

Project templates

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Why use templates?

- Avoid repetitive tasks
- Standardize project structure
- Include configuration files:
 - Pytest (`pytest.ini`)
 - Sphinx (`conf.py`)
- Include Makefile to automate further steps:
 - Build Sphinx documentation
 - Create virtual environments
 - Initialize git repositories
 - Deploy packages to the PyPI





COOKIE CUTTER

- Not just for Python
- Flexible
 - Edit files in template directory
 - Local
 - Remote

Cookiecutter prompts

```
from cookiecutter import main  
main.cookiecutter(TEMPLATE_REPO)
```

```
project [PROJECT_NAME]:  
>? "My Project"
```

- `cookiecutter()` arguments:
 - `template` : git repository url or path

Cookiecutter prompts

```
from cookiecutter import main  
main.cookiecutter(TEMPLATE_REPO)
```

```
project [PROJECT_NAME]: "My Project"
```

```
Select license:
```

```
1 - MIT
```

```
2 - BSD
```

```
3 - GPL3
```

```
Choose from 1, 2, 3 (1, 2, 3) [1]:
```

```
>?
```

- `cookiecutter()` arguments:
 - `template` : git repository url or path

Cookiecutter defaults

```
from cookiecutter import main  
  
main.cookiecutter(  
    TEMPLATE_REPO_URL,  
    no_input=True  
)
```

- `cookiecutter()` arguments:
 - `template` : git repository url or path
 - `no_input`
 - Suppress prompts
 - Use `cookiecutter.json` defaults
 - Key-value pairs

Override defaults

```
from cookiecutter import main  
  
main.cookiecutter(  
    TEMPLATE_REPO,  
    no_input=True,  
    extra_context={'KEY': 'VALUE'}  
)
```

```
{  
    "project": "Your project's name",  
    "license": ["MIT", "BSD", "GPL3"]  
}
```

- `cookiecutter()` arguments:
 - `template` : git repository url or path
 - `no_input`
 - Suppress prompts
 - Use `cookiecutter.json` defaults
 - Key-value pairs
 - `extra_context`
 - Override defaults

Access JSON files

```
from json import load  
from pathlib import Path  
  
# Local JSON file to dictionary  
load(Path(JSON_PATH).open()).values()  
  
# List local cookiecutter.json keys  
[*load(Path(JSON_PATH).open())]
```

```
from requests import get  
  
# Remote JSON file to dictionary  
get(JSON_URL).json().values()  
  
# List remote cookiecutter.json keys  
[*get(JSON_URL).json()]
```

Jinja2 template

```
{"project": "Project Name",
"author": "Your name (or your organization/company/team)",
"repo": "{{ cookiecutter.project.lower().replace(' ', '_') }}",
}
```

```
project = "My Project"
project.lower().replace(' ', '_')
```

```
my_project
```

Cookiecutter example

```
from cookiecutter.main import cookiecutter
cookiecutter('https://github.com/marskar/cookiecutter', no_input=True,
            extra_context={'project': 'PROJECT_NAME', 'author': 'AUTHOR_NAME'})
```

```
$ cookiecutter https://github.com/marskar/cookiecutter --no-input \
project="PROJECT_NAME" author="AUTHOR_NAME" \
user=USER_NAME description="DESCRIPTION"
```

Cookiecutter example

```
from cookiecutter.main import cookiecutter
cookiecutter('gh:marskar/cookiecutter', no_input=True,
            extra_context={'project': 'PROJECT_NAME', 'author': 'AUTHOR_NAME',
                           'user': 'USER_NAME', 'description': 'DESCRIPTION'})
```

```
$ cookiecutter gh:marskar/cookiecutter --no-input \
project="PROJECT_NAME" author="AUTHOR_NAME" \
user=USER_NAME description="DESCRIPTION"
```



Project structure

```
...  
???.docs  
?  ???.Makefile  
?  ??_.static  
?  ???.authors.rst  
?  ???.changelog.rst  
?  ???.conf.py  
?  ???.index.rst  
?  ???.license.rst  
???.requirements.txt  
???.setup.cfg  
???.setup.py
```

```
???.src  
?  ???.template  
?      ???.__init__.py  
?      ???.template.py  
???.tests  
      ???.conftest.py  
      ???.test_template.py
```

```
$ make html
```

Navigation

Description

License

Authors

Changelog

Module Reference

Test Reference

Quick search

 Go

PROJECT_NAME

Welcome to the documentation of PROJECT_NAME!

Note:

This is the main page of your project's [Sphinx](#) documentation. It is formatted in [reStructuredText](#). Add additional pages by creating rst-files in `docs` and adding them to the [toctree](#) below. Use then [references](#) in order to link them from this page, e.g. [Contributors](#) and [Changelog](#).

It is also possible to refer to the documentation of other Python packages with the [Python domain syntax](#). By default you can reference the documentation of [Sphinx](#), [Python](#), [NumPy](#), [SciPy](#), [matplotlib](#), [Pandas](#), [Scikit-Learn](#). You can add more by extending the `intersphinx_mapping` in your Sphinx's `conf.py`.

The pretty useful extension [autodoc](#) is activated by default and lets you include documentation from docstrings. Docstrings can be written in [Google style](#) (recommended!), [NumPy style](#) and [classical style](#).



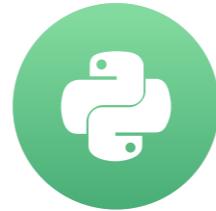
Read the Docs

Let's practice using project templates!

CREATING ROBUST PYTHON WORKFLOWS

Executable projects

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Run a module

```
def print_name_and_file():
    print('Name is', __name__,
          'and file is', __file__)
if __name__ == "__main__":
    print_name_and_file()
```

```
$ python -m prj.src.pkg.main
```

```
Name is __main__ and file is
/Users/USER/prj/src/pkg/main.py
```

```
prj
??? src
??? pkg
??? __init__.py
??? main.py
```

Top-level imports

```
# Import module into __main__.py
(from prj.src.pkg.main
import print_name_and_file)

if __name__ == "__main__":
    print_name_and_file()
```

```
$ python -m prj
```

Name is prj.src.pkg.main and file is
/Users/USER/prj/src/pkg/main.py

```
prj
??__main__.py
??src
??pkg
??__init__.py
??main.py
```

Import error

```
# Import module into __main__.py
(from src.pkg.main
import print_name_and_file)

if __name__ == "__main__":
    print_name_and_file()
```

```
$ python -m pkg
```

```
...
ModuleNotFoundError:
No module named 'src'
```

```
prj
??__main__.py
??src
    ??pkg
        ??__init__.py
        ??main.py
```



Run zipped project

```
import zipapp  
  
zipapp.create_archive('prj')
```

```
$ python -m zipapp prj  
$ python prj.pyz
```

Name is src.pkg.main and file is
prj.pyz/src/pkg/main.py

```
prj  
???. __main__.py  
???. src  
      ???. pkg  
         ???. __init__.py  
         ???. main.py
```



Pass arguments to projects

```
import sys  
  
if __name__ == "__main__":  
    print(sys.argv)
```

```
$ python -m zipapp prj  
$ python prj.pyz hello
```

```
['prj.pyz', 'hello']
```

1. Include a command-line interface (CLI) in `__main__.py`
2. Use `zipapp` to create zipped project
3. Pass shell arguments to project

Zipapp main argument

```
import os
import zipapp

os.remove('prj/__main__.py')

zipapp.create_archive('prj', main='src.pkg.main:print_name_and_file')
```

```
$ rm prj/__main__.py
$ python -m zipapp prj --main src.pkg.main:print_name_and_file
```

Zipapp set interpreter

```
import zipapp  
  
zipapp.create_archive('prj', interpreter="/usr/bin/env python")
```

```
$ python -m zipapp prj --python "/usr/bin/env python"  
$ ./prj.pyz
```

Name is src.pkg.main and file is ./prj.pyz/src/pkg/main.py

Self-contained zipped projects

```
import zipapp  
  
zipapp.create_archive('prj', interpreter="/usr/bin/env python")
```

```
$ python -m pip install --requirement prj/requirements.txt --target prj  
$ python -m zipapp prj --python "/usr/bin/env python"  
$ ./prj.pyz
```

Name is src.pkg.main and file is ./prj.pyz/src/pkg/main.py

Let's make an executable project!

CREATING ROBUST PYTHON WORKFLOWS

Notebook pipelines

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Jupyter nbconvert

- Can be used as a Python library, e.g. our `nbconv()` function
- Can execute notebooks

```
$ jupyter nbconvert --execute --to notebook input.ipynb --output output.ipynb
```

- Cannot pass arguments to code in notebooks



Injected parameters

In [1]:

injected-parameters ✘

Parameters
PARAMETER = VALUE

```
$ papermill input.ipynb output.ipynb --parameters PARAMETER VALUE
```

Default parameters

In [1]:

parameters ✖

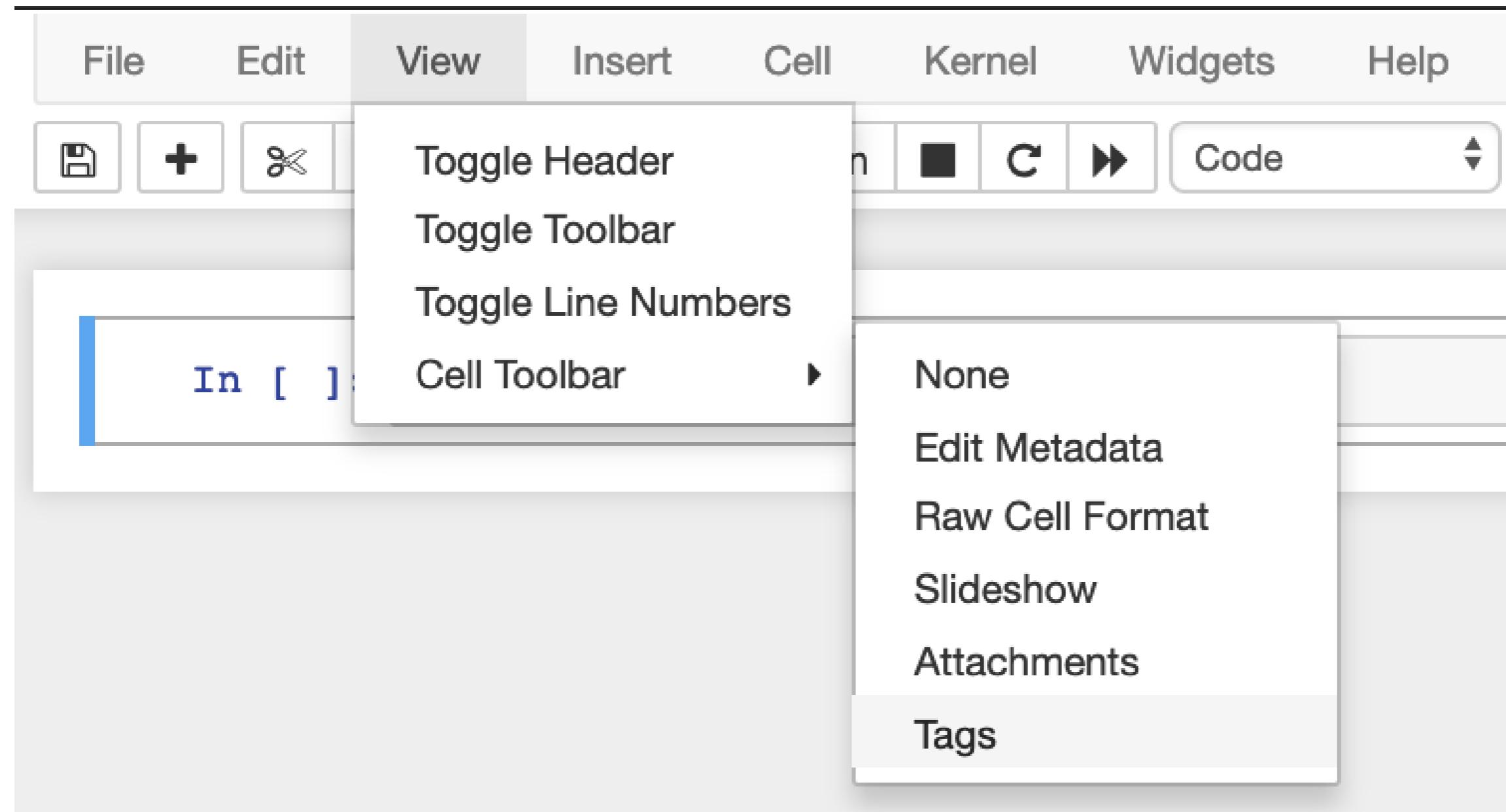
```
1 # This cell is tagged `parameters`  
2 alpha = 0.1  
3 ratio = 0.1
```

...

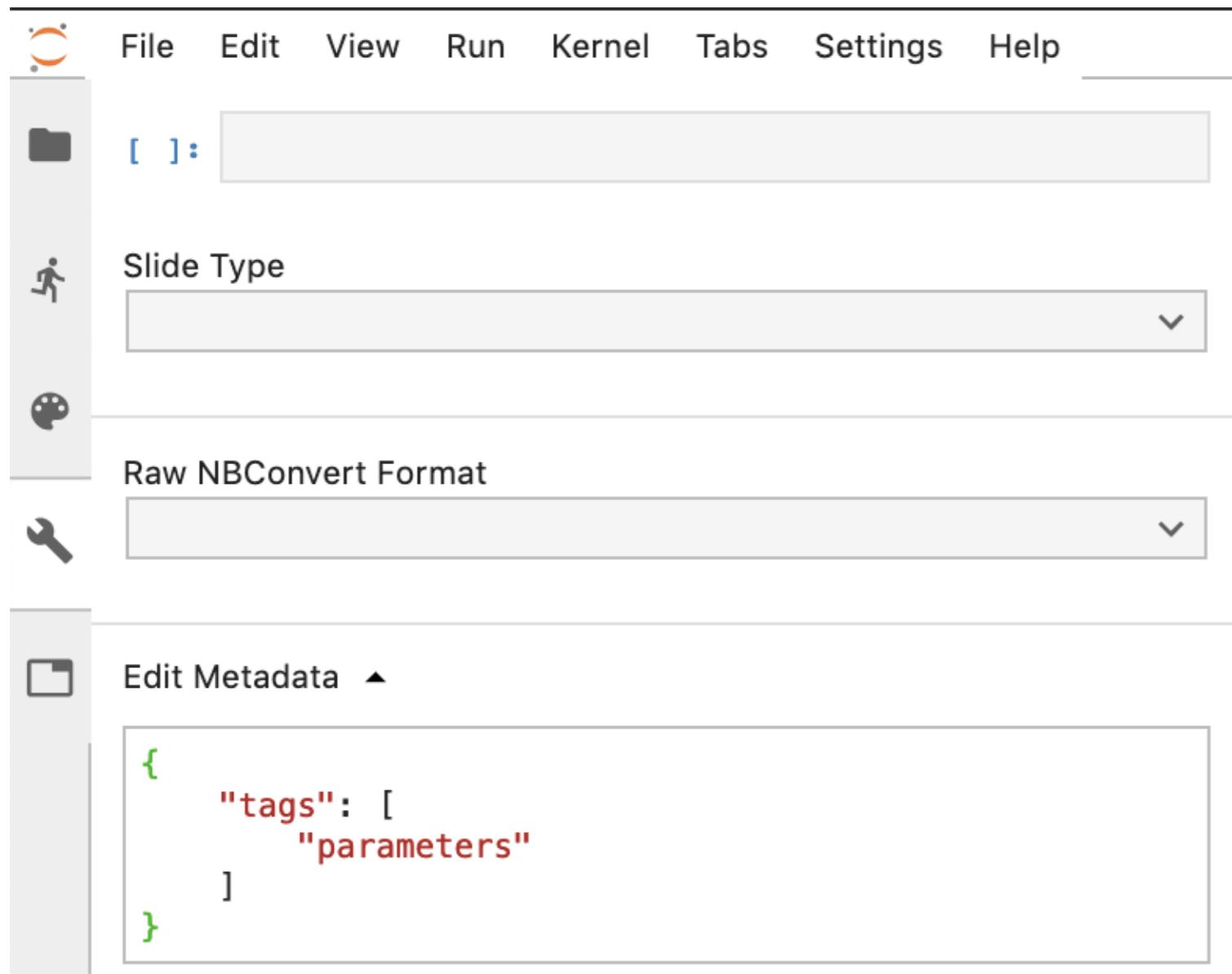
Add tag

```
$ papermill input.ipynb output.ipynb --parameters alpha 0.2
```

Classic notebook interface



JupyterLab interface



- Edit metadata (JupyterLab)

```
{  
  "tags": [  
    "parameters"  
  ]  
}
```

Jupyter nbformat

```
import nbformat

nb = nbformat.read('NOTEBOOK.ipynb',
                    as_version=4)

nb.cells[0].metadata = {'tags':
                        ['parameters']}

nb.cells[0].source = "alpha = 0.4"

nbformat.write(nb, 'NOTEBOOK.ipynb')
```

- `nbformat.read()` : read in a notebook
- Edit the first cell
 - Add a parameters tag to `metadata`
 - Add a default parameter to `source`
- `nbformat.write()` : overwrite the original

Execute notebook

| <code>pm.execute_notebook()</code> | \$ papermill |
|---------------------------------------|--|
| <code>input_path: str</code> | <code>NOTEBOOK_PATH</code> |
| <code>output_path: str</code> | <code>OUTPUT_PATH</code> |
| <code>cwd: Any = None</code> | <code>--cwd</code> |
| <code>parameters: Any = None</code> | <code>-p, --parameters</code> |
| <code>kernel_name: Any = None</code> | <code>-k, --kernel</code> |
| <code>report_mode: Any = False</code> | <code>--report-mode / --not-report-mode</code> |

Parametrize

```
import papermill as pm  
  
names = ['alpha', 'ratio']  
values = [0.6, 0.4]  
  
param_dict = dict(zip(names, values))  
  
pm.execute_notebook(  
    'IN.ipynb',  
    'OUT.ipynb',  
    kernel_name='python3',  
    parameters=param_dict  
)
```

- Save parameter names and values as lists
- Create a dictionary of custom parameters
- Pass the dictionary
 - To the `execute_notebook()` function
 - As its `parameters` argument

Overwrite defaults

In [1]:

parameters ✖

```
# Parameters  
alpha = 0.6
```

In [2]:

injected-parameters ✖

```
# Parameters  
alpha = 0.8
```

Notebook parameters

```
# Parameters  
dataset_name = "diabetes"  
model_type = "ensemble"  
model_name = "RandomForestRegressor"  
hyperparameters = {"max_depth": 3, "n_estimators": 100, "random_state": 0}
```

Use parameters inside notebooks

```
from importlib import import_module
from typing import Optional, Dict

def get_model(model_type, model_name, hyperparameters=None):
    model = getattr(import_module('sklearn.'+model_type), model_name)
    return model(**hyperparameters) if hyperparameters else model()

keys = ['model_type', 'model_name', 'hyperparameters']
vals = [model_type, model_name, hyperparameters]
model = get_model(**dict(zip(keys, vals)))
```



scrapbook

- `sb.glue('alpha', alpha)` : record a variable
- `sb.read_notebook('NOTEBOOK.ipynb')` : return a `scrapbook.models.Notebook` object
 - `scraps` : a dictionary of recorded values
 - `scrap_dataframe` : a dataframe of recorded values
 - `papermill_metrics` : a dataframe of execution times
 - `parameter_dataframe` : a dataframe of notebook parameters

Summarize

```
import papermill as pm  
  
names = ['alpha', 'ratio']  
values = [0.6, 0.4]  
  
param_dict = dict(zip(names, values))  
  
pm.execute_notebook(  
    'IN.ipynb',  
    'OUT.ipynb',  
    kernel_name='python3',  
    parameters=param_dict  
)
```

```
import scrapbook as sb  
  
nb = sb.read_notebook('OUT.ipynb')  
  
nb.parameter_dataframe
```

| | name | value | type | filename |
|---|-------|-------|-----------|-----------|
| 2 | alpha | 0.6 | parameter | OUT.ipynb |
| 3 | ratio | 0.4 | parameter | OUT.ipynb |

**Let's practice using
papermill and
scrapbook!**

CREATING ROBUST PYTHON WORKFLOWS

Parallel computing

CREATING ROBUST PYTHON WORKFLOWS



Martin Skarzynski

Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Parallel computing

- Execute multiple jobs at once (in parallel)
- Decrease code execution time

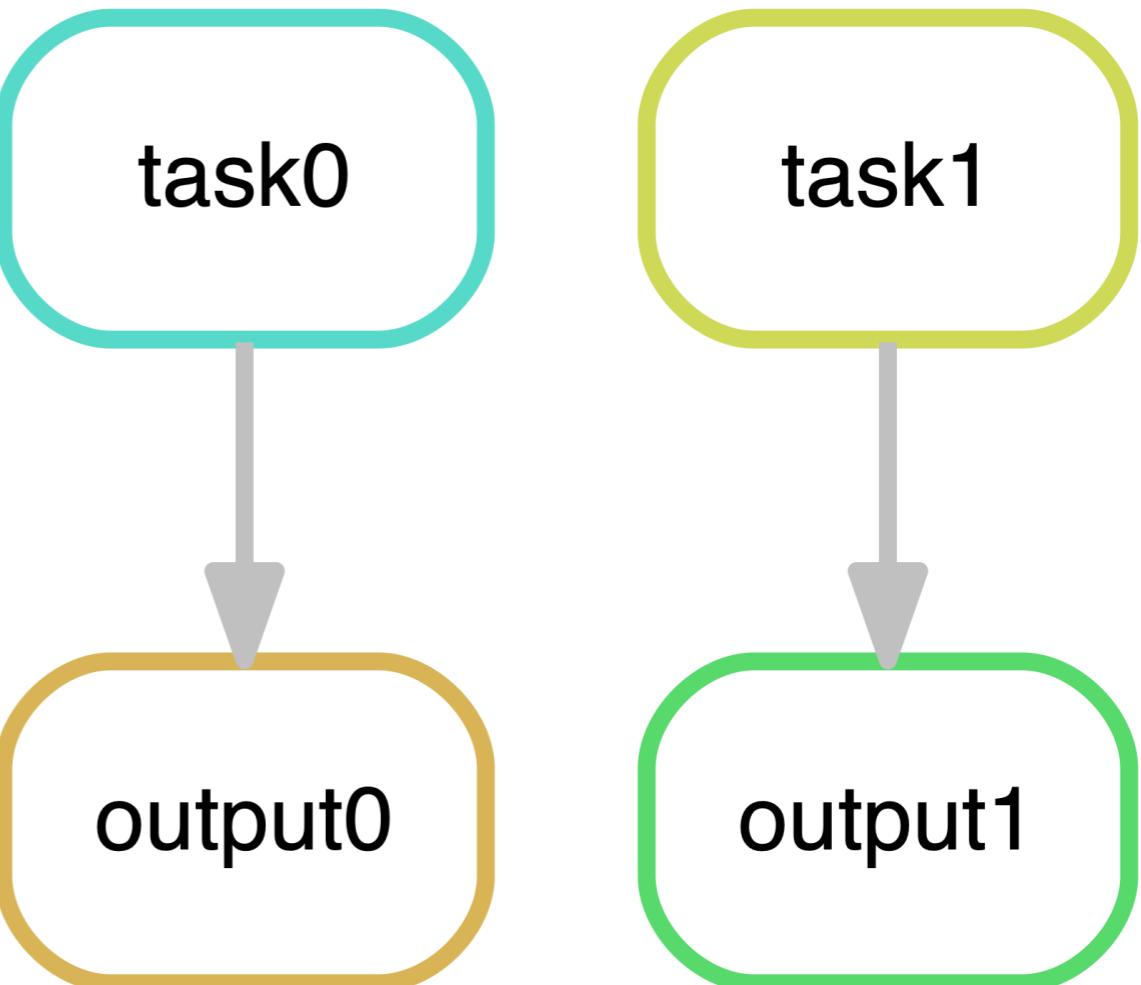
Example:

- Run multiple Make recipes in parallel

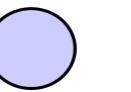
```
$ make --jobs 2
```

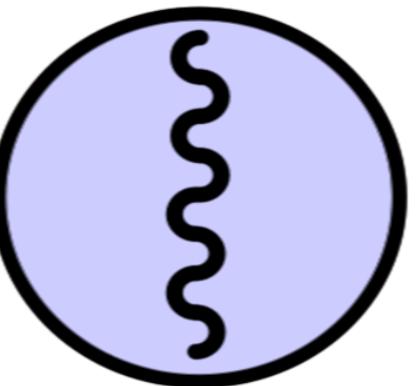
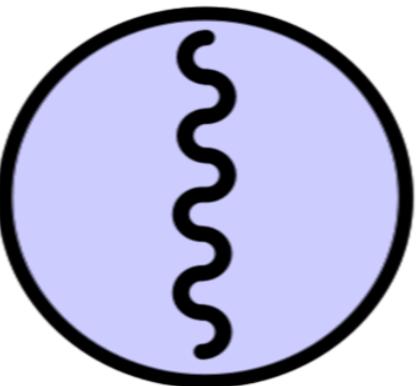
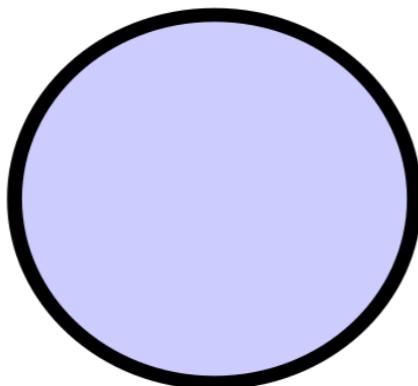
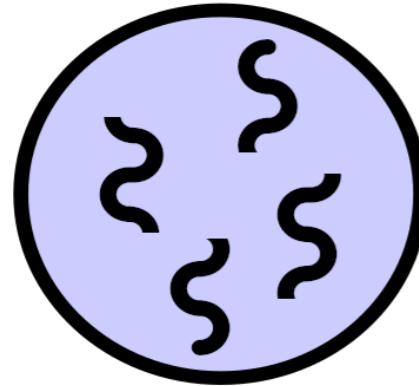
Two parallel computing options:

- Multiprocessing
- Multithreading



Threads and processes

- Thread ~ task 
- Multithreading
 - Like multitasking
 - Assign multiple tasks to one worker
- Process ~ worker 
- Multiprocessing
 - Like teamwork
 - Give each worker a task



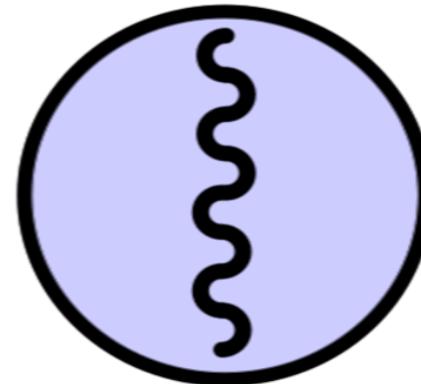
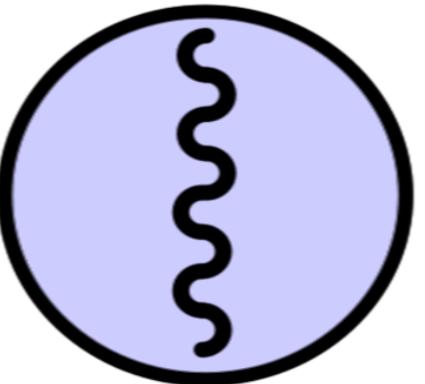
Multiprocessing

```
import time  
  
def task(duration):  
    time.sleep(duration)
```

Thread



- Process ~ worker 
- Multiprocessing
 - Like teamwork
 - Give each worker a task



Multiprocessing

```
import time
from multiprocessing import Pool
from itertools import repeat

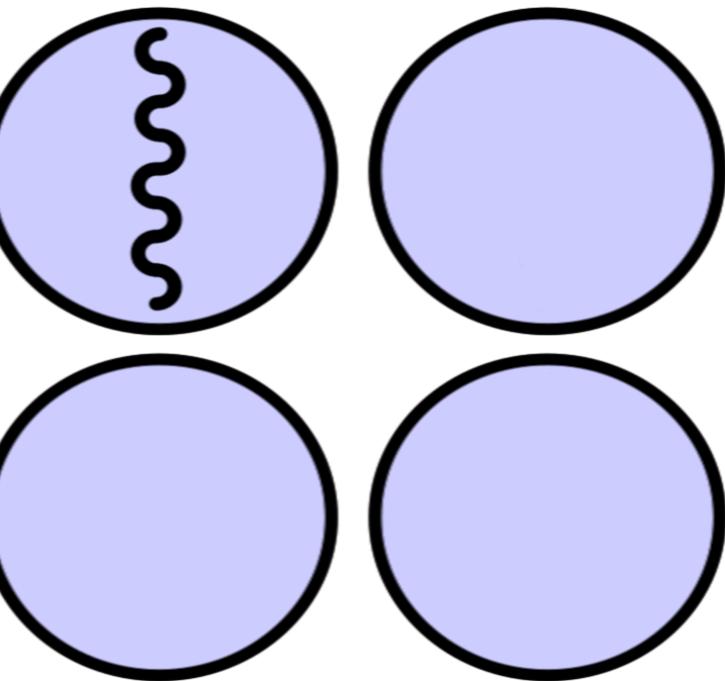
def split_tasks(n_workers, n_tasks, task_duration):
    start = time.time()
    Pool(n_workers).map(task, repeat(task_duration, n_tasks))
    end = time.time()
    print("Workers:", n_workers, "Tasks:", n_tasks, "Seconds:", round(end - start))
```

Sequential execution

```
split_tasks(n_workers=1,  
            n_tasks=4, task_duration=2)
```

Workers: 1 Tasks: 4 Seconds: 8

- 1 worker
- 4 tasks
- 1 task at a time



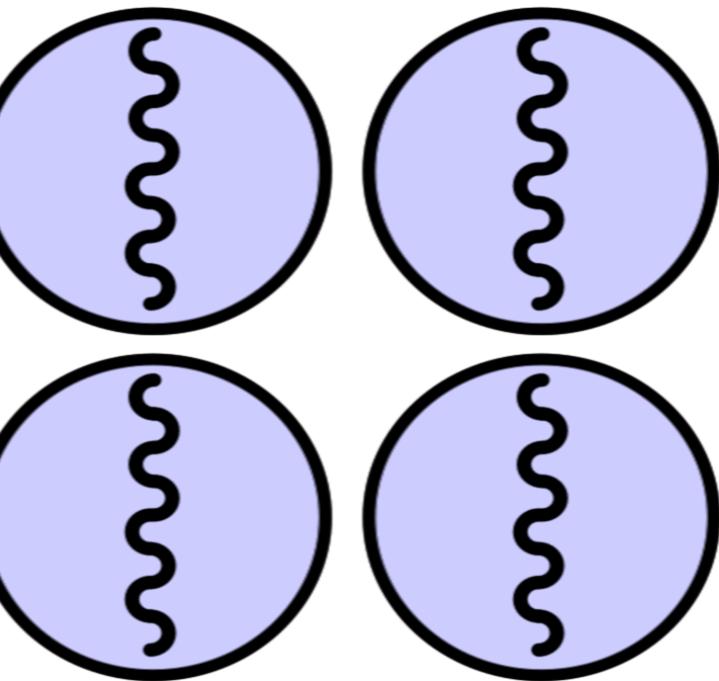
Parallel execution

```
split_tasks(n_workers=4,  
            n_tasks=4, task_duration=2)
```

Workers: 4 Tasks: 4 Seconds: 2

```
from multiprocessing import Pool  
  
Pool(n_workers).map(FUNCTION, ITERABLE)
```

- 4 workers
- 4 tasks
- 4 tasks at a time



Parallelize scikit-learn

```
from dask.distributed import Client  
from sklearn.externals import joblib
```



Parallelize scikit-learn

```
from dask.distributed import Client
from sklearn.externals import joblib

Client(n_workers=1,
       threads_per_worker=4,
       processes=False)
with joblib.parallel_backend('dask'):
    MODEL.fit(x_train, y_train)
```

1. Instantiate the `Client` class
 - Number of workers (`n_workers`)
 - Set `threads_per_worker` ratio
 - Enable threading (`processes=False`)
2. As part of a `with` statement
 - Pass `'dask'` to `parallel_backend()`
3. Inside the context of the `with` statement
 - Call a model instance's `fit()` method

Dask collections

- Can be used interactively with minimal setup
- Dask bags resemble unordered tuples and are limited to one process per thread
- Numpy and Pandas can handle more than one thread per process
- Replace Numpy arrays and Pandas dataframes with analogous Dask Collections

| Dask collection | Similar to | Default scheduler | Advantage |
|-----------------|-------------------|-------------------|--------------------|
| Bag | tuple (unordered) | Multiprocessing | 1 thread / process |
| Array | Numpy Array | Threaded | Easy data sharing |
| DataFrame | Pandas DataFrame | Threaded | Easy data sharing |

Pandas dataframes

```
import pandas as pd  
  
df = pd.read_csv('FILENAME.csv')  
  
(df  
 .groupby('COLUMN_NAME')  
 .mean()  
)
```

1. Import `pandas`
2. Read in csv file as `df`
3. Chain methods
 - `groupby()`
 - `mean()`

Dask dataframes

```
import dask.dataframe as dd  
  
df = dd.read_csv('FILENAME*.csv')  
  
(df  
 .groupby('GROUP')  
 .mean()  
 .compute()  
)
```

1. Import `dask.dataframe`
2. Read in csv file(s) as `df`
3. Chain methods
 - o `groupby()`
 - o `mean()`
 - o `compute()`

Persist dask dataframe

```
import dask.dataframe as dd  
  
df = dd.read_csv('FILENAME*.csv')  
  
df = df.persist()  
  
(df  
    .groupby('GROUP')  
    .mean()  
    .compute()  
)
```

1. Import `dask.dataframe`
2. Read in csv file(s) as `df`
3. Store `df` on disk with `persist()`
4. Chain methods
 - `groupby()`
 - `mean()`
 - `compute()`

Let's practice using Dask!

CREATING ROBUST PYTHON WORKFLOWS

Wrap-up

CREATING ROBUST PYTHON WORKFLOWS

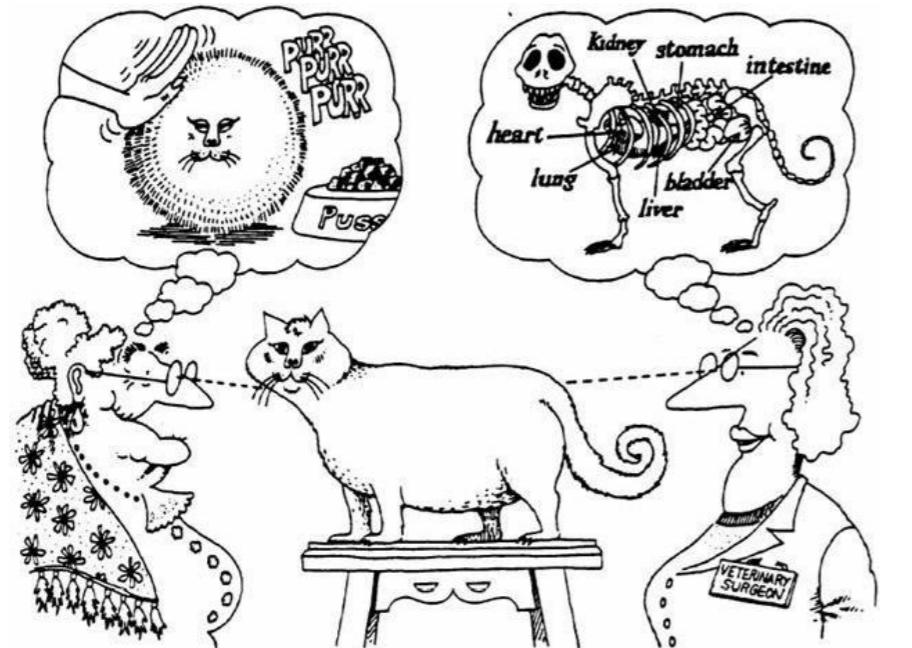


Martin Skarzynski

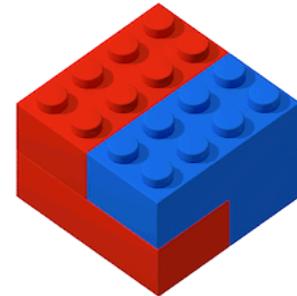
Co-Chair, Foundation for Advanced
Education in the Sciences (FAES)

Principles

- DRY (Don't repeat yourself)
- Modularity
- Abstraction



Booch, G. et al. *Object-Oriented Analysis and Design with Applications*. Addison-Wesley, 2007, p. 45.

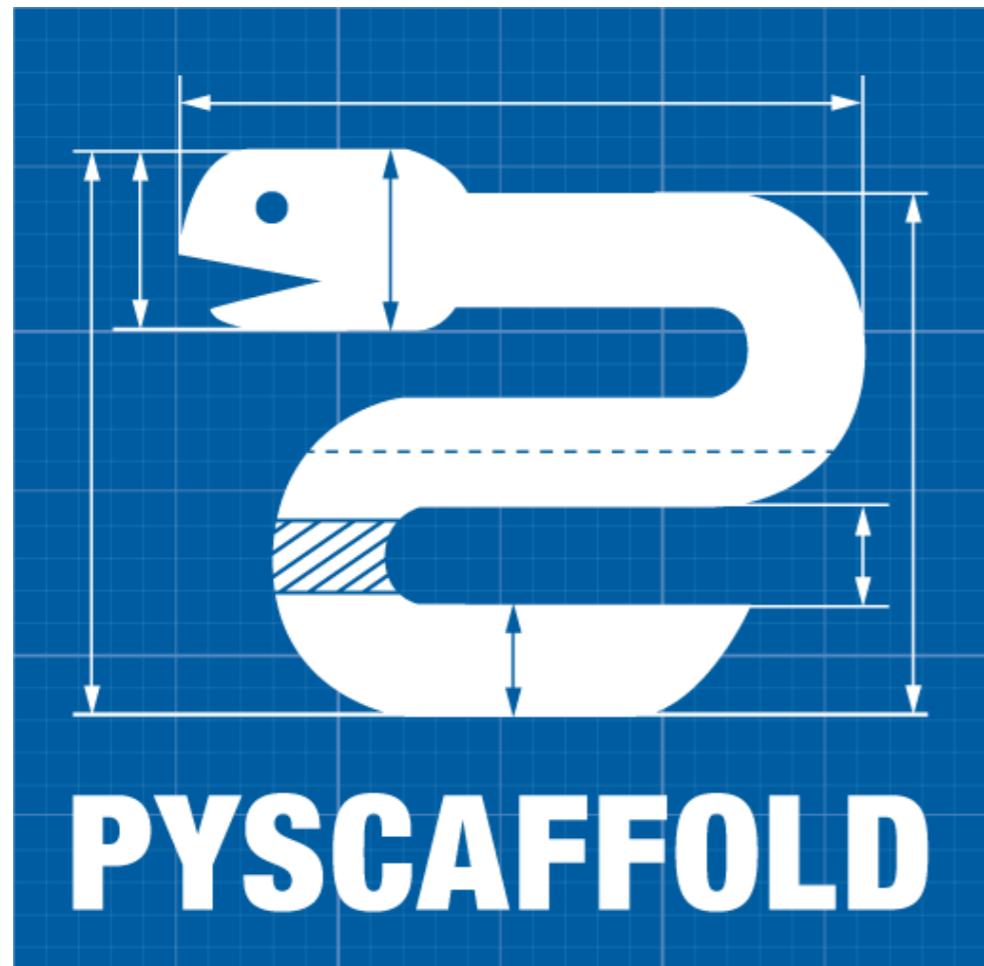


Documentation



- Includes:
 - Docstrings `"""`
 - Type hints `x: int`

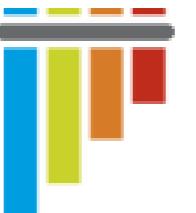
Project templates



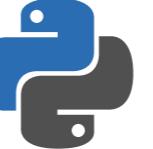
- Includes:
 - Docstrings `"""`
 - Type hints `x: int`

Tests

- `pytest` testing framework
 - `pytest.ini` configuration file
 - `doctest` : run docstring examples
 - `mypy` : check types



pytest

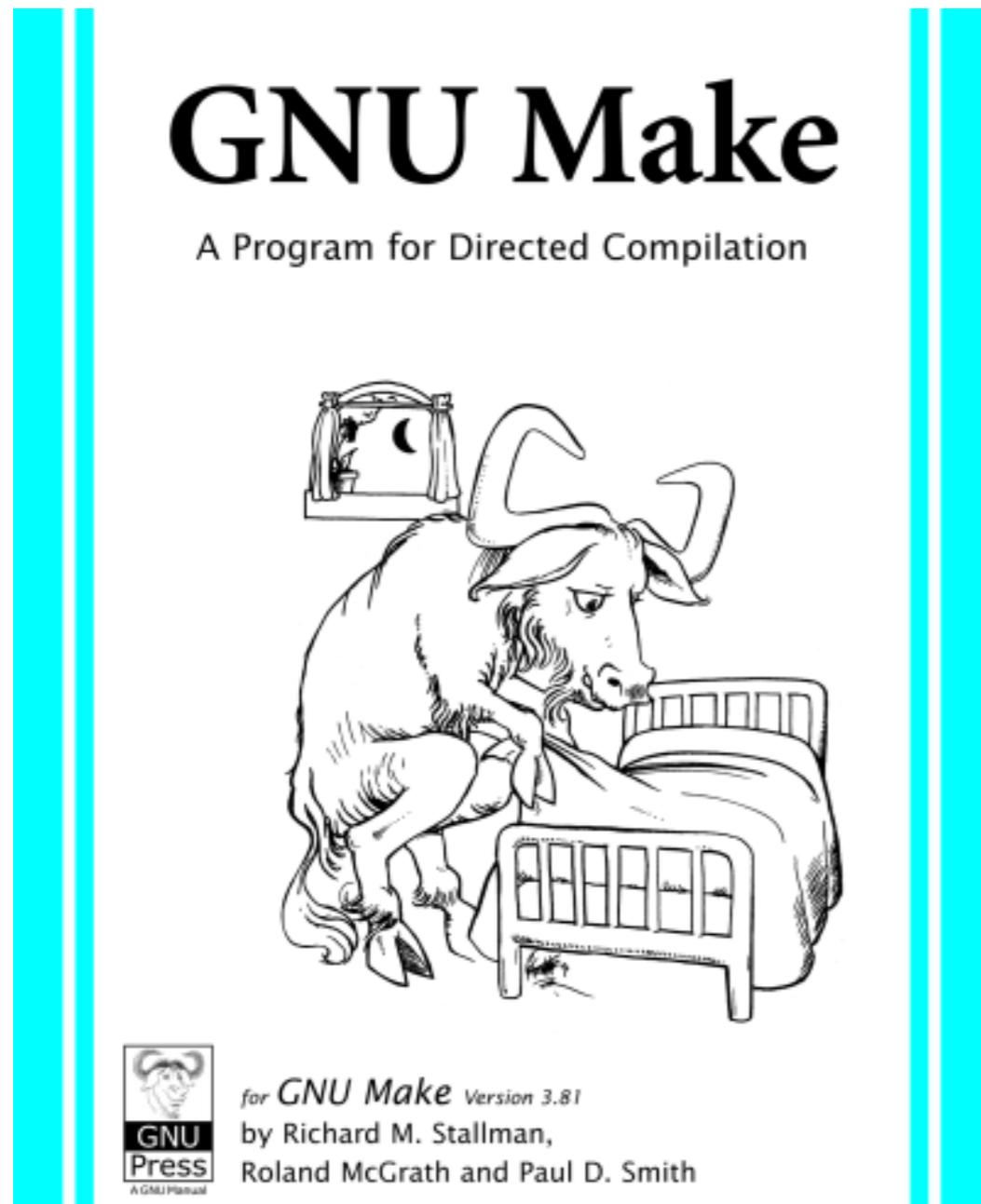
 : my[py]

Jupyter notebooks

- Create and edit notebooks
 - `nbformat`
- Convert notebooks to other formats
 - `nbconvert`
- Execute notebooks with parameters
 - `papermill`
- Access notebook data
 - `scrapbook`
- Check out `rmarkdown` !



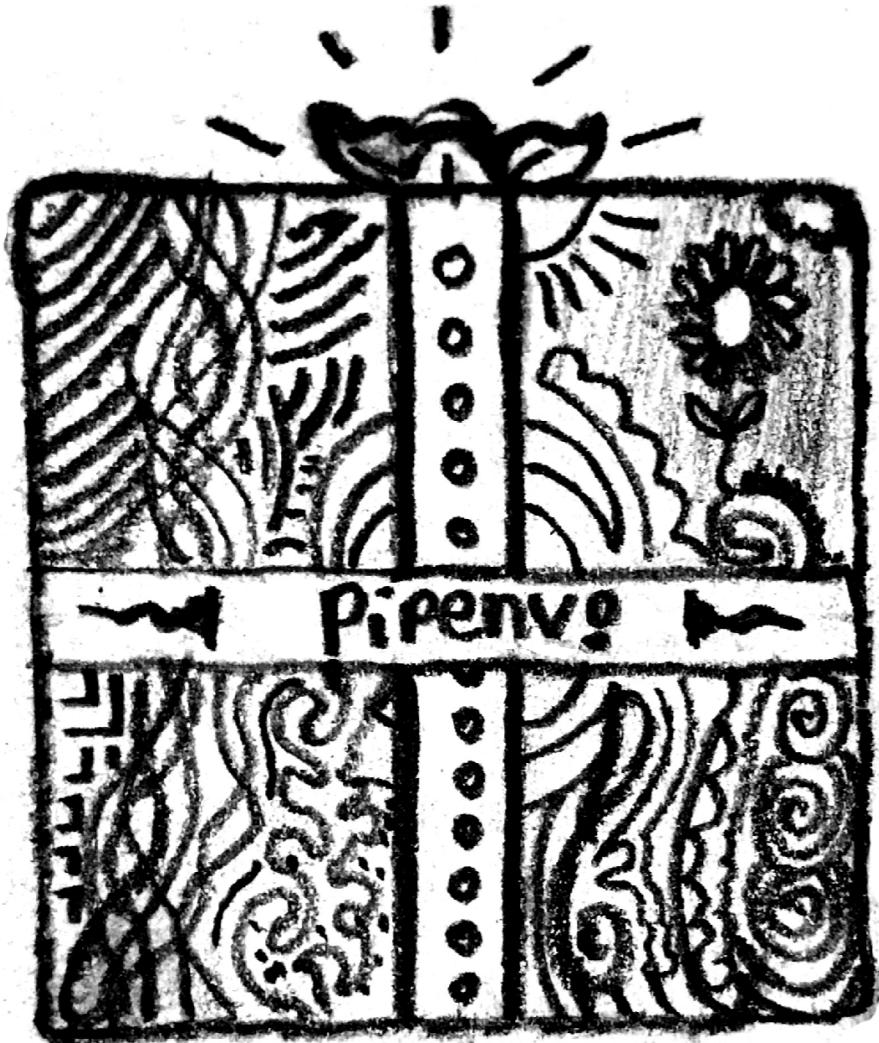
Pipelines



Snakemake



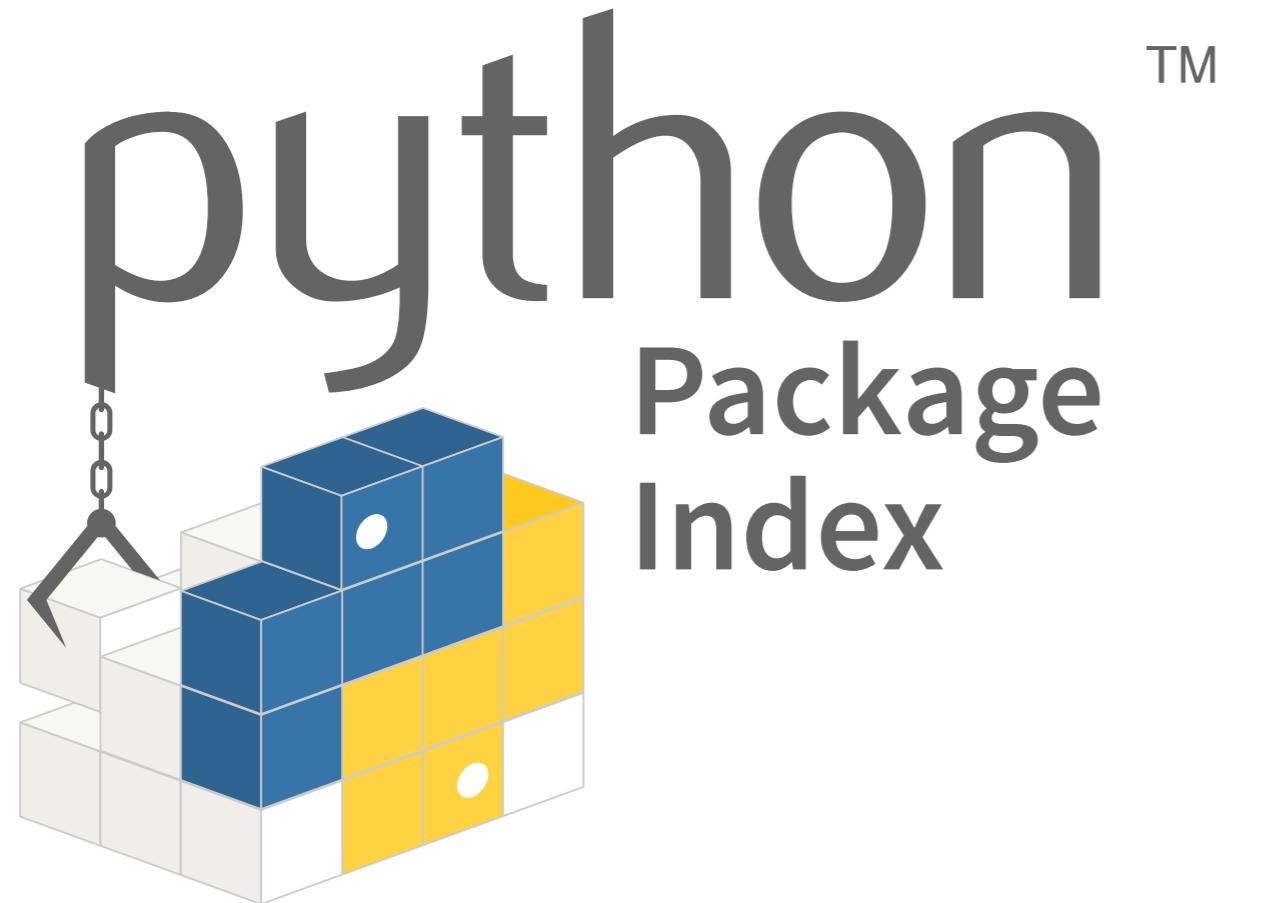
Virtual environments



- Create virtual Python environments
 - `venv` , `virtualenv` , or `pipenv`
- Install Python packages
 - `pip` or `pipenv`
- Not limited to Python

CONDA

Packaging



- Package Python code
 - `setuptools`
- Deploy packages to PyPI
 - `twine`



Poetry

Keep learning!

CREATING ROBUST PYTHON WORKFLOWS