Jupyter Tutorial

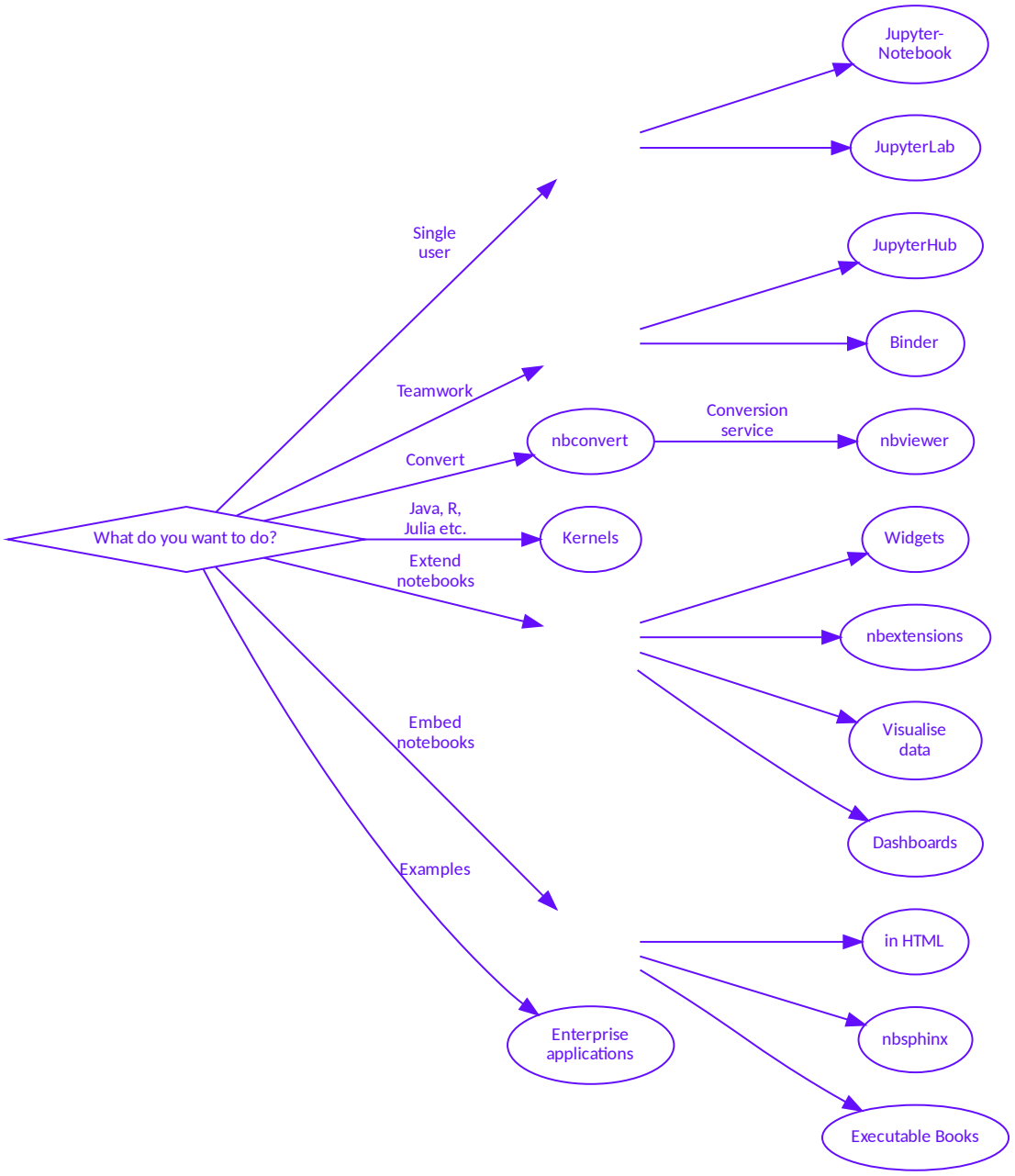
Release 24.1.0

Veit Schiele

Apr 16, 2024

CONTENTS

Jupyter notebooks are growing in popularity with data scientists and have become the de facto standard for rapid prototyping and exploratory analysis. They inspire experiments and innovations enormously and as well they make the entire research process faster and more reliable. In addition, many additional components are created that expand the original limits of their use and enable new uses.



CHAPTER

ONE

INTRODUCTION

1.1 Target group

The users of Jupyter notebooks are diverse, from data scientists to data engineers and analysts to system engineers. Their skills and workflows are very different. However, one of the great strengths of Jupyter notebooks is that they allow these different experts to work closely together in cross-functional teams.

Data scientists
: explore data with different parameters and summarise the results.

Data engineers
: check the quality of the code and make it more robust, efficient and scalable.

Data analysts
: use the code provided by data engineers to systematically analyse the data.

System engineers
: provide the research platform based on the *JupyterHub* on which the other roles can perform their work.

In this tutorial we address system engineers who want to build and run a platform based on Jupyter notebooks. We then explain how this platform can be used effectively by data scientists, data engineers and analysts.

1.2 Why Jupyter?

How can these diverse tasks be simplified? You will hardly find a tool that covers all of these tasks, and several tools are often required even for individual tasks. Therefore, on a more abstract level, we are looking for more general patterns for tools and languages with which data can be analysed and visualised and a project can be documented and presented. This is exactly what we are aiming for with Project Jupyter.

The Jupyter project started in 2014 with the aim of creating a consistent set of open source tools for scientific research, reproducible workflows, computational narratives and data analysis. In 2017, Jupyter received the ACM Software Systems Award – a prestigious award which, among other things, shares with Unix and the web.

To understand why Jupyter notebooks are so successful, let's take a closer look at the core functions:

Jupyter Notebook Format
: Jupyter Notebooks are an open, JSON-based document format with full records of the user's sessions and the code they contain.

Interactive Computing Protocol
: The notebook communicates with the computing kernel via the *Interactive Computing Protocol*, an open network protocol based on JSON data via ZMQ and WebSockets.

Kernels
: Kernels are processes that execute interactive code in a specific programming language and return the output to the user.

See also:

- Jupyter celebrates 20 years

1.3 Jupyter infrastructure

A platform for the above-mentioned use cases requires an extensive infrastructure that not only allows the provision of the kernel and the parameterisation, time control and parallelisation of notebooks, but also the uniform provision of resources.

This tutorial provides a platform that enables fast, flexible and comprehensive data analysis beyond Jupyter notebooks. At the moment, however, we are not yet going into how it can be expanded to include streaming pipelines and domain-driven data stores.

However, you can also create and run the examples in the Jupyter tutorial locally.

1.4 Workspace

Setting up the workspace includes installing and configuring IPython and *Jupyter notebooks*, *nbextensions* and *ipywidgets*.

CHAPTER

TWO

WHAT'S NEW?

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

1.1.0

- Jupyter-Tutorial 1.1.0
- Fix PDF structure
- 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/

CHAPTER

THREE

NOTEBOOK

Jupyter Notebooks extend the console-based approach to interactive computing with a web-based application, with which the entire process can be recorded: from developing and executing the code to documenting and presenting the results.

3.1 Install Jupyter Notebook

3.1.1 Create a virtual environment with `jupyter`

Python virtual environments allow Python packages to be installed in an isolated location for a specific application, rather than installing them globally. So you have your own installation directories and do not share libraries with other virtual environments:

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

3.1.2 Start `jupyter notebook`

```
$ 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
```

Your standard web browser will then open with this URL.

When the notebook opens in your browser, the notebook dashboard is displayed with a list of the notebooks, files and subdirectories in the directory in which the notebook server was started. In most cases you want to start a notebook server in your project directory.

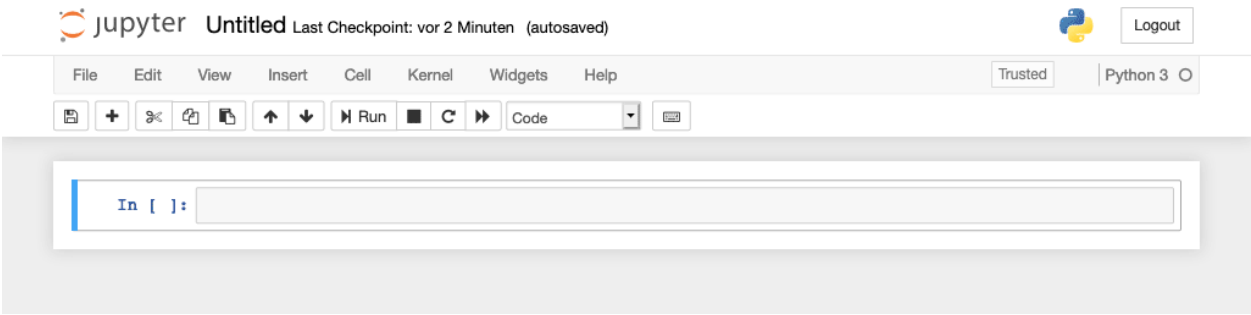
3.2 Create notebook

After the notebook server has started, we can create our first notebook.

3.2.1 Create a notebook

In your standard browser you should see the notebook dashboard with the *New* menu on the right. All notebook kernels are listed in this menu, but initially probably only *Python 3*.

After you have selected *New* → *Python 3*, a new notebook `Untitled.ipynb` will be created and displayed in a new tab:



3.2.2 Renaming the notebook

Next you should rename this notebook by clicking on the title *Untitled*:

3.2.3 The notebook user interface

There are two important terms used to describe Jupyter Notebooks: *cell* and *kernel*:

Notebook kernel
Computational engine that executes the code contained in a notebook.

Notebook cell
Container for text to be displayed in a notebook or for code to be executed by the notebook's kernel.

Code
contains code to be executed in the kernel, and the output which is shown below.

In front of the code cells are brackets that indicate the order in which the code was executed.

In [ ]:
indicates that the code has not yet been executed.

In [*]:
indicates that the execution has not yet been completed.

> **Warning:** The output of cells can be used in other cells later. Therefore, the result depends on the order. If you choose a different order than the one from top to bottom, you may get different results later when you e.g. select *Cell → Run All*.

Markdown
contains text formatted with Markdown, which is interpreted as soon as *Run* is pressed.

3.2.4 What's an `ipynb` file?

This file describes a notebook in JSON format. Each cell and its contents including pictures are listed there along with some metadata. You can have a look at them if you select the notebook in the dashboard and then click on *edit*. For example the JSON file for `my-first-notebook.ipynb` looks like this:

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

(continues on next page)

(continued from previous page)


```
    }
   ],
   "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 Save and checkpoints

When you click on *Save and Checkpoint*, your `*.ipynb` file will be saved. But what is the checkpoint all about?

Every time you create a new notebook, a file is also created, which usually automatically saves your changes every 120 seconds. This checkpoint is usually located in a hidden directory called `.ipynb_checkpoints/`. This checkpoint file therefore enables you to restore your unsaved data in the event of an unexpected problem. You can go back to one of the last checkpoints in *File* → *Revert to Checkpoint*.

3.2.6 Tips and tricks

1. Give the notebook a title (*`# MY TITLE`*) and a meaningful foreword to describe the content and purpose of the notebook.
2. Create headings and documentation in Markdown cells to structure your notebook and explain your workflow steps. It doesn't matter whether you do this for your colleagues or for yourself in the future.
3. Use *Table of Contents (2)* from the *List of extensions* to create a table of contents.
4. Use the notebook extension *setup*.
5. Use snippets from the list of extensions to add more frequently used code blocks, for example typical import instructions, easy to insert.

3.3 Keyboard shortcuts

If you know the Jupyter keyboard shortcuts, you can work much more efficiently with notebooks. Jupyter notebooks have two different keyboard input modes:

- In **edit mode** you can enter code or text in a cell. This is indicated by a green cell border.
- **Command mode** binds the keyboard to notebook-level commands and is indicated by a gray cell border with a blue left border.

Command mode	
This mode is available with .	
f	find and replace
	enter edit mode
--f, --p, p	open the command palette
-	run cell, select below
-, -	run selected cells
-	run cell and insert below
y	change cell to code
m	change cell to markdown
r	change cell to raw
1, 2 etc.	change cell to heading 1, heading 2, etc.
k, ↑	select cell above
j, ↓	select cell below
-k, -↑	extend selected cells above
-j, -↓	extend selected cells below
-a	select all cells
a	insert cell above
b	insert cell below
x	cut selected cells
c	copy selected cells
-v	paste cells above
v	paste cells below
z	undo cell deletion
d d	delete selected cells
-m	merge selected cells, or current cell with cell below if only one cell is select
-s, s	save and checkpoint
l	toggle line numbers
o	toggle output of selected cells
-o	toggle output scrolling of selected cells
h	show keyboard shortcuts
i i	interrupt the kernel
0 0	restart the kernel (with dialog)
-v	dialog for paste from system clipboard
, q	close the pager

Edit mode	
This mode becomes available with .	
	code completion or indent
-	tooltip

continues on next page

Table 2 – continued from previous page

Edit mode	
-]	indent
-[	dedent
-a	select all
-z	undo
-/	comment
-d	delete whole line
-u	undo selection
	toggle overwrite flag
-↑	go to cell start
-↓	go to cell end
-←	go one word left
-→	go one word right
-	delete word before
-	delete word after
--z	redo
--u	redo selection
-k	emacs-style line kill
-	delete line left of cursor
-	delete line right of cursor
-m,	enter command mode
--f, --p	open the command palette
-	run cell, select below
-	run selected cells
-	run selected cells
-	run cell and insert below
---	split cell at cursor(s)
-s	save and checkpoint
↓	move cursor down
↑	move cursor up

3.3.1 Own keyboard shortcuts

You can also define your own keyboard shortcuts in *Help → Edit Keyboard Shortcuts*.

See also:

- Keyboard Shortcut Customization

3.4 Jupyter paths and configuration

Configuration files are usually stored in the `~/.jupyter` directory. However, another directory can be specified with the environment variable `JUPYTER_CONFIG_DIR`. If Jupyter cannot find a configuration in `JUPYTER_CONFIG_DIR`, Jupyter runs through the search path with `/SYS.PREFIX/etc/jupyter/` and then for Unix `/usr/local/etc/jupyter/` and `/etc/jupyter/`, for Windows `%PROGRAMDATA%\jupyter\`.

You can have the currently used configuration directories listed with:

```
$ jupyter --paths
config:
```

(continues on next page)

(continued from previous page)

```
    /Users/veit/.jupyter
    /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/bin/../etc/jupyter
    /usr/local/etc/jupyter
    /etc/jupyter
...
```

3.4.1 Create the configuration files

You can create a standard configuration with:

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

More generally, configuration files can be created for all Jupyter applications with `jupyter` *`APPLICATION`* `--generate-config`.

3.4.2 Change the configuration

... by editing the configuration file

e.g. in `jupyter_notebook_config.py`:

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

If the values are saved as `list`, `dict` or `set`, they can also be supplemented with `append`, `extend`, `prepend`, `add` and `update`, e.g.:

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

... with the command line

for example:

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

There are aliases for frequently used options such as for `--port` or `--no-browser`.

The command line options override options set in a configuration file.

See also:

traitlets.config

3.5 Parameterisation and scheduling

With *JupyterLab* you can use the *Jupyter Scheduler* for parameterisation and time-controlled execution. For Jupyter Notebooks, papermill is available.

3.5.1 Install

```
$ pipenv install papermill
Installing papermill...
Adding papermill to Pipfile's [packages]...
✓ Installation Succeeded
...
```

3.5.2 Use

1. Parameterise

   The first step is to parameterise the notebook. For this purpose the cells are tagged as *parameters* in *View* → *Cell Toolbar* → *Tags*.

2. Inspect

   You can inspect a notebook for example with:

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

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

3. Execute

   There are two ways to run a notebook with parameters:

   - … via the Python API

     The `execute_notebook` function can be called to execute a notebook with a dict of parameters:

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

     for example for `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"),
             )
     ```

     The result is `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,
            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
```

See also:

- Workflow reference

- … via CLI

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

Alternatively, a YAML file can be specified with the parameters, for example `params.yaml`:

Listing 1: params.yaml

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

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

With `-b`, a base64-encoded YAML string can be provided, containing the parameter values:

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

See also:

- CLI reference

You can also add a timestamp to the file name:

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

This creates an output file whose file name contains a timestamp, for example `output_2023-06-26_15:57:33.ipynb`.

Finally, you can use `crontab -e` to execute the two commands automatically at certain times, for example on the first day of every month:

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

4. Store

   Papermill can store notebooks in a number of locations including S3, Azure data blobs, and Azure data lakes. Papermill allows new data stores to be added over time.

   See also:

   - papermill Storage
   - Extending papermill through entry points

3.6 Testing

3.6.1 Concepts

Test Case
tests a single scenario.

See also:

- pytest fixtures

Test Fixture
is a consistent test environment.

Test Suite
is a collection of several test cases.

Test Runner
runs through a test suite and presents the results.

3.6.2 Notebooks

Unit tests

[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)


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 "/tmp/ipykernel_8759/2216555184.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 0x1065794d0>
```

Alternatively, ipython-unittest can also be used. This enables the following *Cell Magics* to be used in iPython:

- `%%unittest_main` executes test cases that are defined in a cell
- `%%unittest_testcase` creates a test case with the function defined in a cell and executes it
- `%%unittest` converts Python `assert` to unit test functions
- `%%external` to perform external unit tests
- `%%write {mode}` to write external files

[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)
```

(continues on next page)

(continued from previous page)

```
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>
```

By default, Docstring separates the unit test methods in this magic. However, if docstrings are not used, the *Cell Magics* create one for each `assert` method.

These *Cell Magics* support optional arguments:

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

  puts the cursor in front of P cells (default `-1` corresponds to the next cell)

  However, this only works if jupyter_dojo is also installed.

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

  sets the *ooutput stream* (default is: `sys.stdout`)

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

  defines the name of the TestCase for `%%unittest` and `%%unittest_testcase`

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

  outputs the source code after the transformations

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]:
```
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` imports the Python debugger
2. `pdb.set_trace()` creates a breakpoint that starts the Python debugger.

See also:

- pdb – The Python Debugger

Mock

Mock objects promote tests based on the behaviour of objects. The Python library mock allows you to replace parts of the system under test with mock objects and make statements about their use.

Installation

mock is included in the Python standard library since Python 3.3. For older versions of Python you can install it with:

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

Example

In our example, we want to check whether the working days from Monday to Friday are determined correctly.

1. First we import `datetime` and `Mock`:

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

2. Then we define two test days:

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

3. Now we define a method to check the working days, where Python's datetime library treats Mondays as `0` and Sundays as 6:

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

4. Then we mock `datetime`:

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

5. Finally, we test our two mock objects:

[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()
```

(continues on next page)

(continued from previous page)

```
AssertionError:
```

See also:

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

mock.ANY

With mock.ANY you can check whether a value is present at all without having to check an exact value:

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


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

See also:

In test_report.py of the OpenStack container service Zun you will find more practical examples for `ANY`.

`patch` decorator

To create mock classes or objects, the `patch` decorator can be used. In the following examples, the output of `os.listdir` is mocked. For this, the file `example.txt` does not have to be present in the directory:

[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()
```

Alternatively, the return value can also be defined separately:

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


test_listdir()
```

See also:

You can use responses to create mock objects for the Requests library.

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),
    ]
```

(continues on next page)

(continued from previous page)

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

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

Test parameterisation

[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 0.00s
```

Reference

%%run_pytest ...

IPython magic that executes first the cell and then `run_pytest`. Arguments passed in the cell are passed directly to pytest. The Magics should have been imported with `import ipytest.magics` beforehand.

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

runs the tests in the existing module (by default `main`) with pytest.

Arguments:

- `module`: the module that contains the tests. If not specified, `__main__` is used.
- `filename`: Filename of the file containing the tests. If nothing is specified, the `__file__`attribute of the passed module is used.
- `pytest_options`: additional options passed to pytest.
- `pytest_plugins`: additional pytest plugins.

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

Arguments:

- `doctest`: If `True` is specified, angegeben wird, then doctests are searched for.
- `items`: The *globals* object that contains the tests. If `None` is specified, the *globals* object is obtained from the call stack.

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

deletes those tests whose names match the specified pattern.

In IPython, the results of all evaluations are saved in global variables, unless they are explicitly deleted. This behavior implies that if tests are renamed, the previous definitions will still be found if they are not deleted. This method aims to simplify this process.

An effective method is `clean_tests` to start with a cell, then define all test cases and finally `run_tests` call them. That way, renaming tests works as expected.

Arguments:

- `pattern`: A glob pattern that is used to find the tests to delete.
- `items`: The *globals* object that contains the tests. If `None` is specified, the globals object is obtained from the call stack.

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

collects all test cases and sends them to `unittest.TestSuite`.

The arguments are the same as for `ipytest.run_tests`.

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

compares two objects and throws an exception if they are not the same.

The method `ipytest.get_assert_function` determines the assert implementation to be used depending on the following arguments:

- `a, b`: the two objects to be compared.
- `args, kwargs`: (Keyword) arguments that are passed to the underlying test function.

ipytest.get_assert_function(a, b)

determines the assert function to be used depending on the arguments.

If one of the objects is `numpy.ndarray`, `pandas.Series`, `pandas.DataFrame` or `pandas.Panel` the assert functions provided by `numpy` and `pandas` will be returned.

`ipytest.unittest_assert_equals(a, b)`

compares two objects using the `assertEqual` method of `unittest.TestCase`.

Hypothesis

Hypothesis is a library that allows you to write tests that are parameterised from a source of examples. Then simple and comprehensible examples are generated, which can be used to fail your tests and to find errors with little effort.

Example

To test lists with floating point numbers, many examples are tried, but only a simple example is given in the report for each bug (unique exception type and position):

[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):

/tmp/ipykernel_8817/1742712940.py:2:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
↪ - - - - -

ls = [9.9792015476736e+291, 1.7976931348623157e+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)
```

(continues on next page)

(continued from previous page)

```
E       assert inf <= 1.7976931348623157e+308
E        +  where 1.7976931348623157e+308 = max([9.9792015476736e+291, 1.
↪7976931348623157e+308])
E       Falsifying example: test_mean(
E           ls=[9.9792015476736e+291, 1.7976931348623157e+308],
E       )

/tmp/ipykernel_8817/1742712940.py:4: AssertionError
=========================================== warnings summary␣
↪=========================================
../../../../../../.local/share/virtualenvs/python-311-6zxVKbDJ/lib/python3.11/site-
↪packages/_pytest/config/__init__.py:1204
  /Users/veit/.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
====================================== short test summary info␣
↪======================================
FAILED t_97777e739d3141398e41c86d782e924f.py::test_mean - assert inf <= 1.
↪7976931348623157e+308
1 failed, 1 warning in 0.63s
```

Installation

```
$ pipenv install hypothesis
```

Alternatively, Hypothesis can also be installed with extras, for example

```
$ pipenv install hypothesis"[numpy, pandas]"
```

Note:

If you haven't installed pipenv yet, you can find instructions on how to do this in Install pipenv.

See also:

- Hypothesis for the Scientific Stack

CHAPTER

FOUR

JUPYTERLAB

JupyterLab is an extensible, feature-rich editor for creating and editing *Jupyter Notebooks*:

- You can arrange multiple documents and activities side by side in your workspace using tabs.
- Code consoles provide temporary scratchpads for running code interactively, which can also be linked to a *notebook kernel*.
- There is also preview of CSV and Vega files.
- *JupyterLab extensions* can customise or enhance any part of JupyterLab.

JupyterLab currently uses the same Notebook document format as the classic *Jupyter Notebook*. Notebook 7 will replace the classic Jupyter Notebook format.

See also:

Migrating to Notebook 7

4.1 Install JupyterLab

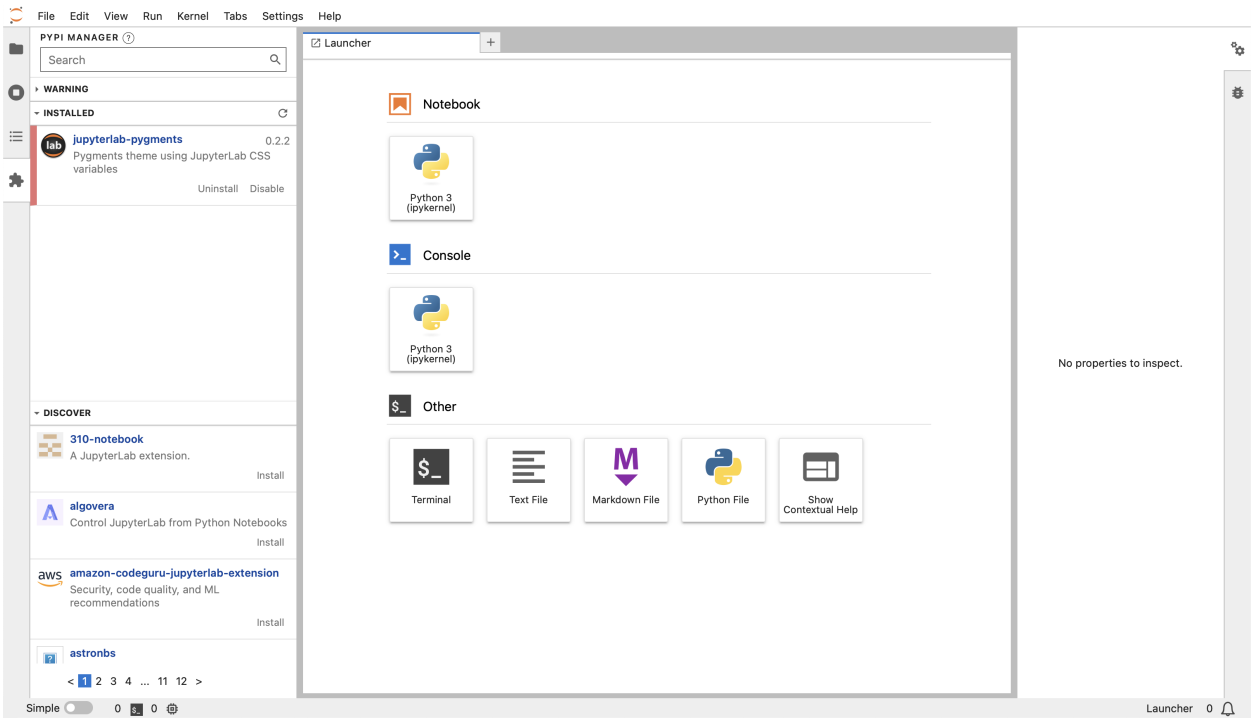
4.1.1 Creating a virtual environment with JupyterLab

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

4.1.2 Start JupyterLab

```
$ 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
```

Your default web browser will then open with this URL.



4.1.3 Localisation

Since version 3.0, JupyterLab offers the possibility to set the display language of the interface. To do this, the appropriate language packages must be installed, for example:

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

In the language-packs repository you will find a list of the available language packs.

Then you can select the newly installed language in *Settings* → *Language*.

4.2 JupyterLab extensions

JupyterLab is designed as an extensible environment. In doing so, JupyterLab extensions can customise any part of JupyterLab. They can provide new themes, file viewers and editors, or renderers for rich outputs in *Notebook*.

See also:

JupyterLab Extensions by Examples

4.2.1 Installing extensions

A JupyterLab extension contains JavaScript that is installed in JupyterLab and executed in the browser. Most JupyterLab extensions can be installed with pip. These packages can also contain server-side components that are required for the extension to function.

Since JupyterLab 4, the default extension manager uses PyPI as a source for the available extensions and pip to install them. An extension is listed if the Python package has the trove classifier `Framework :: Jupyter :: JupyterLab :: Extensions :: Prebuilt`.

Warning: It does not check if the extension is compatible with the current JupyterLab version.

Danger: Installing an extension allows arbitrary code to be executed on the server, kernel and browser. Therefore, avoid installing extensions that you do not trust.

4.2.2 Configuring the Extension Manager

By default, there are two extension managers provided by JupyterLab:

pypi
: Default setting that allows the installation from pypi.org.

readonly
: shows the installed extensions with the possibility to disable or enable them.

You can specify the manager with the command line option `--LabApp.extension_manager`, for example `jupyter lab --LabApp.extension_manager=readonly`.

When searching for extensions in the extension manager, JupyterLab usually shows all search results and any source extension can be installed. However, to increase security, JupyterLab can be configured so that extensions can only be activated using the block or allow lists.

You can define the loading of the lists with `blocked_extensions_uris` or `allowed_extensions_uris`, which contain a list of comma-separated URIs, for example `--LabServerApp.blocked_extensions_uris=http://example.com/blocklist.json` with the following `blocklist.json` file:

```
{
  "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"
    }
  ]
}
```

Another example shows an `allowlist.json` file that allows all extensions of the JupyterLab organisation:

```
{
  "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 on JupyterHub

JupyterLab works by default with *JupyterHub* 1.0 and can even run alongside classic *notebooks*.

When JupyterLab is used with JupyterHub, additional menu items are displayed in the *File* menu to *Log Out* or to call up the *Hub Control Panel*.

If JupyterLab is not yet the default, you can change the configuration in `jupyterhub_config.py`:

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

4.4 Real-time collaboration

From JupyterLab 4 it is possible to activate Real-Time Collaboration by installing the extension `jupyter_collaboration`. This enables real-time collaboration between multiple clients. In addition, you can see the cursors of others.

See also:

- jupyter_collaboration documentation

4.4.1 Installation

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

To share a document with others, you can copy the URL and send it, or you can additionally install jupyterlab-link-share, which allows you to share the link including the token.

4.5 Scheduler

Jupyter Scheduler is a collection of extensions for programming jobs to run immediately or on a schedule. It has a Lab (client) and a Server extension. Both are needed to schedule and run notebooks. If you install Jupyter Scheduler via the JupyterLab extension manager, you may only install the client extension and not the server extension. Therefore, install Jupyter Scheduler with pip:

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

This will automatically activate the lab and server extensions. You can check this with

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

and

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

1. To create a jog from an open notebook, click the *Create a notebook job* button in the top toolbar of the open notebook.
2. Give your notebook job a name, select the output formats and specify parameters that will be set as local variables when your notebook is executed. This parameterised execution is similar to that used in *Papermill*.
3. To create a job that will run once, select *Run now* and click *Create*.
4. To create a job definition that will run repeatedly on a schedule, select *Run on a schedule*.

CHAPTER

FIVE

JUPYTERHUB

JupyterHub is a multi-user server for Jupyter Notebooks, which can create and manage many different instances of Jupyter Notebooks and which acts as a proxy.

5.1 Installation

1. Install Python3.6 and pip:

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

2. Create service user `jupyter`:

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

3. Switch to the service user `jupyter`:

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

4. Install Pipenv:

```
$  python3 -m pip install --user pipenv
```

   This installs Pipenv in `USER_BASE`.

5. Determine `USER_BASE` and enter it in `PATH`:

```
$  python3 -m site --user-base
/srv/jupyter/.local
```

   Then the `bin` directory must be appended and added to `PATH` in `~/.profile`, so:

```
export PATH=/srv/jupyter/.local/bin:$PATH
```

   Finally, the changed profile is read in with:

```
$  source ~/.profile
```

6. Create a virtual environment and install JupyterHub:

```
$ mkdir jupyterhub_env
$ cd jupyterhub_env
$ pipenv install jupyterhub
```

7. Install `nodejs` and `npm`:

```
$ sudo apt install nodejs npm
$ node -v
v12.22.9
$ npm -v
8.5.1
```

8. Install the HTTP proxy:

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

9. If JupyterLab and Notebook are to run in the same environment, they must also be installed here:

```
$  pipenv install jupyterlab notebook
```

10. Testing the installation:

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

11. Starting the JupyterHub:

```
$  pipenv run jupyterhub
...
[I 2019-07-31 22:47:26.617 JupyterHub app:1912] JupyterHub is now running at http://
↪:8000
```

You can end the process again with `ctrl-c`.

5.2 Configuration

5.2.1 JupyterHub configuration

Create configuration file:

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

See also:

- JupyterHub Configuration Basics
- JupyterHub Networking basics

5.2.2 System Service for JupyterHub

1. Determine the Python virtual environment:

```
$ cd ~/jupyter-tutorial
$ pipenv --venv
/srv/jupyter/.local/share/virtualenvs/jupyter-tutorial-aFv4x91W
```

2. Configure the absolute path to `jupyterhub-singleuser` in the `jupyterhub_config.py` file:

```
c.Spawner.cmd = ['/srv/jupyter/.local/share/virtualenvs/jupyter-tutorial-aFv4x91/
↪bin/jupyterhub-singleuser']
```

3. Add a new systemd unit file `/etc/systemd/system/jupyterhub.service` with the command:

```
$ sudo systemctl edit --force --full jupyterhub.service
```

   Add your corresponding Python environment.

```
[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/.local/share/virtualenvs/jupyterhub-aFv4x91W/bin/jupyterhub -
↪f /srv/jupyter/jupyterhub_env/jupyterhub_config.py

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

4. Loading the configuration with:

```
$ sudo systemctl daemon-reload
```

5. The JupyterHub can be managed with:

```
$ sudo systemctl <start|stop|status> jupyterhub
```

6. To ensure that the service is also loaded during a system start, the following is called:

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

7. To be able to use the `jupyterhub-singleuser` and start your own server, the ix users must be entered in the `jupyter` group, for example with `usermod -aG jupyter` *VEIT*.

Since JupyterHub includes authentication and allows the execution of any code, it should not be executed without SSL (HTTPS). To do this, an official, trustworthy SSL certificate must be created. After you have received and installed a key and a certificate, you don't configure the JupyterHub itself, but the upstream Apache web server.

1. For this purpose, the additional modules are first activated with

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

2. Then the VirtualHost can be configured in `/etc/apache2/sites-available/jupyter.cusy.io.conf`

```
# 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. This VirtualHost is activated with

```
# a2ensite jupyter.cusy.io.conf
```

4. Finally, the status of the Apache web server is checked with

```
# systemctl status apache2
 apache2.service - The Apache HTTP Server
   Loaded: loaded (/lib/systemd/system/apache2.service; enabled; vendor preset:␣
```

(continues on next page)

(continued from previous page)

```
↪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)
    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.
```

5.2.3 Cookie Secret

The cookie secret is used to encrypt the browser cookies that are used for authentication.

1. The cookie secret can e.g. be created with

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

2. The file should not be readable by either `group` or `anonymous`:

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

3. Finally it will be entered in the `jupyterhub_config.py` file:

```
c.JupyterHub.cookie_secret_file = 'jupyterhub_cookie_secret'
```

5.2.4 Proxy authentication token

The hub authenticates its requests to the proxy using a secret token that the hub and proxy agree on. Usually, the proxy authentication token does not need to be set, as the hub itself generates a random key. This means that the proxy has to be restarted every time unless the proxy is a subprocess of the hub.

1. Alternatively, the value can e.g. can be generated with

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

2. It can then be entered in the configuration file, for example with

```
c.JupyterHub.proxy_auth_token =
↪'18a0335b7c2e7edeaf7466894a32bea8d1c3cff4b07860298dbe353ecb179fc6'
```

5.3 systemdspawner

The systemdspawner allows JupyterHub to create single user notebook servers with systemd. You get isolation and security without having to use Docker, rkt or similar. In addition, systemdspawner offers other features:

- the maximum allowed memory and CPU per person can be limited via cgroups and checked with `systemd-cgtop`.
- all get their own */tmp* directory to increase isolation
- notebook servers can be started as specific local users on the system
- the use of sudo in notebooks can be restricted
- the paths that can be read and written to can be restricted
- logs for each notebook can be managed

5.3.1 Requirements

systemdspawner requires systemd v211; the security-related functions require systemd v228. You can check which version of systemd is available on your system with

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

To limit memory and CPU allocations, certain kernel options must also be available. This can be checked with check-kernel.bash.

If the default setting `c.SystemdSpawner.dynamic_users = False` is used, the server is started with the local Unix user account. Therefore, this spawner requires that all users, already have a local account on the machine. With `c.SystemdSpawner.dynamic_users = True`, on the other hand, no local user accounts are required; they are dynamically created by systemd as needed.

5.3.2 Installation and Configuration

You can install systemdspawner with

```
$ pipenv install jupyterhub-systemdspawner
```

Then you can activate it in `jupyterhub_config.py` with

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

There are many other configuration options open to you, for example

`c.SystemdSpawner.mem_limit = '4G'`
specifies the maximum amount of memory that can be used by each user. The `None` setting disables the memory limit.

Although individual users should be able to use as much memory as possible, it is still useful to set a memory limit of 80–90% of the total physical memory. This prevents one user from accidentally crippling the machine single-handedly.

`c.SystemdSpawner.cpu_limit = 4.0`
A floating point number that specifies the number of CPU cores that each user can use.

c.SystemdSpawner.user_workingdir = '/home/*USERNAME*'
The directory where each user's notebook server is started. This directory is also what users see when they open their notebook servers. Normally this is the user's home directory.

c.SystemdSpawner.username_template = 'jupyter-*USERNAME*'
Template for the Unix user name under which each user should be created.

c.SystemdSpawner.default_shell = '/bin/bash'
The default shell used for the terminal in the notebook. Sets the environment variable SHELL to this value.

c.SystemdSpawner.extra_paths = ['/home/*USERNAME*/conda/bin']
List of paths to prepend to the PATH environment variable for the spawned notebook server. This is easier than setting the env property because you want to add PATH and not replace it completely. This is very useful if you want to include a virtualenv or conda installation in the user's PATH by default.

c.SystemdSpawner.unit_name_template = 'jupyter-*USERNAME*-singleuser'
Systemd service unit name template for each user notebook server. This allows differentiation between multiple JupyterHubs with systemd spawners on the same machine. Should only contain [a-zA-Z0-9_-].

c.SystemdSpawner.unit_extra_properties = *'LimitNOFILE': '16384'*
Dict of key-value pairs used to add arbitrary properties to spawned JupyerHub units – see also man systemd-run for details on properties.

c.SystemdSpawner.isolate_tmp = True
Setting this to True will create a separate, private /tmp directory for each user. This is very useful to protect against accidental leakage of otherwise private information.

This requires systemd version > 227. If you enable this in earlier versions, spawning will fail.

c.SystemdSpawner.isolate_devices = True
If you set this option to True, a separate, private /dev directory will be created for each user. This prevents users from accessing hardware devices directly, which could be a potential source of security problems. /dev/null, /dev/zero, /dev/random and the ttyp pseudo devices are already mounted, so most users should not notice any change if this is enabled.

c.SystemdSpawner.disable_user_sudo = True
Setting this option to True will prevent users from being able to use sudo or other means to become other users. This helps limit the damage done by compromising a user's credentials if they also have sudo privileges on the machine – a web-based exploit can now only damage the user's own data.

This requires systemd version > 228. If you enable this in earlier versions, spawning will fail.

c.SystemdSpawner.readonly_paths = ['/']
List of file system paths to be mounted read-only for the user's notebook server. This overrides any existing file system permissions. Subpaths of paths that are mounted readonly can be marked as readwrite with readwrite_paths. This is useful for marking / as read-only and listing only those paths to which notebook users are allowed to write. If the paths listed here do not exist, you will get an error message.

This requires systemd version > 228. If you enable this feature in earlier versions, spawning will fail. Up to systemd version 231 it can also only contain directories and no files.

c.SystemdSpawner.readwrite_paths = ['/home/*USERNAME*']
List of file system paths to be mounted read-only for the user's notebook server. This only makes sense if readonly_paths is used to make some paths read-only. This does not override the file system permissions – the user must have the appropriate permissions to write to these paths.

This requires systemd version > 228. If you enable this feature in earlier versions, spawning will fail. Up to systemd version 231, it can also only contain directories and not files.

See also:

- systemdspawner

5.4 Create service nbviewer

1. Configuring the notebook viewer as a JupyterHub service has the advantage that only users who have previously logged on to JupyterHub can call up the `nbviewer` instance. This can be used to protect access to notebooks as a 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
: The path name under which the notebook viewer can be reached: `/services/NAME`

url
: Protocol, address and port used by `nbviewer`

cwd
: The path to the `nbviewer` repository

command
: Command to start `nbviewer`

5.5 ipyparallel

This section provides an overview of ipyparallel which supports different types of parallelisation, including:

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

5.5.1 Installation

1. Installation

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

2. Activate notebook server extension:

```
$ pipenv run jupyter serverextension enable --py ipyparallel
Enabling: ipyparallel.nbextension
- Writing config: /Users/veit/.jupyter
    - Validating...
      ipyparallel.nbextension  OK
```

3. Install notebook extension:

```
$ pipenv run jupyter nbextension install --py ipyparallel
...
- Validating: OK

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

          jupyter nbextension enable ipyparallel --py
```

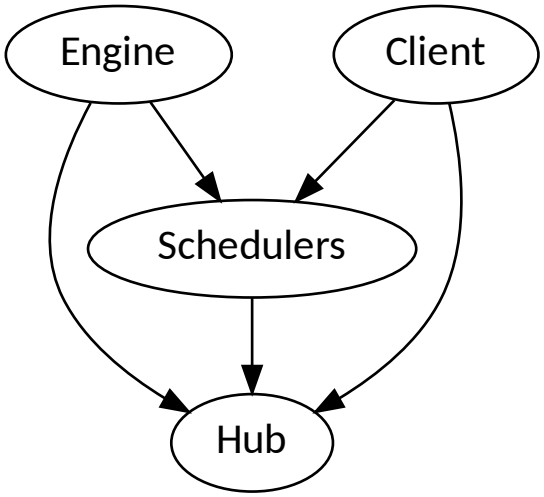
4. Activate notebook extension:

```
$ pipenv run jupyter nbextension enable --py ipyparallel
Enabling tree extension ipyparallel/main...
      - Validating: OK
```

5.5.2 Overview

Architecture

The `IPython.parallel` architecture consists of four components:



IPython-Engine

The IPython engine is an extension of the IPython kernel for Jupyter. The module waits for requests from the network, executes code and returns the results. IPython parallel extends the Jupyter messaging protocol with native Python object serialisation and adds some additional commands. Several engines are started for parallel and distributed computing.

IPython-Hub

The main job of the hub is to establish and monitor connections to clients and engines.

IPython-Schedulers

All actions that can be carried out on the engine go through a scheduler. While the engines themselves block when user code is executed, the schedulers hide this from the user to provide a fully asynchronous interface for a number of engines.

IPython-Client

There is a main object `Client` to connect to the cluster. Then there is a corresponding `View` for each execution model. These `Views` allow users to interact with a number of engines. The two standard views are:

- `ipyparallel.DirectView` class for explicit addressing
- `ipyparallel.LoadBalancedView` class for target-independent scheduling

Start

1. Starting the IPython Hub:

```
$ pipenv 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
: The database in which the IPython tasks are managed. In addition to the in-memory database `DictDB`, `MongoDB` and `SQLite` are further options.

`ipcontroller-client.json`
: Configuration file for the IPython client

`ipcontroller-engine.json`
: Configuration file for the IPython engine

Task-Schedulers
: The possible routing scheme. `leastload` always assigns tasks to the engine with the fewest open tasks.

Alternatively, `lru` (Least Recently Used), `plainrandom`, `twobin` and `weighted` can be selected, the latter two also need NumPy.

This can be configured in `ipcontroller_config.py`, for example with `c.TaskScheduler.scheme_name = 'leastload'` or with

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

2. Starting the IPython controller and the engines:

```
$ pipenv 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 systems

Besides the possibility to start `ipcontroller` and `ipengine` locally, see Starting a cluster with SSH, there are also the profiles for `MPI`, `PBS`, `SGE`, `LSF`, `HTCondor`, `Slurm`, `SSH` and `WindowsHPC`.

This can be configured in `ipcluster_config.py` for example with `c.IPClusterEngines.engine_launcher_class = 'SSH'` or with

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

See also:

MPI

3. Starting the Jupyter Notebook and loading the IPython-Parallel-Extension:

```
$ pipenv 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. Finally the cluster with the `default` profile can be started in the browser at the URL `http://localhost:8888/tree/docs/parallel/ipyparallel#ipyclusters`.

5.5.3 Check the 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']
```

5.5.4 Configuration

For the configuration, a configuration file is created for the client and engine when the IPython hub is started, usually in `~/.ipython/profile_default/security/`.

1. If we don't want to use the `default` profile, we should first create a new IPython profile with:

```
$ pipenv 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`
   : includes the configuration files for *Parallel Computing* (`ipengine`, `ipcontroller` ETC. (et cetera)).

2. When the IPython controller and the engines are started, the files `ipcontroller-engine.json` and `ipcontroller-client.json` are generated in `~/.ipython/profile_default/security/`.

`ipcluster` in `mpiexec/mpirun` mode

1. Creating the profile:

```
$ pipenv 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. Editing of `ipcluster_config.py`:

   1. so that the MPI launcher can be used:

```
c.IPClusterEngines.engine_launcher_class = 'MPIEngineSetLauncher'
```

3. The cluster can then be started with:

```
$ pipenv 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
```

5.5.5 IPython's `Direct` interface

Create a `DirectView`

[1]:
```
import ipyparallel as ipp


rc = ipp.Client()
```

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

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

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

Use all engines:

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

`map()` function

Python's builtin `map()` function can be applied to a sequence of elements and is usually easy to parallelise.

Please note that the `DirectView` version of `map()` does not do automatic load balancing. You may have to use `LoadBalancedView` for this.

[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
```

5.5.6 ipyparallel magics

[1]:
```
import ipyparallel as ipp


rc = ipp.Client()
```

[2]:
```
with rc[:].sync_imports():
    from matplotlib.pyplot import plot
    from numpy.linalg import eigvals
    from numpy.random import rand
    from numpy.random import random
```

```
importing plot from matplotlib.pyplot on engine(s)
importing eigvals from numpy.linalg on engine(s)
importing rand from numpy.random on engine(s)
importing random from numpy.random on engine(s)
```

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

[4]:
```
%px eigvals(a)
```

```
Out[0:2]: array([0.92645255, 0.48484728])
```

```
Out[3:2]: array([0.23462225, 0.7614717 ])
```

```
Out[1:2]: array([1.36472418, 0.17225772])
```

```
Out[2:2]: array([0.88215574, 0.01329861])
```

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

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

```
[stdout:1] hi
```

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

```
[stdout:3] hi
```

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

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

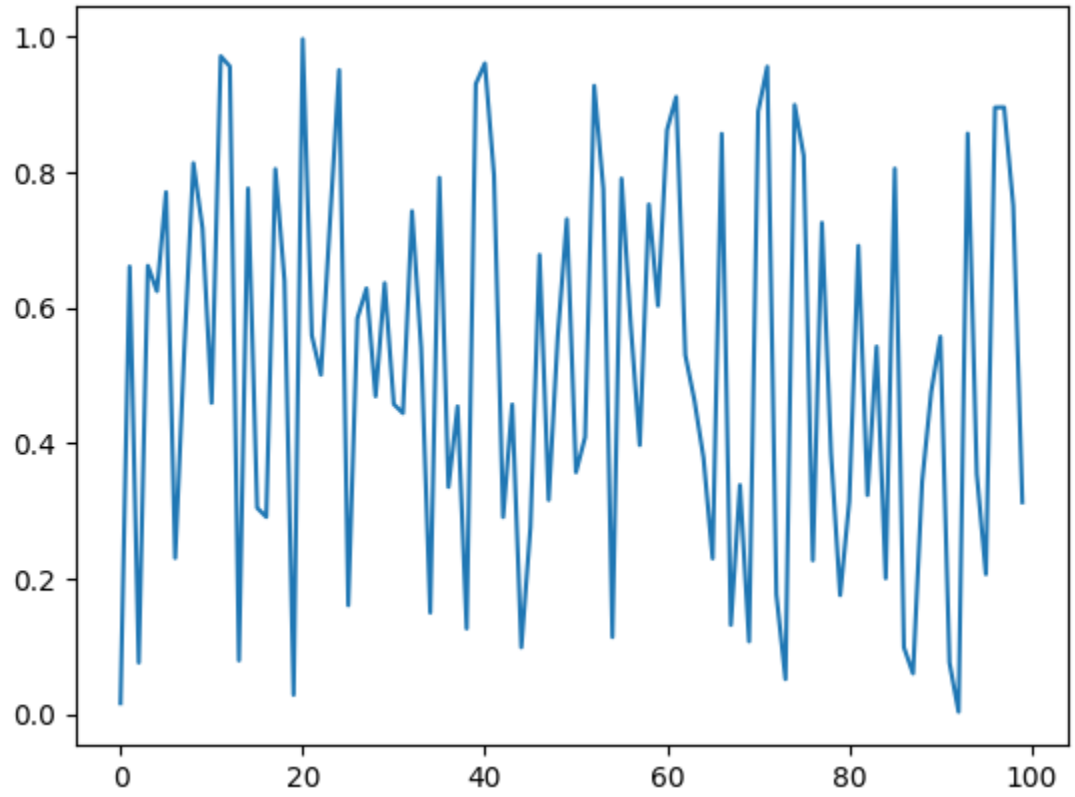
```
Out[3:5]: [<matplotlib.lines.Line2D at 0x12c926f50>]
```

```
Out[2:5]: [<matplotlib.lines.Line2D at 0x1180bc3d0>]
```

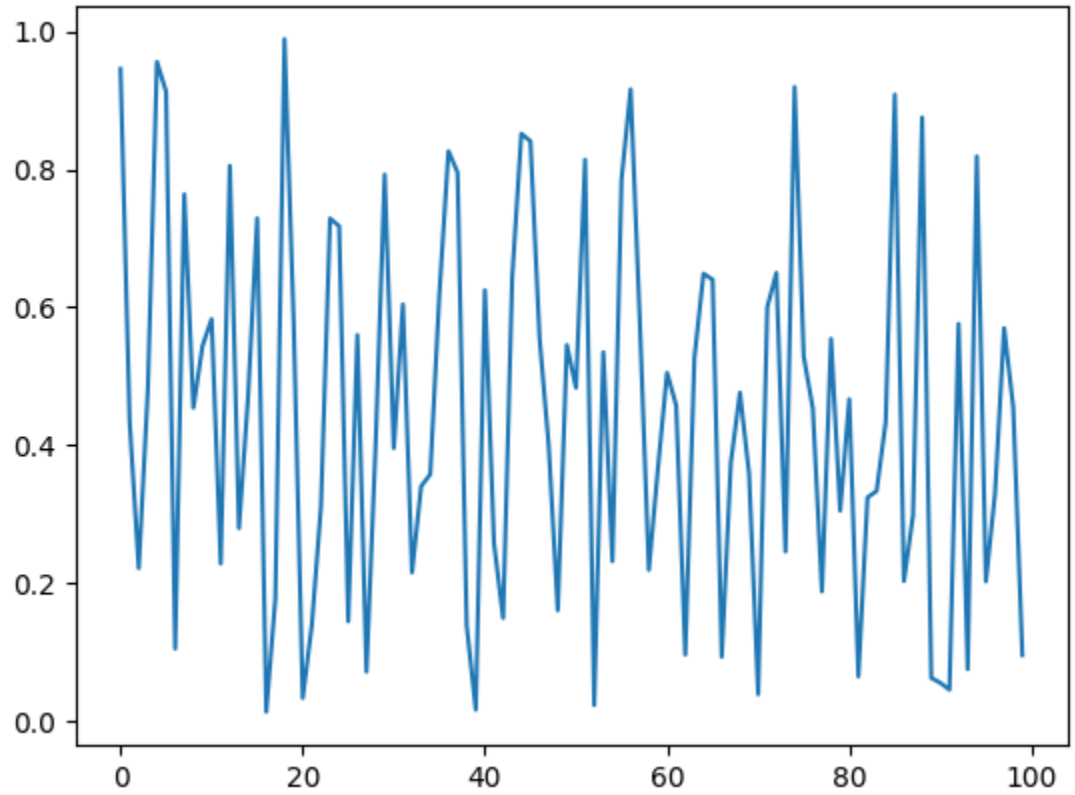
```
Out[1:5]: [<matplotlib.lines.Line2D at 0x116a39b10>]
```

```
Out[0:5]: [<matplotlib.lines.Line2D at 0x1164c5e50>]
```

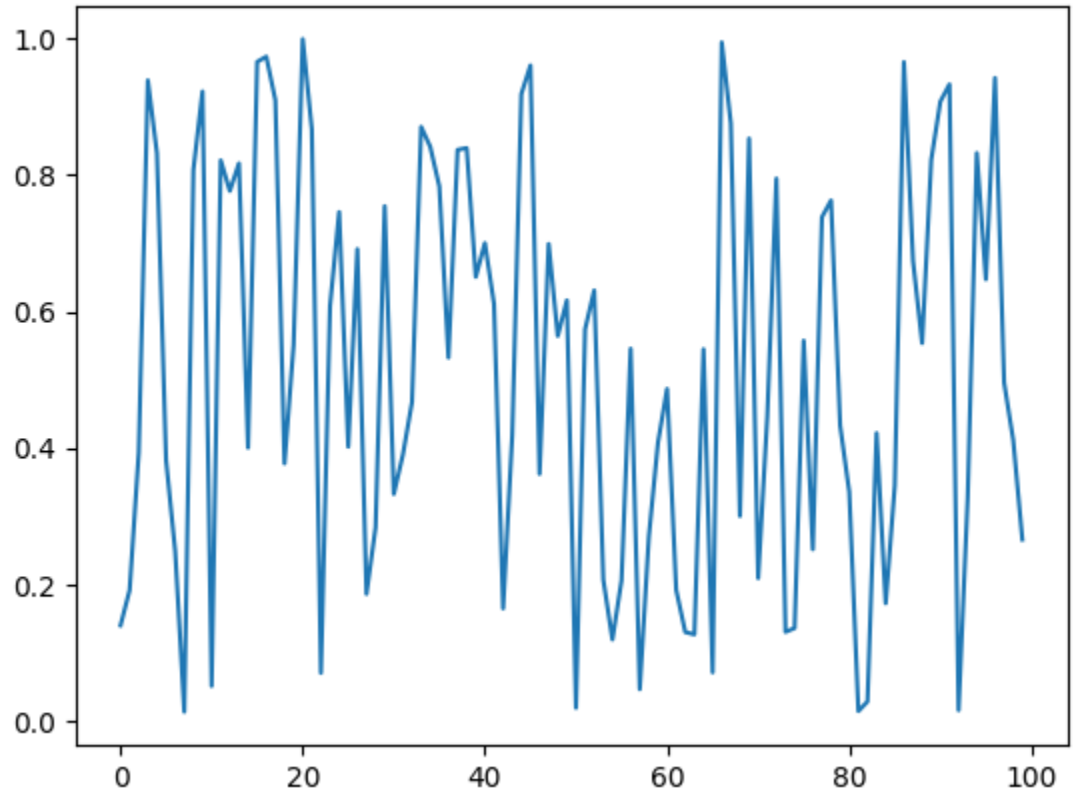
```
[output:1]
```



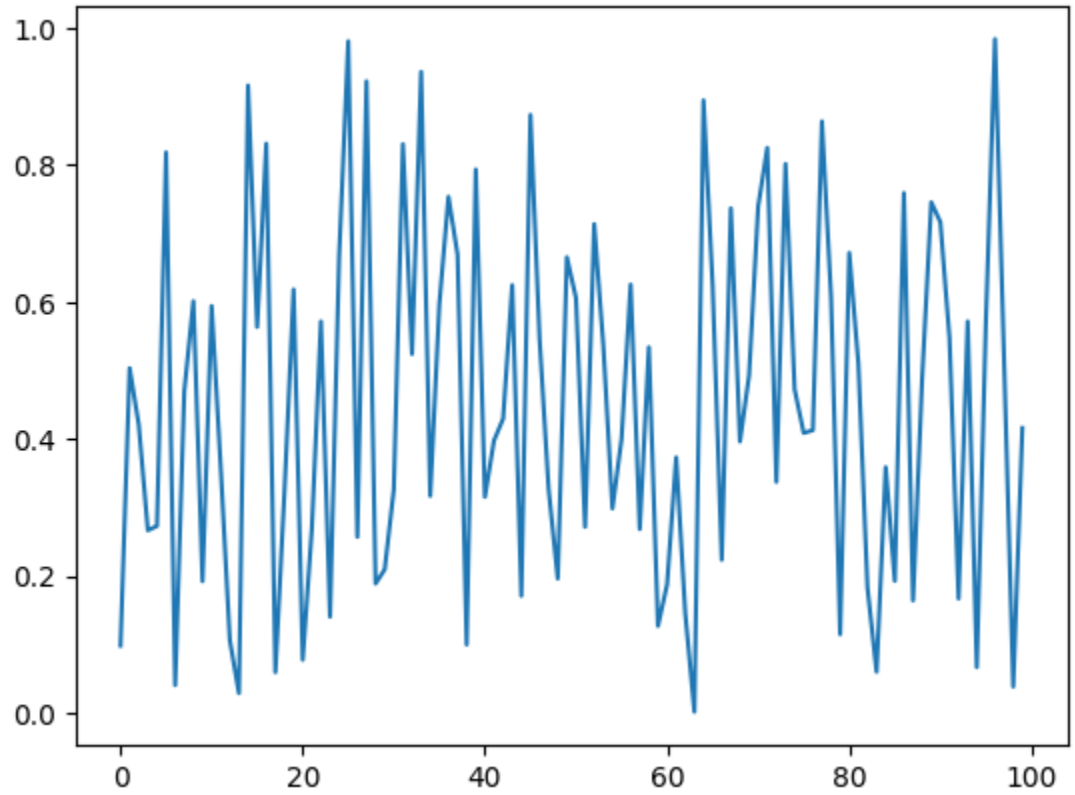
[output:2]



[output:0]



[output:3]



%%px cell magic

`--targets`, `--block` and `--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")
```

```
[stdout: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(%px): pending>
```

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

```
Out[0:8]: 1687880859.822293
```

```
Out[1:8]: 1687880859.82328
```

```
Out[2:8]: 1687880859.825772
```

```
Out[3:7]: 1687880859.823294
```

[13]:
```
%%px --block --group-outputs=engine
a = random((2,2))
ev = eigvals(a)
print(ev)
ev.max()
```

```
[stdout:0] [-0.28021424  0.98267777]
```

```
Out[0:9]: 0.9826777676672127
```

```
[stdout:1] [-0.21385117  0.65604043]
```

```
[stdout:3] [-0.01307275  1.11437227]
```

```
Out[1:9]: 0.6560404264968436
```

```
[stdout:2] [ 1.0290232  -0.55640908]
```

```
Out[3:8]: 1.1143722684611335
```

```
Out[2:9]: 1.0290231951538584
```

%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
```

%pxconfig

[16]:
```
%pxconfig --block
%px print("hi")
```

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

```
[stdout:1] hi
```

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

```
[stdout:3] hi
```

[17]:
```
%pxconfig --targets ::2
%px print("hi")
```

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

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

[18]:
```
%pxconfig --noblock
%px print("hi")
```

[18]:
```
<AsyncResult(%px): pending>
```

[19]:
```
%pxresult
```

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

(continues on next page)

(continued from previous page)

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

%autopx

[20]:
```
dview = rc[:]
```

[21]:
```
dview.block=True
```

[22]:
```
%autopx
```

```
%autopx enabled
```

[23]:
```
max_evals = []
for i in range(100):
    a = rand(10, 10)
    a = a + a.transpose()
    evals = eigvals(a)
    max_evals.append(evals[0].real)
```

[7]:
```
print(f"Average max eigenvalue is: {sum(max_evals)/len(max_evals)}")
```

```
[stdout:0] Average max eigenvalue is: 10.16283266814223
```

```
[stdout:1] Average max eigenvalue is: 10.107913323305741
```

```
[stdout:2] Average max eigenvalue is: 10.107616513254618
```

```
[stdout:3] Average max eigenvalue is: 10.040409820953181
```

Multiple active views

Magics of `ipyparallel` are assigned to certain `DirectView` objects. However, the active view can be changed by calling the `activate()` method on a view.

[25]:
```
even = rc[::2]
even.activate()
%px print("hi")
```

[25]:
```
<AsyncResult(%px): pending>
```

[26]:
```
even.block = True
%px print("hi")
```

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

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

If you activate the view, you can also specify a suffix so that it can be assigned to a whole range of magics without replacing the existing ones.

[27]:
```
rc.activate()
```

[27]:
```
<DirectView all>
```

[28]:
```
even.activate("_even")
%px print("hi")
```

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

```
[stdout:1] hi
```

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

```
[stdout:3] hi
```

[29]:
```
%px_even print("We aren't odd!")
```

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

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

This suffix is used at the end of all magics, e.g.`%autopx_even`, `%pxresult_even` etc.

For the sake of simplicity, also `Client` has an `activate()` method that creates a `DirectView` with `block = True`, activates it, and returns the new view.

The initial magics that are registered when a client is created are the result of the call `rc.activate()` with standard arguments.

Engines as kernel

Engines are actually the same object as IPython kernels, with the only exception that engines connect to a controller, while regular kernels bind their sockets directly to connections to their front end.

Sometimes you will connect your front end directly to an engine for debugging or analysing the interaction more directly. You can also do this by instructing the engine to bind its kernel to your frontend as well:

[30]:
```
%px import ipyparallel as ipp; ipp.bind_kernel()
%px %qtconsole
```

Note:

Be careful with this statement, as it starts as many QtConsoles as there are engines available.

Alternatively, you can also display the connection information and determine how you can establish a connection to the engines, depending on where they live and where you are:

[31]:
```
%px %connect_info
```

```
[stdout:0] {
  "shell_port": 63143,
  "iopub_port": 63148,
  "stdin_port": 63154,
  "control_port": 63167,
  "hb_port": 63156,
  "ip": "127.0.0.1",
  "key": "defc3484-e06d43284f54a3939ad7eec2",
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": ""
}

Paste the above JSON into a file, and connect with:
    $> jupyter <app> --existing <file>
or, if you are local, you can connect with just:
    $> jupyter <app> --existing /srv/jupyter/.ipython/profile_default/security/kernel-
↪11237.json
or even just:
    $> jupyter <app> --existing
if this is the most recent Jupyter kernel you have started.
```

```
[stdout:1] {
  "shell_port": 63145,
  "iopub_port": 63149,
  "stdin_port": 63153,
  "control_port": 63165,
  "hb_port": 63155,
  "ip": "127.0.0.1",
  "key": "defc3484-e06d43284f54a3939ad7eec2",
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": ""
}

Paste the above JSON into a file, and connect with:
    $> jupyter <app> --existing <file>
or, if you are local, you can connect with just:
    $> jupyter <app> --existing /srv/jupyter/.ipython/profile_default/security/kernel-
↪11238.json
or even just:
    $> jupyter <app> --existing
if this is the most recent Jupyter kernel you have started.
```

```
[stdout:2] {
  "shell_port": 63147,
  "iopub_port": 63152,
  "stdin_port": 63160,
  "control_port": 63166,
  "hb_port": 63161,
```

(continues on next page)

(continued from previous page)

```
  "ip": "127.0.0.1",
  "key": "defc3484-e06d43284f54a3939ad7eec2",
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": ""
}

Paste the above JSON into a file, and connect with:
    $> jupyter <app> --existing <file>
or, if you are local, you can connect with just:
    $> jupyter <app> --existing /srv/jupyter/.ipython/profile_default/security/kernel-
↪11239.json
or even just:
    $> jupyter <app> --existing
if this is the most recent Jupyter kernel you have started.
```

```
[stdout:3] {
  "shell_port": 63150,
  "iopub_port": 63159,
  "stdin_port": 63162,
  "control_port": 63169,
  "hb_port": 63163,
  "ip": "127.0.0.1",
  "key": "defc3484-e06d43284f54a3939ad7eec2",
  "transport": "tcp",
  "signature_scheme": "hmac-sha256",
  "kernel_name": ""
}

Paste the above JSON into a file, and connect with:
    $> jupyter <app> --existing <file>
or, if you are local, you can connect with just:
    $> jupyter <app> --existing /srv/jupyter/.ipython/profile_default/security/kernel-
↪11240.json
or even just:
    $> jupyter <app> --existing
if this is the most recent Jupyter kernel you have started.
```

5.5.7 Task interface

The task interface to the cluster presents the engines as a fault-tolerant, dynamic load balancing of Workers. In contrast to the direct interface, the task interface does not have direct access to individual engines. As the IPython scheduler assigns the workers, the interface becomes simpler and more powerful at the same time.

The best part, however, is that both interfaces can be used at the same time to leverage their respective strengths. If calculations do not depend on previous results, the task interface is ideal:

Create an `LoadBalancedView` instance

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

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

[3]:
```
rc = ipp.Client(url_file='/Users/veit/.ipython/profile_mpi/security/ipcontroller-client.
↪json')
```

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

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

`load_balanced_view` is the default view.

See also:

- Views

Fast and easy parallelism

`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,...]
```

Dependencies

Note:

Please note that the pure ZeroMQ scheduler does not support any dependencies.

Function dependencies

`UnmetDependency`

`@ipp.require` decorator

`@ipp.depend` decorator

`dependent` object

`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)
```

> **See also:** Some parallel workloads can be described as Directed acyclic graph (DAG). In DAG Dependencies we describe using an example how NetworkX is used to represent the task dependencies as DAG.

`ImpossibleDependency`

`retries` and `resubmit`

Schedulers

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

Scheme	Description
`lru`	**Least Recently Used**: Always assigns the workers to the last used engine. Similar to *round robin*, however, it does not take into account the runtime of each individual task.
`plainrar`	**Plain Random**: Randomly selects the engine to be run.
`twobin`	**Two-Bin Random**: Requires `numpy`. Randomly select two engines and use `lru`. This is often better than the purely random distribution, but requires more computational effort.
`leastloa`	**Least Load**: Standard scheme that the engine always assigns tasks with the fewest outstanding tasks.
`weightec`	**Weighted Two-Bin Random**: Weighted **Two-Bin Random** scheme.

5.5.8 AsyncResult object

`apply()` returns in the `noblock` mode an `AsyncResult` object. This allows inquiries with the `get()` method at a later point in time. In addition, metadata occurring during execution is also collected in this object.

The AsyncResult object provides a number of useful functions for parallelisation that can be accessed through Python's multiprocessing.pool.AsyncResult:

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
```

Metadata

Client.metadata

Timing

Iterable map results

[2]:
```
from __future__ import print_function

import time

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)
```

(continues on next page)

(continued from previous page)

```
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()
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, 3]
sleeping for `id` seconds on each engine
0: 0.005
1: 1.012
2: 2.011
3: 3.013
running with one call per task
task 0 on engine 0: 0.007
task 1 on engine 3: 0.016
task 2 on engine 2: 0.029
task 3 on engine 1: 0.042
task 4 on engine 0: 0.053
task 5 on engine 3: 0.073
task 6 on engine 2: 0.092
task 7 on engine 1: 0.118
```

(continues on next page)

(continued from previous page)

```
task 8 on engine 0: 0.139
...
```

[3]:
```
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))
```

[3]:
```
58.028845747399714
```

1. `map(sq, X)` computes the square of each item in the list.
2. `reduce(add, sqX) / len(X)` calculates the mean by adding the list of `AsyncMapResult` and dividing by the number.
3. Square root of the resulting number.

See also:

If you want to expand the results of `AsyncResult` or `AsyncMapResult` you can do so with the `msg_ids` attribute. You can find an example for this at ipyparallel/docs/source/examples/customresults.py.

5.5.9 MPI

Often, a parallel algorithm requires moving data between the engines. One way is to push and pull over the `DirectView`. However, this is slow because all of the data has to get through the controller to the client and then back to the final destination.

A much better option is to use the Message Passing Interface (MPI). IPython's parallel computing architecture was designed from the ground up to integrate with MPI. This notebook gives a brief introduction to using MPI with IPython.

Requirements

- A standard MPI implementation like OpenMPI or MPICH.

  For Debian/Ubuntu these can be installed with

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

  or

  ```
  $ sudo apt install mpich
  ```

  Alternatively, OpenMPI or MPICH can also be installed with Spack: the packages are `openmpi` or `mpich`.

- mpi4py

Starting the engines with activated MPI

Automatic start with `mpiexec` and `ipcluster`

This can be done with, for example

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

For this, however, a corresponding profile must first be created; see *configuration*.

Automatic start with PBS and `ipcluster`

The `ipcluster` command also offers integration in PBS. You can find more information about this in Starting IPython Parallel on a traditional cluster.

Example

The following notebook cell calls `psum.py` with the following content:

```
from mpi4py import MPI
import numpy as np

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')
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]
```

CHAPTER

SIX

BINDER

Binder provides an easy way to share computer environments with everyone. Binder is used for

Teaching and training
Binder can be used to share links to interactive data analysis environments. This is great for workshops, tutorials and courses and allows you to get students up to speed with the code much more quickly.

Technical documentation
Binder tools can be used to make documentation and demonstrations of tools interactive.

Open educational resources
Binder can provide publicly available interactive educational materials, enabling richer experiences.

Reproducible scientific analysis
Binder allows you to share an interactive environment along with your code and analysis. You can share a link where others can reproduce and interact with your work. For example, the Neurolibre project uses Binder to reproduce neuroscience analyses.

Binder provides a full open-source infrastructure stack. The main tools are

`BinderHub`
provides the Binder service in the cloud.

See also:

- Repository
- Docs
- Examples

`repo2docker`
creates reproducible Docker images from a Git repository.

See also:

- Repository
- Docs

mybinder.org
public BinderHub deployment.

CHAPTER

SEVEN

NBCONVERT

nbconvert
: converts notebooks to other formats

7.1 Installation

```
$ pipenv install nbconvert
```

Important: To be able to use all functions of `nbconvert`, Pandoc and TeX (especially XeLaTeX) are required. These must be installed separately.

7.1.1 Install Pandoc

`nbconvert` uses Pandoc to convert Markdown to formats other than HTML.

```
$ sudo apt install pandoc
```

```
$ brew install pandoc
```

7.1.2 Install Tex

For the conversion to PDF, `nbconvert` uses the Tex ecosystem in preparation: A `.tex` file is created, which is converted into a PDF by the XeTeX engine.

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

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

See also:

MacTeX

7.2 Use on the command line

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

latex
: creates a `NOTEBOOK_NAME.tex` file and possibly images as PNG files in a folder. With `--template` you can choose between one of two templates:

  --template article
  : default

    Latex article, derived from the Sphinx how-to

  --template report
  : Latex report with table of contents and chapters

pdf
: creates a PDF over latex. Supports the same templates as `latex`.

slides
: creates Reveal.js slides.

script
: kconverts the notebook into an executable script. This is the easiest way to create a Python script or a script in another language.

Note: If a notebook contains *Magics*, then this can possibly only be carried out in one Jupyter session.

We can for example convert foo.ipynb into a Python script with:

```
$ pipenv run jupyter nbconvert --to script workspace/ipython/mypackage/foo.ipynb
[NbConvertApp] Converting notebook docs/basics/ipython/mypackage/foo.ipynb to script
[NbConvertApp] Writing 245 bytes to docs/basics/ipython/mypackage/foo.py
```

The result is then `foo.py` with:

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

# # `foo.ipynb`

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

# In[2]:
def has_ip_syntax():
    listing = get_ipython().getoutput("ls")
    return listing

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

Note: In order to assign notebook cells to slides, you should select *View → Cell Toolbar → Slideshow*. Then a menu is displayed in each cell at the top right with the options: *Slide, Sub-Slide, Fragment, Skip, Notes*.

Note: Lecture notes require a local copy of `reveal.js`. The following option can be specified so that `nbconvert` can find this: `--reveal-prefix /path/to/reveal.js`.

Further details for `FORMAT` are `asciidoc`, `custom`, `html`, `markdown`, `notebook`, and `rst`.

7.3 nb2xls

nb2xls converts Jupyter notebooks into Excel files (`.xlsx`) taking into account pandas DataFrames and Matplotlib outputs. However, the input cells are not converted and only part of the Markdown is converted.

7.4 Own exporters

See also:

Customizing exporters allows you to write your own exporters.

CHAPTER

EIGHT

NBVIEWER

nbviewer
: *nbconvert* as web service: Renders Jupyter notebooks as static web pages.

8.1 Installation

1. The Notebook Viewer requires several binary packages that have to be installed on our system, for

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

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

1. The Jupyter Notebook Viewer can then be installed in a new virtual environment with:

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

Note: The notebook app outputs the error `AttributeError: module 'tornado.gen' has no attribute 'Task'` with current versions of Tornado. This error does not occur with `tornado<6.0`, , see also Delete Terminal Not Working with Tornado version 6.0.1:

```
$ pipenv install "tornado<6.0"
```

Now `nbviewer` can also be installed:

```
$ pipenv install nbviewer
```

2. For testing, the server can be started with:

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

8.2 Extending the Notebook Viewer

The notebook viewer can be extended to include providers, see Extending the Notebook Viewer.

8.3 Access control

If the viewer is run as *Create service nbviewer*, only users who have authenticated themselves on the JupyterHub can access the nbviewer's notebooks.

CHAPTER

NINE

KERNELS

The Jupyter team manages the IPython-Kernel. In addition to Python, many other languages can be used in notebooks. The following Jupyter kernels are widely used:

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

A list of available kernels can be found in Jupyter kernels.

See also:

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

9.1 Install, view and start the kernel

9.1.1 Install a kernel

Kernels are searched for in the following directories, for example:

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

To make your new environment available as a Jupyter kernel in one of the directories, you should install ipykernel:

```
$ pipenv install ipykernel
```

You can then register your kernel, for example with

```
$ pipenv run python -m ipykernel install --prefix=/srv/jupyter/.ipython/kernels --name␣
↪python311 --display-name 'Python 3.11 Kernel'
```

--prefix=*/PATH/TO/KERNEL*
: specifies the path where the Jupyter kernel is to be installed.

--user
: installs the kernel for the current user and not system-wide

name *NAME*
: gives a name for the `kernelspec`. This is required in order to be able to use several IPython kernels at the same time.

`ipykernel install` creates a `kernelspec` file in JSON format for the current Python environment, for example:

```
{
 "display_name": "My Kernel",
 "language": "python"
 "argv": [
  "/srv/jupyter/.ipython/kernels/python311_kernel-7y9G693U/bin/python",
  "-m",
  "ipykernel_launcher",
  "-f",
  "{connection_file}"
 ],
}
```

display_name
: The name of the kernel as it should be displayed in the browser. In contrast to the kernel name used in the API, it can contain any Unicode characters.

language
: The name of the language of the kernel. If no suitable `kernelspec` key is found when loading notebooks, a kernel with a suitable language is used. In this way, a notebook written for a Python or Julia kernel can be linked to the user's Python or Julia kernel, even if it does not have the same name as the author's.

argv
: A list of command line arguments used to start the kernel. `{connection_file}` refers to a file that contains the IP address, ports, and authentication key required for the connection. Usually this JSON file is saved in a safe place of the current profile:

  `connection_file` refers to a file containing the IP address, ports and authentication key needed for the connection. Typically, this JSON file is stored in a secure location of the current profile:

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

interrupt_mode
: can be either `signal` or `message` and specifies how a client should interrupt the execution of a cell on this kernel.

signal
sends an interrupt, e.g. `SIGINT` on *POSIX* systems

message
sends an `interrupt_request`, see also Kernel Interrupt.

env
`dict` with environment variables to be set for the kernel. These are added to the current environment variables before the kernel starts.

metadata
`dict` with additional attributes for this kernel. Used by clients to support the kernel selection. Metadata added here should have a namespace for the tool to read and write that metadata.

You can edit this `kernelspec` file at a later time.

9.1.2 Show available kernels

```
$ pipenv run jupyter kernelspec list
Available kernels:
  mykernel    /Users/veit/Library/Jupyter/kernels/mykernel
  python2    /Users/veit/Library/Jupyter/kernels/python2
  python3    /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/bin/../share/
↪jupyter/kernels/python3
```

9.1.3 Start kernel

```
$ pipenv run jupyter console --kernel mykernel
Jupyter console 6.0.0
Python 2.7.15 (default, Oct 22 2018, 19:33:46)
...

In [1]:
```

With `ctrl-d` you can exit the kernel again.

9.1.4 Delete kernel

```
$ pipenv run jupyter kernelspec uninstall mykernel
```

9.1.5 Uninstall the Standard kernel

If not already done, a configuration file can be created, for example with

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

Then you can add the following line to this configuration file:

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

9.2 What's new in Python 3.8?

In Python 3.8, the syntax is simplified and support for C libraries is also improved. Below is a brief overview of some of the new features. You can get a complete overview in What's New In Python 3.8.

9.2.1 Installation

Check

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

```
Python 3.8.0
```

or

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

9.2.2 Assignment Expressions: Walrus operator :=

So far, e.g. `env_base` can be determined by pip as follows:

[ ]:
```
import os
```

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

This can now be simplified with:

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

Multiple nested `if`, such as in cpython/Lib/copy.py, can also be avoided. This

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

[ ]:
```
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)
```

(continues on next page)

(continued from previous page)

```
            if reductor:
                rv = reductor()
            else:
                raise Error(
                    "un(deep)copyable object of type %s" % cls)
```

becomes that:

[ ]:
```
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* parameters

In Python 3.8 a function parameter can be specified position-related with `/`. Several Python functions implemented in C do not allow keyword arguments. This behavior can now be emulated in Python itself, e.g. for the pow() function:

[ ]:
```
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` support = for self-documenting expressions and debugging

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

9.2.5 Debug and release build use the same ABI

So far, a consistent application binary interface (ABI) should be guaranteed by Spack. However, this did not include using Python in the debug build. Python 3.8 now also supports ABI compatibility for debug builds. The `Py_TRACE_REFS` macro can now be set with the `./configure --with-trace-refs` option.

9.2.6 New C API

PEP 587 adds a new C API for configuring the Python initialisation, which offers more precise control of the entire configuration and better error reports.

9.2.7 Vectorcall – a fast protocol for CPython

The protocol is not yet fully implemented; this will probably come with Python 3.9. However, you can already get a full description in PEP 590.

9.2.8 Update – or not?

The following is a brief overview of the problems you may encounter when switching to Python 3.8:

Missing packages

- opencv-python

Bugs

- Python 3.7.1 was released 4 months after the first major release with a long list of bug fixes . Something similar is to be expected with Python 3.8.

Syntax

- Very few code analysis tools and autoformatters can already handle the syntax changes of Python 3.8

Why update anyway?

Since the upgrade will take some time, it can be tempting to postpone the move indefinitely. Why should you concern yourself with incompatibilities in new versions when your current version works reliably?

The problem is that your Python is not supported indefinitely, nor will the libraries you use will support all older Python versions indefinitely. And the longer you delay an update, the bigger and riskier it will be. Therefore, the update to the new major version of Python is usually recommended a few months after the first release.

9.2.9 Porting

See also:

- Porting to Python 3.8

9.3 What's new in Python 3.9?

With Python 3.9, a new release cycle is used for the first time: in the future, new releases will appear annually (see also PEP 602). The developers hope that they will get faster feedback on new features.

With the first published release candidate, Python should also have a stable binary interface (application binary interface, ABI): there should no longer be any ABI changes in the 3.9 series, which means that extension modules no longer have to be recompiled for each version.

You can find more information in What's New In Python 3.9.

In the following, I'll give you a brief overview of some of the new features.

9.3.1 Installation

Check

[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 and Update Operators

Operators for the built-in dict class are now similar to those for concatenating lists: Merge (`|`) and Update (`|=`). This eliminates various disadvantages of the previous methods `dict.update`, `{**d1, **d2}` and `collections.ChainMap`.

Example ipykernel/ipykernel/kernelapp.py

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

(continues on next page)

(continued from previous page)

```
    '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.'
    ),
})
```

can be simplified with:

```
[ ]: 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}},
            "stderr auf das Nullgerät umleiten"),
}}
```

Example matplotlib/legend.py

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

can be simplified with:

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

9.3.3 PEP 616: `removeprefix()` and `removesuffix()` for string methods

With str.removeprefix(prefix) and str.removesuffix(suffix) you can easily remove prefixes and suffixes. Similar methods have also been added for `bytes`, `bytearray` objects, and `collections.UserString`. All in all, this should lead to less fragile, better performing and more readable code.

Example find_recursionlimit.py

```
[ ]: if test_func_name.startswith("test_"):
         print(test_func_name[5:])
     else:http://localhost:8888/notebooks/docs/workspace/jupyter/kernels/python39.ipynb
     ↪#Beispiel-find_recursionlimit.py
         print(test_func_name)
```

can be simplified with:

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

Example deccheck.py

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

can be simplified with:

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

9.3.4 PEP 585: Additional generic types

In *Type Annotations*, for example `list` or `dict` can be used directly as generic types – they no longer have to be imported separately from `typing`. Importing `typing` is thus deprecated.

Example

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

9.3.5 PEP 617: New PEG parser

Python 3.9 now uses a PEG (Parsing Expression Grammar) parser instead of the previous LL parser. This has i.a. the following advantages:

- the parsing of abstract syntax trees (AST) is simplified considerably
- Left recursion becomes possible
- The creation of concrete syntax trees (CST) is possible

The new parser is therefore more flexible and should be used primarily when designing new language functions. The ast module is already using the new parser without the output having changed.

In Python 3.10, the old parser and all functions that depend on it – mainly the obsolete parser module - are deleted. Only in Python 3.9 you can return to the LL parser on the command line with `-X oldparser` or with the environment variable `PYTHONOLDPARSER=1`.

9.3.6 PEP 615: Support for the IANA Time Zone Database in the Standard Library

The new zoneinfo brings support for the IANA time zone database to the standard library.

[5]:
```
from zoneinfo import ZoneInfo
from datetime import datetime, timedelta
```

Pacific Daylight Time:

[6]:
```
dt = datetime(2020, 10, 31, 12, tzinfo=ZoneInfo("America/Los_Angeles"))
print(dt)
```

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

[7]:
```
dt.tzname()
```

[7]:
```
'PDT'
```

Pacific Standard Time:

[8]:
```
dt += timedelta(days=7)
print(dt)
```

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

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

```
PST
```

9.4 What's New In Python 3.10

See also:

- What's New In Python 3.10

[1]:
```
import sys
assert sys.version_info[:2] >= (3, 10)
```

9.4.1 Better error messages

Syntax Errors

- When parsing code that contains unclosed parentheses or brackets the interpreter now includes the location of the unclosed bracket of parentheses instead of displaying `SyntaxError: unexpected EOF`.
- `SyntaxError` exceptions raised by the interpreter will now highlight the full error range of the expression that constitutes the syntax error itself, instead of just where the problem is detected.
- Specialised messages for `SyntaxError` exceptions have been added e.g. for
  - missing `:` before blocks
  - unparenthesised tuples in comprehensions targets
  - missing commas in collection literals and between expressions
  - missing `:` and values in dictionary literals
  - usage of = instead of == in comparisons
  - usage of * in f-strings

Indentation Errors

- Many `IndentationError` exceptions now have more context.

Attribute Errors

- `AttributeError` will offer suggestions of similar attribute names in the object that the exception was raised from.

Name Errors

- `NameError` will offer suggestions of similar variable names in the function that the exception was raised from.

9.4.2 Structural Pattern Matching

Many functional languages have a `match` expression, for example Scala, Rust, F#.

A `match` statement takes an expression and compares it to successive patterns given as one or more case blocks. This is superficially similar to a switch statement in C, Java or JavaScript, but much more powerful.

`match`

The simplest form compares a subject value against one or more literals:

[2]:
```
def http_error(status):
      match status:
          case 400:
              return "Bad request"
          case 401:
              return "Unauthorized"
```

(continues on next page)

(continued from previous page)

```
            case 403:
                return "Forbidden"
            case 404:
                return "Not found"
            case 418:
                return "I'm a teapot"
            case _:
                return "Something else"
```

Note:

Only in this case _ acts as a wildcard that never fails and **not** as a variable name.

The cases not only check for equality, but rebind variables that match the specified pattern. For example:

[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
```

«If this poorly-designed feature is really added to Python, we lose a principle I've always taught students: ‹if you see an undocumented constant, you can always name it without changing the code's meaning.› The Substitution Principle, learned in algebra? It'll no longer apply.» – Brandon Rhodes

«… the semantics of this can be quite different from switch. The cases don't simply check equality, they rebind variables that match the specified pattern.» – Jake VanderPlas

Symbolic constants

Patterns may use named constants. These must be dotted names to prevent them from being interpreted as capture variable:

[4]:
```
from enum import Enum

class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

color = Color(2)

match color:
    case color.RED:
        print("I see red!")
    case color.GREEN:
        print("Grass is green")
```

(continues on next page)

(continued from previous page)

```
    case color.BLUE:
        print("I'm feeling the blues :(")
```

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

«… “case CONSTANT” actually matching everything and assigning to a variable named CONSTANT»
– Armin Ronacher

See also:

- Structural pattern matching for Python
- PEP 622 – Structural Pattern Matching superseded by
- 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

   For Ubuntu & Debian:

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

2. R packages

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

3. Deploy the kernel

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

You can also deploy the kernel system-wide:

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

See also:

- IRkernel Installation

CHAPTER

TEN

IPYWIDGETS

ipywidgets are interactive widgets for Jupyter notebooks. They extend notebooks by the possibility that users can enter data themselves, manipulate data and see the changed results.

10.1 Examples

IPython includes an interactive widget architecture that combines Python code running in the kernel and JavaScript/HTML/CSS running in the browser. These widgets allow users to interactively examine their code and data.

10.1.1 Interact function

`ipywidgets.interact` automatically creates user interface (UI) controls to interactively explore code and data.

[1]:
```
from __future__ import print_function

import ipywidgets as widgets

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

In the simplest case, `interact` automatically generates controls for function arguments and then calls the function with those arguments when you interactively edit the controls. The following is a function that returns its only argument `x`.

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

Slider

If you specify a function with an integer keyword argument (`x=10`), a slider is generated and bound to the function parameter:

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

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

Checkbox

If you specify `True` or `False`, interact generates a checkbox:

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

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

Text area

If you pass a string, `interact` generates a text area:

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

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

10.1.2 Decorator

`interact` can also be used as a decorator. This way you can define a function and interact with it in a single setting. As the following example shows, `interact` also works with functions that have multiple arguments:

[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 list

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

10.2.1 Numeric widgets

There are a variety of IPython widgets that are designed to display numeric values. The integer widgets have a similar naming scheme as their counterparts with floating point numbers. You can find the respective integer equivalent by replacing `Float` with`Int` in the widget name.

IntSlider

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

The `FloatLogSlider` has a scale that makes it easy to use a slider for a wide range of positive numbers. `min` and `max` refer to the minimum and maximum exponents of the base and the `value` refers to the actual value of the slider.

[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,
    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 Boolean widgets

There are three widgets that are designed to display Boolean values.

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

The `Valid` widget offers a read-only display.

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

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

10.2.3 Selection widgets

There are several widgets for selecting single values and two for multiple values. All inherit from the same base class.

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",
    readout=True,
)
```

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

SelectionRangeSlider

index is a tuple of minimum and maximum values.

[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:', icons=('check', 'check'), options=('Slow', 'Regular',
↪ 'Fast'), tooltips=('...
```

SelectMultiple

Several values can be selected by holding down the shift and/or ctrl (or command) keys and clicking the mouse or arrow keys.

[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

There are several widgets that can be used to display strings. The widgets `Text` and`Textarea` accept input; the widgets `HTML` and`HTMLMath` display a string as HTML (`HTMLMath` also renders mathematical formulas).

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

The `Label` widget is useful for custom descriptions that are similar in style to the built-in descriptions.

[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

The `Output` widget can record and display `stdout`, `stderr` and IPython.display.

You can attach the output to an output widget or delete it programmatically.

[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': '...
```

You can find detailed documentation in Output widgets.

10.2.8 Play/Animation-Widget

The `Play` widget is useful for running animations that you want to run at a specific speed. In the following example, a slider is linked to the player.

[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

The date picker widget works in Chrome, Firefox and IE Edge, but not currently in Safari because it does not support `input type="date"`.

[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` enables the use of a game controller as an input device.

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

```
Controller()
```

10.2.12 Container/layout widgets

These widgets are used to store other widgets called `children`.

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` and`Tab`

Unlike the other widgets previously described, the container widgets `Accordion` and`Tab` update their `selected_index` attribute when the user changes the accordion or tab; In addition to user input, the `selected_index` can also be set programmatically.

If `selected_index = None` is chosen, all accordions will be closed or all tabs will be deselected.

In the following notebook cells the value of `selected_index` of the tab and/or accordion is displayed.

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

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

Nesting tabs and accordions

Tabs and accordions can be nested as deeply as you want. The following example shows some tabs with an accordion as `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` cannot be used to represent a data type, but only for `on_click`. With the `print` function the docstring of `on_click` can be output.

[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.
```

Examples

Button clicks are stateless, i.e. they transfer messages from the front end to the back end. If you use the `on_click` method, a button will be displayed that will print the message as soon as it is clicked.

[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 are IPython traitlets. To make changes, the `observe` method of the widget can be used to register a callback. You can see the docstring for observe below.

You can find more information at 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
            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

To link widget attributes, you can simply link them together.

Linking traitlet attributes in the 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 attributes on the client side

There might be a delay while synchronizing Traitlet attributes due to communication with the server. However, you can also link the widget attributes to the link widgets directly in the browser. The Javascript links with `jslink` are retained even if widgets are embedded in HTML websites without a kernel.

[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 Continuous updates

Some widgets offer a `continuous_update` attribute with the ability to continuously update values. In the following example we can see that the delayed controls only transmit their value after the user has dragged the slider or sent the text field. The continuous slides transfer their values continuously as soon as they are changed.

[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 Custom widget

The widget framework is based on the Comms framework, which enables the kernel to send and receive JSON to the front end. In order to create a custom widget, the widget must be defined both in the browser and in the Python kernel.

See also:

- Low Level Widget Tutorial.

10.4.1 Python kernel

DOMWidget

To define a widget, it must inherit from the `Widget` or `DOMWidget` base class. If the widget is to be displayed in the Jupyter notebook, your widget should inherit from `DOMWidget`. The `DOMWidget` class itself inherits from the `Widget` class.

`_view_name`

By adopting `DOMWidget`, the widget framework is **not** informed which front-end widget should be linked to the back-end widget.

Instead, you have to specify this yourself using one of the following attributes:

- `_view_name`
- `_view_module`
- `_view_module_version`

and if applicable

- `_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 is a framework with which Python classes can have attributes with type checking, dynamically calculated default values and callbacks when changed. The `sync=True` keyword argument tells the widget framework to synchronise the value with the browser; without it, the browser would not learn anything about `_view_name` or `_view_module`.

10.4.2 Frontend (JavaScript)

Models and Views

The front end of the IPython widget framework depends heavily on Backbone.js. Backbone.js is an Model View Controller (MVC) framework that automatically synchronises widgets defined in the backend with generic Backbone.js models in the frontend: the previously defined `_view_name` characteristic is used by the widget framework to display the corresponding Backbone.js-View and link it to the model.

Import `@jupyter-widgets/base`

First you have to use the `@jupyter-widgets/base` module with the `define` method of RequireJS.

[2]:
```
%%javascript
define('hello', ["@jupyter-widgets/base"], function(widgets) {

});
```

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

Define view

Next we define the widget view class and we inherit from `DOMWidgetView` with the `.extend` method.

[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` method

Finally, we still have to override the basic `render` method to define a custom rendering logic. A handle to the standard DOM element of the widget can be called with `this.el`. The `el` property is the DOM element associated with the view.

[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

The widget can now be displayed like any other widget with

[5]:
```
HelloWidget()
```

```
HelloWidget()
```

10.4.4 Stateful widget

There's not much you can do with the example above. To change this, you have to make the widget stateful. Instead of a static Hello World! Message, a string specified by the backend should be displayed. To do this, a new traitlet is first added. Use the name of `value` here to stay consistent with the rest of the widget framework and to allow your widget to be used with interaction.

10.4.5 Create Jupyter widgets from a template

A Cookiecutter is available with widget-cookiecutter. It contains an implementation for a placeholder widget *Hello World*. It also makes it easier for you to pack and distribute your Jupyter widgets.

10.5 ipywidgets libraries

Popular widget libraries are

qplot
2-D plotting library for Jupyter notebooks

- bqplot

ipycanvas
Interactive canvas elements in Jupyter notebooks

10.5.1 ipycanvas

provides the Web-Canvas-API. However, there are some differences:

- The Canvas widget exposes the CanvasRenderingContext2D API directly
- The entire API is written in `snake_case` instead of `camelCase`, so for example `canvas.fillStyle = 'red'` written in Python becomes `canvas.fill_style = 'red'` in JavaScript.

Installation

```
$ pipenv install ipycanvas
Installing ipycanvas...
...
```

Creating canvas elements

Before we can start creating canvas elements, first a note about the canvas grid. The origin of a grid is in the upper left corner at the coordinate `(0,0)`. All elements are placed relative to this origin.

There are four methods of drawing rectangles:

- `fill_rect(x, y, width, height=None)` draws a filled rectangle
- `stroke_rect(x, y, width, height=None)` draws a rectangular outline
- `fill_rects(x, y, width, height=None)` draws filled rectangles
- `stroke_rects(x, y, width, height=None)` draws rectangular outlines

With `height=None`, the same value is used as with `width`.

For `*_rects`, `x`, `y`, `width` and `height` are integers, lists of integers or numpy arrays.

[17]:
```
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)
```

(continues on next page)

(continued from previous page)

```
canvas
```

```
Canvas()
```

[18]:
```
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()
```

[19]:
```
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()
```

Since `Canvas` is an ipywidget, it can

- appear several times in a notebook
- change the attributes
- Link changed attributes to other widget attributes

Delete canvas

[20]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(120, 100))
# Perform some drawings...
canvas.clear()
```

[21]:
```
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()
```

Shapes

The available drawing commands are:

- `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)`:

Draw circles

There are four different ways to draw circles:

- `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)`

With `*_arcs`, `x`, `y` and `radius` are NumPy arrays, lists or scalar values.

[22]:
```
from math import pi

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()
```

Drawing paths

A path is a list of points connected by line segments that can be different shapes, straight or curved, closed or open, different widths and colors. The following functions are available:

```
begin_path ()
close_path () adds a straight line to the path leading to the beginning of the␣
↪current path
stroke () draws the shape along the contour
fill (rule) draws the shape using a fill within the path
```

- `begin_path()` creates a new path
- `close_path()` adds a straight line to the path leading to the beginning of the current path
- `stroke()` draws the shape along the contour
- `fill(rule)` draws the shape using a fill within the path

[23]:
```
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()
```

Examples

arc

[24]:
```
from math import pi

from ipycanvas import Canvas


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()
```

bezier_curve_to

[25]:
```
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()
```

Styles and colors

Colors

`Canvas` has two color attributes, one for strokes and one for areas; the transparency can also be changed.

- `stroke_style` expects a valid HTML color. The default is black.
- `fill_style` expects a valid HTML color. The default is black.
- `global_alpha` indicates the transparency. The default is 1.0.

Lines

Line style

Lines can be described by the following attributes:

- `line_width`
- `line_cap`
- `line_join`
- `miter_limit`
- `get_line_dash()`
- `set_line_dash(segments)`
- `line_dash_offset`

Line width

[26]:
```
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()
```

Line end

[27]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(320, 360))

# Possible line_cap values
line_caps = ["butt", "round", "square"]

canvas.scale(2)

# Draw guides
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()
```

Line connection

defines the appearance of the corners where lines meet.

[28]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(320, 360))

# Possible line_join values
line_joins = ["round", "bevel", "miter"]
```

(continues on next page)

(continued from previous page)

```
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

    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()
```

Line pattern

[29]:
```
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()
```

text

There are two methods of designing text:

- `fill_text(text, x, y, max_width=None)`
- `stroke_text(text, x, y, max_width=None)`

[30]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(400, 50))

canvas.font = "32px serif"
canvas.fill_text("Drawing from Python is cool!", 10, 32)
canvas
```

```
Canvas()
```

[31]:
```
from ipycanvas import Canvas


canvas = Canvas(size=(400, 50))

canvas.font = "32px serif"
canvas.stroke_text("Hello There!", 10, 32)
canvas
```

```
Canvas()
```

font indicates the current text style. The default value is “12px sans-serif”. text_align specifies the text alignment. Possible values are “start”, “end”, “left”, “right” or “center”. text_baseline indicates the alignment with the baseline. Possible values are “top”, “hanging”, “middle”, “alphabetic”, “ideographic” and “bottom”. The default value is “alphabetic”. direction indicates the text direction. Possible values are “ltr”, “rtl”, “inherit”. The default value is “inherit”.

Of course, fill_style and stroke_style can also be used to color the text.

- `font` indicates the current text style. The default value is `"12px sans-serif"`.
- `text_align` specifies the text alignment. Possible values are `"start"`, `"end"`, `"left"`, `"right"` or `"center"` .
- `text_baseline` indicates the alignment with the baseline. Possible values are `"top"`, `"hanging"`, `"middle"`, `"alphabetic"`, `"ideographic"` and `"bottom"`. The default value is `"alphabetic"`.
- `direction` indicates the text direction. Possible values are `"ltr"`, `"rtl"`, `"inherit"`. The default value is `"inherit"`.

Of course, `fill_style` and `stroke_style` can also be used to color the text.

Images

... from a Numpy array

With `put_image_data(image_data, dx, dy)` an image can be displayed, where `image_data` specifies a Numpy array and `dx` and `dy` the upper left corner of the image.

[32]:
```
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)

canvas = Canvas(size=(image_data.shape[0], image_data.shape[1]))
canvas.put_image_data(image_data, 0, 0)

canvas
```

```
Canvas()
```

Status

The status can be specified with two values:

- `save()` saves the status of the canvas element.
- `restore()` restores the last saved status of the canvas element. This method can be called as often as required.

Transformations

- `translate(x, y)` moves the canvas element
- `rotate(angle)` rotates the canvas element clockwise
- `scale(x, y=None)` scales the canvas element

See also:

- API reference

`pythreejs`
Jupyter Three.js bridge

- pythreejs

ipyvolume
IPyvolume is a Python library for visualizing 3D volumes and glyphs (for example 3D scatter plots).

- ipyvolume

ipyleaflet
Jupyter-Leaflet.js bridge

- ipyleaflet

ipywebrtc
WebRTC and MediaStream API for Jupyter notebooks

10.5.2 ipywebrtc

ipywebrtc provides WebRTC and the MediaStream-API in Jupyter notebooks. This allows e.g. to create screenshots from a MediaStream and analyse them further with skimage. With `ipywebrtc` you can not only read video, image, audio and widget data but also record stream objects. It even provides a simple chat function.

Installation

`ipywebrtc` is installed in both the kernel and the Jupyter environment:

```
$ pipenv install ipywebrtc
```

Examples

Example `VideoStream`

[1]:
```
from ipywebrtc import VideoStream
```

[2]:
```
bbb = VideoStream.from_url("https://github.com/maartenbreddels/ipywebrtc/raw/master/
↪docs/source/Big.Buck.Bunny.mp4")
```

[3]:
```
bbb
```

```
VideoStream(video=Video(value=b'https://github.com/maartenbreddels/ipywebrtc/raw/
↪master/docs/source/Big.Buck.B...
```

Record

A record button can be created with `MediaRecorder.record`, for videos with:

[4]:
```
from ipywebrtc import VideoRecorder


recorder = VideoRecorder(stream=bbb)
recorder
```

```
VideoRecorder(stream=VideoStream(video=Video(value=b'https://github.com/
↪maartenbreddels/ipywebrtc/raw/master/d...
```

Save

The stream can either be saved via the download button or programmatically, for example with:

[ ]:
```
recorder.save("bbb.webm")
```

Example `WidgetStream`

A WidgetStream creates a MediaStream from any widget.

[6]:
```
from ipywebrtc import VideoStream, WidgetStream
```

[7]:
```
from pythreejs import (
    AmbientLight,
    DirectionalLight,
    Mesh,
    MeshLambertMaterial,
    OrbitControls,
    PerspectiveCamera,
    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,...
```

The following `webgl_stream` is updated after something changes in the scene above. You can do this by moving the ball with the mouse.

[8]:
```
webgl_stream = WidgetStream(widget=renderer)
webgl_stream
```

```
WidgetStream(widget=Renderer(camera=PerspectiveCamera(children=(DirectionalLight(color=
↪'white', intensity=0.5,...
```

Alternatively, you can also use a slider:

[9]:
```
from ipywidgets import FloatSlider


slider = FloatSlider(
    value=7.5,
    step=0.1,
    description="Test:",
    disabled=False,
    continuous_update=False,
    orientation="horizontal",
    readout=True,
    readout_format=".1f",
)

slider
```

```
FloatSlider(value=7.5, continuous_update=False, description='Test:', readout_format=
↪'.1f')
```

ipysheet
Interactive tables to use IPython widgets in tables of Jupyter notebooks.

10.5.3 ipysheet

ipysheet connects ipywidgets with tabular data. It basically adds two widgets: a *Cell widget* and a *Sheet widget*. There are also auxiliary functions for creating table rows and columns as well as for formatting and designing cells.

Installation

`ipysheet` can be easily installed with Pipenv:

```
$ pipenv install ipysheet
```

Import

[1]:
```
import ipysheet
```

Cell formatting

[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', ...
```

Examples

Interactive table

[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]])
```

Table search

[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
```

(continues on next page)

(continued from previous page)

```
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(column_
↪end=0, column_start=0, row_star...
```

Plot editable tables

[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"},
}

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...
```

For further reading

- Interactive spreadsheets in Jupyter
- GitHub
- Docs

ipydatagrid
ipydatagrid is a fast and versatile datagrid widget.

ipyvuetify
Vuetify widgets in Jupyter notebooks

10.5.4 ipyvuetify

ipyvuetify provides Jupyter widgets based on vuetify UI components and implementing Google's Material Design with the Vue.js-Framework framework.

Installation

```
$ pipenv install ipyvuetify
Installing ipyvuetify...
...
$ pipenv run jupyter nbextension enable --py --sys-prefix ipyvuetify
Enabling notebook extension jupyter-vuetify/extension...
      - Validating: OK
```

Examples

Imports

[1]:
```
from threading import Timer

import ipyvuetify as v
import ipywidgets

from traitlets import Any, List, Unicode
```

Menu

[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 = [
    v.ListItem(children=[v.ListItemTitle(children=[f"Item {i}"])])
    for i in range(1, 5)
```

(continues on next page)

(continued from previous page)

```
]

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'], layout=No...
```

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"]),
    ]
)
```

```
Layout(children=[Btn(children=['primary'], color='primary', layout=None),␣
↪Btn(children=['error'], color='error...
```

[4]:
```
v.Layout(
    children=[
        v.Btn(color="primary", flat=True, children=["flat"]),
        v.Btn(color="primary", round=True, children=["round"]),
```

(continues on next page)

(continued from previous page)

```
        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', layout=None),␣
↪Btn(children=['round'], color='primary'...
```

[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'], layout=None, loading=False)], layout=None)
```

[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"])]),
```

(continues on next page)

(continued from previous page)

```
        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
↪'], layout=None)], layout=...
```

[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=[
        v.Btn(slot='activator', color='primary', children=[
           'tooltip'
        ]),
        'Insert tooltip text here'
    ])
])
```

```
Layout(children=[Btn(children=[Icon(children=['fingerprint'], layout=None,␣
↪left=True), 'Icon left'], color='pr...
```

[8]:
```
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"]),
            ],
        ),
```

(continues on next page)

(continued from previous page)

```
        v.Tooltip(
            bottom=True,
            children=[
                v.Btn(slot="activator", color="primary", children=["tooltip"]),
                "Insert tooltip text here",
            ],
        ),
    ]
)
```

```
Layout(children=[Btn(children=[Icon(children=['fingerprint'], layout=None,␣
↪left=True), 'Icon left'], color='pr...
```

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)

ipywidgets.jslink((slider, "v_model"), (slider2, "v_model"))

v.Container(
    children=[
        slider,
        slider2,
    ]
)
```

```
Container(children=[Slider(layout=None, v_model=25), Slider(layout=None, thumb_
↪label=True, v_model=25)], layou...
```

Tabs

[10]:
```
lorum_ipsum = "Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do␣
↪eiusmod tempor incididunt ut labore et dolore magna aliqua."
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'], layout=None), Tab(children=['Tab 2'],␣
↪layout=None), Tab(children=['Tab ...
```

Accordion

[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'], l...
```

You can search for all available components and attributes in the Vuetify documentation. Ipyvuetify is based on the syntax of Vue.js- and Vuetify, but there are also some differences:

Description	Vuetify	ipyvuetify
Component names are written in CamelCase and the v- prefix is removed	`<v-list-tile .../>`	`ListTile(...)`
Child components and text are defined in the child traitlet	`<v-btn>text <v-icon .../></v-btn>`	`Btn(children=['text', Icon(...)])`
Flag attributes require a Boolean value	`<v-btn round ...`	`Btn(round=True ...`
Attributes are snake_case	`<v-menu offset-y ...`	`Menu(offset_y=True ...`
The `v_model` attribute (value in ipywidgets) receives the value directly	`<v-slider v-model= "some_property" ...`	`Slider(v_model=25...`
The scope of `slot` cannot currently be specified	`<v-menu><template slot:activator="{ on }"><v-btn v-on=on>`	`Menu(children=[Btn(slot='activator', ...), ...]`
Event listeners are defined with `on_event`	`<v-btn @click='someMethod()' ...`	`def some_method (widget, event, data): mit button. on_event('click', some_method)`
Regular HTML tags can be created with the `Html` class	`<div>...</div>`	`Html(tag='div', children=[...])`
An underscore must be added to the `class` and `style` attributes	`<v-btn class="mr-3" style="..." >`	`Btn(class_='mr-3', style_='...')`

VuetifyTemplate

You can get a closer match with the Vue/Vuetify API with `VuetifyTemplate`. For this you create a subclass of `VuetifyTemplate` and define your own traitlets. The traitlets can be accessed via the template as if they were in a Vue model. Methods can be defined with the prefix `vue_`, for example `def vue_button_click(self, data)`, which can then be called with `@click="button_click(e)"`. In the following I show you a table with search, sorting and number of lines:

[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)
    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">
```

(continues on next page)

(continued from previous page)

```
                    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]
        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"]
            ),
        },
```

(continues on next page)

(continued from previous page)

```
    ],
    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
↪'], layout=None, single_lin...
```

ipympl
: ipympl or jupyter-matplotlib offer interactive widgets for matplotlib.

10.5.5 ipympl

Since the Jupyter widget ecosystem is developing too quickly, the Matplotlib developers have decided to outsource the support to a separate module: ipympl or jupyter-matplotlib.

Installation

ipympl is installed in both the kernel and the Jupyter environment

```
$ pipenv install ipympl
Installing ipympl...
Adding ipympl to Pipfile's [packages]...
✓ Installation Succeeded
...
```

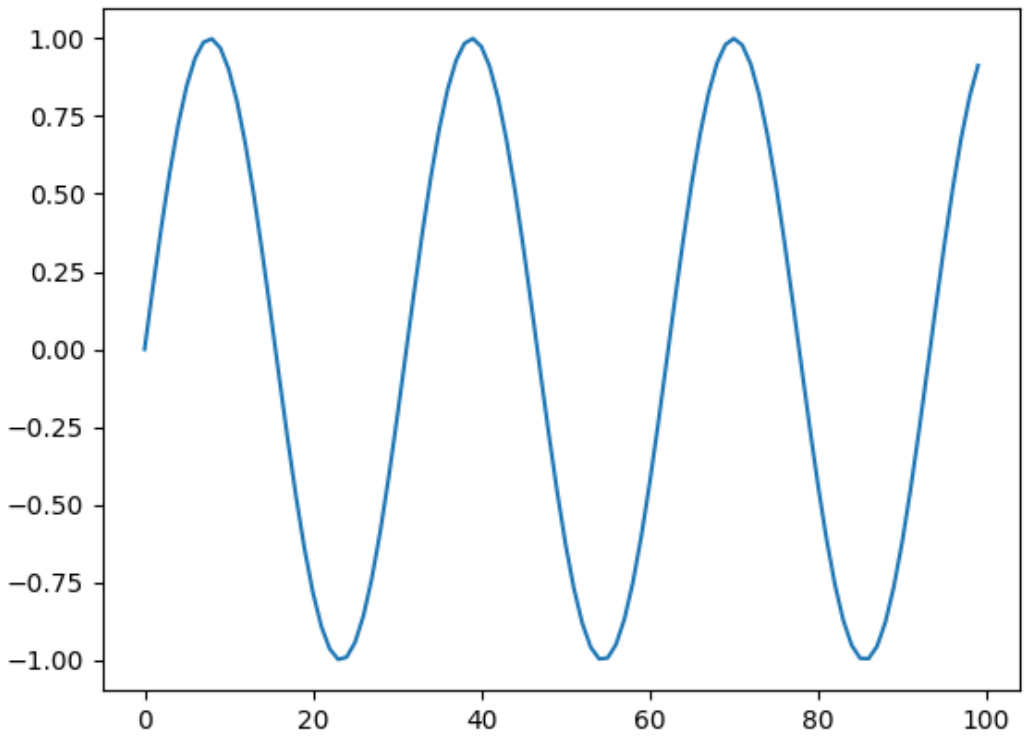
Then you can activate the Jupyter backend in notebooks by using the following *Matplotlib-Magic*:

[1]:
```
%matplotlib widget
```

Examples

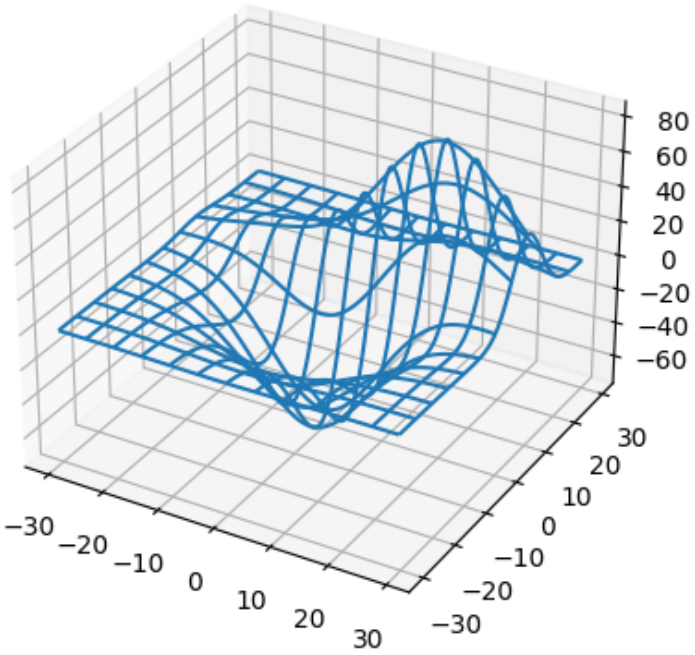
Simple Matplotlib interaction

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


plt.figure(1)
plt.plot(np.sin(np.linspace(0, 20, 100)))
plt.show()
```



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()
```



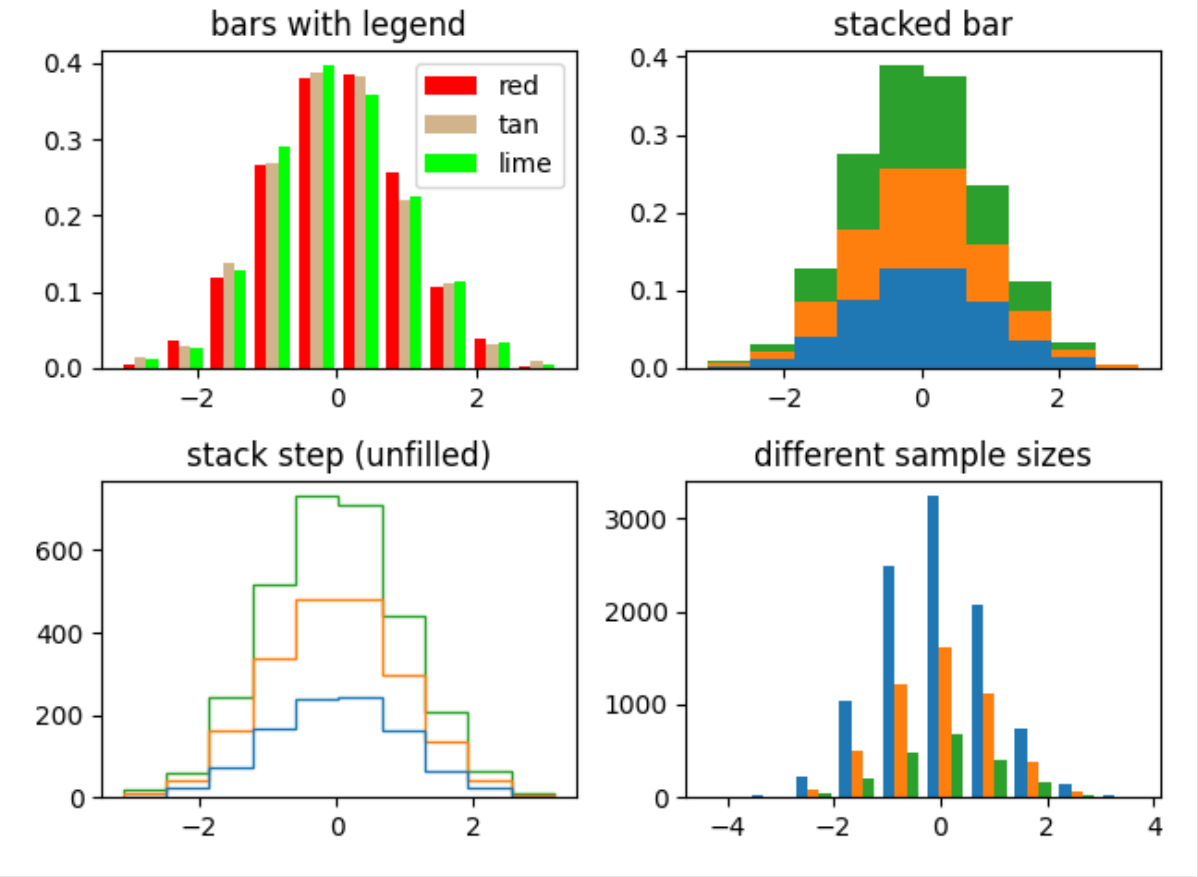
More complex example from the Matplotlib gallery

[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})
ax0.set_title("bars with legend")

ax1.hist(x, n_bins, density=1, histtype="bar", stacked=True)
ax1.set_title("stacked bar")
```

(continues on next page)

(continued from previous page)

```
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()
```



10.6 Embed Jupyter widgets

Jupyter widgets can be serialised and then embedded in other contexts:

- static web pages
- Sphinx documentation
- HTML converted notebooks on Nbviewer

The npm package `@jupyter-widgets/html-manager` allows embedding in two different ways:

- embedding the standard elements that can be used on any website

- embedding with RequireJS also for custom widgets.

10.6.1 Embed widgets in HTML pages

The widgets menu provides several options for this:

Save Notebook Widget State
A notebook file is saved with the current widget status as metadata. This allows to be rendered with the widgets in the browser.

Clear Notebook Widget State
The widget status metadata is deleted from the notebook file.

Embed widgets
The menu item offers a dialog box with an HTML page on which the current widgets are embedded. The RequireJS embedder is used to support custom widgets.

Note: The first script tag loads RequireJS from a CDN. However, RequireJS should be made available on the site itself and this script tag should be deleted.

Note: The second script tag loads the RequireJS widget embedder. This defines suitable modules and then sets up a function for rendering all widget views contained on the page.

If you only embed standard widgets and don't use RequireJS, you can replace the first two script tags with a script tag that loads the standard script.

Download Widget State
The option downloads a JSON file that contains the serialized status of all widget models currently in use in the `application/vnd.jupyter.widget-state+json` format specified in the `@jupyter-widgets/schema` npm package.

10.6.2 Sphinx integration

Jupyter Sphinx

jupyter_sphinx enables jupyter-specific functions in Sphinx. It can be installed with `pip`.

Configuration

Adds `jupyter_sphinx.embed_widgets` to the list of extensions in the `conf.py` file.

Then you can use the following directives in reStructuredText:

`ipywidgets-setup`

```
from ipywidgets import Button, IntSlider, VBox, jsdlink
```

`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])
```

Example

```
.. 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])
```

Options

The `ipywidgets-setup` and `ipywidgets-display` directives have the following options:

`ipywidgets-setup`
: with the option `:show:` to display the setup code as a code block

`ipywidgets-display`
: with the following options:

`:hide-code:`
: doesn't show the code, only the widget

Widget

`:code-below:`
: shows the code after the widget

`:alt:`
: Alternate text if the widget cannot be rendered

See also:

Options

CHAPTER

ELEVEN

NBEXTENSIONS

Jupyter Notebook Extensions contains a collection of extensions. These are mostly written in Javascript and are loaded locally in your browser.

See also:

- Docs
- Github

11.1 Installation

1. Installation with Pipenv:

```
$ pipenv install jupyter_contrib_nbextensions
Installing jupyter_contrib_nbextensions...
...
```

2. Installation of the associated Javascript and CSS files:

```
$ pipenv 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. Check the installation:

```
$ pipenv 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
```

(continues on next page)

(continued from previous page)

```
    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

```
$ pipenv run 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
...
$ pipenv run jupyter nbextension enable --py latex_envs --user
Enabling notebook extension latex_envs/latex_envs...
      - Validating: OK
```

5. yapf Code Prettyfier

   … for Python:

```
$ pipenv install yapf
Installing yapf...
Collecting yapf
  Downloading https://files.pythonhosted.org/packages/79/22/
↪d711c0803b6c3cc8c96eb54509f23fec1e3c078d5bfc6eb11094e762e7bc/yapf-0.28.0-py2.py3-
↪none-any.whl (180kB)
Installing collected packages: yapf
Successfully installed yapf-0.28.0
```

   … for 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
```

   … for 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

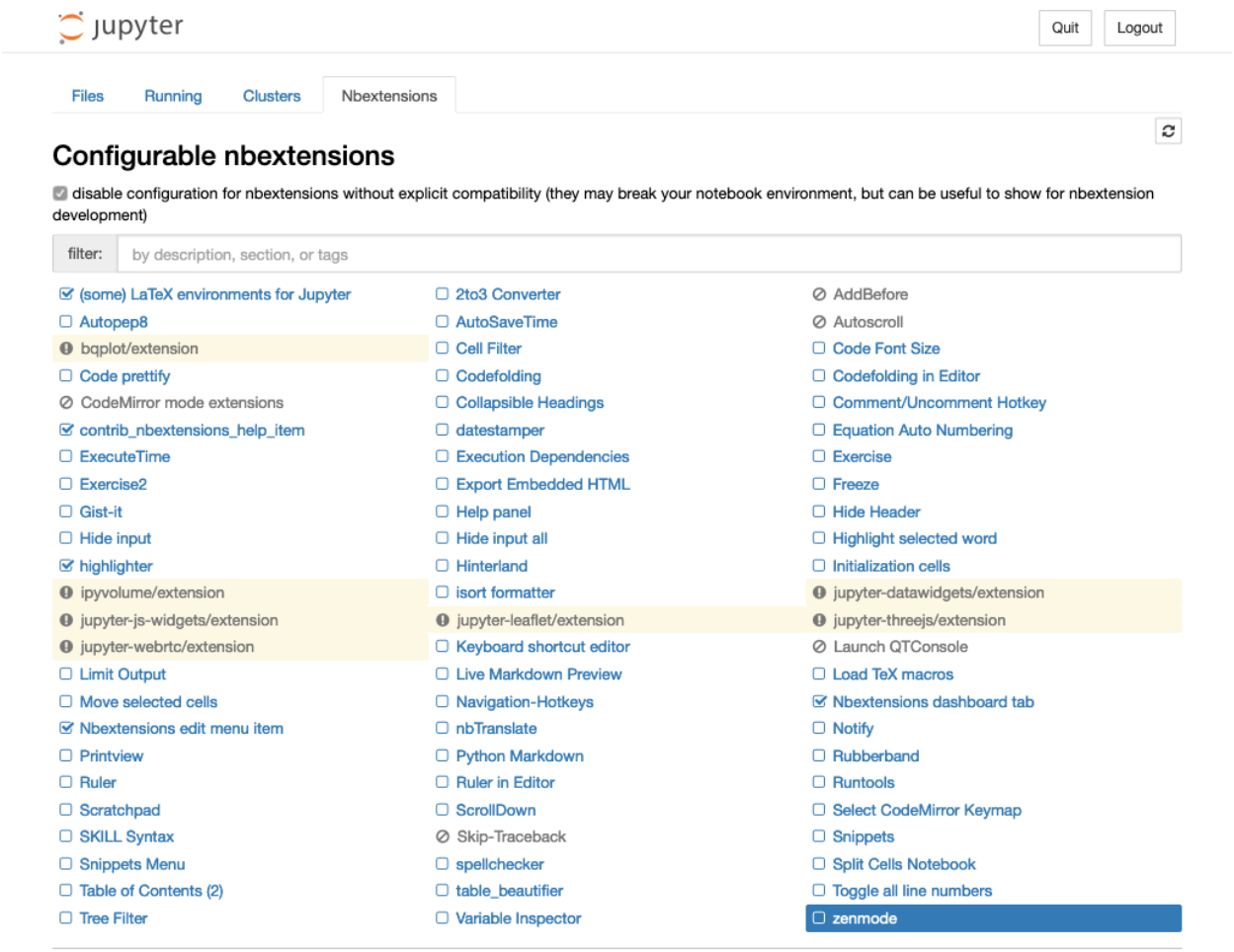
```
$ pipenv run jupyter nbextension install https://rawgit.com/jfbercher/small_
↪nbextensions/master/highlighter.zip  --user
Downloading: https://rawgit.com/jfbercher/small_nbextensions/master/highlighter.zip␣
↪-> /var/folders/_4/cs4t3m8d4ys8lcs67r3lghtw0000gn/T/tmpn9qrcrdz/highlighter.zip
Extracting: /var/folders/_4/cs4t3m8d4ys8lcs67r3lghtw0000gn/T/tmpn9qrcrdz/
↪highlighter.zip -> /Users/veit/Library/Jupyter/nbextensions
$ pipenv run jupyter nbextension enable highlighter/highlighter
Enabling notebook extension highlighter/highlighter...
      - Validating: OK
```

7. nbTranslate

```
$ pipenv install jupyter_latex_envs --upgrade --user
Installing jupyter_latex_envs...
...
$ pipenv run jupyter nbextension install --py latex_envs --user
Installing /srv/jupyter/.local/share/virtualenvs/jupyterhub-aFv4x91W/lib/python3.5/
↪site-packages/latex_envs/static -> latex_envs
...
$ pipenv run jupyter nbextension enable --py latex_envs
```

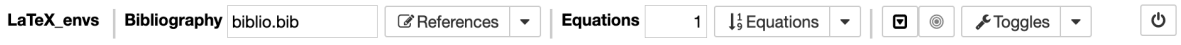
11.2 List of extensions

You can activate and configure the notebook extensions by clicking on the *Nbextensions* tab. There you have access to the extensions, which can be activated/deactivated via checkboxes. In addition, documentation and configuration options are displayed for each extension.



Hereinafter I will give a brief overview of some of the notebook extensions.

(some) LaTeX environments for Jupyter notebook
: enables the use of Markdown cells for LaTeX commands and environments. In addition, two menus are added: *LaTeX_envs* for quick selection of the suitable LaTeX environment and *Some configuration options* for further options:



The notebook can then be exported as an HTML or LaTeX document.

The configuration of the LaTeX environments is done in `user_envs.json` and for the styles in `latex_env.css`. Additional environments can be added in `user_envs.json` or in `thmsInNb4.js` (→ LaTeX-Environments doc).

jupyter-autopep8
: formats/beautifies Python code in cells. The extension uses autopep8 and can therefore only be used with a Python kernel.

A Code Prettifier
: formats/beautifies code notebook code cells. The current notebook kernel is used, which is why the Prettifier package used must be available in this kernel. Sample implementations are provided for ipython, R, and Javascript kernels.

Limit Output
: limits the number of characters that a code cell outputs as text or HTML. This also breaks endless loops. You

can set the number of characters with the `ConfigManager`:

```
from notebook.services.config import ConfigManager
cm = ConfigManager().update('notebook', {'limit_output': 1000})
```

Nbextensions edit menu item

adds an edit menu to open the configuration page of `nbextensions`.

Printview

adds an icon to display the print preview of the current notebook in a new browser tab.

Ruler

adds a ruler after a certain number of characters. The number of characters can be specified with the `ConfigManager`:

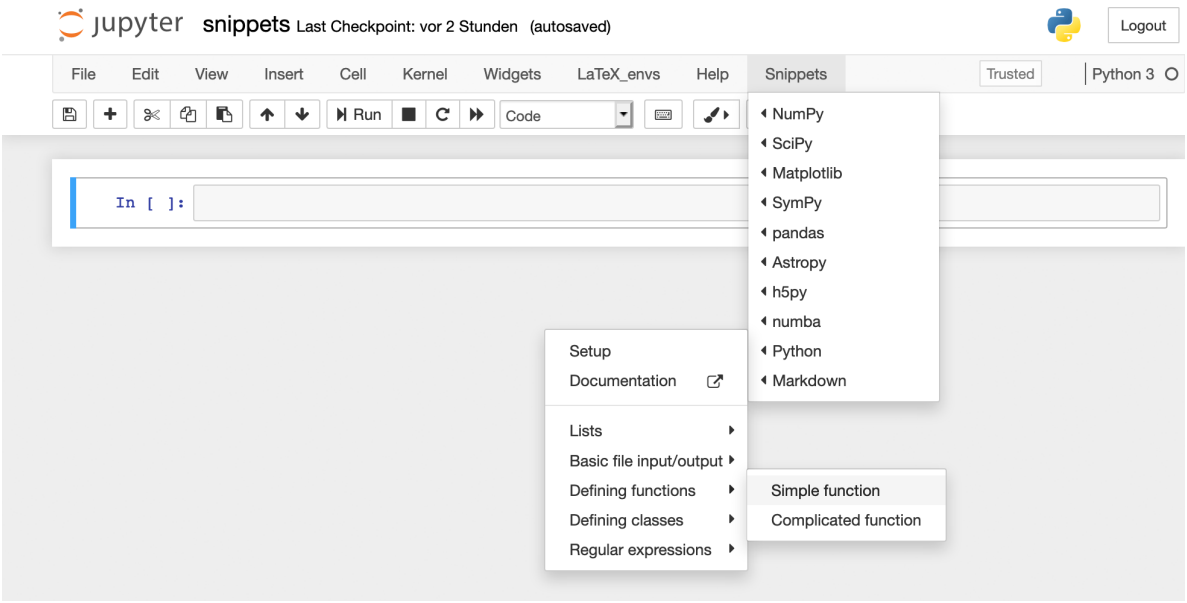
```
from notebook.services.config import ConfigManager
ip = get_ipython()
cm = ConfigManager(parent=ip)
cm.update('notebook', {"ruler_column": [80]})
```

Scratchpad notebook extension

adds a note cell to the notebook. In this cell you can run code from the current kernel without changing the document.

Snippets

adds a configurable menu item to notebooks to insert snippets, boilerplate and code examples.



You can also define your own menu items, see also Snippets.

Table of Contents (2)

makes it possible to collect all headings and display them in a floating window, as a sidebar or in a navigation menu.

If headings shouldn't be displayed in the table of contents, you can do this with:

```
## My title <a class="tocSkip">
```

The table of contents can also be exported by specifying a corresponding template, e.g.

```
$ jupyter nbconvert mynotebook.ipynb --template toc2
```

General documentation on templates can be found in Customizing exporters.

Tree-filter
filters in the Jupyter dashboard by file name.

A 2to3 converter
converts Python2 to Python3 code in a code cell using the lib2to3 library.

Codefolding
enables code folding in code cells.

```
In [ ]: class MyClass(object):
            """
            This is a test class
            """
            def afun(param1):
            """something gets computed here"""
                return param1ram1**2
```

Usually code folding is retained when exporting with nbconvert. This can either be changed in `jupyter_nbconvert_config.py` with:

```
c.CodeFoldingPreprocessor.remove_folded_code=True = True
```

or on the command line with:

```
$ jupyter nbconvert --to html --CodeFoldingPreprocessor.remove_folded_code=True␣
↪mynotebook.ipynb
```

Collapsible Headings
enables notebooks to have collapsible sections separated by headings.

Datestamper
inserts the current time and date in one cell.

Hinterland
enables autocompletion.

Variable Inspector
collects all defined variables and displays them in a floating window.

Purpose
automatically loads a series of latex commands from the `latexdefs.tex` file when a notebook is opened.

11.3 Create plugin

In addition to the existing notebook extensions, other plugins can also be added. The directory in which `jupyter_contrib_nbextensions/nbextensions` is located can be found with `pip show`:

```
$ pipenv run pip show jupyter_contrib_nbextensions
Name: jupyter-contrib-nbextensions
Version: 0.5.1
Summary: A collection of Jupyter nbextensions.
Home-page: https://github.com/ipython-contrib/jupyter_contrib_nbextensions.git
```

(continues on next page)

(continued from previous page)

```
Author: ipython-contrib and jupyter-contrib developers
Author-email: jupytercontrib@gmail.com
License: BSD
Location: /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/lib/python3.7/
↪site-packages
Requires: lxml, jupyter-contrib-core, nbconvert, jupyter-latex-envs, jupyter-core,␣
↪pyyaml, jupyter-nbextensions-configurator, notebook, traitlets, jupyter-highlight-
↪selected-word, tornado, ipython-genutils
Required-by:
```

This directory contains the individual notebook extensions, e.g. with the following structure:

```
$ tree
.
├── main.js
├── main.yaml
└── readme.md
```

main.js

contains the actual logic of the extension, e.g .:

```
define([
    'require',
    'base/js/namespace',
], function (
    requirejs
    $,
    Jupyter,
) {
    "use strict";

    // 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);
    };
```

(continues on next page)

(continued from previous page)

```
    return {
        load_ipython_extension : load_ipython_extension
    };
});
```

main.yaml
yaml file that describes the extension for the Jupyter Extensions Configurator.

```
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
```

More information about the options supported by the configurator can be found on GitHub: jupyter_nbextensions_configurator.

readme.md
Markdown file that describes the extension and how it can be used. This is also displayed in the *Nbextensions* tab.

See also:

- Notebook extension structure

11.3.1 Setup Jupyter Notebook Extension

This is an extension that fixes some problems when working with notebooks that Joel Grus presented at JupyterCon 2018: I Don't Like Notebooks:

- it asks you to name the notebook
- it creates a template to improve the documentation
- it imports and configures frequently used libraries

Installation

1. Find out where the notebook extensions are installed:

```
$ pipenv run pip show jupyter_contrib_nbextensions
Name: jupyter-contrib-nbextensions
Version: 0.5.1
Summary: A collection of Jupyter nbextensions.
Home-page: https://github.com/ipython-contrib/jupyter_contrib_nbextensions.git
Author: ipython-contrib and jupyter-contrib developers
Author-email: jupytercontrib@gmail.com
License: BSD
Location: /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/lib/
↪python3.7/site-packages
```

(continues on next page)

(continued from previous page)

```
Requires: lxml, jupyter-contrib-core, nbconvert, jupyter-latex-envs, jupyter-core,␣
↪pyyaml, jupyter-nbextensions-configurator, notebook, traitlets, jupyter-highlight-
↪selected-word, tornado, ipython-genutils
Required-by:
```

2. Download the Setup directory in `jupyter_contrib_nbextensions/nbextensions/`.
3. Install the extension with

```
$ pipenv run 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. Activate the *Setup* extension in *Nbextensions*.

Finally you can create a new notebook, which then has the following structure: *setup.ipynb*.

See also:

- 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` is based on GoldenLayout, a multi-screen layout manager for web applications.

11.5.1 Installation

`ipylayout` can be easily installed with pipenv:

```
$ pipenv install ipylayout
Installing ipylayout...
...
```

If not already done, `ipywidgets` will also be installed.

11.5.2 Example

For the following example you also need the Python packages `ipyleaflet` and `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 0x110a4df10>
```

[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)')
```

(continues on next page)

(continued from previous page)

```
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 Webanwendungen.

Installation

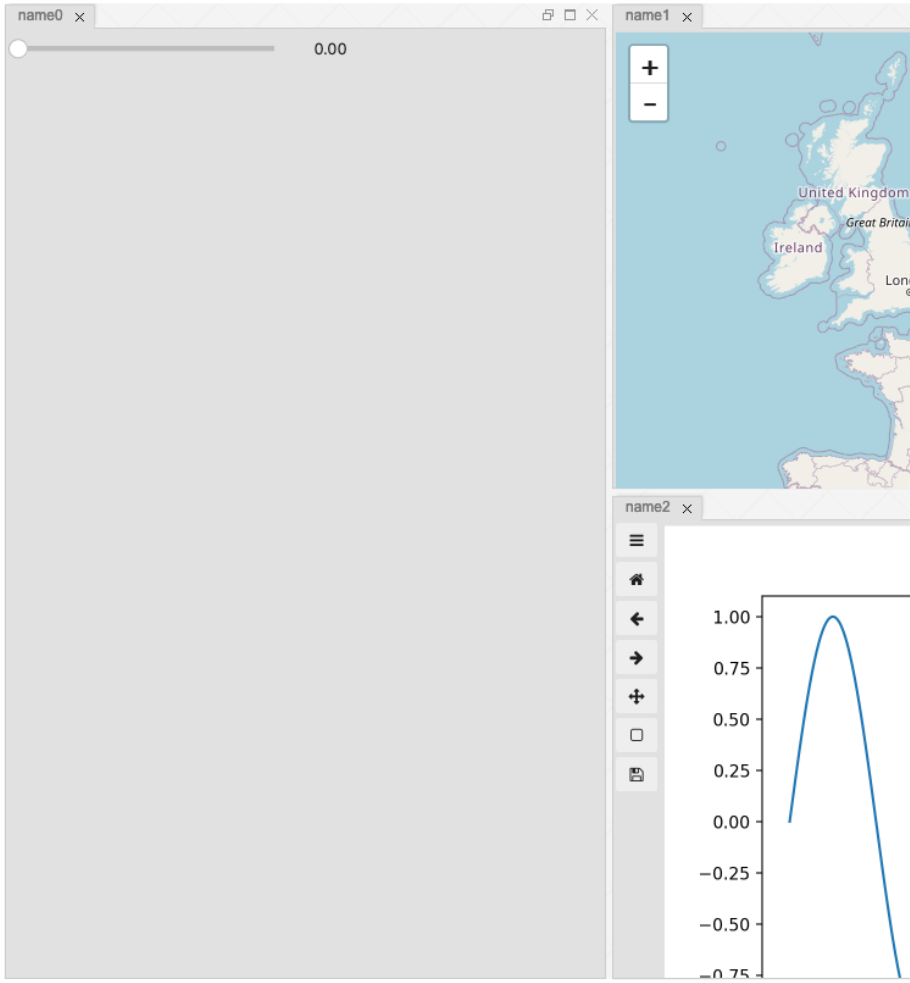
`ipylayout` kann einfach mit pipenv installiert werden:

```
$ 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 `ipympl`.



`ipylayout` can also be used together with *Voilà*:

CHAPTER

TWELVE

VISUALISE DATA

We have outsourced the visualisation of data to a separate tutorial: PyViz Tutorial.

CHAPTER

THIRTEEN

DASHBOARDS

Jupyter Dashboards Layout Extension
: Add-on for Jupyter notebooks, with which outputs (text, plots, widgets, ETC.) can be arranged in a design grid or in report form.

Appmode
: Jupyter extension that turns notebooks into web applications.

nbviewer
: It's great for viewing static reports, but it can also be used in conjunction with *interaktiven Widgets*.

Panel
: was developed on the basis of Bokeh and Param and offers a toolkit especially for creating apps and dashboards, which not only supports bokeh plots, see also Panel: A high-level app and dashboarding solution for the PyData ecosystem.

Voilà
: was developed by QuantStack, see also And voilà!.

jupyter-flex
: Jupyter extension that turns notebooks into dashboards.

With this tabular overview you can quickly compare the activities and licenses of the various libraries.

Table 1: GitHub-Insights: Dashboards

Name	Stars	Contributors	Commit activity	License
Jupyter Dashboards Layout Extension	Stars 980	contributors 16	commit activity 0/year	license not identifiable by github
Appmode	Stars 431	contributors 10	commit activity 6/year	license MIT
nbviewer	Stars 2.2k	contributors 83	commit activity 0/year	license not identifiable by github
Panel	Stars 4.2k	contributors 165	commit activity 1.1k/year	license BSD-3-Clause
Voilà	Stars 5.2k	contributors 64	commit activity 73/year	license not identifiable by github
jupyter-flex	Stars 312	contributors 4	commit activity 20/year	license Apache-2.0

13.1 Jupyter Dashboards

The Jupyter Dashboards Layout Extension is an add-on for Jupyter notebooks, with which outputs (text, plots, widgets, ETC.) can be arranged in a design grid or in report form. It saves the information on the layout directly in the notebook so that other users of this extension can also see the notebook in the same layout. For examples of dashboards, see Jupyter Dashboards Demos.

13.1.1 Use case

The Jupyter dashboards should solve the following problem:

1. Alice creates a Jupyter notebook with plots and interactive widgets.
2. Alice arranges the notebook cells in a grid or report format.
3. Alice provides the dashboard on a dashboard server.
4. Bob calls up the dashboard on the Jupyter Dashboards Server and interacts with Alice Dashboard application.
5. Alice updates her Jupyter notebook and then makes the dashboard available again on the dashboard server.

Note: For steps 3–5, Jupyter Dashboards Bundler and Jupyter Dashboards Server are also required; however, both are now retired and should not be used any longer.

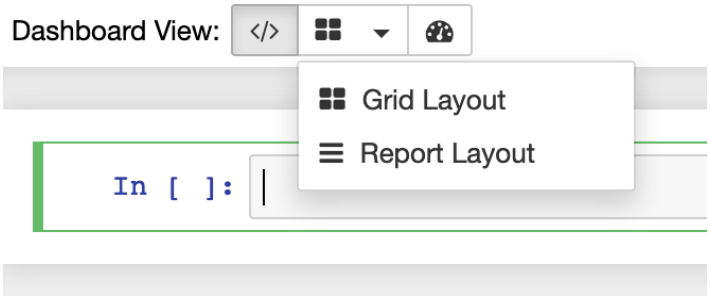
The roadmap for the *Voila-Gridstack-Template* is to support the entire specification for the Jupyter dashboards. Currently, however, the Voilà gridstack template is still in an early stage of development, see also And voilà!.

Installation of Jupyter dashboards

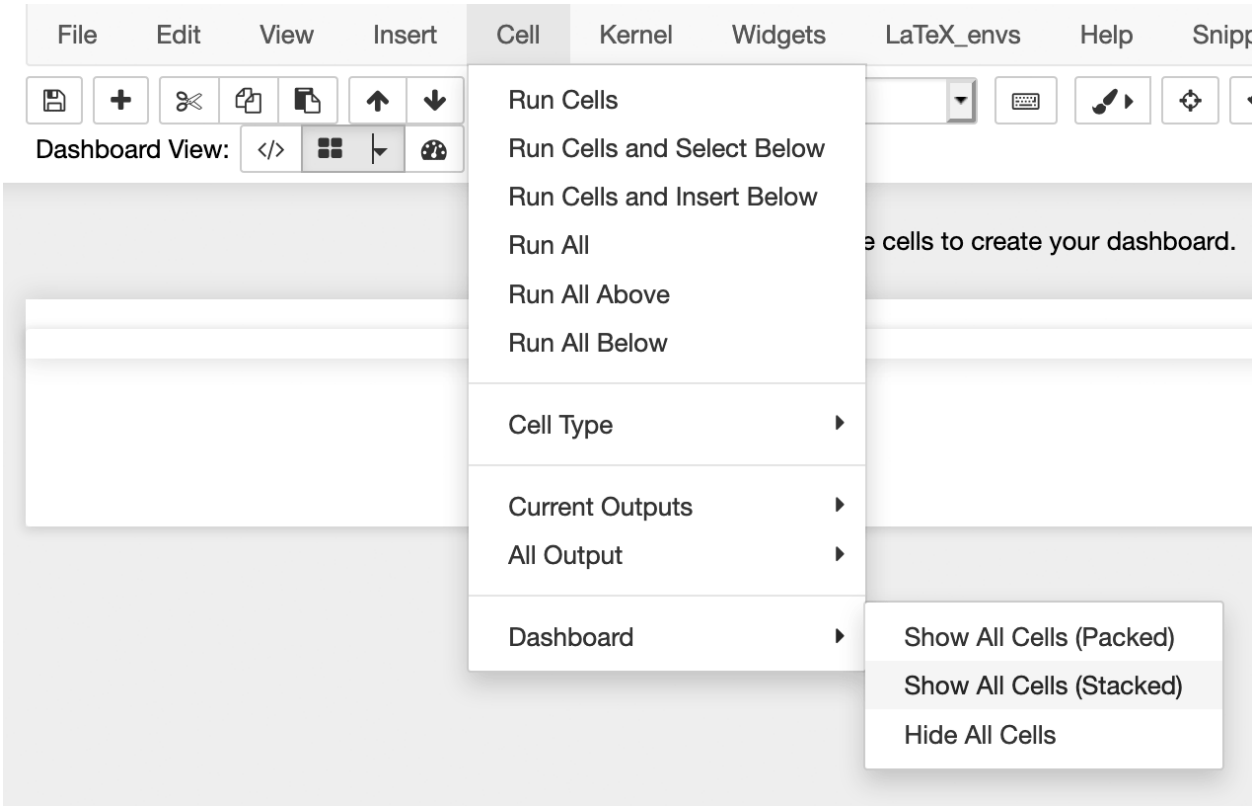
```
$ pipenv install jupyter_dashboards
Installing jupyter_dashboards...
...
$ pipenv run jupyter dashboards quick-setup --sys-prefix
...
Enabling notebook extension jupyter_dashboards/notebook/main...
      - Validating: OK
$ pipenv run jupyter nbextension enable jupyter_dashboards --py --sys-prefix
Enabling notebook extension jupyter_dashboards/notebook/main...
      - Validating: OK
```

Create dashboard layouts

You can use a normal notebook with markdown and code cells. When you run the cells, text, charts, widgets, etc. are generated. Then you can choose in the *Dashboard View* either *Grid Layout* or *Report Layout*:



With the *Grid Layout* you can change the size of the cells in the grid and move them. You can also use *Cell* → *Dashboard*:

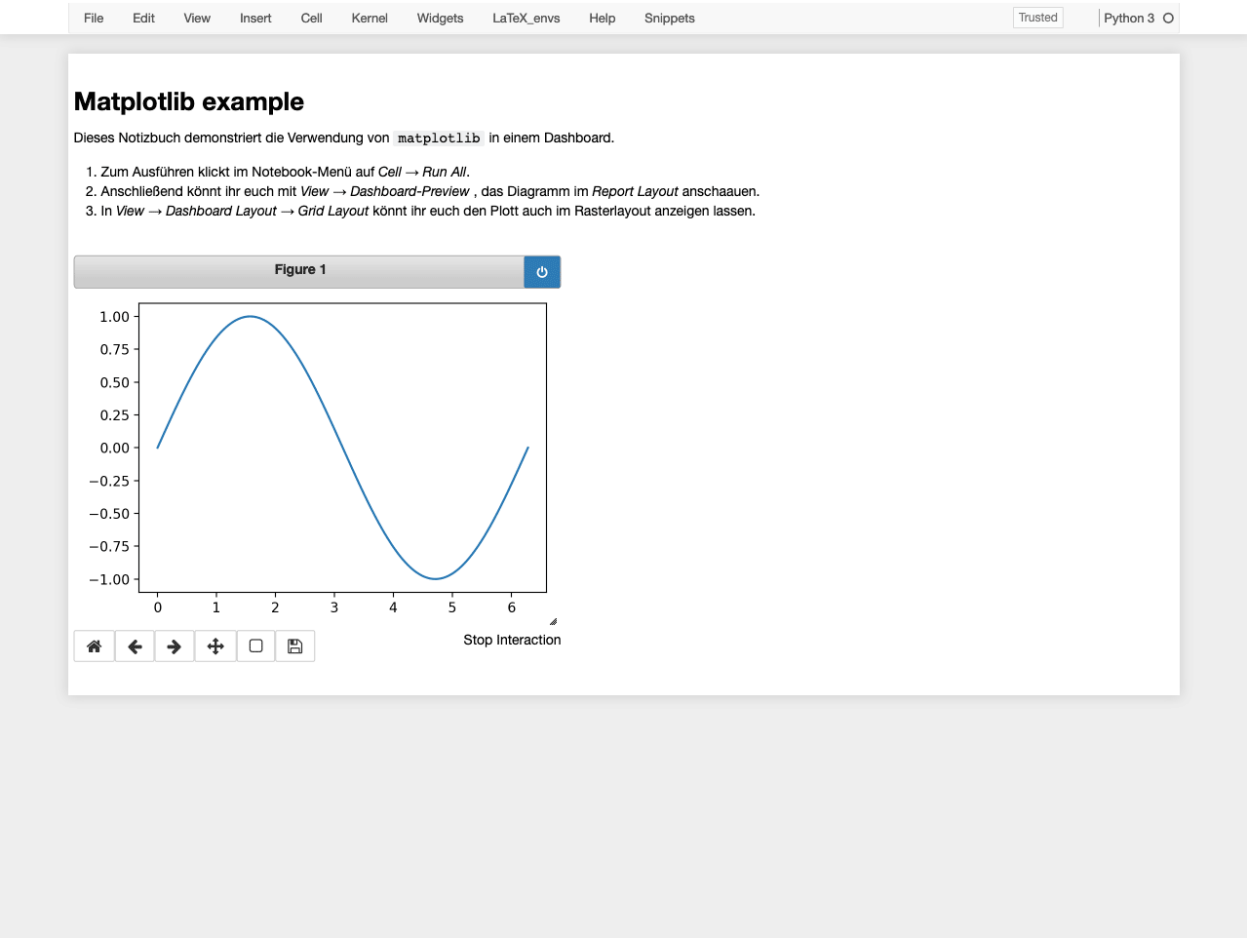


In the *Report Layout* you can show or hide cells.

In both layouts you can click on *MORE INFO* to get additional information:

> **Move cell:** Click and drag ✣ to move a cell. Hold **Shift** to drag from anywhere on a cell.
> **Resize cell:** Click and drag cell edges or corners to resize.
> **Hide cell:** Click ✖ to hide a cell from the dashboard view.
> **Show cell:** Click ✚ to return a hidden cell to the dashboard view.
> **Hide/Show all:** Select the options under **Cell > Dashboard** to hide or show all cells.
> **Edit cell:** Click ✎ to jump to the Notebook and edit the code.

With *Dashboard Preview* you get a preview, for example for the *Matplotlib example*:



View → Notebook brings you back to the notebook editor.

Matplotlib example

This notebook demonstrates the use of `matplotlib` in a Jupyter dashboard.

1. To run it, click on *Cell → Run All* in the Notebook menu.
2. Then you can use *View → Dashboard-Preview* to view the diagram in the *Report Layout*.
3. In *View → Dashboard Layout → Grid Layout* you can also display the plot in the *Grid Layout*.

[1]:
```
%matplotlib notebook
```

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

[3]:
```
x = np.linspace(0, 2*np.pi, 2000)
y = np.sin(x)
```

[4]:
```
fig, ax = plt.subplots(figsize=(5,3.5))
ax.plot(x, y)
plt.tight_layout()
```

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

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

13.2 Appmode

Jupyter extension that turns notebooks into web applications.

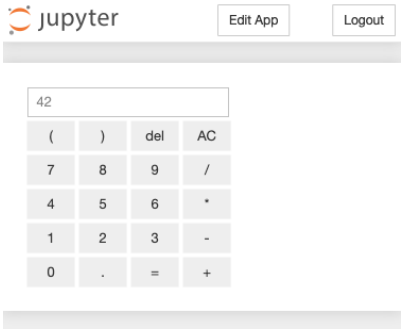
13.2.1 app-example.ipynb

[1]:
```
from __future__ import division

import ipywidgets as ipw


output = ipw.Text(
    placeholder="0", layout=ipw.Layout(width="190px"), disabled=True
)


def on_click(btn):
    if btn.description == "=":
        try:
            output.value = str(eval(output.value))
        except:
            output.value = "ERROR"
    elif btn.description == "AC":
        output.value = ""
    elif btn.description == "del":
        output.value = output.value[:-1]
    else:
        output.value = output.value + btn.description


def mk_btn(description):
    btn = ipw.Button(description=description, layout=ipw.Layout(width="45px"))
    btn.on_click(on_click)
    return btn


row0 = ipw.HBox([mk_btn(d) for d in ("(", ")", "del", "AC")])
row1 = ipw.HBox([mk_btn(d) for d in ("7", "8", "9", " / ")])
row2 = ipw.HBox([mk_btn(d) for d in ("4", "5", "6", " * ")])
row3 = ipw.HBox([mk_btn(d) for d in ("1", "2", "3", " - ")])
row4 = ipw.HBox([mk_btn(d) for d in ("0", ".", "=", " + ")])
ipw.VBox((output, row0, row1, row2, row3, row4))
```

```
VBox(children=(Text(value='', disabled=True, layout=Layout(width='190px'), placeholder='0
↪'), HBox(children=(Bu...
```

13.2.2 Example

When you click on *Appmode* the notebook *app-example.ipynb* becomes a clear web application for a calculator:



13.2.3 Installation

For the Jupyter service `appmode` must be installed with

```
$ pipenv install appmode
Installing appmode...
Collecting appmode
...
Installing collected packages: appmode
Successfully installed appmode-0.6.0
Adding appmode to Pipfile's [packages]...
Locking [dev-packages] dependencies...
Locking [packages] dependencies...
Updated Pipfile.lock (ea9a56)!
Installing dependencies from Pipfile.lock (ea9a56)...
     ████████████████████████████████████████████████████████████████ 128/128 -␣
↪00:09:58
...
$ pipenv run jupyter nbextension enable --py --sys-prefix appmode
Enabling notebook extension appmode/main...
      - Validating: OK
$ pipenv run jupyter serverextension enable --py --sys-prefix appmode
Enabling: appmode.server_extension
- Writing config: /Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/bin/../
↪etc/jupyter
    - Validating...
      appmode.server_extension  OK
```

13.2.4 Configuration

Server-side configuration

The server can be configured with the following three options:

Appmode.trusted_path
runs the app mode only for notebooks under this path; Default setting: *no restrictions*.

Appmode.show_edit_button
displays *Edit App* button in app mode; Default setting: `True`.

Appmode.show_other_buttons
shows other buttons in app mode, for example *Logout*; Default setting: `True`.

You can find more information about the server configuration in *Jupyter paths and configuration*.

Client-side configuration

The UI elements can also be adapted on the client side in the custom.js file, for example with:

```
// Hides the edit app button.
$('#appmode-leave').hide();

// Hides the kernel busy indicator.
$('#appmode-busy').hide();

// Adds a loading message.
$('#appmode-loader').append('<h2>Loading...</h2>');
```

Note: Hiding the *Edit App* button does not prevent users from exiting app mode by manually changing the URL.

13.3 Panel

Panel was developed on the basis of Bokeh and Param and offers a toolkit especially for creating apps and dashboards, which not only supports bokeh plots.

See also:

- Panel Announcement
- Panel: A high-level app and dashboarding solution for the PyData ecosystem.

13.3.1 Installation

You can install Panel in the virtual environment of your Jupyter kernel with:

```
$ pipenv install panel
Installing panel...
Collecting panel
...
Installing collected packages: param, pyviz-comms, pyct, markdown, bokeh, panel
Successfully installed bokeh-1.3.4 markdown-3.1.1 panel-0.6.2 param-1.9.1 pyct-0.4.6␣
↪pyviz-comms-0.7.2
...
```

For some of the following examples additional packages are required such as Holoviews and hvPlot. They can be installed with:

```
$ pipenv install "holoviews[recommended]"
Installing holoviews[recommended]...
...
Installing collected packages: param, pyviz-comms, kiwisolver, cycler, pyparsing,␣
↪matplotlib, pyct, markdown, packaging, bokeh, panel, holoviews
Successfully installed bokeh-1.3.4 cycler-0.10.0 holoviews-1.12.5 kiwisolver-1.1.0␣
↪markdown-3.1.1 matplotlib-3.1.1 packaging-19.1 panel-0.6.2 param-1.9.1 pyct-0.4.6␣
↪pyparsing-2.4.2 pyviz-comms-0.7.2
...
$ pipenv install hvplot
Installing hvplot...
Collecting hvplot
...
Installing collected packages: hvplot
Successfully installed hvplot-0.4.0
...
```

Examples

1. Download

   ```
   $ pipenv run panel sampledata
   Creating /Users/veit/.bokeh/data directory
   Using data directory: /Users/veit/.bokeh/data
   Fetching 'CGM.csv'
   Downloading: CGM.csv (1589982 bytes)
    1589982   [100.00%%]
   ...
   ```

2. View

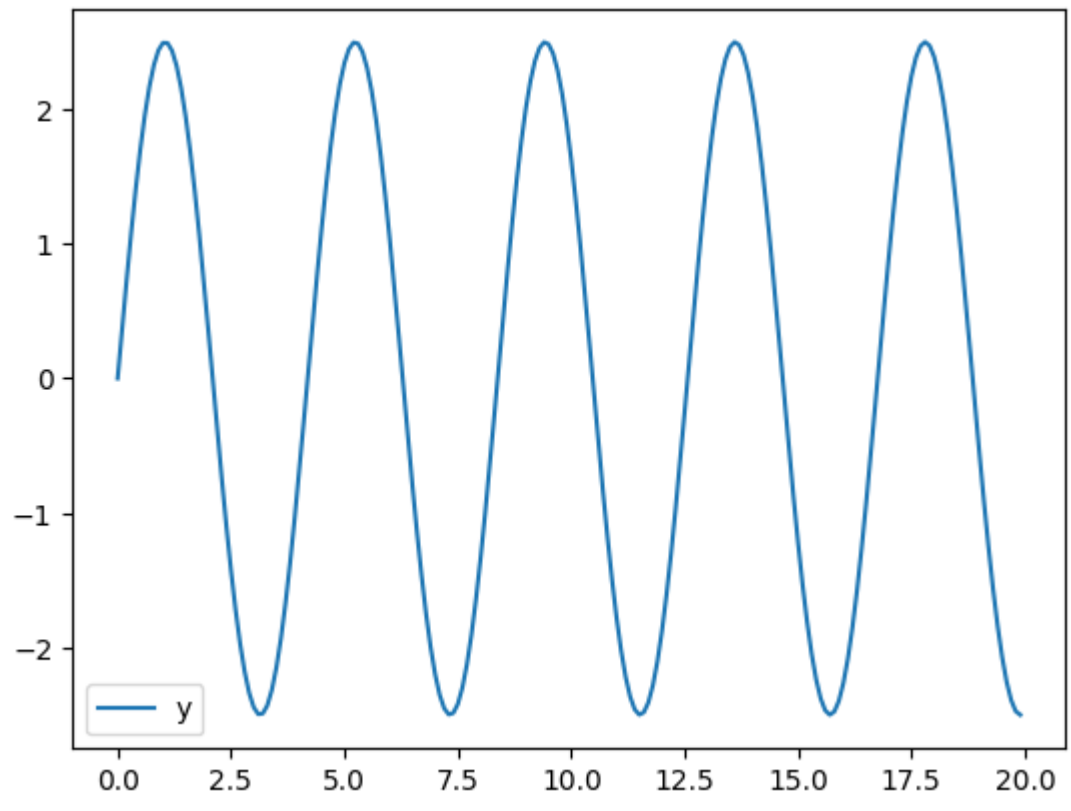
   Then you can look at the examples, for example `Introduction.ipynb` with

   ```
   $ pipenv run panel serve panel-examples/getting_started/Introduction.ipynb
   2019-08-18 10:55:44,056 Starting Bokeh server version 1.3.4 (running on Tornado 6.0.
   ↪3)
   2019-08-18 10:55:44,067 Bokeh app running at: http://localhost:5006/Introduction
   2019-08-18 10:55:44,067 Starting Bokeh server with process id: 86677
   ```

13.3.2 Overview

You can add interactive controls in a panel. This allows you to create simple interactive apps, but also complex multi-page dashboards. We'll start with a simple example of a function for drawing a sine wave with Matplotlib:

[1]:
```
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)


sine(1.5, 2.5)
```

[1]:


Interactive panels

If we wanted to try many combinations of these values to understand how frequency and amplitude affect this graph, we could reevaluate the above cell many times. However, this would be a slow and tedious process. Instead of having to re-enter the values in the code each time, it is advisable to adjust the values interactively with the help of sliders. With such a panel app you can easily examine the parameters of a function. The function of `pn.interact` is similar to 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')
```

As long as a live Python process is running, dragging these widgets calls the `sine` callback function and evaluates the combination of parameter values you selected and displays the results. With such a panel you can easily examine all functions that provide a visual result of a supported object type (see Supported object types and libraries, for example Matplotlib, Bokeh, Plotly, Altair or various text and image types.

Components of panels

`interact` is handy, but what if you want more control over how it looks or works? First, let's see what `interact` is actually created by grabbing the object and viewing its representation:

[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')
```

We can see here that the `interact` call has created an `pn.Column` object that consists of a WidgetBox (with 3 widgets) and a `pn.Row` Matplotlib figure. The control panel is compositional, so you can mix and match these components as you like by adding as many objects as needed:

[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)
```

Note that the widgets remain linked to their plot, even if they are in a different notebook cell:

[5]:
```
i[0][2]
```

[5]:
```
IntSlider(end=100, name='n', start=5, value=200)
```

New panels

With this compositional approach, you can combine different components such as widgets, charts, text and other elements that are needed for an app or a dashboard in any way you want. The `interact` example is based on a reactive programming model in which an input for the function changes and the control panel reactively updates the output of the function. `interact` is a handy way to automatically build widgets from the arguments for your function. However, Panel also provides a more explicit reactive API that allows you to define connections between widgets and function arguments, and then manually create the resulting dashboard from scratch.

In the following example we explicitly declare every component of an app:

1. Widgets
2. a function for calculating sine values
3. Column and row containers
4. the finished `sine_panel` app.

Widget objects have several parameters (current value, allowed ranges, etc.), and here we use the `depends` Panel decorator to declare that the input values of the function should come from the `value` parameters of the widgets. Now, when the function and widgets are displayed, the panel automatically updates the displayed output if one of the inputs changes:

[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)
```

(continues on next page)

(continued from previous page)

```
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

The above panels all work in a notebook cell, but unlike `ipywidgets` and other approaches, `Panel` apps work on standalone servers as well. The above app can, for example, be started as a separate web server with:

[7]:
```
sine_panel.show()
```

```
Launching server at http://localhost:59683
```

[7]:
```
<panel.io.server.Server at 0x14dfdba90>
```

This will start the Panel server and open a browser window with the application.

Or you can just indicate what you want to see on the website. `servable()`, and then the shell command to start a server with this object `pipenv run panel serve --show example.ipynb`:

[8]:
```
sine_panel.servable();
```

The semicolon prevents another copy of the sine field from being displayed here in the notebook.

Declarative Panels

The above compositional approach is very flexible, but it links domain-specific code (the parts about sine waves) to the widget display code. This is common in prototypical projects, but in projects where the code is going to be used in many different contexts, parts of the code that relate to the underlying domains (i.e. the application or research area) should be separated from those that are tied to certain display technologies (such as Jupyter notebooks or web servers).

For such uses, Panel supports objects that have been declared with the separate Param library. This offers a possibility to independently record and declare the parameters of your objects (code, parameters, application and dashboard technology). For example, the above code can be captured in an object that declares the ranges and values of all parameters as well as the generation of the diagram independently of the panel library or any other type of interaction with the object:

[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))
```

(continues on next page)

(continued from previous page)

```
    def view(self):
        return sine(self.frequency, self.amplitude, self.n)


sine_obj = Sine()
```

The `Sine` class and `sine_obj` instance are not dependent on Panel, Jupyter or any other GUI or web toolkit – they simply declare facts about a particular domain (for example that sine waves take frequency and amplitude parameters and that the amplitude is a number greater or equals zero). That information is then enough for Panel to create an editable and viewable representation for this object without having to specify anything that depends on the domain-specific details contained in the `Sine` class and the `sine_obj` -Instance are not dependent on Panel, Jupyter or any other GUI or web toolkit. They simply declare facts about a certain range (for example, that sine waves take frequency and amplitude parameters, and that the amplitude is a number greater than or equal to zero). That information is enough for Panel to create an editable and viewable representation for this object without having to specify anything that depends on the domain-specific details contained outside of `sine_obj`:

[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)
```

In order to support a certain domain, you can create hierarchies of such classes, in which all parameters and functions are summarised that you need for different object families. Both parameters and code are adopted in the classes, regardless of a specific GUI library or even the existence of a GUI at all. This approach makes it convenient to maintain a large code base that can be fully viewed and edited with Panel, in a way that can be maintained and customised over time.

Linking plots and actions between panels

The above approaches each work with a variety of displayable objects, including images, equations, tables, and charts. In each case, the panel provides interactive functionality using widgets and updates the objects displayed accordingly, making very few assumptions about what is actually displayed. Panel also supports a broader and more dynamic interactivity in which the displayed object itself is interactive, for example JavaScript-based diagrams of Bokeh and Plotly.

For example, if we replace the matplotlib wrapper that came with pandas with the Bokeh wrapper hvPlot, we automatically get interactive plots that allow *zooming*, *panning* and *hovering*:

[11]:
```
import hvplot.pandas


def hvplot(df, **kwargs):
    return df.hvplot()


pn.interact(sine, view_fn=hvplot)
```

(continued from previous page)

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)
```

These interactive actions can be combined with more complex interactions in a plot (for example `tap`, `hover`) to make it easier to explore data and uncover connections. For example, we can use HoloViews to create a more comprehensive version of the hvPlot example that is dynamically updated to show the position on the circle as we hover over the sine curve:

[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"
        )
        return circle * triangle * labels

    vline = hv.DynamicMap(hv.VLine, streams=[tap]).opts(color="black")

    return (plot * vline).opts(toolbar="right")
```

(continues on next page)

(continued from previous page)

```
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 Interactions

The `interact` function (`panel.interact`) automatically creates controls for interactively browsing code and data.

[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

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

At the simplest level, `interact` controls are automatically generated for function arguments, and the function is then called with those arguments when you interactively edit the controls. To use `interact` you need to define a function that you want to examine. Here is a function that returns the only argument: `x`.

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

If you pass this function as the first argument together with an integer keyword argument `x=10` to `interact` a slider is generated and bound to the function parameters.

[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')
```

If you move the slider, the function is called, which outputs the current value of `x`. If you pass `True` or `False` `interact` generates a check box:

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

[4]:
```
Column
    [0] Column
        [0] Checkbox(name='x', value=True)
    [1] Row
        [0] Str(bool, name='interactive00142')
```

When you pass a string, `interact` generates a text area.

[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` can also be used as a *Decorator*. In this way you can define a function as well as determine the type of interaction. As the following example shows, `interact` works also with functions that have multiple arguments.

[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)
```

(continues on next page)

(continued from previous page)

```
        [1] FloatSlider(end=3.0, name='y', start=-1.0, value=1.0)
    [1] Row
        [0] Str(tuple, name='interactive00200')
```

Layout of interactive widgets

The `interact` function returns a panel that contains the widgets and the display output. By indexing these panels we can lay out the objects exactly how we want:

[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')
```

Set arguments with `fixed`

There may be times when you want to examine a function using `interact`, but want to set one or more of its arguments to specific values. This can be achieved using the `fixed` function:

[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

If you pass certain values, `interact` use automatically the appropriate widget, for example a checkbox for `True` or `IntSlider` for integer values. So you don't have to explicitly specify the appropriate widget:

[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')
```

This example shows how the keyword arguments are processed by `interact`:

1. If the keyword argument is an instance of `Widget` with a `value` attribute, this widget is used. Any widget with a `value` attribute can be used, including custom ones.
2. Otherwise, the value is treated as a *Widget Abbreviation* that is converted to a widget before use.

The following table gives an overview of the various *Widget Abbreviations*:

Keyword argument	Widget
`True` or `False`	`Checkbox`
`"Hi Pythonistas!"`	`Text`
Integer value as `min`, `max`,`step`, `value`	`IntSlider`
Floating-point `min`, `max`,`step`, `value`	`FloatSlider`
`["apple", "pear"]` or `{"one": 1, "two": 2}`	`Dropdown`

13.3.4 Widgets

`Panel` offers a wide range of widgets for precise control of parameter values. The widget classes use a consistent API that allows broad categories of widgets to be treated as interchangeable. For example, to select a value from a list of options, you can use `SelectWidget`, a `RadioButtonGroup`widget, or an equivalent widget interchangeably.

Like all other components in `Panel`, `Widget` objects can also synchronise their state both in the notebook and on the bokeh server:

[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')
```

If you change the text value, the corresponding parameter is automatically updated:

[3]:
```
widget.value
```

[3]:
```
'A string'
```

Updating the parameter value also updates the widget:

[4]:
```
widget.value = "Another string"
```

Callbacks and links

In order to notice a parameter change, we can call a function `widget.param.watch` with the parameter to be observed:

[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)
```

If we change now `widget.value`, the resulting event is output.

[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, in combination with objects, enable the easy creation of interactive dashboards and visualisations. For more information on defining callbacks and links between widgets and other components, see the User Guide.

Widgets

To put several widgets together, they can be added to a `Row`-, `Column`- or `Tabs` panel. For more information on the layout of widgets and control panels, see 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 categories

The supported widgets can be divided into different categories based on their compatible APIs.

Option selection

With option selection widgets you can select one or more values from a `list` or a `dictionary`. All widgets of this type have `options` and `value` parameters.

Options	Widget	Description
Individual values		With these widgets you can choose a value from a `list` or a `dictionary`
	`AutocompleteInp`	selects one `value` rom an automatically completed text field
	`DiscretePlayer`	displays controls of `mediaplayer`, that allow you to play and step through the options available
	`DiscreteSlider`	selects a value with a slider
	`RadioButtonGrou`	selects a value from a series of mutually exclusive toggle keys
	`RadioBoxGroup`	selects a value from a series of mutually exclusive check boxes
	`Select`	selects a value from a drop-down menu
Multiple values		With these widgets you can select several values from a `list` or a `dictionary`
	`CheckBoxGroup`	select values by activating the corresponding check boxes
	`CheckButtonGrou`	select values by toggling the corresponding buttons
	`CrossSelector`	select values by moving items between two lists
	`MultiSelect`	select values by marking them in a list

Type-based selectors

Type-based selectors offer the possibility of choosing a value according to its type. All selectors have a `value`. In addition to the type, the widgets in this category can also have other forms of validation, for example the upper and lower limits of sliders.

	Types	Widget	Description
Individual values			allows the selection of a single `value` type
	Numerically		Numeric selectors are limited by `start` and `end` values
		`IntSlider`	selects an integer value within a specified range with a slider
		`FloatSlide`	uses a slider to select a floating point value within a specified range
		`Player`	displays controls of the `mediaplayer`, that allow you to play and cycle through a range of integer values
	Boolean values		
		`Checkbox`	toggles a single condition between `True`/`False` by checking a box
		`Toggle`	toggles a single condition between `True`/`False` by clicking a button
	Date		
		`DatetimeIr`	selects a date value using a text box and the browser's date picker utility
		`DatePicker`	enters a date/time value as text and analyse it with a predefined formatter
		`DateSlider`	selects a date value within a specified range with a slider
	Text		
		`TextInput`	enters any character string via a text entry field
	Other		
		`ColorPicke`	selects a color using the browser's color-picking utilities
		`FileInput`	uploads a file from the frontend and make the data and MIME type available in Python
		`LiteralInp`	enters any Python literal via a text entry field, which will then be parsed in Python
Areas			enables the selection of a value range of the corresponding type, which is saved as a `(lower, upper)` tuple for the `value` parameter
	Numerically		
		`IntRangeSl`	selects an integer range with a slider with two handles
		`RangeSlide`	selects a floating point area using a slider with two handles
	Dates		
		`DateRangeS`	selects a date range using a slider with two handles
Other			
		`Audio`	displays an audio player to which an audio file has been assigned locally or remotely, and allows access and control of the player status
		`Button`	allows events to be triggered when the button is clicked; unlike other widgets, it does not have an `value` parameter

13.3.5 Parameterisation

Panel supports the use of parameters and dependencies between parameters, expressed in a simple way by param, to encapsulate dashboards as declarative, stand-alone classes.

Parameters are Python attributes that have been extended using the `param` library to support types, ranges, and documentation. This is just the information you need to automatically create widgets for each parameter.

Parameters and widgets

For this purpose, some parameterised classes with different parameters are declared first:

[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)
```

(continues on next page)

(continued from previous page)

```
    multiple_files = param.MultiFileSelector(
        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
```

As you can see, the declaration of parameters only depends on the separate `param` library. Parameters are a simple idea with a few properties critical to creating clean, usable code:

- The `param` library is written in pure Python with no dependencies, which makes it easy to include in any code without tying it to a specific GUI or widgets library, or to Jupyter notebooks.
- Parameter declarations focus on semantic information that is relevant to your domain. In this way, you avoid contaminating domain-specific code with anything that binds it to a specific display or interaction with it.
- Parameters can be defined wherever they make sense in your inheritance hierarchy, and you can document them once, enter them and limit them to a certain area. All these properties are inherited from any base class. For example, all parameters work the same here, regardless of whether they were declared in `BaseClass` or `Example`. This makes it easier to provide this metadata once and prevents it from being duplicated anywhere in the code where areas or types need to be checked or documentation saved.

If you then decide to use these parameterised classes in a notebook or web server environment, you can easily display and edit the parameter values as an optional additional step with `import panel`:

[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)
```

(continues on next page)

(continued from previous page)

```
        [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')
        [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})
```

As you can see, Panel does not need to have knowledge of your domain-specific application, nor of the names of your parameters. It simply shows widgets for all parameters that have been defined for this object. By using Param with Panel, an almost complete separation between your domain-specific code and your display code is achieved, which considerably simplifies the maintenance of both over a longer period of time. Here even the `msg` behavior of the buttons was declared declaratively as an action that can be called regardless of whether it is used in a GUI or in another context.

Interaction with the above widgets is only supported in the notebook and on the bokeh server. However, you can also export static renderings of the widgets to a file or a website.

If you edit values in this way, you have to run the notebook cell by cell by default. When you get to the cell above, edit the values as you wish and execute the following cells, in which these parameter values are referred to, your interactively selected settings are used:

[3]:
```
Example.unbounded_int
```

[3]:
```
23
```

[4]:
```
Example.num_int
```

[4]:
```
50000
```

To work around this and automatically update all widgets generated from the parameter, you can pass the `param` object:

[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)
```

Custom widgets

In the previous section we saw how parameters can be automatically converted into widgets. This is possible because the Panel internally manages an assignment between parameter types and widget types. However, sometimes the standard widget doesn't provide the most convenient user interface, and we want to give Panel an explicit hint on how a parameter should be rendered. This is possible with the `widgets` argument for the `Param` panel. With the `widgets` keyword we can declare an association between the parameter name and the desired widget type.

As an example we can assign a `RadioButtonGroup` and a `DiscretePlayer` to a `String` and a `Number` selector.

[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
↪'...})
```

It is also possible to pass arguments to the widget to customise it. Instead of passing the widget, pass a dictionary with the options you want. Uses the `type` keyword to map the widget:

[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 dependencies

Declaring parameters is usually just the beginning of a workflow. In most applications, these parameters are then linked to a computation. To express the relationship between a computation and the parameters on which it depends, the `param.depends` decorator for parameterized methods can be used. This decorator gives panels and other `param`-based libraries (e.g. HoloViews) an indication that the method should be recalculated if a parameter is changed.

As a simple example with no additional dependencies, let's write a small class that returns an ASCII representation of a sine wave that depends on `phase` and `frequency` parameters. When we pass the `.view` method to a panel, the view is automatically recalculated and updated as soon as one or more of the parameters change:

[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
        )


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)
```

The parameterised and annotated `view` method can return any type provided by the Pane objects. This makes it easy to link parameters and their associated widgets to a plot or other output. Parameterised classes can therefore be a very useful pattern for encapsulating part of a computational workflow with an associated visualisation and for declaratively expressing the dependencies between the parameters and the computation.

By default, a Param area (*Pane*) shows widgets for all parameters with a `precedence` value above the value `pn.Param.display_threshold`, so you can use `precedence` to automatically hide parameters. You can also explicitly choose which parameters should contain widgets in a certain area by passing an `parameters` argument. For example, this code outputs a `phase` widget, keeping `sine.frequency` the initial value 1:

[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)
```

Another common pattern is linking the values of one parameter to another parameter, for example when there are dependencies between parameters. In the following example we define two parameters, one for the continent and one for the country. Since we would like the selection of valid countries to change when we change continent, let's define a method to do this for us. To connect the two, we express the dependency using the `param.depends` decorator and then use `watch=True` to ensure that the method is executed when the continent is changed.

We also define a `view` method that returns an HTML iframe showing the country using Google Maps.

[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"],
    }

    @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
```

(continues on next page)

(continued from previous page)

```
    [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)
```

Whenever the continent changes, the `_update_countries` method for changing the displayed country list is now executed, which in turn triggers an update of the `view` method.

[11]:
```python
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):
        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 sub-objects

`Parameterized` objects often have parameter values that are `Parameterized` objects themselves and form a tree-like structure. With the control panel you can not only edit the parameters of the main object, but also access sub-objects. Let's first define a hierarchy of `Shape` classes that will draw a bokeh plot of the selected `Shape`:

[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):
        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})
```

Now that we have multiple `Shape` classes we can create instances of them and create a `ShapeViewer` to choose between. We can also declare two methods with parameter dependencies that update the plot and the plot title. It should be noted that the `param.depends` decorator can not only depend on parameters on the object itself, but can also be expressed on certain parameters on the subobject, for example `shape.radius` or with `shape.param` on parameters of the subobject.

[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)
```

Now that we have a class with sub-objects, we can display them as usual. Three main options control how the sub-object is rendered:

- `expand`: whether the sub-object is expanded during initialisation (`default=False`)
- `expand_button`: whether there should be a button to toggle the extension; otherwise it is set to the initial `expand` value (`default=True`)
- `expand_layout`: A layout type or instance to extend the plot in (`default=Column`)

Let's start with the standard view, which has a toggle button to expand the sub-object:

[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)
```

Alternatively, we can offer a completely separate `expand_layout` instance for a param area, which with the `expand` and `expand_button` option always remains expanded. This allows us to separate the main widgets and the sub-object's widgets:

[15]:
```
viewer = ShapeViewer()

expand_layout = pn.Column()
```

(continues on next page)

(continued from previous page)

```
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 objects build on param, which allows them to be specified by parameters so that users can flexibly edit to control the output displayed. In addition to the parameters specific to each component and component class, all components define a common set of parameters to control the size and style of the rendered view.

[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 components

css_classes

The `css_classes` parameter enables a panel component to be assigned to one or more CSS classes. CSS can be specified directly in the notebook or as a reference to an external CSS file by passing it to the Panel extension with `raw_css` or `css_files` as a list. Outside a notebook, in an external module or library, we can attach configuration parameters with `pn.config.raw_css` and `pn.config.js_files`.

To demonstrate this usage, let's define a CSS class named `widget-box`:

[2]:
```
css = """
.widget-box {
  background: #f0f0f0;
  border-radius: 5px;
  border: 1px black solid;
}
"""

pn.extension(raw_css=[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

[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

If we just want to give the component a background, we can define one as a hex string:

[4]:
```
pn.Column(styles={"background": "#f0f0f0", "width": "100", "height": "100"})
```

[4]:
```
Column(styles={'background': '#f0f0f0', ...})
```

style

Certain components, especially markup-related panes, provide a `style` parameter that can be used to define CSS styles that are applied to the HTML container of the window content, e.g. the `Markdown` pane:

[5]:
```
pn.pane.Markdown("### Cusy: DevOps", styles={"font-family": "sans-serif"})
```

[5]:
```
Markdown(str, styles={'font-family': '...})
```

Component size and layout

The size of the components and their spacing are also controlled by a number of parameters that are shared by all components.

margin

The `margin` parameter can be used to create space around an element, which is defined as the number of pixels in the order top, right, bottom and left, e.g.

[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 dimensioning with `width` and `height`

By default, all components use either automatic or absolute resizing. Panels generally take up as much space as the components they contain, and text- or image-based panels adjust to the size of their content. To set a fixed size for a component, it is usually sufficient to set a width or height. In certain cases, however, `sizing_mode='fixed'` must be specified explicitly.

[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),
    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` can have the following values:

- `fixed`: The component is not responsive. The original width and height are retained regardless of subsequent events that resize the browser window. This is the default behavior and just uses the specified width and height.
- `stretch_width`: The component resizes to stretch it to the available width without maintaining the aspect ratio. The height of the component depends on the type of component and can be fixed or tied to the content of the component.
- `stretch_height`: The component is resized appropriately to fit the available height, but without maintaining the aspect ratio. The width of the component depends on the type of component and can be fixed or tied to the content of the component.
- `stretch_both`: The component is responsive, regardless of width or height, and occupies all available horizontal and vertical space, even if this changes the aspect ratio of the component.
- `scale_height`: The component is resized appropriately to stretch it to the available height while maintaining the original or provided aspect ratio.
- `scale_width`: The component is resized appropriately to stretch it to the available width while maintaining the original or provided aspect ratio.
- `scale_both`: The component is resized to the available width and height, while maintaining the original or provided aspect ratio.

[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` are a very versatile component that can be used to easily create fixed or responsive distances between objects. Like all other components `Spacer` support both absolute and responsive mode:

[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` and `HSpacer` ensure an attractive vertical or horizontal distance. With these components we can place objects equidistantly on a layout and shrink the empty area when the browser is resized:

[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 and export

One of the main design goals for Panel was to enable a seamless transition between interactively prototyping a dashboard and deploying it as a standalone server app. This notebook shows how to interactively display panels, embed static output, save a snapshot, and serve it as a separate web server app.

Configure output

Panel objects are automatically displayed in a notebook and use Jupyter Comms to support communication between the rendered app and the Jupyter kernel. The display of a panel object in the notebook is simple: it only has to load the `panel.extension` first in order to initialise the required JavaScript in the notebook context.

[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

Optional dependencies

In order to be able to use certain components such as Vega, LaTeX and Plotly-Plots, the corresponding Javascript components must also be loaded. To do this, you can simply include them as part of the call to `pn.extension`:

[2]:
```
pn.extension("vega", "katex")
```

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

Initialise JS and CSS

Additional CSS and Javascript can also be specified with `css_files`, `js_files` and `raw_css`. `js_files` should be specified as a dictionary mapping from the exported JS module name to the URL with the JS components, while `css_files` can be defined as a list:

[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

With this `raw_css` argument you can define a list of strings with CSS that should be published as part of the notebook and the app.

Providing keyword arguments with `extension` is equivalent to specifying with `pn.config`. `pn.config` is the preferred approach to add Javascript and CSS files outside of a notebook:

[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"
]
```

Display in the notebook

Once `extension` is loaded, panel objects that are placed at the end of a cell are displayed:

[5]:
```
pane = pn.panel("<marquee>Here is some custom HTML</marquee>")

pane
```

[5]:
```
Markdown(str)
```

The display function

To avoid having to put a panel in the last row of a notebook cell, you can use the IPython `display` function:

[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

Finally, it is also possible to display a panel object as a bokeh server app in the notebook. To do this, call the `.app` method in the panel object and enter the URL of your notebook server:

[7]:
```
pn.io.notebook.show_server(panel=pane, notebook_url="localhost:8888")
```

Data type cannot be displayed: application/vnd.holoviews_exec.v0+json, text/html

[7]:
```
<panel.io.server.Server at 0x1095da090>
```

The app is now executed in an instance of the Bokeh server that is separate from the Jupyter notebook kernel, so that you can quickly test whether the entire functionality of your app works both in the notebook and in the server context.

Display in an interactive Python window (REPL)

If you work via the command line, extensive displays are not automatically displayed inline, as is the case in a notebook. However, you can interact with your panel components if you start a Bokeh server instance and use the `show` method to open a separate browser window. The method has the following arguments:

- `port`: `int`, (optional): allows a specific port to be specified (`default=0` select any open port)
- `websocket_origin`: `str` or `list(str)` (optional): A list of hosts that can connect to the websocket. This is necessary when a server app is embedded in an external website. If not specified, `localhost` is used.
- `threaded`: `boolean` (optional, `default=False`): `True` starts the server in a separate thread and allows you to interact with the app.

The `show` call returns either a Bokeh server instance (`threaded=False`) or an `StoppableThread` instance (`threaded=True`), both provide a `stop` method for stopping the server instance.

Starting a server from the command line

Panel (and Bokeh) provide a CLI command to deploy a Python script, app directory, or Jupyter notebook with a Bokeh or Panel app. To start a server using the CLI, simply do the following:

```
$ pipenv run panel serve app.ipynb
```

To turn a notebook into a deployable app, simply attach to one or more panel objects `.servable()`, which adds the app to bokehs `curdoc`. This makes it easy to create dashboards interactively in a notebook and then seamlessly provide them to the Bokeh server.

Session status

- `panel.state` exposes some of the internal Bokeh server components to users.
- `panel.state.curdoc` allows access to the current bokeh.document.

Embed

Panel generally needs either the Jupyter kernel or a Bokeh server running in the background to enable interactive behavior. However, for simple apps it is also possible to capture the entire widget status so that the app can be used entirely from Javascript. To demonstrate this, let's create a simple app that simply takes a slider value, multiplies that by 5, and then displays the result:

[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 0x10ce13850>
```

If you try the above widget, you will find that it only has three different status `0`, `5` and `10`. This is because embedding attempts to limit the number of options for non-discrete or semi-discrete widgets to a maximum of three values by default. This can be changed with the `max_opts` argument of the `embed` method. The full options for the `embed` method are:

- `max_states`: Maximum number of states to be embedded
- `max_opts`: Maximum number of states for a single widget
- `json`: Specifies whether the data should be exported to json files
- `save_path`: Path to save JSON files (`default='./'`)
- `load_path`: Path or URL from which the JSON files are loaded (as `save_path` unless otherwise specified)

As you can easily imagine, a combinatorial explosion of the statuses can quickly occur with several widgets, so that the output is limited to around 1000 statuses by default. For larger apps, the status can also be exported to JSON files. For example, if you want to make the app available on a website, specify `save_path` where the JSON file should be saved and `load_path` where the JS code should search for the files.

Save

If you don't need an actual server or just want to export a static snapshot of a panel app, you can use the save method which can be used to export the app to a standalone HTML or PNG file.

By default, the generated HTML file depends on loading the JavaScript code for BokehJS from the online CDN repository to reduce the file size. If you need to work in a networked or non-networked environment, you can choose to use INLINE resources instead of CDN:

[10]:
```
from bokeh.resources import INLINE


pane.save("deploy-panel.html", resources=INLINE)
pane.save("test.png")
```

To export the `png` file you also need Selenium and PhantomJS:

```
$ pipenv install selenium
Installing 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
```

In addition, you can use the `save` method together with the `embed` option to embed the app status in the app or to save it in JSON files, which can be deployed together with the exported HTML code. You have the following options:

- `resources`: `bokeh.resources`, e.g. `CDN` or `INLINE`
- `embed`: Boolean value, whether the status should be saved in the file or not.
- `max_states`: The maximum number of states to be embedded
- `max_opts`: The maximum number of states for a single widget
- `embed_json`: Boolean value as to whether the data should be exported as a JSON file (`default=True`).

13.3.8 Pipelines

In *parameterisation* is described how classes are created, which declare the parameters and link calculations or visualisations. In this section you will learn how you can connect several such panels with a pipeline to express complex workflows in which the output of one stage is fed into the next stage.

[1]:
```
import panel as pn
import param


pn.extension("katex")
```

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]:
```
pipeline = pn.pipeline.Pipeline()
```

While we saw earlier how methods are linked to the `param.depends` decorator, pipelines use a different decorator and a convention for displaying the objects. The `param.output` decorator provides a way to annotate the methods of a class by declaring its output. `Pipeline` uses this information to determine what outputs are available to be fed into the next stage of the workflow. In the following example, the class `Stage1` has two parameters (`a` and `b`) and an output `c`. The decorator's signature allows a number of different ways to declare the outputs:

- `param.output()`: If output is declared with no arguments, the method returns output that inherits the name of the method and does not make any specific type declarations.
- `param.output(param.Number)`: When declaring an output with a specific parameter or a Python type, the output is declared with a specific type.
- `param.output(c=param.Number)`: If an output is declared with a keyword argument, you can overwrite the method name as the name of the output and declare the type.

It is also possible to declare several parameters as keywords or as tuples:

- `param.output(c=param.Number, d=param.String)`
- `param.output(('c', param.Number), ('d', param.String))`

In the example below, the output is simply the result of multiplying the two inputs (`a` and `b`) that produce the output `c`. In addition, we declare a `view` method that returns a `LaTeX` pane. Finally, a `panel` method returns a Panel object that render both the parameters and the view.

[3]:
```
class Stage1(param.Parameterized):
    a = param.Number(default=5, bounds=(0, 10))

    b = param.Number(default=5, bounds=(0, 10))

    @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)
```

(continues on next page)

(continued from previous page)

```
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)
```

In summary, we followed a few conventions to create this stage of our pipeline:

1. Declare a parameterised class with some input parameters
2. Declare and name one or more output methods
3. Declare a `panel` method that returns a View of the object that the pipeline can render.

Now that the object has been instantiated, we can also ask it about its outputs:

[4]:
```
stage1.param.outputs()
```

[4]:
```
{'c': (<param.Number at 0x158083880>,
  <bound method Stage1.output of Stage1(a=5, b=5, name='Stage100954')>,
  0),
 'd': (<param.Number at 0x158083640>,
  <bound method Stage1.output of Stage1(a=5, b=5, name='Stage100954')>,
  1)}
```

We can see that `Stage1` declared an output with the name `c` of the type `Number` that can be accessed using the `output` method. Now let's add `stage1` with `add_stage` to our pipeline:

[5]:
```
pipeline.add_stage("Stage 1", stage1)
```

For a pipeline, however, we still need at least one `stage2` that processes the result of `stage1`. Therefore a parameter `c` should be declared from the result of `stage1`. As a further parameter, we define `exp` and a `view` method again, which depends on the two parameters and the `panel` method.

[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)
```

(continues on next page)

(continued from previous page)

```
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)
```

Also, we now add `stage2` to the `pipeline` object:

[7]:
```
pipeline.add_stage("Stage 2", stage2)
```

We now have a two-stage pipeline where the output `c` is passed from `stage1` to `stage2`. Now we can display the pipeline with `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)
```

The rendering of the pipeline shows a small diagram with the available workflow stages and the *Previous* and *Next* buttons to switch between the individual phases. This enables navigation even in more complex workflows with many more phases.

Above we instantiated each level individually. However, if the pipeline is to be deployed as a server app, the stages can also be declared as part of the constructor:

[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)
```

(continues on next page)

(continued from previous page)

```
        [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)
```

The pipeline stages can either be `Parameterized` instances or `Parameterized` classes. With instances, however, you have to make sure that the update of the parameters of the class also updates the current status of the class.

13.3.9 Templates

If you want to provide a panel app or a dashboard as a bokeh application, it is rendered in a standard template that refers to the JS and CSS resources as well as the actual panel object. If you want to adapt the layout of the provided app or if you want to provide several separate panels in one app, the `Template` component of Panel allows you to adapt this standard template.

Such a template is defined with Jinja, whereby you can extend or even completely replace the standard template. Here is an example:

```
<!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 }}
        {% 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 }}
```

(continues on next page)

(continued from previous page)

```
        {% for root in doc.roots %}
            {{ embed(root) | indent(10) }}
        {% endfor %}
        {% endfor %}
    {% endblock %}
    {{ plot_script | indent(8) }}
    {% endblock %}
</body>
{% endblock %}
</html>
```

The template defines a number of user-defined blocks that can be supplemented or overwritten by `extends`:

Use custom templates

[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

Once we have Panel loaded, we can start defining a custom template. It is usually easy to customise an existing template by overwriting certain blocks. With `{% extends base %}` we declare that we are only expanding an existing template and not defining a new one.

In the following case, we are expanding the `postamble` block of the header to load an additional resource and the `contents` block to redefine the arrangement of the components:

[2]:
```
template = """
{% extends base %}

<!-- head -->
{% block postamble %}
<link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.3.1/css/
↪bootstrap.min.css">
{% endblock %}
```

(continues on next page)

(continued from previous page)

```
<!-- body -->
{% block contents %}
<h1>Custom template for multiple apps</h1>
<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 %}
"""
```

Using the `embed` macro, we have defined two different `roots` in the template. In order to be able to render the template, we must first create the `pn.Template` object with the HTML template and then integrate the two `roots` objects.

[3]:
```
tmpl = pn.Template(template)

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)
```

A button is rendered in the notebook with which you can start a local server to check whether the output meets your expectations.

If the template is larger, it is often easier to create it in a separate file. You can use the Jinja2 template loading mechanism by defining an environment together with a `loader`:

[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 Running Panel in the browser with WASM

Panel lets you write dashboards and other applications in Python that are accessed through a web browser. Normally, the Python interpreter runs as a separate Jupyter or Bokeh server process and communicates with the JavaScript code running in the client browser. However, Python can also be run directly in the browser using WASM (WebAssembly), without the need for a separate server.

Panel uses Pyodide for this and PyScript for rendering.

Converting panel applications

Future versions of Panel can convert your Panel application from one or more Python scripts or Notebook files, including *Templates*, into an HTML file using `panel convert`. The only requirements are:

- they only import global modules and packages and no relative imports from other scripts or modules
- the libraries have been compiled for Pyodide or are available as *Python wheels <Wheel>* on the Python Package Index (PyPI).

Example

In the following example we will convert the *Overview* notebook into a standalone HTML page with

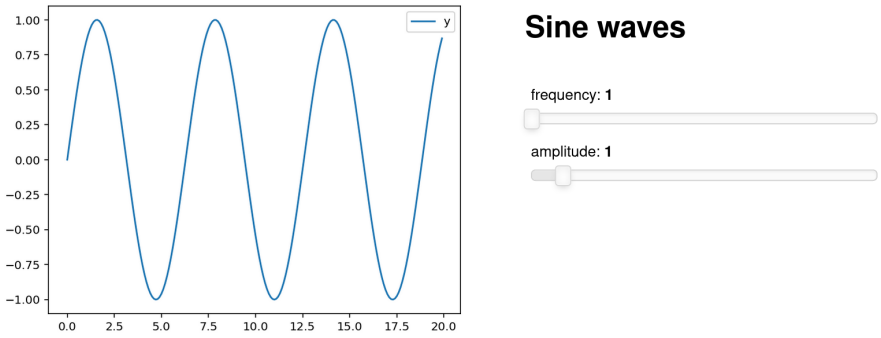
```
$ panel convert overview.ipynb --out pyodide
Column
    [0] Column
        [0] FloatSlider(end=3.0, name='frequency', start=-1.0, value=1.0)
        [1] FloatSlider(end=3.0, name='amplitude', start=-1.0, value=1.0)
        [2] IntSlider(end=100, name='n', start=5, value=200)
    [1] Row
        [0] Matplotlib(Figure, name='interactive00114')
Launching server at http://localhost:40405
```

Now you can open `http://localhost:40405` in your browser and try out the app:

You can now add the `pyodide/overview.html` file to your Github pages or similar – no separate server is required.

See also:

- Awesome Panel/Webassembly Apps



Options

In the following I explain some of the options of `panel convert`.

`--to`
: The format to convert to. There are three options, each with different advantages and disadvantages:

  `pyodide` (default)
  : The application is run with pyodide in the main thread. This option is less performant than `pyodide-worker`, but produces a fully self-contained HTML file that does not need to be hosted on a static file server, such as Github Pages.

  `pyodide-worker`
  : generates HTML and JS files, but includes a web worker that runs in a separate thread. This is the most powerful option, but the files must be hosted on a static file server.

  `pyscript`
  : creates an HTML file that uses PyScript. This creates standalone HTML files with `<py-env>` and `<py-script>` tags containing the dependencies and application code. This output is the most readable and should have the same performance as the `pyodide` option.

`-out`
: The directory to write the files to.

`--pwa`
: adds files that make the application a Progressive Web App.

  Progressive Web Apps (PWAs) provide a way for your web apps to behave almost like a native app, both on mobile devices and on the desktop. `panel convert` has a `--pwa` option that generates the files necessary to turn your panel and pyodide app into a PWA.

`--skip-embed`
: skips embedding pre-rendered content in the converted file.

  Panel embeds pre-rendered content in the HTML page and replaces it with live components once the page is loaded. However, this can take a long time. If you want to disable this behaviour and render a blank page first, use the `--skip-embed` option.

`--index`
: creates an index when you convert several applications at once, so you can easily navigate between them.

`--requirements`
: Explicit requirements to add to the converted file or to a `requirements.txt` file.

By default, requirements are derived from code.

If a library uses an optional import that cannot be derived from your application's list of imports, you must specify an explicit list of dependencies.

Note: panel and its dependencies, including NumPy and Bokeh, are loaded automatically, which means that the explicit requirements for the above application would be as follows:

```
$ panel convert overview.ipynb --out pyodide --requirements pandas matplotlib
```

AAlternatively, you can provide a `requirements.txt` file:

```
$ panel convert overview.ipynb --out pyodide --requirements requirements.txt
```

`--watch`
Observe the source files.

You can get a complete overview with `panel convert -u`.

Tip: If the converted application does not work as expected, you can usually find the errors in the browser console, see Finding Your Browser's Developer Console.

See also:

Answers to the most frequently asked questions about Python in the browser can be found in the

- Pyodide FAQ
- PyScript FAQ

13.3.11 FastAPI integration

Panel usually runs on a Bokeh-Server, which in turn runs on Tornado. However, it is also often useful to embed a Panel app into a large web application, such as a FastAPI web server. Integration with FastAPI is easier compared to others such as Flask, as it is a more lightweight framework. Using Panel with FastAPI requires only a little more effort than notebooks and Bokeh servers.

Configuration

Before we start adding a Bokeh application to our FastApi server, we need to set up some of the basic configuration in `fastAPI/main.py`:

1. First, we import all the necessary elements:

Listing 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. Next, we define `app` as an instance of `FastAPI` and define the path to the template directory:

```
app = FastAPI()
templates = Jinja2Templates(directory="templates")
```

3. Now we create our first routine via an asynchronous function and refer it to our BokehServer:

```
@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. As you can see from the code, a Jinja2 template `fastAPI/templates/base.html` is expected. This can have the following content, for example:

Listing 2: fastAPI/templates/base.html

```
<!DOCTYPE html>
<html>
    <head>
        <title>Panel in FastAPI: sliders</title>
    </head>
    <body>
        {{ script|safe }}
    </body>
</html>
```

5. Let's now return to our `fastAPI/main.py` file to start our bokeh server with `pn.serve()`:

```
pn.serve(
    {"/app": createApp},
    address="127.0.0.1",
    port=5000,
    show=False,
    allow_websocket_origin=["127.0.0.1:8000"],
)
```

createApp
: calls up our panel app in this example, but this is not covered until the next section.

address, port
: Address and port at which the server listens for requests; in our case `http://127.0.0.1:5000`.

show=False
: ensures that the Bokeh server is started but is not immediately displayed in the browser.

allow_websocket_origin
: lists the hosts that can connect to the websocket. In our example, this should be `fastApi`, so we use `127.0.0.1:8000`.

6. Now we define the `sliders` app based on a standard template for FastAPI apps, which shows how Panel and FastAPI can be integrated:

fastAPI/sliders/sinewave.py
: a parameterised object that represents your existing code:

Listing 3: fastAPI/sliders/sinewave.py

```python
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))

    def __init__(self, **params):
        super(SineWave, self).__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
```

(continues on next page)

(continued from previous page)

```
        )
        return x, y
```

fastAPI/sliders/pn_app.py
creates an app function from the `SineWave` class:

Listing 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. Finally, we return to our `fastAPI/main.py` and import the `createApp` function:

Listing 5: fastAPI/main.py

```
from sliders.pn_app import createApp
```

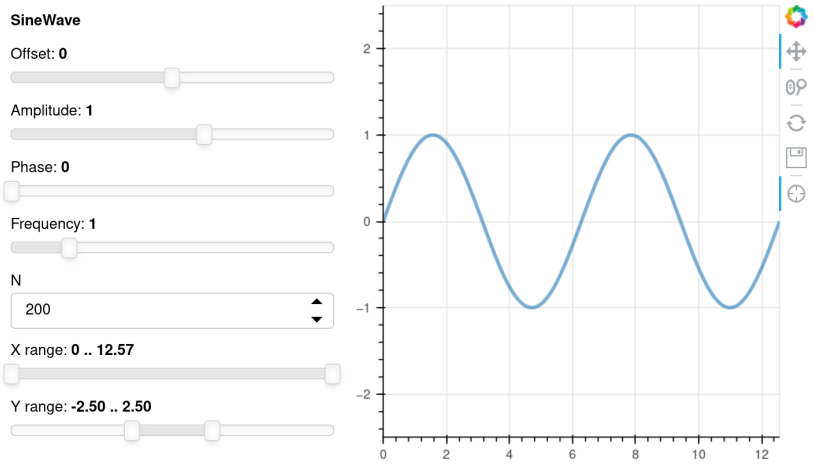
The file structure should now look like this:

```
fastAPI
├── main.py
├── sliders
│   ├── pn_app.py
│   └── sinewave.py
└── templates
    └── base.html
```

You can now start the server with:

```
$ bin/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.
```

You should then see the following in your web browser under the URL `http://127.0.0.1:8000`:


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

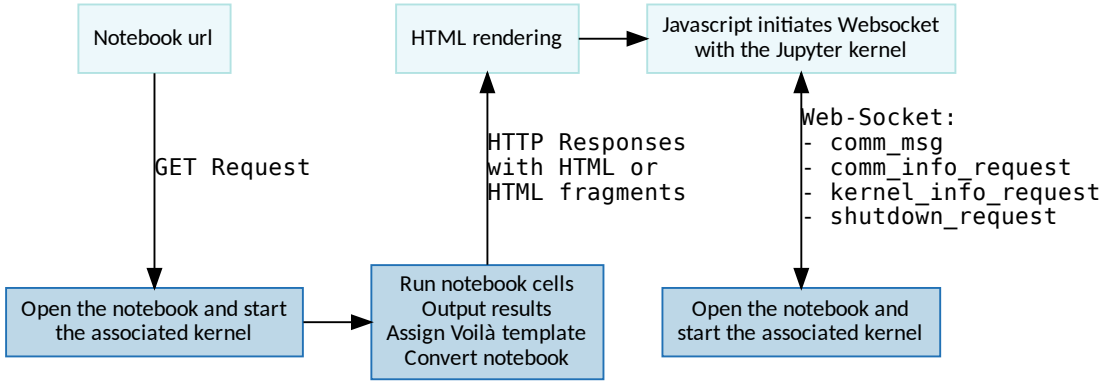
13.4 Voilà

Voilà was developed by QuantStack.

13.4.1 Features

- Voilà supports *interactive Jupyter widgets*, including round trips to the kernel. Custom widgets like bqplot, ipyleaflet, ipyvolume, ipympl, *ipysheet*, plotly, ipywebrtc etc. are also supported.
- Voilà does not allow arbitrary code execution by dashboard users.
- Voilà is based on Jupyter standard protocols and file formats and works with any *Jupyter-Kernel*: C++, Python, Julia. This makes it a language-independent dashboard system.
- Voilà is expandable. It contains a flexible *Template* system for creating extensive layouts.

13.4.2 Execution model



An important aspect of this execution model is that the frontend cannot specify which code is executed by the backend. Unless otherwise specified with the option `--strip-sources=False`, the source code of the rendered notebook does not even reach the frontend. The Voilà instance of `jupyter_server` does not allow execution requests by default.

Warning: The current version of Voilà does not respond to the first `GET` request until all cells have been executed. This can take longer. However, work is being done to enable progressive rendering, see feat: progressive cell rendering.

See also:

- Voilà Gallery
- And voilà!

Installation and use

Installation

voilà can be installed with:

```
$ pipenv install voila
Installing voila...
...
```

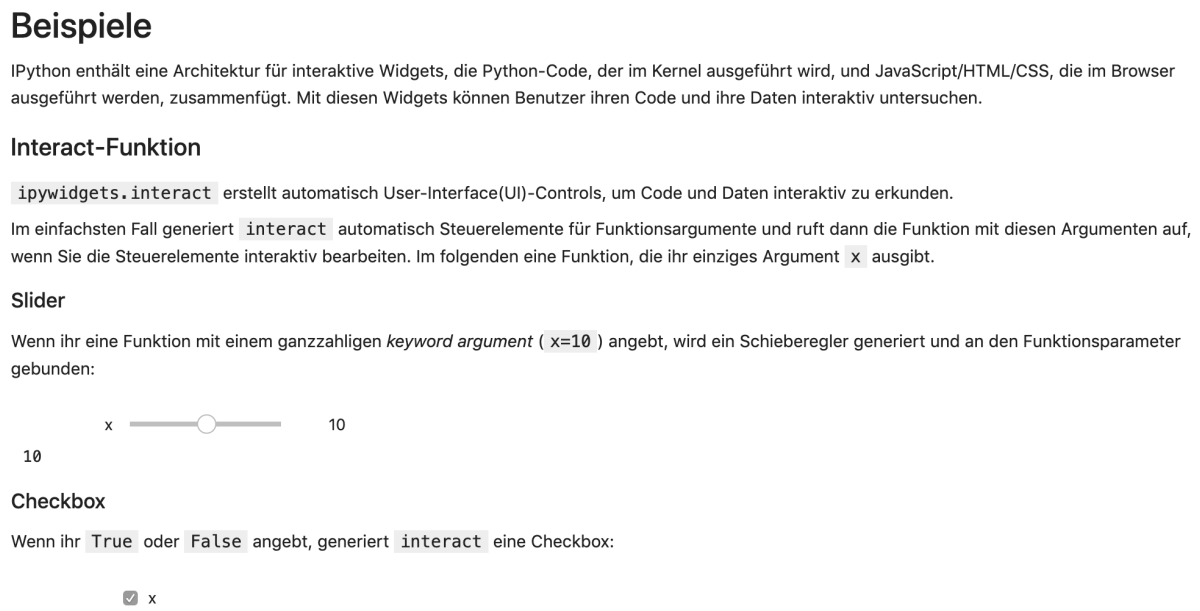
Start

... as a stand-alone application

You can check the installation, e.g. with:

```
$ pipenv run voila docs/dashboards/voila/bqplot_vuetify_example.ipynb
...
[Voila] Voilà is running at:
http://localhost:8866/
```

Your standard browser should open and display the `voila` examples from our tutorial:



Alternatively, you can also display a directory with all the notebooks it contains:

```
$ pipenv 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

It is also possible to display the source code with:

```
$ pipenv run voila --strip_sources=False docs/dashboards/voila/bqplot_vuetify_example.
↪ipynb
...
```

Note: Note that the code is only displayed. Voilà does not allow users to edit or run the code.

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);
```


```
10
```

Checkbox

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

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


```
True
```

Textbereich

Usually the `light` theme is used; however, you can also choose the `dark` theme:

```
$ pipenv run voila --theme=dark docs/dashboards/voila/bqplot_vuetify_example.ipynb
...
```

... as an extension of the Jupyter server

Alternatively you can start voilà as an extension of the Jupyter server:

```
$ pipenv run jupyter notebook
...
```

Then you can call up voilà, e.g. under the URL `http://localhost:8888/voila`.

Templating

Voilà gridstack

gridstack.js is a jQuery plugin for widget layouts. This enables multi-column drag and drop grids and customizable layouts suitable for Bootstrap v3. It also works with knockout.js and touch devices.

The Gridstack Voilà template uses the metadata of the notebook cells to design the notebook's layout. It is supposed to support the entire specification for the outdated *Jupyter Dashboards*.

voila-vuetify

voila-vuetify is a template for using Voilà with the Material Design Component Framework Vuetify.js.

Installation

```
$ pipenv install bqplot ipyvuetify voila-vuetify
```

Usage

To use `voila-vuetify` in a notebook, you first have to import `ipyvuetify`:

```
import ipyvuetify as v
```

Then you can create a layout, for example with:

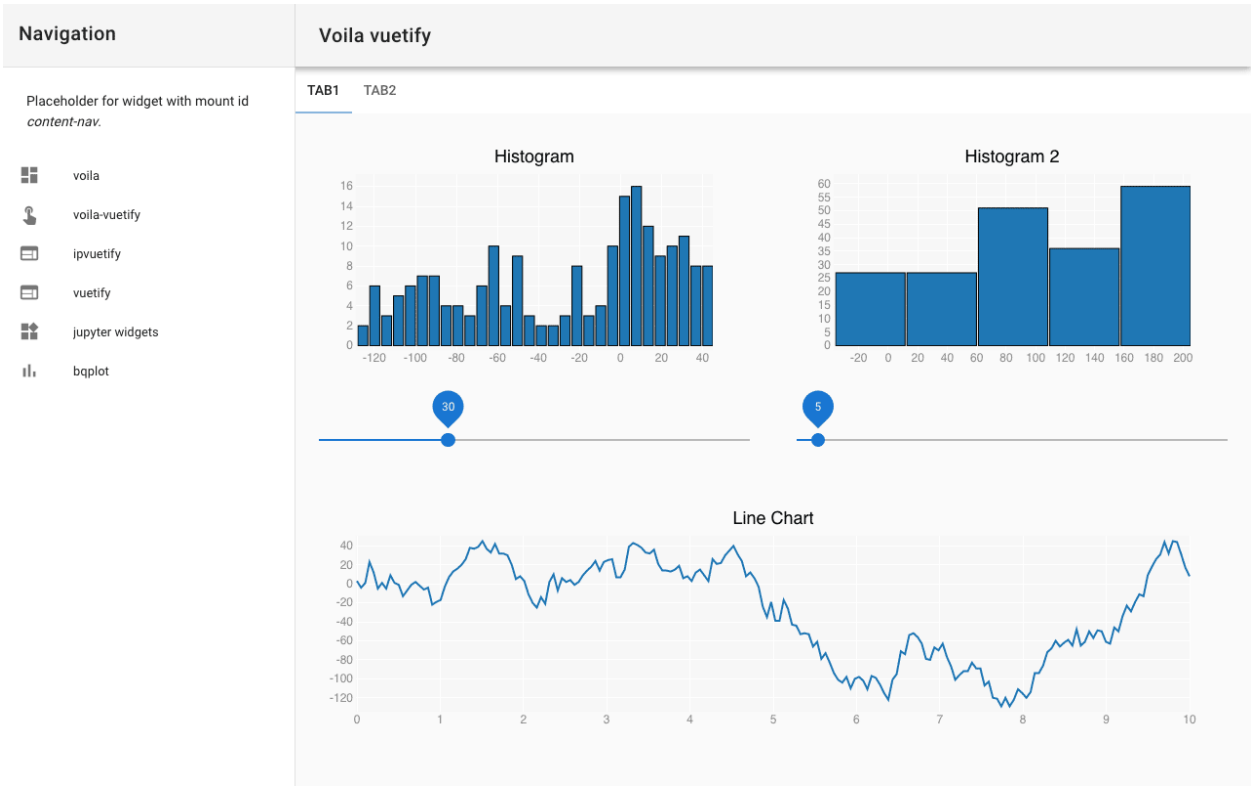
```
v.Tabs(_metadata={'mount_id': 'content-main'}, children=[
    v.Tab(children=['Tab1']),
    v.Tab(children=['Tab2']),
    v.TabItem(children=[
        v.Layout(row=True, wrap=True, align_center=True, children=[
            v.Flex(xs12=True, lg6=True, xl4=True, children=[
                fig, slider
            ]),
            v.Flex(xs12=True, lg6=True, xl4=True, children=[
                figHist2, sliderHist2
            ]),
            v.Flex(xs12=True, xl4=True, children=[
                fig2
            ]),
        ])
    ]),
    v.TabItem(children=[
        v.Container(children=['Lorum ipsum'])
    ])
])
```

You can use *bqplot_vuetify_example.ipynb* with:

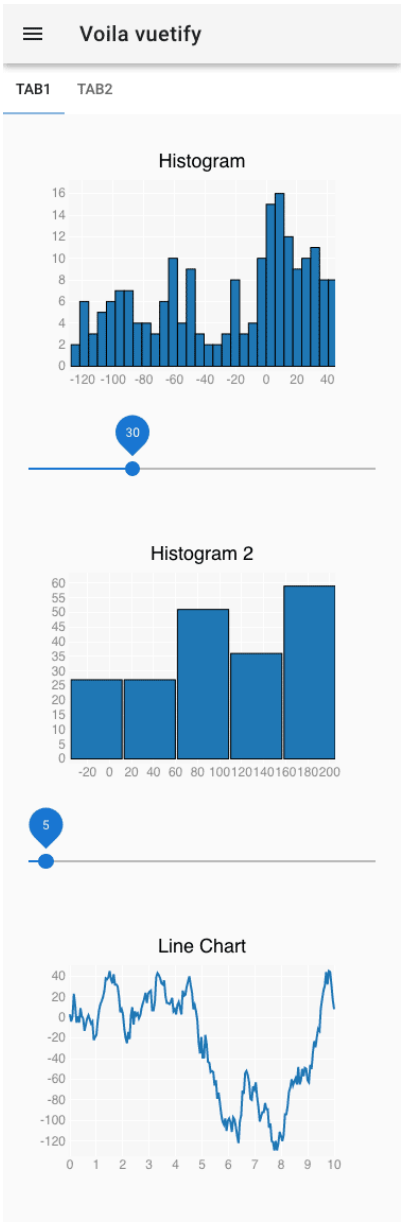
```
$ pipenv run voila --template vuetify-default bqplot_vuetify_example.ipynb
```

Then your standard browser will open the URL `http://localhost:8866/` and show you the plots in Responsive Material Design.

Example for Voilà-vuetify with the monitor resolution of a laptop MDPI screen:



Example for Voilà-vuetify with the monitor resolution of an iPhone X:



voila-debug

voila-debug is a template for displaying debug information when working on Voilà applications.

Installation

```
$ pipenv install voila-debug
```

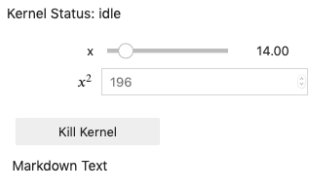
Usage

You can use the template *debug.ipynb* with:

```
$ pipenv run voila --template=debug --VoilaExporter.template_file=debug.tpl
```

This will open your default browser with the URL `localhost:8866`.

Then you can take a closer look at how it works at `http://localhost:8866/voila/render/docs/dashboards/voila/debug.ipynb`.



In addition to an example widget, it contains a code cell for exiting the kernel:

```
import os

def kill_kernel(change):
    os._exit(0)

button = widgets.Button(description="Kill Kernel")
button.on_click(kill_kernel)
button
```

voila-reveal

voila-reveal is a template for slideshows based on RevealJS.

Installation

```
$ pipenv install voila-reveal
```

Usage

You can use the template with:

```
$ pipenv run voila --template=reveal reveal.ipynb
```

Additional options can be used to override the default settings, for example to change the default value for transition `Fade` to `Zoom` with:

```
$ pipenv run voila --template=reveal --VoilaConfiguration.resources="{'reveal': {
↪'transition': 'zoom'}}" reveal.ipynb
```

If configuration options are to be saved permanently, a `conf.json` file can be created in `share/jupyter/voila/templates/reveal/`:

```
{
  "traitlet_configuration": {
    "resources": {
      "reveal": {
        "scroll": false,
        "theme": "simple",
        "transition": "zoom"
      }
    }
  }
}
```

You can then turn your notebook into a slideshow in *View* → *Cell Toolbar* → *Slideshow*. In a cell toolbar you can choose between

Slide
: left to right

Sub-Slide
: top to bottom

Fragment
: stops inside a slide

Notes
: Speaker notes opened in a new window when the presenter press the `t` key

If you want to publish your slideshow on binder, you must write the following tag in the metadata of the notebook in *Edit* → *Edit Notebook Metadata*:

```
"rise": {
    "autolaunch": true
}
```

You can also use the chalkboard reveal plugin in the metadata of the notebook:

```
"rise": {
  "enable_chalkboard": true
}
```

Create your own templates

A Voilà template is a folder that is located in the virtual environment at `share/jupyter/voila/templates` and for example, contains the following:

```
/Users/veit/.local/share/virtualenvs/jupyter-tutorial--q5BvmfG/share/jupyter/voila/
↪templates/mytheme
├── conf.json
├── nbconvert_templates
│   └── voila.tpl
├── static
│   ├── mytheme.js
│   └── mytheme.css
└── templates
    ├── 404.html
    ├── browser-open.html
    ├── error.html
    ├── page.html
    └── tree.html
```

conf.json
: Configuration file that for example refers to the basic template:

```
{"base_template": "default"}
```

nbconvert_templates
: Custom templates for *nbconvert*.

static
: Directory for static files.

templates
: Custom tornado templates.

bqplot_vuetify_example.ipynb

Import

[1]:
```
import ipyvuetify as v
```

First histogram plot

[2]:
```
import bqplot
import ipywidgets as widgets
import numpy as np

from bqplot import pyplot as plt


n = 200
```

(continues on next page)

(continued from previous page)

```
x = np.linspace(0.0, 10.0, n)
y = np.cumsum(np.random.randn(n) * 10).astype(int)

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=147.0, min=-75.0))]...
```

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)
```

Line chart

[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':...
```

Add `BrushIntervalSelector`

[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]
```

(continues on next page)

(continued from previous page)

```
brushintsel.observe(update_range, "selected")
fig2.interaction = brushintsel
```

Second histogram plot

[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)
```

Set up voilà vuetify layout

The Voila vuetify template does not show the output of the Jupyter Notebook, only the widget with the `mount_id` metadata.

[7]:
```
v.Tabs(
    _metadata={"mount_id": "content-main"},
    children=[
        v.Tab(children=["Tab1"]),
```

(continues on next page)

(continued from previous page)

```
        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]),
                    ],
                )
            ]
        ),
        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())
```

13.5 jupyter-flex

Jupyter extension that turns notebooks into dashboards:

- uses Markdown headers and Jupyter notebook cell tags to define the layout and components of the dashboard
- flexible and easy way to define row- and column-oriented layouts
- uses *nbconvert* for static reports
- uses *Voilà* for dynamic applications with a Jupyter *kernel*
- *ipywidgets* support

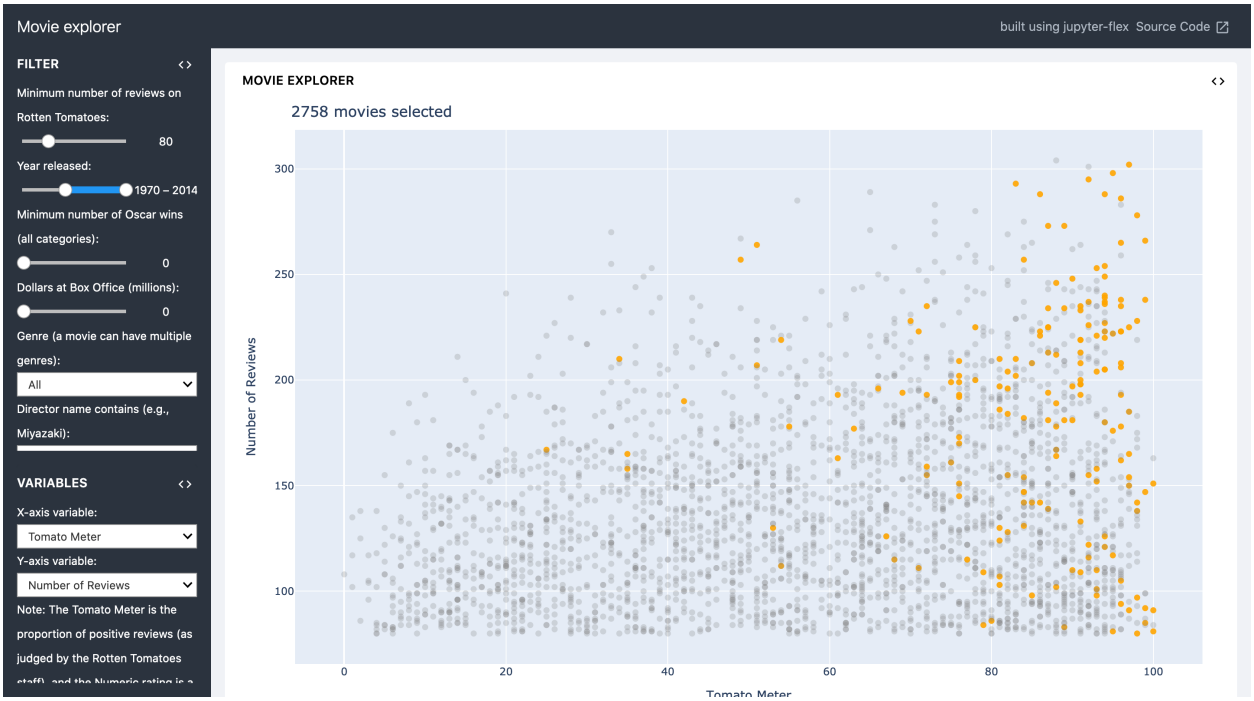
See also:

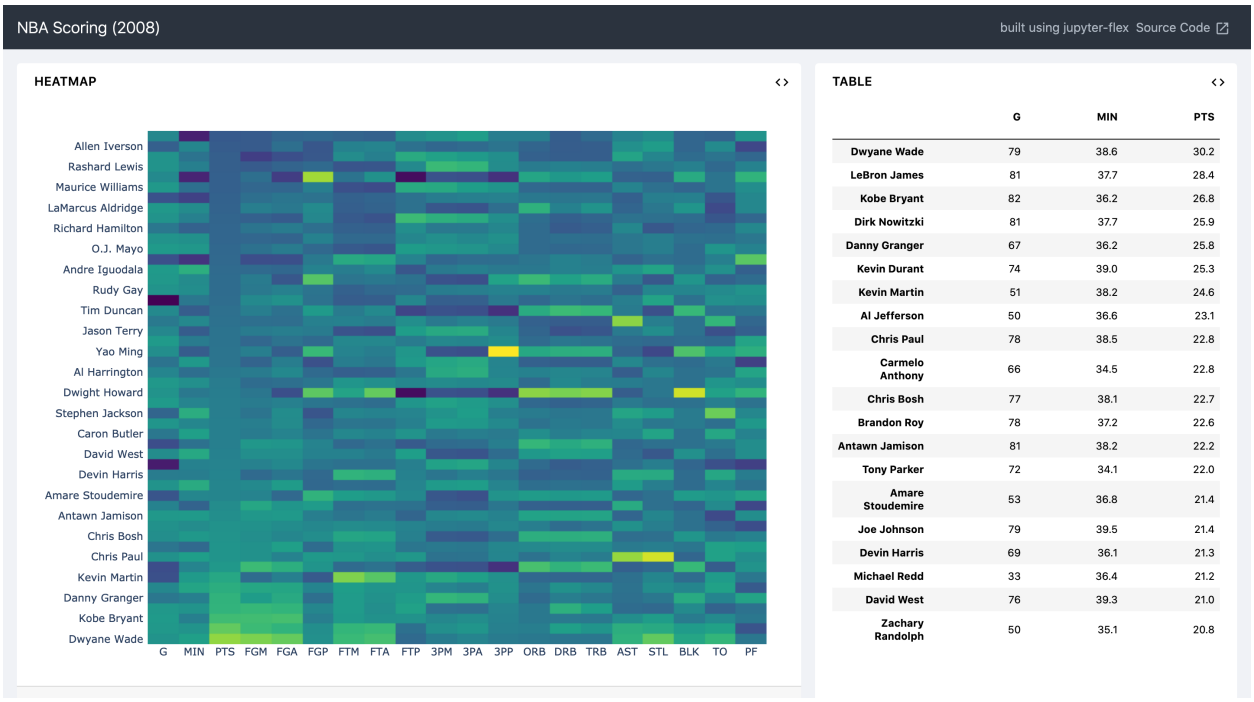
- Docs
- GitHub

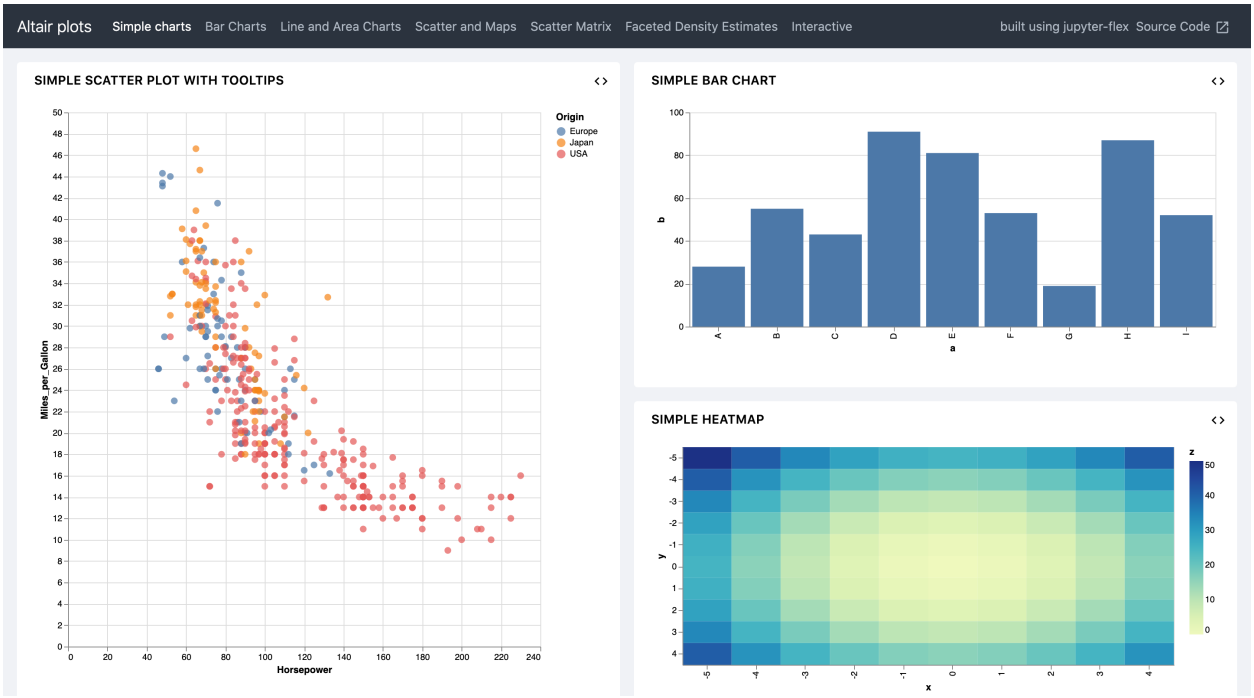
13.5.1 Examples

13.5.2 Installation

```
$ pipenv install jupyter-flex
```


Movie explorer
built using jupyter-flex Source Code
FILTER
Minimum number of reviews on Rotten Tomatoes:
80
Year released:
1970 – 2014
Minimum number of Oscar wins (all categories):
0
Dollars at Box Office (millions):
0
Genre (a movie can have multiple genres):
All
Director name contains (e.g., Miyazaki):
VARIABLES
X-axis variable:
Tomato Meter
Y-axis variable:
Number of Reviews
Note: The Tomato Meter is the proportion of positive reviews (as judged by the Rotten Tomatoes
MOVIE EXPLORER
2758 movies selected
Number of Reviews
300
250
200
150
100
0
20
40
60
80
100
Tomato Meter


NBA Scoring (2008)
built using jupyter-flex Source Code
HEATMAP
Allen Iverson
Rashard Lewis
Maurice Williams
LaMarcus Aldridge
Richard Hamilton
O.J. Mayo
Andre Iguodala
Rudy Gay
Tim Duncan
Jason Terry
Yao Ming
Al Harrington
Dwight Howard
Stephen Jackson
Caron Butler
David West
Devin Harris
Amare Stoudemire
Antawn Jamison
Chris Bosh
Chris Paul
Kevin Martin
Danny Granger
Kobe Bryant
Dwyane Wade
G MIN PTS FGM FGA FGP FTM FTA FTP 3PM 3PA 3PP ORB DRB TRB AST STL BLK TO PF
TABLE
G MIN PTS
Dwyane Wade 79 38.6 30.2
LeBron James 81 37.7 28.4
Kobe Bryant 82 36.2 26.8
Dirk Nowitzki 81 37.7 25.9
Danny Granger 67 36.2 25.8
Kevin Durant 74 39.0 25.3
Kevin Martin 51 38.2 24.6
Al Jefferson 50 36.6 23.1
Chris Paul 78 38.5 22.8
Carmelo Anthony 66 34.5 22.8
Chris Bosh 77 38.1 22.7
Brandon Roy 78 37.2 22.6
Antawn Jamison 81 38.2 22.2
Tony Parker 72 34.1 22.0
Amare Stoudemire 53 36.8 21.4
Joe Johnson 79 39.5 21.4
Devin Harris 69 36.1 21.3
Michael Redd 33 36.4 21.2
David West 76 39.3 21.0
Zachary Randolph 50 35.1 20.8


Altair plots
Simple charts
Bar Charts
Line and Area Charts
Scatter and Maps
Scatter Matrix
Faceted Density Estimates
Interactive
built using jupyter-flex
Source Code
SIMPLE SCATTER PLOT WITH TOOLTIPS
Origin
Europe
Japan
USA
Miles_per_Gallon
Horsepower
SIMPLE BAR CHART
b
a
A
B
C
D
E
F
G
H
I
SIMPLE HEATMAP
y
x
z

CHAPTER

FOURTEEN

SPHINX

Sphinx, a documentation tool that converts reStructuredText into HTML or PDF, EPub and man pages. The Jupyter tutorial is also created with Sphinx.

Originally developed for Python documentation, Sphinx is now used in almost all Python projects, including NumPy and SciPy, Matplotlib, pandas and SQLAlchemy.

With *nbsphinx*, Jupyter Notebooks can also be integrated into Sphinx. *Executable Books*, on the other hand, is a collection of open-source tools that allow you to write Markdown and Jupyter Notebooks, execute content and insert it into your book, among other things.

14.1 nbsphinx

nbsphinx is a Sphinx extension that provides a parser for `*.ipynb` files: Jupyter Notebook code cells are displayed in both HTML and LaTeX output. Notebooks with no output cells saved are automatically created during the Sphinx build process.

14.1.1 Installation

```
$ pipenv install sphinx nbsphinx
```

Requirements

- *nbconvert*

14.1.2 Configuration

Configure Sphinx

1. Creating a documentation with Sphinx:

   ```
   $ pipenv run python -m sphinx.cmd.quickstart
   ```

2. The Sphinx configuration file `conf.py` is then located in the newly created directory. In this, `nbsphinx` is added as an extension and notebook checkpoints are excluded:

```
extensions = [
    ...
    "nbsphinx",
]
...
exclude_patterns = [
    ...
    "**/.ipynb_checkpoints",
]
```

You can find an example in the /conf.py file of the Jupyter tutorial.

You can make further configurations for nbsphinx.

Timeout

In the standard setting of nbsphinx, the timeout for a cell is set to 30 seconds. You can change this for your Sphinx project in the conf.py file with nbsphinx_timeout = 60.

Alternatively, you can also specify this for individual code cells in the metadata of the code cell:

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "nbsphinx": {
     "timeout": 60
   }
  }
 ]
}
```

If the timeout is to be deactivated, -1 can be specified.

Custom formats

Libraries such as jupytext save notebooks in other formats, for example as R-Markdown with the suffix Rmd. So that these can also be executed by nbsphinx, further formats can be specified in the Sphinx configuration file conf.py with nbsphinx_custom_formats, for example

```
import jupytext


nbsphinx_custom_formats = {
    ".Rmd": lambda s: jupytext.reads(s, ".Rmd"),
}
```

Configure cells

Don't show cell

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
    "nbsphinx": "hidden"
   }
  }
 ]
}
```

nbsphinx-toctree

With this instruction Sphinx will create a table of contents within a notebook cell, for example

```
{
 "cells": [
  {
   "cell_type": "markdown",
   "metadata": {
    "nbsphinx-toctree": {
      "maxdepth": 2
    },
   "source": [
    "The following title is rendered as ``toctree caption``.\n",
    "\n",
    "## Content\n",
    "\n",
    "[A notebook](a-notebook.ipynb)\n",
    "\n",
    "[An external HTML link](https://jupyter-tutorial.readthedocs.io/)\n"
   ]
   }
  }
 ]
}
```

Further options you will find in the Sphinx documentation.

14.1.3 Build

1. Now you can add your `*.ipynb` file in the table of contents of your `index.rst` file, see for example jupyter-tutorial/notebook/testing/index.rst
2. Finally, you can generate the pages, for example HTML with `$ pipenv run python -m sphinx` *SOURCE_DIR BUILD_DIR* or `$ pipenv run python -m sphinx` *SOURCE_DIR BUILD_DIR* `-j` NUMBER_OF_PROCESSES where `-j` is the number of processes to run in parallel.

   If you want to create a LaTeX file, you can do so with `$ pipenv run python -m sphinx` *SOURCE_DIR BUILD_DIR* `-b latex`.

3. Alternatively, you can have the documentation generated automatically with `sphinx-autobuild`. It can be installed with `$ pipenv run python -m pip install sphinx-autobuild`.

   The automatic creation can then be started with `$ pipenv run python -m sphinx_autobuild` *`SOURCE_DIR BUILD_DIR`*.

   This starts a local web server that provides the generated HTML pages at `http://localhost:8000/`. And every time you save changes in the Sphinx documentation, the corresponding HTML pages are regenerated and the browser view is updated.

   You can also use this to automatically generate the LaTeX output: `$ pipenv run python -m sphinx_autobuild` *`SOURCE_DIR BUILD_DIR`* `-b latex`.

4. Another alternative is publication on readthedocs.org.

   To do this, you first have to create an account at https://readthedocs.org/ and then connect your GitLab, Github or Bitbucket account.

Markdown cells

Equations

Equations can be specified *inline* between $ characters, for example

```
$\text{e}^{i\pi} = -1$
```

Equations can also be expressed line by line, for example

```
\begin{equation}
\int\limits_{-\infty}^\infty f(x) \delta(x - x_0) dx = f(x_0)
\end{equation}
```

See also:

- Equation Numbering

Quotes

`nbsphinx` supports the same syntax for quotations as nbconvert:

```
<cite data-cite="kluyver2016jupyter">Kluyver et al. (2016)</cite>
```

Alert boxes

```
<div class="alert alert-block alert-info">

**Note**

This is a notice!
</div>

<div class="alert alert-block alert-success">

**Success**

This is a success notice!
</div>

<div class="alert alert-block alert-warning">
```

(continues on next page)

(continued from previous page)

```
**Warning**

This is a warning!
</div>

<div class="alert alert-block alert-danger">

**Danger**

This is a danger notice!
</div>
```

Links to other notebooks

```
a link to a notebook in a subdirectory](subdir/notebook-in-a-subdir.ipynb)
```

Links to `*.rst` files

```
[reStructuredText file](rst-file.rst)
```

Links to local files

```
[Pipfile](Pipfile)
```

Code cells

Javascript

Javascript can be used for the generated HTML, for example:

```
%%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 Galleries

nbsphinx provides support for creating thumbnail galleries from a list of Jupyter notebooks. This functionality is based on Sphinx-Gallery and extends nbsphinx to work with Jupyter notebooks instead of Python scripts.

Sphinx-Gallery also directly supports Matplotlib, seaborn and Mayavi.

Installation

Sphinx-Gallery can be installed for Sphinx 1.8.3 with

```
$ pipenv install sphinx-gallery
```

Configuration

In order for Sphinx-Gallery to be used, it must also be entered into the `conf.py` file:

```
extensions = [
    "nbsphinx",
    "sphinx_gallery.load_style",
]
```

You can then use Sphinx-Gallery in two different ways:

1. With the reStructuredText directive `.. nbgallery::`.

   See also:

   Thumbnail Galleries

2. In a Jupyter notebook, by adding an `nbsphinx-gallery` tag to the metadata of a cell:

   ```
   {
       "tags": [
           "nbsphinx-gallery"
       ]
   }
   ```

14.2 Executable Books

Executable Books is a collection of open-source tools that facilitate the publication of computational narratives using the Jupyter ecosystem, primarily:

Jupyter Book
: *Sphinx* distribution that allows you to write content in Markdown and Jupyter Notebooks, execute content and insert it into your book.

  See also:

  - jupyterbook.org is the landing page of the project.
  - gallery.jupyterbook.org is a gallery of Jupyter books.
  - github.com/executablebooks/jupyter-book is the project's repository.

MyST
: is an extensible semantic variant of Markdown designed for scientific and computational narratives. MyST-Markdown is a language- and implementation-independent variant of Markdown supported by several tools.

  See also:

  - mystmd.org is the landing page of the project.
  - spec.mystmd.org describes the MyST specification.

- MyST Enhancement Proposals is a process for proposing and deciding on changes to the MyST specification.

JupyterLab MyST Extension
: renders Markdown cells in *JupyterLab* using MyST Markdown, including interactive references, notes, figure numbering, tabs, cards and grids.

See also:

- github.com/executablebooks/jupyterlab-myst

CHAPTER

FIFTEEN

USE CASES

In some companies, Jupyter notebooks are used to explore the ever-increasing amounts of data. These include:

- 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

CHAPTER

SIXTEEN

INDEX

INDEX