": 2}`	`Dropdown`

13.3.4 Widgets

`Panel` bietet eine breite Palette von Widgets zur präzisen Steuerung von Parameter-Werten. Die Widget-Klassen verwenden eine konsistente API, mit der breite Kategorien von Widgets als austauschbar behandelt werden können. Um beispielsweise einen Wert aus einer Liste von Optionen auszuwählen, können Sie ein `SelectWidget`, ein `RadioButtonGroup`-Widget oder ein gleichwertiges Widget austauschbar verwenden.

Wie alle anderen Komponenten in `Panel`, können auch `Widget`-Objekte ihren Zustand sowohl im Notebook als auch auf dem Bokeh-Server synchronisieren:

[1]:
```
import panel as pn


pn.extension()
```

⚠

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

⚠

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

⚠

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

[2]:
```
widget = pn.widgets.TextInput(name="A widget", value="A string")
widget
```

[2]:
```
TextInput(name='A widget', value='A string')
```

Wenn ihr den Textwert ändert, wird der entsprechende Parameter automatisch aktualisiert:

[3]:
```
widget.value
```

[3]:
```
'A string'
```

Durch Aktualisieren des Parameter-Werts wird auch das Widget aktualisiert:

[4]:
```
widget.value = "Another string"
```

Callbacks und Links

Um eine Parameter-Änderung mitzubekommen, können wir `widget.param.watch` mit dem zu beobachtenden Parameter un einer Funktion aufrufen:

[5]:
```
from __future__ import print_function


widget.param.watch(print, "value")
```

[5]:
```
Watcher(inst=TextInput(name='A widget', value='Another string'), cls=<class 'panel.
↪widgets.input.TextInput'>, fn=<built-in function print>, mode='args', onlychanged=True,
↪ parameter_names=('value',), what='value', queued=False, precedence=0)
```

Wenn wir `widget.value` nun ändern, wird das resultierende *Event* ausgegeben.

[6]:
```
widget.value = "A"
```

```
Event(what='value', name='value', obj=TextInput(name='A widget', value='A'),␣
↪cls=TextInput(name='A widget', value='A'), old='Another string', new='A', type='changed
↪')
```

PanelWidgets ermöglichen in Kombination mit Objekten die einfache Erstellung interaktiver Dashboards und Visualisierungen. Weitere Informationen zum Definieren von Rückrufen und Verknüpfungen zwischen Widgets und anderen Komponenten findet ihr im Benutzerhandbuch zu Verknüpfungen .

Widgets

Um mehrere Widgets zusammenzustellen, können sie zu einem`Row`-, `Column`- oder `Tabs`-Panel hinzugefügt werden. Weitere Informationen zum Layout von Widgets und Bedienfeldern findet ihr in Declare Custom Widgets.

[7]:
```
slider = pn.widgets.FloatSlider(name="Another widget", width=200)
pn.Column(widget, slider, width=200)
```

[7]:
```
Column(width=200)
    [0] TextInput(name='A widget', value='A')
    [1] FloatSlider(name='Another widget', width=200)
```

Widget-Kategorien

Die unterstützten Widgets können anhand ihrer kompatiblen APIs in verschiedene Kategorien eingeteilt werden.

Optionsauswahl

Mit Optionsauswahl-Widgets könnt ihr einen oder mehrere Werte aus einer `list` oder einem `dictionary` auswählen. Alle Widgets dieses Typs haben `options` und `value`-Parameter.

Optionen	Widget	Beschreibung
Einzelwerte		Mit diesen Widgets könnt ihr einen Wert aus einer `list` oder einem `dictionary` auswählen
	`AutocompleteI`	wählt einen `value` aus einem sich automatisch vervollständigenden Textfeld
	`DiscretePlaye`	zeigt Steuerelemente des `mediaplayer` an, mit denen ihr die verfügbaren Optionen abspielen und durchgehen könnt
	`DiscreteSlide`	wählt einen Wert mit einem Schieberegler
	`RadioButtonGr`	wählt einen Wert aus einer Reihe sich gegenseitig ausschließender Umschalttasten
	`RadioBoxGroup`	wählt einen Wert aus einer Reihe sich gegenseitig ausschließender Kontrollkästchen aus
	`Select`	wählt einen Wert aus einem Dropdown-Menü
Mehrere Werte		Mit diesen Widgets könnt ihr mehrere Werte aus einer `list` oder einem `dictionary` auswählen
	`CheckBoxGroup`	wählt Werte aus, indem ihr die entsprechenden Kontrollkästchen aktiviert
	`CheckButtonGr`	wählt Werte aus, indem ihr die entsprechenden Schaltflächen umschaltet
	`CrossSelector`	wählt Werte aus, indem ihr Elemente zwischen zwei Listen verschiebt
	`MultiSelect`	wählt Werte aus, indem ihr sie in einer Liste markiert

Typ-basierte Selektoren

Typ-basierte Selektoren bieten die Möglichkeit, einen Wert entsprechend seinem Typ auszuwählen. Alle Selektoren verfügen über `value`. Die Widgets in dieser Kategorie können über den Typ hinaus auch andere Validierungsformen aufweisen, z.B. die obere und untere Grenze von Schiebereglern.

	Typen	Widget	Beschreibung
Einzelwerte			erlaubt die Auswahl eines einzelnen `value`-Typs
	Numerisch		Numerische Selektoren sind durch `start` und `end`-Werte begrenzt
		`IntSlider`	wählt mit einem Schieberegler einen ganzzahligen Wert innerhalb eines festgelegten Bereichs aus
		`FloatSlid`	wählt mit einem Schieberegler einen Gleitkommawert innerhalb eines festgelegten Bereichs aus
		`Player`	zeigt Steuerelemente des `mediaplayer` an, mit denen ihr eine Reihe von Ganzzahlwerten abspielen und durchlaufen könnt
	Boolesche Werte		
		`Checkbox`	Schaltet eine einzelne Bedingung zwischen `True`/`False` um, indem ein Kontrollkästchen aktiviert wird
		`Toggle`	Umschalten einer einzelnen Bedingung zwischen `True`/`False`-Zuständen durch Klicken auf eine Schaltfläche
	Termine		
		`DatetimeI`	wählt mit einem Textfeld und dem Dienstprogramm zur Datumsauswahl des Browsers einen Datumswert aus
		`DatePicke`	gebt einen Datums-/Uhrzeitwert als Text ein und analysiert ihn mit einem vordefinierten Formatierer
		`DateSlide`	wählt mit einem Schieberegler einen Datumswert innerhalb eines festgelegten Bereichs aus.
	Text		
		`TextInput`	gebt eine beliebige Zeichenfolge über ein Texteingabefeld ein
	Andere		
		`ColorPick`	wählt eine Farbe mit den Farbauswahl-Dienstprogrammen des Browsers aus.
		`FileInput`	Ladet eine Datei vom Frontend hoch und macht die Daten und den MIME-Typ in Python verfügbar
		`LiteralIn`	gebt ein beliebiges Python-Literal über ein Texteingabefeld ein, das dann in Python analysiert wird
Bereiche			ermöglicht die Auswahl eines Wertebereichs des entsprechenden Typs, der als `(lower, upper)`-Tupel für den `value`-Parameter gespeichert ist
	Numerisch		
		`IntRangeS`	wählt mit einem Schieberegler mit zwei Ziehpunkten einen ganzzahligen Bereich aus
		`RangeSlid`	wählt mit einem Schieberegler mit zwei Ziehpunkten einen Gleitkommabereich aus
	Termine		
		`DateRange`	wählt mit einem Schieberegler mit zwei Ziehpunkten einen Datumsbereich aus
Andere			
		`Audio`	zeigt einen Audio-Player an, dem eine Audiodatei lokal oder Remote zugewiesen wurde, und ermöglicht den Zugriff und die Steuerung des Player-Status
		`Button`	ermöglicht das Auslösen von Ereignissen, wenn auf die Schaltfläche geklickt wird; im Gegensatz zu anderen Widgets hat es keinen `value`-Parameter

13.3.5 Parametrisierung

Panel unterstützt die Verwendung von Parametern und Abhängigkeiten zwischen Parametern, die von param in einfacher Weise ausgedrückt werden, um Dashboards als deklarative, eigenständige Klassen zu kapseln.

Parameter sind Python-Attribute, die mithilfe der `param`-Bibliothek erweitert wurden, um Typen, Bereiche und Dokumentation zu unterstützen. Dabei handelt es sich lediglich um die Informationen, die ihr zum automatischen Erstellen von Widgets für jeden Parameter benötigt.

Parameter und Widgets

Hierfür werden zuerst einige parametrisierte Klassen mit verschiedenen Parametern deklariert:

[1]:
```
import datetime as dt

import param


class BaseClass(param.Parameterized):
    x = param.Parameter(default=3.14, doc="X position")
    y = param.Parameter(default="Not editable", constant=True)
    string_value = param.String(default="str", doc="A string")
    num_int = param.Integer(50000, bounds=(-200, 100000))
    unbounded_int = param.Integer(23)
    float_with_hard_bounds = param.Number(8.2, bounds=(7.5, 10))
    float_with_soft_bounds = param.Number(
        0.5, bounds=(0, None), softbounds=(0, 2)
    )
    unbounded_float = param.Number(30.01, precedence=0)
    hidden_parameter = param.Number(2.718, precedence=-1)
    integer_range = param.Range(default=(3, 7), bounds=(0, 10))
    float_range = param.Range(default=(0, 1.57), bounds=(0, 3.145))
    dictionary = param.Dict(default={"a": 2, "b": 9})


class Example(BaseClass):
    """An example Parameterized class"""

    timestamps = []

    boolean = param.Boolean(True, doc="A sample Boolean parameter")
    color = param.Color(default="#FFFFFF")
    date = param.Date(
        dt.datetime(2017, 1, 1),
        bounds=(dt.datetime(2017, 1, 1), dt.datetime(2017, 2, 1)),
    )
    select_string = param.ObjectSelector(
        default="yellow", objects=["red", "yellow", "green"]
    )
    select_fn = param.ObjectSelector(default=list, objects=[list, set, dict])
    int_list = param.ListSelector(
        default=[3, 5], objects=[1, 3, 5, 7, 9], precedence=0.5
    )
    single_file = param.FileSelector(path="../../*/*.py*", precedence=0.5)
    multiple_files = param.MultiFileSelector(
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        path="../../*/*.py?", precedence=0.5
    )
    record_timestamp = param.Action(
        lambda x: x.timestamps.append(dt.datetime.now()),
        doc="""Record timestamp.""",
        precedence=0.7,
    )


Example.num_int
```

[1]:
```
50000
```

Wie ihr seht, hängt die Deklaration von Parametern nur von der separaten `param`-Bibliothek ab. Parameter sind eine einfache Idee mit einigen Eigenschaften, die für die Erstellung von sauberem, verwendbarem Code entscheidend sind:

- Die `param`-Bibliothek ist in reinem Python ohne Abhängigkeiten geschrieben, wodurch es einfach ist, sie in jeden Code einzubinden, ohne sie an eine bestimmte GUI- oder Widgets-Bibliothek oder an Jupyter-Notebooks zu binden.
- Parameterdeklarationen konzentrieren sich auf semantische Informationen, die für eure Domäne relevant sind. So vermeidet ihr, dass domänenspezifischer Code durch irgendetwas verunreinigt wird, das ihn an eine bestimmte Art der Anzeige oder Interaktion mit ihm bindet.
- Parameter können überall dort definiert werden, wo sie in eurer Vererbungshierarchie sinnvoll sind, und ihr könnt sie einmal dokumentieren, eingeben und auf einen bestimmten Bereich beschränken. Dabei werden alle diese Eigenschaften von einer beliebigen Basisklasse geerbt. Beispielsweise funktionieren hier alle Parameter gleich, unabhängig davon, ob sie in `BaseClass` oder `Example` deklariert wurden. Dies erleichtert die einmalige Bereitstellung dieser Metadaten und verhindert, dass sie überall im Code dupliziert werden, wo Bereiche oder Typen überprüft oder Dokumentationen gespeichert werden müssen.

Wenn ihr euch dann für die Verwendung dieser parametrisierten Klassen in einer Notebook- oder Webserver-Umgebung entscheidet, könnt ihr mit `import panel` die Parameter-Werte als optionalen zusätzlichen Schritt einfach anzeigen und bearbeiten:

[2]:
```
import panel as pn


pn.extension()

base = BaseClass()
pn.Row(Example.param, base.param)
```



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



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

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

[2]:
```
Row
    [0] Column(margin=(5, 10), name='Example')
        [0] StaticText(value='<b>Example</b>')
        [1] FloatInput(name='Unbounded float', value=30.01)
        [2] LiteralInput(description='X position', name='X', value=3.14)
        [3] LiteralInput(disabled=True, name='Y', value='Not editable')
        [4] TextInput(description='A string', name='String value', value='str')
        [5] IntSlider(end=100000, name='Num int', start=-200, value=50000)
        [6] IntInput(name='Unbounded int', value=23)
        [7] FloatSlider(end=10, name='Float with hard bounds', start=7.5, value=8.2)
        [8] FloatSlider(end=2, name='Float with soft bounds', value=0.5)
        [9] RangeSlider(end=10, name='Integer range', step=1, value=(3, 7), value_end=7,␣
↪value_start=3)
        [10] RangeSlider(end=3.145, name='Float range', value=(0, 1.57), value_end=1.57)
        [11] DictInput(name='Dictionary', type=<class 'dict'>, value={'a': 2, 'b': 9})
        [12] Checkbox(name='Boolean', value=True)
        [13] ColorPicker(name='Color', value='#FFFFFF')
        [14] DatetimeInput(end=datetime.datetime(2017, ..., name='Date', start=datetime.
↪datetime(2017, ..., type=<class 'datetime.datetime'..., value=datetime.datetime(2017,␣
↪...)
        [15] Select(options=OrderedDict([('red', ...]), value='yellow')
        [16] Select(options=OrderedDict([('list', ...]), value=<class 'list'>)
        [17] MultiSelect(name='Int list', options=OrderedDict([('1', ...]), value=[3, 5])
        [18] Select(name='Single file', options=OrderedDict([('hub/jupyter...]), value='.
↪./../hub/jupyterhub_conf...)
        [19] FileSelector(name='Multiple files')
        [20] Button(name='Record timestamp')
    [1] Column(margin=(5, 10), name='BaseClass')
        [0] StaticText(value='<b>BaseClass</b>')
        [1] FloatInput(name='Unbounded float', value=30.01)
        [2] LiteralInput(description='X position', name='X', value=3.14)
        [3] LiteralInput(disabled=True, name='Y', value='Not editable')
        [4] TextInput(description='A string', name='String value', value='str')
        [5] IntSlider(end=100000, name='Num int', start=-200, value=50000)
        [6] IntInput(name='Unbounded int', value=23)
        [7] FloatSlider(end=10, name='Float with hard bounds', start=7.5, value=8.2)
        [8] FloatSlider(end=2, name='Float with soft bounds', value=0.5)
        [9] RangeSlider(end=10, name='Integer range', step=1, value=(3, 7), value_end=7,␣
↪value_start=3)
        [10] RangeSlider(end=3.145, name='Float range', value=(0, 1.57), value_end=1.57)
        [11] DictInput(name='Dictionary', type=<class 'dict'>, value={'a': 2, 'b': 9})
```

Wie ihr seht, muss Panel nicht über Kenntnisse eurer domänenspezifischen Anwendung verfügen, auch nicht über die Namen eurer Parameter. Es werden einfach Widgets für alle Parameter angezeigt, die für dieses Objekt definiert wurden. Durch die Verwendung von Param mit Panel wird somit eine nahezu vollständige Trennung zwischen eurem domänenspezifischen Code und eurem Display-Code erreicht, wodurch die Wartung beider über einen längeren Zeit-

raum erheblich vereinfacht wird. Hier wurde sogar das `msg`-Behavior der Schaltflächen deklarativ festgelegt als eine Aktion, die unabhängig davon, ob sie in einer GUI oder in einem anderen Kontext verwendet wird, aufgerufen werden kann.

Die Interaktion mit den oben genannten Widgets wird nur im Notebook und auf dem Bokeh-Server unterstützt. Ihr könnt jedoch auch statische Renderings der Widgets in eine Datei oder eine Webseite exportieren.

Wenn ihr Werte auf diese Weise bearbeitet, müsst ihr das Notebook standardmäßig Zelle für Zelle ausführen. Wenn ihr zu der obigen Zelle gelangt, bearbeitet ihr die Werte nach euren Wünschen und führt die nachfolgenden Zellen aus, in denen auf diese Parameterwerte verwiesen wird, werden eure interaktiv ausgewählte Einstellungen verwendet:

[3]:
```
Example.unbounded_int
```

[3]:
```
23
```

[4]:
```
Example.num_int
```

[4]:
```
50000
```

Um dies zu umgehen und automatisch alle Widgets zu aktualisieren, die aus dem Parameter generiert wurden, könnt ihr das `param`-Objekt übergeben:

[5]:
```
pn.Row(Example.param.float_range, Example.param.num_int)
```

[5]:
```
Row
    [0] RangeSlider(end=3.145, name='Float range', value=(0, 1.57), value_end=1.57)
    [1] IntSlider(end=100000, name='Num int', start=-200, value=50000)
```

Benutzerdefinierte Widgets

Im vorherigen Abschnitt haben wir gesehen, wie Parameter automatisch in Widgets umgewandelt werden können. Dies ist möglich, da Panel intern eine Zuordnung zwischen Parameter- und Widget-Typen verwaltet. Manchmal bietet das Standard-Widget jedoch nicht die bequemste Benutzeroberfläche, und wir möchten Panel einen expliziten Hinweis geben, wie ein Parameter gerendert werden soll. Dies ist mit dem `widgets`-Argument für das `Param`-Panel möglich. Mit dem `widgets`-Keyword können wir eine Zuordnung zwischen dem Parameter-Namen und dem gewünschten Widget-Typ deklarieren.

Als Beispiel können wir einer `RadioButtonGroup` und einem `DiscretePlayer` einen `String`- und einen `Number`-Selector zuordnen.

[6]:
```
class CustomExample(param.Parameterized):
    """An example Parameterized class"""

    select_string = param.Selector(objects=["red", "yellow", "green"])
    select_number = param.Selector(objects=[0, 1, 10, 100])


pn.Param(
    CustomExample.param,
    widgets={
        "select_string": pn.widgets.RadioButtonGroup,
        "select_number": pn.widgets.DiscretePlayer,
    },
)
```

[6]:
```
Param(ParameterizedMetaclass, name='CustomExample', widgets={'select_string': <class
↪'...})
```

Es ist auch möglich, Argumente an das Widget zu übergeben, um es anzupassen. Anstatt das Widget zu übergeben, übergebt ein Wörterbuch mit den gewünschten Optionen. Verwendet das `type`-Schlüsselwort, um das Widget zuzuordnen:

[7]:
```
pn.Param(
    CustomExample.param,
    widgets={
        "select_string": {
            "type": pn.widgets.RadioButtonGroup,
            "button_type": "primary",
        },
        "select_number": pn.widgets.DiscretePlayer,
    },
)
```

[7]:
```
Param(ParameterizedMetaclass, name='CustomExample', widgets={'select_string': {'type':
↪...})
```

Parameter-Abhängigkeiten

Das Deklarieren von Parametern ist normalerweise nur der Anfang eines Workflows. In den meisten Anwendungen sind diese Parameter dann an eine Berechnung gebunden. Um die Beziehung zwischen einer Berechnung und den Parametern, von denen sie abhängt, auszudrücken, kann der `param.depends`-Dekorator für parametrisierte Methoden verwendet werden. Dieser Dekorator gibt Panel und anderen `param`-basierten Bibliotheken (z.B. HoloViews) einen Hinweis, dass die Methode bei einer Änderung eines Parameters neu bewertet werden sollte.

Als einfaches Beispiel ohne zusätzliche Abhängigkeiten schreiben wir eine kleine Klasse, die eine ASCII-Darstellung einer Sinuswelle zurückgibt, die von `phase` und `frequency`-Parametern abhängt. Wenn wir die `.view`-Methode an ein Panel übergeben wird die Ansicht automatisch neu berechnet und aktualisiert, sobald sich einer oder mehrere der Parameter ändern:

[8]:
```
import numpy as np


class Sine(param.Parameterized):
    phase = param.Number(default=0, bounds=(0, np.pi))
    frequency = param.Number(default=1, bounds=(0.1, 2))

    @param.depends("phase", "frequency")
    def view(self):
        y = np.sin(np.linspace(0, np.pi * 3, 40) * self.frequency + self.phase)
        y = ((y - y.min()) / y.ptp()) * 20
        array = np.array(
            [list((" " * (int(round(d)) - 1) + "*").ljust(20)) for d in y]
        )
        return pn.pane.Str(
            "\n".join(["".join(r) for r in array.T]), height=325, width=500
        )
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
sine = Sine(name="ASCII Sine Wave")
pn.Row(sine.param, sine.view)
```

[8]:
```
Row
    [0] Column(margin=(5, 10), name='ASCII Sine Wave')
        [0] StaticText(value='<b>ASCII Sine Wave</b>')
        [1] FloatSlider(end=3.141592653589793, name='Phase')
        [2] FloatSlider(end=2, name='Frequency', start=0.1, value=1)
    [1] ParamMethod(method, _pane=Str, defer_load=False)
```

Die parametrisierte und mit Anmerkungen versehene `view`-Methode kann einen beliebigen Typ zurückgeben, der vom Pane-Objects bereitgestellt wird. Auf diese Weise können Parameter und die zugehörigen Widgets auf einfache Weise mit einem Plot oder einer anderen Ausgabe verknüpft werden. Parametrisierte Klassen können daher ein sehr nützliches Muster sein, um einen Teil eines Rechenworkflows mit einer zugehörigen Visualisierung zu kapseln und die Abhängigkeiten zwischen den Parametern und der Berechnung deklarativ auszudrücken.

Standardmäßig zeigt ein Param-Bereich (*Pane*) Widgets für alle Parameter mit einem `precedence`-Wert über dem Wert `pn.Param.display_threshold` an, sodass ihr `precedence` verwenden könnt um automatisch Parameter auszublenden. Ihr könnt auch explizit auswählen, welche Parameter Widgets in einem bestimmten Beriech enthalten sollen, indem ihr ein `parameters`-Argument übergebt. Dieser Code gibt beispielsweise ein `phase`-Widget aus wobei `sine.frequency` den Anfangswert 1 beibehält:

[9]:
```
pn.Row(pn.panel(sine.param, parameters=["phase"]), sine.view)
```

[9]:
```
Row
    [0] Column(margin=(5, 10), name='ASCII Sine Wave')
        [0] StaticText(value='<b>ASCII Sine Wave</b>')
        [1] FloatSlider(end=3.141592653589793, name='Phase')
    [1] ParamMethod(method, _pane=Str, defer_load=False)
```

Ein weiteres gängiges Muster ist das Verknüpfen der Werte eines Parameters mit einem anderen Parameter, z.B. wenn Abhängigkeiten zwischen Parametern bestehen. Im folgenden Beispiel definieren wir zwei Parameter, einen für den Kontinent und einen für das Land. Da wir möchten, dass sich die Auswahl der gültigen Länder ändert, wenn wir den Kontinent wechseln, definieren wir eine Methode, um dies für uns zu tun. Um die beiden zu verbinden, drücken wir die Abhängigkeit mithilfe des `param.depends`-Dekorators aus und stellen dann mit `watch=True` sicher, dass die Methode ausgeführt wird, wenn der Kontinent geändert wird.

Wir definieren auch eine `view`-Methode, die einen HTML-Iframe zurückgibt, der das Land mithilfe von Google Maps anzeigt.

[10]:
```
class GoogleMapViewer(param.Parameterized):
    continent = param.ObjectSelector(
        default="Asia", objects=["Africa", "Asia", "Europe"]
    )

    country = param.ObjectSelector(
        default="China", objects=["China", "Thailand", "Japan"]
    )

    _countries = {
        "Africa": ["Ghana", "Togo", "South Africa", "Tanzania"],
        "Asia": ["China", "Thailand", "Japan"],
        "Europe": ["Austria", "Bulgaria", "Greece", "Portugal", "Switzerland"],
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    }

    @param.depends("continent", watch=True)
    def _update_countries(self):
        countries = self._countries[self.continent]
        self.param["country"].objects = countries
        self.country = countries[0]

    @param.depends("country")
    def view(self):
        iframe = """
        <iframe width="800" height="400" src="https://maps.google.com/maps?q={country}&
↪z=6&output=embed"
        frameborder="0" scrolling="no" marginheight="0" marginwidth="0"></iframe>
        """.format(
            country=self.country
        )
        return pn.pane.HTML(iframe, height=400)


viewer = GoogleMapViewer(name="Google Map Viewer")
pn.Row(viewer.param, viewer.view)
```

[10]:
```
Row
    [0] Column(margin=(5, 10), name='Google Map Viewer')
        [0] StaticText(value='<b>Google Map V...)
        [1] Select(name='Continent', options=OrderedDict([('Africa', ...]), value='Asia')
        [2] Select(name='Country', options=OrderedDict([('China', ...]), value='China')
    [1] ParamMethod(method, _pane=HTML, defer_load=False)
```

Immer wenn sich der Kontinent ändert, wird nun die `_update_countries`-Methode zum Ändern der angezeigten Länderliste ausgeführt, was wiederum eine Aktualisierung der `view`-Methode auslöst.

[11]:
```
from bokeh.plotting import figure


class Shape(param.Parameterized):
    radius = param.Number(default=1, bounds=(0, 1))

    def __init__(self, **params):
        super(Shape, self).__init__(**params)
        self.figure = figure(x_range=(-1, 1), y_range=(-1, 1))
        self.renderer = self.figure.line(*self._get_coords())

    def _get_coords(self):
        return [], []

    def view(self):
        return self.figure


class Circle(Shape):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    n = param.Integer(default=100, precedence=-1)

    def __init__(self, **params):
        super(Circle, self).__init__(**params)

    def _get_coords(self):
        angles = np.linspace(0, 2 * np.pi, self.n + 1)
        return (self.radius * np.sin(angles), self.radius * np.cos(angles))

    @param.depends("radius", watch=True)
    def update(self):
        xs, ys = self._get_coords()
        self.renderer.data_source.data.update({"x": xs, "y": ys})


class NGon(Circle):
    n = param.Integer(default=3, bounds=(3, 10), precedence=1)

    @param.depends("radius", "n", watch=True)
    def update(self):
        xs, ys = self._get_coords()
        self.renderer.data_source.data.update({"x": xs, "y": ys})
```

Parameter-Unterobjekte

`Parameterized`-Objekte haben oft Parameter-Werte, die selbst `Parameterized`-Objekte sind und eine baumartige Struktur bilden. Mit dem Bedienfeld könnt ihr nicht nur die Parameter des Hauptobjekts bearbeiten, sondern auch auf Unterobjekt durchgreifen. Definieren wir zunächst eine Hierarchie von `Shape`-Klassen deklarieren, die einen Bokeh-Plot des ausgewählten `Shape` zeichnen:

[12]:
```
from bokeh.plotting import figure


class Shape(param.Parameterized):
    radius = param.Number(default=1, bounds=(0, 1))

    def __init__(self, **params):
        super(Shape, self).__init__(**params)
        self.figure = figure(x_range=(-1, 1), y_range=(-1, 1))
        self.renderer = self.figure.line(*self._get_coords())

    def _get_coords(self):
        return [], []

    def view(self):
        return self.figure


class Circle(Shape):
    n = param.Integer(default=100, precedence=-1)

    def __init__(self, **params):
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        super(Circle, self).__init__(**params)

    def _get_coords(self):
        angles = np.linspace(0, 2 * np.pi, self.n + 1)
        return (self.radius * np.sin(angles), self.radius * np.cos(angles))

    @param.depends("radius", watch=True)
    def update(self):
        xs, ys = self._get_coords()
        self.renderer.data_source.data.update({"x": xs, "y": ys})


class NGon(Circle):
    n = param.Integer(default=3, bounds=(3, 10), precedence=1)

    @param.depends("radius", "n", watch=True)
    def update(self):
        xs, ys = self._get_coords()
        self.renderer.data_source.data.update({"x": xs, "y": ys})
```

Jetzt, da wir mehrere Shape-Klassen haben, können wir Instanzen davon erstellen und einen `ShapeViewer` erstellen, um zwischen ihnen auszuwählen. Wir können auch zwei Methoden mit Parameter-Abhängigkeiten deklarieren, die den Plot und den Plot-Titel aktualisieren. Hierbei ist zu beachten, dass der `param.depends`-Dekorator nicht nur von Parametern am Objekt selbst abhängen kann, sondern auch von bestimmten Parametern am Unterobjekt, z.B. `shape.radius` oder von Parametern des Unterobjekts mit `shape.param` ausgedrückt werden kann.

[13]:
```
shapes = [NGon(), Circle()]


class ShapeViewer(param.Parameterized):
    shape = param.ObjectSelector(default=shapes[0], objects=shapes)

    @param.depends("shape")
    def view(self):
        return self.shape.view()

    @param.depends("shape", "shape.radius")
    def title(self):
        return "## %s (radius=%.1f)" % (
            type(self.shape).__name__,
            self.shape.radius,
        )

    def panel(self):
        return pn.Column(self.title, self.view)
```

Nachdem wir eine Klasse mit Unterobjekten haben, können wir sie wie gewohnt anzeigen. Drei Hauptoptionen steuern, wie das Unterobjekt gerendert wird:

- `expand`: ob das Unterobjekt bei der Initialisierung erweitert wird (`default=False`)
- `expand_button`: ob eine Schaltfläche zum Umschalten der Erweiterung vorhanden sein soll; ansonsten ist es auf den initialen `expand`-Wert festgelegt (`default=True`)

- `expand_layout`: Ein Layout-Typ oder eine Instanz zum Erweitern des Plots in (`default=Column`)

Beginnen wir mit der Standardansicht, die eine Umschaltfläche zum Erweitern des Unterobjekts bietet:

[14]:
```
viewer = ShapeViewer()

pn.Row(viewer.param, viewer.panel())
```

[14]:
```
Row
    [0] Column(margin=(5, 10), name='ShapeViewer')
        [0] StaticText(value='<b>ShapeViewer</b>')
        [1] Row(width=300)
            [0] Select(margin=(5, 0, 5, 10), name='Shape', options=OrderedDict([(
↪'NGon00654',...]), sizing_mode='stretch_width', value=NGon)
            [1] Toggle(align='end', button_type='primary', height_policy='fit',␣
↪margin=(0, 0, 5, 10), max_height=30, max_width=20, name='')
    [1] Column
        [0] ParamMethod(method, _pane=Markdown, defer_load=False)
        [1] ParamMethod(method, _pane=Bokeh, defer_load=False)
```

Alternativ können wir eine völlig getrennte `expand_layout`-Instanz für einen Param-Bereich bieten, die mit `expand` und `expand_button`-Option immer ausgeklappt bleibt. Dies ermöglicht uns, die Haupt-Widgets und die Widgets des Unterobjekts getrennt anzuordnen:

[15]:
```
viewer = ShapeViewer()

expand_layout = pn.Column()

pn.Row(
    pn.Column(
        pn.panel(
            viewer.param,
            expand_button=False,
            expand=True,
            expand_layout=expand_layout,
        ),
        "#### Subobject parameters:",
        expand_layout,
    ),
    viewer.panel(),
)
```

[15]:
```
Row
    [0] Column
        [0] Column(margin=(5, 10), name='ShapeViewer')
            [0] StaticText(value='<b>ShapeViewer</b>')
            [1] Select(name='Shape', options=OrderedDict([('NGon00654',...]), value=NGon)
        [1] Markdown(str)
        [2] Column
            [0] Param(NGon, expand=True, expand_button=False, expand_layout=Column)
    [1] Column
        [0] ParamMethod(method, _pane=Markdown, defer_load=False)
        [1] ParamMethod(method, _pane=Bokeh, defer_load=False)
```

13.3.6 Styling

Panel-Objekte bauen auf param auf, wodurch für sie Parameter angegeben werden können, die Benutzer flexibel bearbeiten können, um die angezeigte Ausgabe zu steuern. Zusätzlich zu den für jede Komponente und Komponentenklasse spezifischen Parametern definieren alle Komponenten einen gemeinsamen Satz von Parametern, um die Größe und den Stil des gerenderten View zu steuern.

[1]:
```
import panel as pn

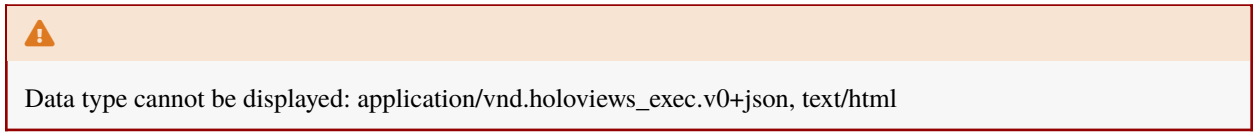

pn.extension()
```



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



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



Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

Styling-Komponenten

`css_classes`

Der`css_classes`-Parameter ermöglicht die Zuordnung einer Panel-Komponente zu einer oder mehreren CSS-Klassen. CSS-kann direkt im Notebook angegeben werden oder als Verweis auf eine externe CSS-Datei, indem sie mit `raw_css` oder `css_files` als Liste an die *Panel extension* übergeben werden. Außerhalb eines Notebooks, in einem externen Modul oder einer Bibliothek, können wir mit `pn.config.raw_css` und `pn.config.js_files` Konfigurationsparameter anhängen.

Um diese Verwendung zu demonstrieren, definieren wir eine CSS-Klasse mit dem Namen, `widget-box`:

[2]:
```
css = """
.widget-box {
  background: #f0f0f0;
  border-radius: 5px;
  border: 1px black solid;
}
"""

pn.extension(raw_css=[css])
```

(Fortsetzung der vorherigen Seite)

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

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

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

[3]:
```
pn.Column(
    pn.widgets.FloatSlider(name="Quantity", margin=(20, 26, 6, 26)),
    pn.widgets.Select(
        name="Fruit",
        options=["Apple", "Pear", "Orange"],
        margin=(10, 26, 6, 26),
    ),
    pn.widgets.Button(name="Run", margin=(34, 26, 20, 26)),
    css_classes=["widget-box"],
)
```

[3]:
```
Column(css_classes=['widget-box'])
    [0] FloatSlider(margin=(20, 26, 6, 26), name='Quantity')
    [1] Select(margin=(10, 26, 6, 26), name='Fruit', options=['Apple', 'Pear', ...],␣
↪value='Apple')
    [2] Button(margin=(34, 26, 20, 26), name='Run')
```

background

Wenn wir der Komponente einfach einen Hintergrund geben möchten, können wir einen als Hex-String definieren:

[4]:
```
pn.Column(styles={"background": "#f0f0f0", "width": "100", "height": "100"})
```

[4]:
```
Column(styles={'background': '#f0f0f0', ...})
```

style

Bestimmte Komponenten, insbesondere markup-bezogene Panes, stellen einen `style`-Parameter zur Verfügung, mit dem CSS-Stile definiert werden können, die auf den HTML-Container des Fensterinhalts angewendet werden, z.B. das `Markdown`-Pane:

[5]:
```
pn.pane.Markdown("### Cusy: DevOps", styles={"font-family": "sans-serif"})
```

[5]:
```
Markdown(str, styles={'font-family': '...})
```

Komponentengröße und Layout

Die Größe der Komponenten und ihr Abstand werden auch über eine Reihe von Parametern gesteuert, die von allen Komponenten gemeinsam verwendet werden.

`margin`

Der `margin`-Parameter kann verwendet werden, um Platz rund um ein Element zu schaffen, das als Anzahl der Pixel in der Reihenfolge oben, rechts, unten und links definiert ist, z.B.:

[6]:
```
pn.Row(
    pn.Column(
        pn.widgets.Button(name="Selector", margin=(20, 16, 20, 26)),
        styles={"background": "#f0f0f0"},
    ),
    pn.Column(
        pn.widgets.Button(name="Widget", margin=(20, 16, 20, 0)),
        styles={"background": "#f0f0f0"},
    ),
    pn.Column(
        pn.widgets.Button(name="Description", margin=(20, 26, 20, 0)),
        styles={"background": "#f0f0f0"},
    ),
)
```

[6]:
```
Row
    [0] Column(styles={'background': '#f0f0f0'})
        [0] Button(margin=(20, 16, 20, 26), name='Selector')
    [1] Column(styles={'background': '#f0f0f0'})
        [0] Button(margin=(20, 16, 20, 0), name='Widget')
    [2] Column(styles={'background': '#f0f0f0'})
        [0] Button(margin=(20, 26, 20, 0), name='Description')
```

Absolute Dimensionierung mit `width` und `height`

Standardmäßig verwenden alle Komponenten entweder die automatische oder die absolute Größenänderung. Bedienfelder nehmen im Allgemeinen so viel Platz ein wie die darin enthaltenen Komponenten, und text- oder bildbasierte Bedienfelder passen sich an die Größe ihres Inhalts an. Um eine feste Größe für eine Komponente festzulegen, ist es normalerweise ausreichend, eine Breite oder Höhe festzulegen. In bestimmten Fällen muss jedoch `sizing_mode='fixed'` explizit angegeben werden.

[7]:
```
pn.Row(
    pn.pane.Markdown(
        "\>CUSY_",
        styles={
            "color": "white",
            "font-weight": "300",
            "background": "black",
            "width": "100px",
            "height": "100px",
            "padding": "10px",
        },
    ),
    pn.pane.GIF("../../ipywidgets/smiley.gif", width=100),
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    pn.widgets.FloatSlider(width=100),
)
```

[7]:
```
Row
    [0] Markdown(str, styles={'color': 'white', ...})
    [1] GIF(str, width=100)
    [2] FloatSlider(width=100)
```

sizing_mode

`sizing_mode` kann die folgenden Werte annehmen:

- `fixed`: Die Komponente ist nicht responsiv. Die ursprüngliche Breite und Höhe wird unabhängig von nachfolgenden Ereignissen zur Größenänderung des Browserfensters beibehalten. Dies ist das Standardverhalten und verwendet einfach die angegebene Breite und Höhe.
- `stretch_width`: Die Komponente passt die Größe an, um sie auf die verfügbare Breite zu strecken, ohne jedoch das Seitenverhältnis beizubehalten. Die Höhe der Komponente hängt vom Typ der Komponente ab und kann fest oder an den Inhalt der Komponente gebunden sein.
- `stretch_height`: Die Größe der Komponente wird ansprechend angepasst, um sie auf die verfügbare Höhe zu erreichen, ohne jedoch das Seitenverhältnis beizubehalten. Die Breite der Komponente hängt vom Typ der Komponente ab und kann fest oder an den Inhalt der Komponente gebunden sein.
- `stretch_both`: Die Komponente ist responsiv, unabhängig von Breite und Höhe, und belegt den gesamten verfügbaren horizontalen und vertikalen Raum, auch wenn sich dadurch das Seitenverhältnis der Komponente ändert.
- `scale_height`: Die Größe der Komponente wird ansprechend angepasst, um sie auf die verfügbare Höhe zu strecken, wobei das ursprüngliche oder bereitgestellte Seitenverhältnis beibehalten wird.
- `scale_width`: Die Größe der Komponente wird ansprechend angepasst, um sie auf die verfügbare Breite zu strecken, wobei das ursprüngliche oder bereitgestellte Seitenverhältnis beibehalten wird.
- `scale_both`: Die Größe der Komponente wird ansprechend auf die verfügbare Breite und Höhe angepasst, wobei das ursprüngliche oder bereitgestellte Seitenverhältnis beibehalten wird.

[8]:
```
pn.pane.Str(styles={"background": "#f0f0f0", "height": "100", "sizing_mode": "stretch_
↪width"})
```

[8]:
```
Str(None, styles={'background': '#f0f0f0', ...})
```

[9]:
```
pn.Column(
    pn.pane.Str(styles={"background": "#f0f0f0", "sizing_mode": "stretch_height"}),␣
↪height=100
)
```

[9]:
```
Column(height=100)
    [0] Str(None, styles={'background': '#f0f0f0', ...})
```

[10]:
```
pn.Column(
    pn.pane.Str(styles={"background": "#f0f0f0", "sizing_mode": "stretch_both"}),␣
↪height=100
)
```

[10]:
```
Column(height=100)
    [0] Str(None, styles={'background': '#f0f0f0', ...})
```

[11]:
```
pn.Column(
    pn.pane.GIF("../../ipywidgets/smiley.gif", sizing_mode="scale_both"),
    styles={"background": "#f0f0f0"},
)
```

[11]:
```
Column(styles={'background': '#f0f0f0'})
    [0] GIF(str, sizing_mode='scale_both')
```

Spacer

`Spacer` sind eine sehr vielseitige Komponente, mit der sich feste oder responsive Abstände zwischen Objekten problemlos herstellen lassen. Wie alle anderen Komponenten unterstützen `Spacer` sowohl den absoluten als auch den responsiven Modus:

[12]:
```
pn.Row(
    1,
    pn.Spacer(width=200),
    2,
    pn.Spacer(width=100),
    3,
    pn.Spacer(width=50),
    4,
    pn.Spacer(width=25),
    5,
)
```

[12]:
```
Row
    [0] Str(int)
    [1] Spacer(width=200)
    [2] Str(int)
    [3] Spacer(width=100)
    [4] Str(int)
    [5] Spacer(width=50)
    [6] Str(int)
    [7] Spacer(width=25)
    [8] Str(int)
```

`VSpacer` und `HSpacer` sorgen für einen ansprechenden vertikalen bzw. horizontalen Abstand. Mit diesen Komponenten können wir Objekte in einem Layout in gleichem Abstand platzieren und den leeren Bereich verkleinern, wenn die Größe des Browsers geändert wird:

[13]:
```
pn.Row(
    "* Item 1\n* Item2",
    pn.layout.HSpacer(),
    "1. First\n2. Second",
    pn.layout.HSpacer(),
)
```

[13]:
```
Row
    [0] Markdown(str)
    [1] HSpacer()
    [2] Markdown(str)
    [3] HSpacer()
```

13.3.7 Deploy und Export

Eines der Hauptentwurfsziele für Panel bestand darin, einen nahtlosen Übergang zwischen interaktivem Prototyping eines Dashboards und der Bereitstellung als eigenständige Server-App zu ermöglichen. In diesem Notebook wird gezeigt, wie Panels interaktiv angezeigt, statische Ausgaben eingebettet, ein Snapshot gespeichert und als separate Webserver-App bereitgestellt werden.

Ausgabe konfigurieren

Panel-Objekte werden automatisch in einem Notebook angezeigt und nutzen Jupyter Comms, um die Kommunikation zwischen der gerenderten App und dem Jupyter-Kernel zu unterstützen. Das Anzeigen eines Panel-Objekts im Notebook ist einfach: es muss nur zunächst die `panel.extension` geladen werden, um das erforderliche JavaScript im Notebook-Kontext zu initialisieren.

[1]:
```
import panel as pn


pn.extension()
```

⚠

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

⚠

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

⚠

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

Optionale Abhängigkeiten

Um bestimmte Komponenten wie Vega, LaTeX und Plotly-Plots verwenden zu können, müssen die entsprechenden Javascript-Komponenten ebenfalls geladen werden. Hierfür könnt ihr sie einfach als Teil des Aufrufs von `pn.extension` angeben:

[2]:
```
pn.extension("vega", "katex")
```

(Fortsetzung der vorherigen Seite)

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

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

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

JS und CSS initialisieren

Auch zusätzliches CSS- und Javascript kann mit `css_files`, `js_files` und `raw_css` angegeben werden. Dabei sollte `js_files` als Dictionary-Mapping vom exportierten JS-Modulnamen zur URL mit den JS-Komponenten angegeben werden während `css_files` als Liste definiert werden kann:

[3]:
```
pn.extension(
    js_files={"deck": "https://unpkg.com/deck.gl@~5.2.0/deckgl.min.js"},
    css_files=[
        "https://api.tiles.mapbox.com/mapbox-gl-js/v0.44.1/mapbox-gl.css"
    ],
)
```

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

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

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

Mit diesem `raw_css`-Argument könnt ihr eine Liste von Zeichenfolgen mit CSS definieren, die als Teil des Notebooks und der App veröffentlicht werden sollen.

Das Bereitstellen von Keyword-Argumenten mit `extension` entspricht dem Festlegen mit `pn.config`. `pn.config` ist der bevorzugte Ansatz um außerhalb eine Notebooks Javascript- und CSS-Dateien hinzuzufügen:

[4]:
```
pn.config.js_files = {"deck": "https://unpkg.com/deck.gl@~5.2.0/deckgl.min.js"}
pn.config.css_files = [
    "https://api.tiles.mapbox.com/mapbox-gl-js/v0.44.1/mapbox-gl.css"
]
```

Anzeige im Notebook

Sobald `extension` geladen ist, werden Panel-Objekte, die am Ende einer Zelle platziert werden, angezeigt:

[5]:
```
pane = pn.panel("<marquee>Here is some custom HTML</marquee>")

pane
```

[5]:
```
Markdown(str)
```

Die `display`-Funktion

Um zu vermeiden, dass ein Panel in die letzte Zeile einer Notebook-Zelle gestellt werden muss, könnt ihr die IPython-`display`-Funktion verwenden:

[6]:
```
def display_marquee(text):
    display(pn.panel("<marquee>{text}</marquee>".format(text=text)))

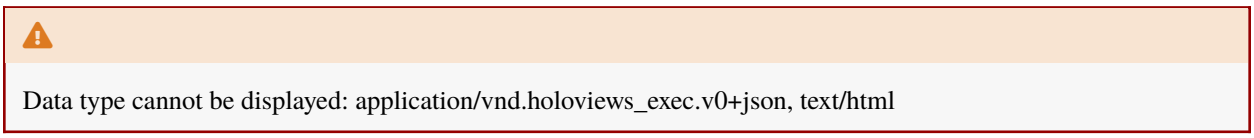

display_marquee("This Panel was displayed from within a function")
```

```
Markdown(str)
```

Inline-Apps

Schließlich ist es auch möglich, mit `pn.io.notebook.show_server` ein Panel-Objekt als Bokeh-Server-App im Notebook anzuzeigen:

[7]:
```
pn.io.notebook.show_server(pane, "localhost:8888")
```



Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

[7]:
```
<panel.io.server.Server at 0x13f623210>
```

Anzeige in einem interaktiven Python-Fenster (REPL)

Wenn ihr über die Befehlszeile arbeitet, werden nicht automatisch umfangreiche Darstellungen inline dargestellt, wie dies in einem Notebook der Fall ist. Ihr könnt jedoch mit euren Panel-Komponenten interagieren, wenn ihr eine Bokeh-Serverinstanz startet und mit der `show`-Methode ein separates Browserfenster öffnet. Die Methode hat folgende Argumente:

- `port`: `int`, (optional): erlaubt die Angabe eines spezifischen Ports (`default=0` wählt einen beliebigen offenen Port)
- `websocket_origin`: `str` oder `list(str)` (optional): Eine Liste von Hosts, die sich mit dem Websocket verbinden können. Dies ist erforderlich wenn eine Server-App in eine externe Website eingebettet wird. Wenn keine Angabe gemacht wird, wird `localhost` verwendet.

- `threaded`: `boolean` (optional, `default=False`): `True` startet den Server in einem separaten Thread und erlaubt euch, interaktiv mit der App agieren zu können.

Der `show`-Aufruf gibt entweder eine Bokeh-Serverinstanz (`threaded=False`) oder eine `StoppableThread`-Instanz (`threaded=True`) zurück, die beide eine `stop`-Methode zum Stoppen der Serverinstanz bereitstellen.

Starten eines Servers in der Befehlszeile

Panel (und Bokeh) stellen einen CLI-Befehl zum Bereitstellen eines Python-Skripts, eines App-Verzeichnisses oder eines Jupyter-Notebooks mit einer Bokeh- oder Panel-App bereit. Um einen Server über die CLI zu starten, führt einfach Folgendes aus:

```
$ uv run panel serve app.ipynb
```

Um ein Notebook in eine bereitstellbare App zu verwandeln, hängt einfach an ein oder mehrere Panel-Objekte `.servable()` an, wodurch die App zu Bokehs `curdoc` hinzufügt. Auf diese Weise ist es einfach, Dashboards interaktiv in einem Notebook zu erstellen und sie dann nahtlos auf dem Bokeh-Server bereitzustellen.

Sitzungsstatus

- `panel.state` macht einige der internen Bokeh-Serverkomponenten für Benutzer verfügbar.
- `panel.state.curdoc` erlaubt den Zugriff auf das aktuelle bokeh.document.

Einbetten

Panel benötigt im Allgemeinen entweder den Jupyter-Kernel oder einen Bokeh-Server, der im Hintergrund ausgeführt wird, um interaktives Verhalten zu ermöglichen. Für einfache Apps ist es jedoch auch möglich, den gesamten Widget-Status zu erfassen, sodass die App vollständig von Javascript aus verwendet werden kann. Um dies zu demonstrieren, erstellen wir eine einfache App, die einfach einen Schiebereglerwert annimmt, diesen mit 5 multipliziert und dann das Ergebnis anzeigt:

[8]:
```
slider = pn.widgets.IntSlider(
    name="Integer to Scientific Notation Converter", start=0, end=10
)


@pn.depends(slider.param.value)
def callback(value):
    return "%d = %e" % (value, value)


row = pn.Row(slider, callback)
```

[9]:
```
row.embed()
```

[9]:
```
<panel.io.notebook.Mimebundle at 0x103a16990>
```

Wenn ihr das obige Widget ausprobieren, werdet ihr feststellen, dass es nur drei verschiedene Status hat, `0`, `5` und `10`. Dies liegt daran, dass beim Einbetten standardmäßig versucht wird, die Anzahl der Optionen für nicht-diskrete oder halb-diskrete Widgets auf höchstens drei Werte zu beschränken. Dies kann mit dem `max_opts`-Argument der `embed`-Methode verändert werden. Die vollständigen Optionen für die `embed`-Methode sind:

- `max_states`: Maximale Anzahl der einzubettenden Zustände
- `max_opts`: Maximale Anzahl von Status für ein einzelnes Widget
- `json`: Gibt an, ob die Daten in json-Dateien exportiert werden sollen

- `save_path`: Pfad zum Speichern von JSON-Dateien (`default='./'`)
- `load_path`: Pfad oder URL, von dem bzw. der die JSON-Dateien geladen werden (wie `save_path` wenn nicht anders angegeben)

Wie ihr euch leicht vorstellen könnt, kann es bei mehreren Widgets schnell zu einer kombinatorischen Explosion der Status kommen, sodass die Ausgabe standardmäßig auf etwa 1000 Status beschränkt ist. Bei größeren Apps können die Status auch in JSON-Dateien exportiert werden. Wenn ihr die App beispielsweise auf einer Website bereitstellen möchtet, gebt mit `save_path` an, wo die JSON-Datei gespeichert werden soll und mit `load_path`, wo der JS-Code nach den Dateien suchen soll.

Speichern

Wenn ihr keinen tatsächlichen Server benötigt oder einfach einen statischen Snapshot einer Panel-App exportieren möchtet, könnt ihr die save-Methode verwenden, mit der die App in eine eigenständige HTML- oder PNG-Datei exportiert werden kann.

Standardmäßig hängt die generierte HTML-Datei vom Laden des JavaScript-Codes für BokehJS aus dem Online-CDNRepository ab, um die Dateigröße zu verringern. Wenn Sie in einer Umgebung mit oder ohne Netzwerk arbeiten müssen, können Sie festlegen, dass INLINERessourcen anstelle von CDN:

[10]:
```
from bokeh.resources import INLINE


pane.save("deploy-panel.html", resources=INLINE)
pane.save("test.png")
```

Für den Export der png-Datei benötigt ihr zusätzlich Selenium und PhantomJS:

```
$ uv add selenium
...
$ npm install -g phantomjs-prebuilt
...
Done. Phantomjs binary available at /usr/local/lib/node_modules/phantomjs-prebuilt/lib/
↪phantom/bin/phantomjs
+ phantomjs-prebuilt@2.1.16
added 81 packages from 76 contributors in 31.121s
```

Darüber hinaus könnt ihr mit der `save`-Methode z.B. auch die `embed`-Option aktivieren um die App-Status in die App einzubetten oder in JSON-Dateien zu speichern, die zusammen mit dem exportierten HTML-Code deployed werden können. U.a. habt ihr folgende Optionen:

- `resources`: `bokeh.resources`, z.B. `CDN` oder `INLINE`
- `embed`: Boolscher Wert, ob die Status in der Datei gespeichert werden sollen oder nicht.
- `max_states`: Die maximale Anzahl der einzubettenden Status
- `max_opts`: Die maximale Anzahl der Status für ein einzelnes Widget
- `embed_json`: Boolscher Wert, ob die Daten als JSON-Datei exportiert werden sollen (`default=True`).

13.3.8 Pipelines

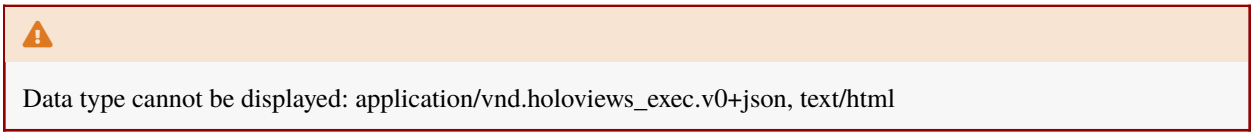
In Parametrisierung wurde beschrieben, wie Klassen erstellt werden, die Parameter deklarieren und mit Berechnungen oder Visualisierungen verknüpfen. In diesem Abschnitt erfahrt ihr, wie ihr mehrere solcher Panels mit einer Pipeline verbinden könnt um komplexe Workflows auszudrücken, bei denen die Ausgabe einer Stufe in die nächste Stufe eingespeist wird.

[1]:
```
import panel as pn
import param


pn.extension("katex")
```







[2]:
```
pipeline = pn.pipeline.Pipeline()
```

Während wir früher bereits gesehen haben, wie Methoden mit dem `param.depends`-Decorator verknüpft werden, verwenden Pipelines einen anderen Decorator und eine Konvention zum Anzeigen der Objekte. Der `param.output`-Decorator bietet eine Möglichkeit, die Methoden einer Klasse mit Anmerkungen zu versehen, indem seine Ausgaben deklariert werden. `Pipeline` verwendet diese Informationen, um zu bestimmen, welche Ausgaben verfügbar sind, um in die nächste Stufe des Workflows eingespeist zu werden. Im folgenden Beispiel hat die Klasse `Stage1` zwei Parameter (`a` und `b`) und eine Ausgabe `c`. Die Signatur des Decorators ermöglicht eine Reihe von verschiedenen Möglichkeiten, die Ausgaben zu deklarieren:

- `param.output()`: Wenn eine Ausgabe ohne Argumente deklariert wird, gibt die Methode eine Ausgabe zurück, die den Namen der Methode erbt und keine spezifischen Typ-Deklarationen vornimmt.
- `param.output(param.Number)`: Beim Deklarieren einer Ausgabe mit einem bestimmten Parameter oder einem Python-Typ wird die Ausgabe mit einem bestimmten Typ deklariert.
- `param.output(c=param.Number)`: Wenn eine Ausgabe mit einem Keyword-Argument deklariert wird, könnt ihr damit den Methodennamen als Namen der Ausgabe überschreiben und den Typ deklarieren.

Es ist auch möglich, mehrere Parameter als Keywords oder als Tupel zu deklarieren:

- `param.output(c=param.Number, d=param.String)`
- `param.output(('c', param.Number), ('d', param.String))`

Im folgenden Beispiel ist die Ausgabe einfach das Ergebnis der Multiplikation der beiden Eingaben (`a` und `b`), die die Ausgabe `c` erzeugen. Zusätzlich deklarieren wir eine `view`-Methode, die einen `LaTeX`-Pane zurückgibt. Schließlich gibt eine `panel`-Methode ein Panel-Objekt zurück, das sowohl die Parameter als auch den View rendern.

[3]:
```
class Stage1(param.Parameterized):
    a = param.Number(default=5, bounds=(0, 10))

    b = param.Number(default=5, bounds=(0, 10))
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    @param.output(("c", param.Number), ("d", param.Number))
    def output(self):
        return self.a * self.b, self.a**self.b

    @param.depends("a", "b")
    def view(self):
        c, d = self.output()
        return pn.pane.LaTeX(
            "${a} * {b} = {c}$\n${a}^{{{b}}} = {d}$".format(
                a=self.a, b=self.b, c=c, d=d
            )
        )

    def panel(self):
        return pn.Row(self.param, self.view)


stage1 = Stage1()
stage1.panel()
```

[3]:
```
Row
    [0] Column(margin=(5, 10), name='Stage')
        [0] StaticText(value='<b>Stage</b>')
        [1] FloatSlider(end=10, name='A', value=5)
        [2] FloatSlider(end=10, name='B', value=5)
    [1] ParamMethod(method, _pane=LaTeX, defer_load=False)
```

Zusammenfassend haben wir einige Konventionen befolgt, um diese Phase unserer Pipeline zu erstellen:

1. Deklarieren einer parametrisierten Klasse mit einigen Eingabeparametern
2. Deklarieren und Benennen einer oder mehrerer Ausgabemethoden
3. Deklarieren einer `panel`-Methode, die einen View des Objekts zurückgibt, das von der Pipeline gerendert werden kann.

Nachdem das Objekt nun instanziiert wurde, können wir es auch nach seinen Ausgaben befragen:

[4]:
```
stage1.param.outputs()
```

[4]:
```
{'c': (<param.Number at 0x13f5fb7c0>,
  <bound method Stage1.output of Stage1(a=5, b=5, name='Stage100954')>,
  0),
 'd': (<param.Number at 0x13f5fb640>,
  <bound method Stage1.output of Stage1(a=5, b=5, name='Stage100954')>,
  1)}
```

Wir können sehen, dass `Stage1` eine Ausgabe mit dem Namen `c` und dem Typ `Number` deklariert, auf die mit der `output`-Methode zugegriffen werden kann. Nun fügen wir `stage1` mit `add_stage` unserer Pipeline hinzu:

[5]:
```
pipeline.add_stage("Stage 1", stage1)
```

Für eine Pipeline benötigen wir jedoch noch mindestens eine `stage2`, das das Ergebnis von `stage1` weiterverarbeitet. Daher sollte ein Parameter `c` aus dem Ergebnis von `stage1` deklariert werden. Als weiteren Parameter definieren wir `exp` und erneut eine `view`-Methode, die von den beiden Parametern und der `panel`-Methode abhängt.

[6]:
```
class Stage2(param.Parameterized):
    c = param.Number(default=5, precedence=-1, bounds=(0, None))

    exp = param.Number(default=0.1, bounds=(0, 3))

    @param.depends("c", "exp")
    def view(self):
        return pn.pane.LaTeX(
            "${%s}^{%s}={%.3f}$" % (self.c, self.exp, self.c**self.exp)
        )

    def panel(self):
        return pn.Row(self.param, self.view)


stage2 = Stage2(c=stage1.output()[0])
stage2.panel()
```

[6]:
```
Row
    [0] Column(margin=(5, 10), name='Stage')
        [0] StaticText(value='<b>Stage</b>')
        [1] FloatSlider(end=3, name='Exp', value=0.1)
    [1] ParamMethod(method, _pane=LaTeX, defer_load=False)
```

Auch `stage2` fügen wir nun dem `pipeline`-Objekt hinzu:

[7]:
```
pipeline.add_stage("Stage 2", stage2)
```

Wir haben nun eine zweistufige Pipeline, bei der der Output `c` von `stage1` an `stage2` übergeben wird. Nun können wir uns die Pipeline anzeigen lassen mit `pipeline.layout`:

[8]:
```
pipeline.layout
```

[8]:
```
Column(sizing_mode='stretch_width')
    [0] Row(sizing_mode='stretch_width')
        [0] Column
            [0] Markdown(str, margin=(0, 0, 0, 5))
            [1] Row(width=100)
        [1] HoloViews(Overlay, backend='bokeh', height=80, sizing_mode='stretch_width')
        [2] Row
            [0] Button(disabled=True, name='Previous', width=125)
            [1] Button(name='Next', width=125)
    [1] Row
        [0] Row
            [0] Column(margin=(5, 10), name='Stage')
                [0] StaticText(value='<b>Stage</b>')
                [1] FloatSlider(end=10, name='A', value=5)
                [2] FloatSlider(end=10, name='B', value=5)
            [1] ParamMethod(method, _pane=LaTeX, defer_load=False)
```

Das Rendering der Pipeline zeigt ein kleines Diagramm mit den verfügbaren Workflow-Stufen sowie die Schaltflächen *Previous* und *Next*, um zwischen den einzelnen Phasen wechseln zu können. Dies ermöglicht die Navigation auch in komplexeren Workflows mit sehr viel mehr Phasen.

Oben haben wir jede Stufe einzeln instanziiert. Wenn die Pipeline jedoch als Server-App deployed werden soll, können die Stufen jedoch auch als Teil des Konstruktors deklariert werden:

[9]:
```
stages = [("Stage 1", Stage1), ("Stage 2", Stage2)]

pipeline = pn.pipeline.Pipeline(stages)
pipeline.layout
```

[9]:
```
Column(sizing_mode='stretch_width')
    [0] Row(sizing_mode='stretch_width')
        [0] Column
            [0] Markdown(str, margin=(0, 0, 0, 5))
            [1] Row(width=100)
        [1] HoloViews(Overlay, backend='bokeh', height=80, sizing_mode='stretch_width')
        [2] Row
            [0] Button(disabled=True, name='Previous', width=125)
            [1] Button(name='Next', width=125)
    [1] Row
        [0] Row
            [0] Column(margin=(5, 10), name='Stage')
                [0] StaticText(value='<b>Stage</b>')
                [1] FloatSlider(end=10, name='A', value=5)
                [2] FloatSlider(end=10, name='B', value=5)
            [1] ParamMethod(method, _pane=LaTeX, defer_load=False)
```

Dabei können die Pipeline-Stufen entweder `Parameterized`-Instanzen oder `Parameterized`-Klassen sein. Bei Instanzen müsst ihr jedoch darauf achten, dass die Aktualisierung der Parameter der Klasse auch den aktuellen Status der Klasse aktualisiert.

13.3.9 Templates

Wenn ihr eine Panel-App oder ein Dashboard als Bokeh-Anwendung bereitstellen wollt, wird diese in einem Standard-Template gerendert, das auf die JS- und CSS-Ressourcen sowie das eigentliche Panel-Objekt verweist. Wenn ihr das Layout der bereitgestellten App anpassen wollt oder mehrere separate Panels in eine App bereitstellen wollt, ermöglicht euch die `Template`-Komponente von Panel das Anpassen dieses Standard-Templates.

Ein solches Template wird mithilfe von Jinja definiert, wobei ihr das Standard-Template erweitert oder sogar vollständig ersetzen könnt. Im Folgenden seht ihr ein Beispiel:

```
<!DOCTYPE html>
<html lang="en">
{% block head %}
<head>
    {% block inner_head %}
    <meta charset="utf-8">
    <title>{% block title %}{{ title | e if title else "Panel App" }}{% endblock %}</
↪title>
    {% block preamble %}{% endblock %}
    {% block resources %}
        {% block css_resources %}
        {{ bokeh_css | indent(8) if bokeh_css }}
        {% endblock %}
        {% block js_resources %}
        {{ bokeh_js | indent(8) if bokeh_js }}
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
        {% endblock %}
    {% endblock %}
    {% block postamble %}{% endblock %}
    {% endblock %}
</head>
{% endblock %}
{% block body %}
<body>
    {% block inner_body %}
    {% block contents %}
        {% for doc in docs %}
        {{ embed(doc) if doc.elementid }}
        {% for root in doc.roots %}
            {{ embed(root) | indent(10) }}
        {% endfor %}
        {% endfor %}
    {% endblock %}
    {{ plot_script | indent(8) }}
    {% endblock %}
</body>
{% endblock %}
</html>
```

Das Template definiert eine Reihe von benutzerdefinierten Blöcken, die durch `extends` ergänzt oder überschrieben werden können:

Benutzerdefinierte Templates verwenden

[1]:
```
import holoviews as hv
import panel as pn


pn.extension()
```



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



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



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

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

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

Sobald wir Panel geladen haben, können wir mit der Definition eines benutzerdefinierten Templates beginnen. Normalerweise ist es am einfachsten, ein vorhandenes Template durch Überschreiben bestimmter Blöcke anzupassen. Mit `{% extends base %}` erklären wir, dass wir lediglich eine vorhandene Vorlage erweitern und keine neue definieren.

Im folgenden Fall erweitern wir den `postamble`-Block des Headers, um eine zusätzliche Ressource zu laden, und den `contents`-Block, um die Anordnung der Komponenten neu zu definieren:

[2]:
```
template = """
{% extends base %}

<!-- Addition to head -->
{% block postamble %}
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
↪bootstrap.min.css">
{% endblock %}

<!-- Addition to body -->
{% block contents %}
{{ app_title }}
<p>This is a Panel app with a custom template allowing us to compose multiple Panel␣
↪objects into a single HTML document.</p>
<br>
<div class="container">
  <div class="row">
    <div class="col-sm">
      {{ embed(roots.A) }}
    </div>
    <div class="col-sm">
      {{ embed(roots.B) }}
    </div>
  </div>
</div>
{% endblock %}
"""
```

Mithilfe des `embed`-Makros haben wir zwei verschiedene `roots` in der Vorlage definiert. Um die Vorlage rendern zu können, müssen wir nun zuerst das `pn.Template`-Objekt mit dem HTML-Template erstellen und dann die beiden `roots`-Objekte einbinden.

[3]:
```
tmpl = pn.Template(template)

tmpl.add_variable('app_title', '<h1>Custom Template App</h1>')
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
tmpl.add_panel('A', hv.Curve([1, 2, 3]))
tmpl.add_panel('B', hv.Curve([1, 2, 3]))
tmpl.servable()
```

[3]:
```
Template
    [A] HoloViews(Curve, height=300, sizing_mode='fixed', width=300)
    [B] HoloViews(Curve, height=300, sizing_mode='fixed', width=300)
```

Wenn das Template größer ist, ist es oft einfacher, es in einer separaten Datei zu erstellen. Ihr könnt den Lademechanismus für Jinja2-Vorlagen verwenden, indem ihr ein Environment zusammen mit einem `loader` definiert:

[4]:
```
from jinja2 import Environment, FileSystemLoader


env = Environment(loader=FileSystemLoader("."))
jinja_template = env.get_template("sample_template.html")

tmpl = pn.Template(jinja_template)

tmpl.add_panel("A", hv.Curve([1, 2, 3]))
tmpl.add_panel("B", hv.Curve([1, 2, 3]))

tmpl
```

[4]:
```
Template
    [A] HoloViews(Curve, height=300, sizing_mode='fixed', width=300)
    [B] HoloViews(Curve, height=300, sizing_mode='fixed', width=300)
```

13.3.10 Panel im Browser ausführen mit WASM

Mit Panel könnt ihr Dashboards und andere Anwendungen in Python schreiben, auf die über einen Webbrowser zugegriffen wird. Normalerweise läuft der Python-Interpreter als separater Jupyter- oder Bokeh-Serverprozess und kommuniziert mit dem JavaScript-Code, der im Client-Browser läuft. Python kann jedoch mit WASM (WebAssembly) auch direkt im Browser ausgeführt werden, ohne dass ein separater Server erforderlich ist.

Panel nutzt hierfür Pyodide und für das Rendering PyScript.

Konvertieren von Panel-Anwendungen

Zukünftige Versionen von Panel kann eure Panel-Anwendung aus ein oder mehreren Python-Skripten oder Notebook-Dateien einschließlich *Templates* mit `panel convert` in eine HTML-Datei umwandeln. Die einzige Voraussetzungen sind:

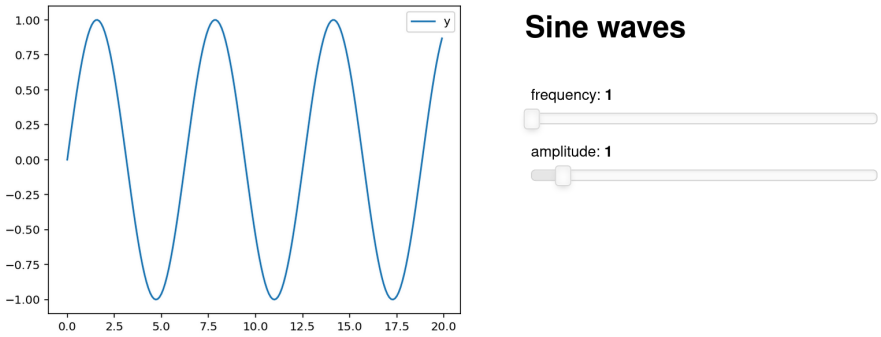
- sie importieren nur globale Module und Pakete und keine relativen Importe von anderen Skripten oder Modulen
- die Bibliotheken wurden für Pyodide kompiliert oder sind als Python wheels auf dem Python Package Index (PyPI) verfügbar.

Beispiel

In folgenden Beispiel werden wir das *Deploy und Export*-Notebook in eine eigenständige HTML-Seite konvertieren mit

```
$ uv run panel convert deploy.ipynb --out pyodide
Column
    [0] Column
        [0] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [1] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
        [2] IntSlider(end=100, name='n', start=5, value=200)
    [1] Row
        [0] Matplotlib(Figure, name='interactive00114')
Launching server at http://localhost:40405
```

Nun könnt ihr `http://localhost:40405` in eurem Browser öffnen und die App auszuprobieren:



Ihr könnt nun die Datei `pyodide/deploy.html` in eure Github-Seiten o.Ä. einfügen – es ist kein separater Server erforderlich.

Siehe auch

- Awesome Panel/Webassembly Apps

Optionen

Im Folgenden erläutere ich einige der Optionen von `panel convert`.

`--to`
: Das Format, in das konvertiert werden soll. Es gibt drei Optionen, die jeweils unterschiedliche Vor- und Nachteile haben:

  `pyodide` (Standard)
  : Die Anwendung wird mit Pyodide im Haupt-Thread ausgeführt. Diese Option ist weniger performant als `pyodide-worker`, erzeugt aber eine völlig eigenständige HTML-Datei, die nicht auf einem statischen Dateiserver, wie z.B. Github Pages, gehostet werden müssen.

  `pyodide-worker`
  : erzeugt HTML- und JS-Dateien, die jedoch einen Web-Worker enthält, der in einem separaten Thread läuft. Dies ist die leistungsfähigste Option, aber die Dateien müssen auf einem statischen Dateiserver gehostet werden.

`pyscript`
: erzeugt eine HTML-Datei, die PyScript nutzt. Dies erzeugt eigenständige HTML-Dateien mit <py-env>- und <py-script>-Tags, die die Abhängigkeiten und den Anwendungscode enthalten. Diese Ausgabe ist am lesbarsten und sollte die gleiche Leistung wie die Option `pyodide` haben.

`-out`
: Das Verzeichnis, in das die Dateien geschrieben werden sollen.

`--pwa`
: Fügt Dateien hinzu, die die Anwendung zu einer Progressive Web-Apps machen.

  Progressive Webanwendungen (PWAs) bieten eine Möglichkeit für eure Webanwendungen, sich fast wie eine native Anwendung zu verhalten, sowohl auf mobilen Geräten als auch auf dem Desktop. `panel convert` hat eine `--pwa`-Option, die die notwendigen Dateien generiert, um eure Panel- und Pyodide-Anwendung in eine PWA zu verwandeln.

`--skip-embed`
: Überspringt das Einbetten von vorgerenderten Inhalten in der konvertierten Datei.

  Panel bettet vorgerenderte Inhalte in die HTML-Seite ein und ersetzt diese durch Live-Komponenten, sobald die Seite geladen ist. Dies kann jedoch sehr lange dauern. Wenn ihr dieses Verhalten deaktivieren und zunächst eine leere Seite rendern möchtet, verwendet die Option `--skip-embed`.

`--index`
: erstellt einen Index wenn ihr mehrere Anwendungen auf einmal konvertiert, damit ihr leicht zwischen den Anwendungen navigieren könnt.

`--requirements`
: Explizite Anforderungen, die der konvertierten Datei oder einer `requirements.txt`-Datei hinzugefügt werden sollen.

  Standardmäßig werden die Anforderungen aus dem Code abgeleitet.

  Wenn eine Bibliothek einen optionalen Import verwendet, der nicht aus der Liste der Importe eurer Anwendung abgeleitet werden kann, müsst ihr eine explizite Liste der Abhängigkeiten angeben.

  Bemerkung

  panel und seine Abhängigkeiten, einschließlich NumPy und Bokeh, werden automatisch geladen, D.H. (das heißt) die expliziten Anforderungen für die obige Anwendung würden wie folgt aussehen:

  ```
  $ uv run panel convert deploy.ipynb --out pyodide --requirements pandas␣
  ↪matplotlib
  ```

  Alternativ könnt ihr auch eine `requirements.txt`-Datei bereitstellen:

  ```
  $ uv run panel convert deploy.ipynb --out pyodide --requirements requirements.txt
  ```

`--watch`
: Beobachten der Quelldateien.

Eine vollständige Übersicht erhaltet ihr mit `panel convert -u`.

Tipp

Wenn die konvertierte Anwendung nicht wie erwartet funktioniert, könnt ihr die Fehler meist in der Browser-Konsole finden, S.A. Finding Your Browser's Developer Console.

Siehe auch

Antworten auf die am häufigsten gestellten Fragen zu Python im Browser findet ihr in den

- Pyodide FAQ
- PyScript FAQ

13.3.11 FastAPI-Integration

Panel läuft üblicherweise auf einem Bokeh-Server, der wiederum auf Tornado läuft. Es ist jedoch auch oft nützlich, eine Panel-App in eine große Webanwendung einzubetten, wie z.B. einen FastAPI-Webserver. Die Integration in FastAPI ist einfacher im Vergleich zu anderen wie z.B. Flask, da es ein leichtgewichtigeres Framework ist. Die Verwendung von Panel mit FastAPI erfordert nur ein wenig mehr Aufwand als bei Notebooks und Bokeh-Servern.

Konfiguration

Bevor wir mit dem Hinzufügen einer Bokeh-Anwendung zu unserem FastApi-Server beginnen, müssen wir einige der grundlegenden Konfigurationen in `fastAPI/main.py` einrichten:

1. Zunächst importieren wir alle erforderlichen Elemente:

Quellcode 1: fastAPI/main.py

```
import panel as pn
from bokeh.embed import server_document

from fastapi import FastAPI, Request
from fastapi.templating import Jinja2Templates
```

2. Als nächstes definieren wir `app` als Instanz von `FastAPI` und definieren den Pfad zum Vorlagenverzeichnis:

```
app = FastAPI()
templates = Jinja2Templates(directory="templates")
```

3. Nun erstellen wir unsere erste Routine über eine asynchrone Funktion und verweisen sie an unseren Bokeh-Server:

```
@app.get("/")
async def bkapp_page(request: Request):
    script = server_document("http://127.0.0.1:5000/app")
    return templates.TemplateResponse(
        "base.html", {"request": request, "script": script}
    )
```

4. Wie ihr dem Code ansehen könnt, wird ein Jinja2-Template `fastAPI/templates/base.html` erwartet. Dieses kann z.B. folgenden Inhalt haben:

Quellcode 2: fastAPI/templates/base.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Panel in FastAPI: sliders</title>
    </head>
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    <body>
        {{ script|safe }}
    </body>
</html>
```

5. Kehren wir nun zurück zu unserer `fastAPI/main.py`-Datei um mit `pn.serve()` unseren Bokeh-Server zu starten:

```
    {"/app": createApp},
    port=5000,
    allow_websocket_origin=["127.0.0.1:8000"],
    address="127.0.0.1",
    show=False,
)
```

`createApp`
ruft in diesem Beispiel unsere Panel-App auf, die jedoch erst im nächsten Abschnitt behandelt wird.

`address`, `port`
Adresse und Port, an dem der Server auf Anfragen lauscht; in unserem Fall also `http://127.0.0.1:5000`.

GGF. (Gegebenenfalls) muss noch die Umgebungsvariable `BOKEH_ALLOW_WS_ORIGIN` gesetzt werden mit:

```
$ export BOKEH_ALLOW_WS_ORIGIN=127.0.0.1:5000
```

`show=False`
sorgt dafür, dass der Bokeh-Server zwar gestartet wird, jedoch nicht unmittelbar im Browser angezeigt wird.

`allow_websocket_origin`
listet die Hosts auf, die sich mit dem Websocket verbinden können. In unserem Beispiel soll das `fastApi` sein, also verwenden wir `127.0.0.1:8000`.

6. Nun definieren wir die `sliders`-App auf Basis einer Standardvorlage für FastAPI-Apps, die zeigt, wie Panel und FastAPI integriert werden können:

`fastAPI/sliders/sinewave.py`
ein parametrisiertes Objekt, das euren bereits vorhandenen Code darstellt:

Quellcode 3: fastAPI/sliders/sinewave.py

```
import numpy as np
import param
from bokeh.models import ColumnDataSource
from bokeh.plotting import figure


class SineWave(param.Parameterized):
    offset = param.Number(default=0.0, bounds=(-5.0, 5.0))
    amplitude = param.Number(default=1.0, bounds=(-5.0, 5.0))
    phase = param.Number(default=0.0, bounds=(0.0, 2 * np.pi))
    frequency = param.Number(default=1.0, bounds=(0.1, 5.1))
    N = param.Integer(default=200, bounds=(0, None))
    x_range = param.Range(default=(0, 4 * np.pi), bounds=(0, 4 * np.pi))
    y_range = param.Range(default=(-2.5, 2.5), bounds=(-10, 10))

```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
    def __init__(self, **params):
        super().__init__(**params)
        x, y = self.sine()
        self.cds = ColumnDataSource(data=dict(x=x, y=y))
        self.plot = figure(
            height=400,
            width=400,
            tools="crosshair, pan, reset, save, wheel_zoom",
            x_range=self.x_range,
            y_range=self.y_range,
        )
        self.plot.line("x", "y", source=self.cds, line_width=3, line_alpha=0.6)

    @param.depends(
        "N",
        "frequency",
        "amplitude",
        "offset",
        "phase",
        "x_range",
        "y_range",
        watch=True,
    )
    def update_plot(self):
        x, y = self.sine()
        self.cds.data = dict(x=x, y=y)
        self.plot.x_range.start, self.plot.x_range.end = self.x_range
        self.plot.y_range.start, self.plot.y_range.end = self.y_range

    def sine(self):
        x = np.linspace(0, 4 * np.pi, self.N)
        y = (
            self.amplitude * np.sin(self.frequency * x + self.phase)
            + self.offset
        )
        return x, y
```

fastAPI/sliders/pn_app.py
erstellt eine App-Funktion aus der `SineWave`-Klasse:

Quellcode 4: fastAPI/sliders/pn_app.py

```
import panel as pn

from .sinewave import SineWave


def createApp():
    sw = SineWave()
    return pn.Row(sw.param, sw.plot).servable()
```

7. Schließlich kehren wir zu unserer `fastAPI/main.py` zurück und importieren die `createApp`-Funktion:

Quellcode 5: fastAPI/main.py

```
3 from sliders.pn_app import createApp
```

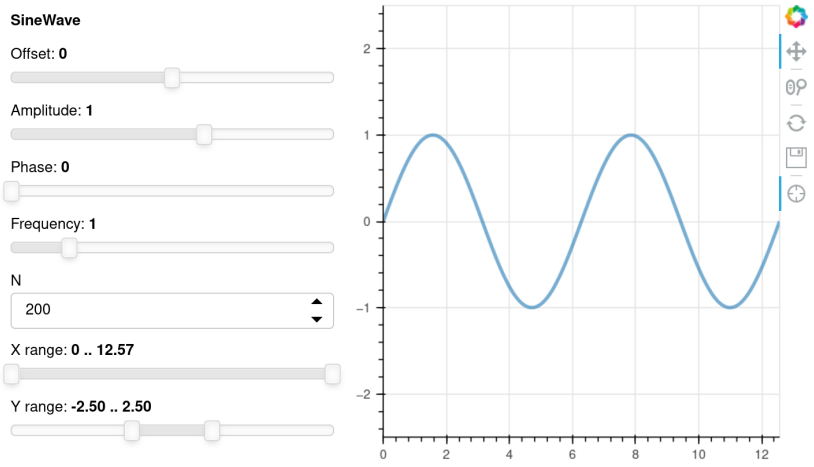
Die Dateistruktur sollte nun folgendermaßen aussehen:

```
fastAPI
├── main.py
├── sliders
│   ├── pn_app.py
│   └── sinewave.py
└── templates
    └── base.html
```

Ihr könnt den Server nun starten mit:

```
$ uv run uvicorn main:app --reload
INFO:     Will watch for changes in these directories: ['/srv/jupyter/jupyter-tutorial/
↪docs/web/dashboards/panel/fastAPI']
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [218214] using StatReload
Launching server at http://127.0.0.1:5000
INFO:     Started server process [218216]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

Anschließend solltet ihr im Web-Browser unter der URL `http://127.0.0.1:8000` folgendes sehen:


SineWave
Offset: 0
Amplitude: 1
Phase: 0
Frequency: 1
N
200
X range: 0 .. 12.57
Y range: -2.50 .. 2.50
2
1
0
−1
−2
0
2
4
6
8
10
12

KAPITEL 14

Sphinx

Sphinx ist ein Dokumentationswerkzeug, das python-basics:document/rest in HTML oder PDF, EPub und Man-Pages umwandelt. Auch das Jupyter-Tutorial wird mit Sphinx erstellt.

Ursprünglich wurde Sphinx für die Dokumentation von Python entwickelt. Heute wird es in fast allen Python-Projekten verwendet, darunter NumPy and SciPy, Matplotlib, Pandas und SQLAlchemy.

Mit *nbsphinx* können auch Jupyter Notebooks in Sphinx integriert werden. *Executable Books* ist hingegen eine Sammlung von Open-Source-Tools, die euch U.A. ermöglichen, Markdown und Jupyter Notebooks zu schreiben, Inhalte auszuführen und in euer Buch einzufügen.

Tipp

cusy Seminare:

- Software-Dokumentation mit Sphinx
- Technisches Schreiben

14.1 nbsphinx

nbsphinx ist eine Sphinx-Erweiterung, die einen Parser für `*.ipynb`-Dateien bereitstellt: Jupyter Notebook-Code-Zellen werden sowohl in der HTML- wie auch in der LaTeX-Ausgabe angezeigt. Notebooks ohne gespeicherte Ausgabezellen werden automatisch während des Sphinx-Build-Prozesses erstellt.

14.1.1 Installation

```
$ uv add sphinx nbsphinx
```

Requirements

- *nbconvert*

14.1.2 Konfiguration

Sphinx konfigurieren

1. Erstellen einer Dokumentation mit Sphinx:

```
$ mkdir docs
$ cd docs
$ uv run python -m sphinx.cmd.quickstart
```

2. Danach befindet sich in `docs` die Sphinx-Konfigurationsdatei `conf.py`. In dieser wird `nbsphinx` als Erweiterung hinzugefügt und automatisch generierte Notebooks ausgeschlossen:

```
extensions = [
    "...",
    "nbsphinx",
]
...
exclude_patterns = [
    "...",
    "**/.ipynb_checkpoints",
]
```

   Ein Beispiel findet ihr in der `/conf.py`-Datei dieses Jupyter-Tutorials.

Ihr könnt noch weitere Konfigurationen für `nbsphinx` vornehmen.

Timeout
: In der Standardeinstellung von `nbsphinx` ist der Timeout für eine Zelle auf 30 Sekunden eingestellt. Ihr könnt dies für euer Sphinx-Projekt in der `conf.py`-Datei ändern mit `nbsphinx_timeout = 60`.

  Alternativ könnt ihr dies auch für einzelne Code-Zellen in den Metadaten der Code-Zelle angeben:

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "nbsphinx": {
     "timeout": 60
   },
  }
 ],
}
```

  Soll das Timeout deaktiviert werden, kann `-1` angegeben werden.

Benutzerdefinierte Formate
: Bibliotheken wie z.B. jupytext speichern Notebooks in anderen Formaten ab, z.B. als *R-Markdown* mit dem Suffix `Rmd`. Damit diese von `nbsphinx` ebenfalls ausgeführt werden können, können in der Sphinx-Konfigurationsdatei `conf.py` mit `nbsphinx_custom_formats` weitere Formate angegeben werden, z.B.

```
import jupytext
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
nbsphinx_custom_formats = {
    ".Rmd": lambda s: jupytext.reads(s, ".Rmd"),
}
```

Zellen konfigurieren

Zelle nicht anzeigen

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
    "nbsphinx": "hidden"
   },
  }
 ],
}
```

nbsphinx-toctree

Mit dieser Anweisung könnt ihr innerhalb einer Notebook-Zelle von Sphinx ein Inhaltsverzeichnis erstellen lassen, z.B.

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
    "nbsphinx-toctree": {
      "maxdepth": 2
    }
   "source": [
    "Der folgende Titel wird als ``toctree caption`` gerendert.\n",
    "\n",
    "## Inhalt\n",
    "\n",
    "[Ein Notebook](ein-notebook.ipynb)\n",
    "\n",
    "[Ein externer HTML-Link](https://jupyter-tutorial.readthedocs.io/)\n",
   ]
   },
  }
 ],
}
```

Weitere Optionen findet ihr in der Sphinx-Dokumentation.

14.1.3 Build

1. Nun könnt ihr im Inhaltsverzeichnis eurer `index.rst`-Datei eure `*.ipynb`-Datei hinzufügen, siehe z.B. jupyter-tutorial/notebook/testing/index.rst.
2. Schließlich könnt ihr die Seiten generieren, z.B. HTML mit `$ uv run python -m sphinx` *SOURCE_DIR BUILD_DIR* oder `$ uv run python -m sphinx` *SOURCE_DIR BUILD_DIR* `-j` *NUMBER_OF_PROCESSES*,

wobei `-j` die Zahl der Prozesse angibt, die parallel ausgeführt werden sollen.

Wenn ihr eine LaTeX-Datei erzeugen wollt, könnt ihr dies mit `$ uv run python -m sphinx SOURCE_DIR BUILD_DIR -b latex}`.

3. Alternativ könnt ihr euch mit `sphinx-autobuild` die Dokumentation auch automatisch generieren lassen. Es kann installiert werden mit

```
$ uv add sphinx-autobuild
```

Anschließend kann die automatische Erstellung gestartet werden mit `$ uv run python -m sphinx_autobuild SOURCE_DIR BUILD_DIR`.

Dadurch wird ein lokaler Webserver gestartet, der die generierten HTML-Seiten unter `http://localhost:8000/` bereitstellt. Und jedes Mal, wenn ihr Änderungen in der Sphinx-Dokumentation speichert, werden die entsprechenden HTML-Seiten neu generiert und die Browseransicht aktualisiert.

Ihr könnt dies auch nutzen, um die LaTeX-Ausgabe automatisch zu erstellen: `$ uv run python -m sphinx_autobuild SOURCE_DIR BUILD_DIR -b latex`.

4. Eine andere Alternative ist die Publikation auf readthedocs.org.

Hierfür müsst ihr zunächst ein Konto unter https://about.readthedocs.com erstellen und dann euer GitLab-, Github- oder Bitbucket-Konto verbinden.

Markdown-Zellen

Gleichungen

Gleichungen können *inline* zwischen $-Zeichen angegeben werden, z.B.

```
$\text{e}^{i\pi} = -1$
```

Und auch zeilenweise können Gleichungen ausgedrückt werden z.B.

```
\begin{equation}
\int\limits_{-\infty}^\infty f(x) \delta(x - x_0) dx = f(x_0)
\end{equation}
```

Siehe auch

- Equation Numbering

Zitate

`nbsphinx` unterstützt dieselbe Syntax für Zitate wie nbconvert:

```
<cite data-cite="kluyver2016jupyter">Kluyver et al. (2016)</cite>
```

Alarmierungsboxen

```
<div class="alert alert-block alert-info">

**Note**

This is a notice!
</div>
```

(Fortsetzung auf der nächsten Seite)

(Fortsetzung der vorherigen Seite)

```
<div class="alert alert-block alert-success">

**Success**

This is a success notice!
</div>

<div class="alert alert-block alert-warning">

**Warning**

This is a warning!
</div>

<div class="alert alert-block alert-danger">

**Danger**

This is a danger notice!
```

Links zu anderen Notebooks

```
[a link to a notebook in a subdirectory](subdir/notebook-in-a-subdir.ipynb)
```

Links zu `*.rst`-Dateien

```
[reStructuredText file](rst-file.rst)
```

Links zu lokalen Dateien

```
[pyproject.toml](pyproject.toml)
```

Code-Zellen

Javascript

Für das generierte HTML kann Javascript verwendet werden, z.B.:

```
%%javascript

var text = document.createTextNode("Hello, I was generated with JavaScript!");
// Content appended to "element" will be visible in the output area:
element.appendChild(text);
```

14.1.4 Galerien

nbsphinx bietet Unterstützung für die Erstellung von Thumbnail-Galerien aus einer Liste von Jupyter-Notebooks. Diese Funktionalität basiert auf Sphinx-Gallery und erweitert diese, um mit Jupyter-Notebooks statt mit Python-Skripten zu arbeiten.

Sphinx-Gallery unterstützt auch direkt Matplotlib, seaborn und Mayavi.

Installation

Sphinx-Gallery lässt sich für Sphinx 1.8.3 installieren mit

```
$ uv add sphinx-gallery
```

Konfiguration

Damit Sphinx-Gallery genutzt werden kann, muss sie zudem noch in die `conf.py` eingetragen werden:

```
extensions = [
    "nbsphinx",
    "sphinx_gallery.load_style",
]
```

Anschließend könnt ihr Sphinx-Gallery auf zwei verschiedene Arten nutzen:

1. Mit der reStructuredText-Direktive `.. nbgallery::`.

   Siehe auch

   Thumbnail Galleries

2. In einem Jupyter-Notizbuch, indem ein `nbsphinx-gallery`-Tag zu den Metadaten einer Zelle hinzugefügt wird:

   ```
   {
       "tags": [
           "nbsphinx-gallery"
       ]
   }
   ```

14.2 Executable Books

Executable Books ist eine Sammlung von Open-Source-Tools, die die Veröffentlichung von computergestützten Narrativen mithilfe des Jupyter-Ökosystems erleichtern, vor allem:

Jupyter Book
: *Sphinx*-Distribution, die euch ermöglicht, Inhalte in Markdown und Jupyter Notebooks zu schreiben, Inhalte auszuführen und in euer Buch einzufügen.

Siehe auch

- jupyterbook.org ist die Landing Page des Projekts.
- gallery.jupyterbook.org ist eine Galerie von Jupyter-Books.
- github.com/executablebooks/jupyter-book ist das Repository des Projekts.

MyST
: ist eine erweiterbare, semantische Variante von Markdown, die für wissenschaftliche und computergestützte Narrative entwickelt wurde. MyST-Markdown ist eine sprach- und implementierungsunabhängige Variante von Markdown, die von verschiedenen Tools unterstützt wird.

Siehe auch

- mystmd.org ist die Landing Page des Projekts.
- spec.mystmd.org beschreibt die MyST-Spezifikation.
- MyST Enhancement Proposals ist ein Prozess, um Änderungen an der MyST-Spezifikation vorzuschlagen und darüber zu entscheiden.

JupyterLab MyST Extension
rendert in *JupyterLab* Markdown-Zellen mit MyST-Markdown, einschließlich interaktiver Referenzen, Hinweisen, Nummerierung von Abbildungen, Tabs, Cards und Grids.

Siehe auch

- github.com/jupyter-book/jupyterlab-myst

KAPITEL 15

Anwendungsbeispiele

In einigen Unternehmen werden Jupyter Notebooks genutzt, um explorativ die immer größer werdenden Datenmengen zu erschließen. Hierzu gehören:

- Netflix
  - Beyond Interactive: Notebook Innovation at Netflix
  - Part 2: Scheduling Notebooks at Netflix
- Bloomberg BQuant platform
  - Bloomberg BQuant (BQNT)
- PayPal
  - PayPal Notebooks: Data science and machine learning at scale, powered by Jupyter
- Société Générale
  - Jupyter & Python in the corporate LAN

KAPITEL 16

Stichwortverzeichnis

Stichwortverzeichnis

J

JUPYTER_CONFIG_DIR, 14

N

Notebook-Kernel, **11**
Notebook-Zelle, **11**

T

Test Case (*Testfall*), **18**
Test Fixture (*Prüfvorrichtung*), **18**
Test Runner, **18**
Test Suite, **18**

U

Umgebungsvariable
    JUPYTER_CONFIG_DIR, 14