



Organising, documenting and distributing code

ASPP 2023, Heraklion

= *How to make your
code (more) usable*



Why bother?





Why bother?



Contents

usability features:

1) separate, individually usable projects

2) clean folder and file structure

3) error-free importing of code

4) readability



Contents

usability features:

1) separate, individually usable projects



2) clean folder and file structure

3) error-free importing of code

4) readability



Organise what?



packages

documentation

Project 1

notebooks

code

Organise what?



pandas = 1.2.4

packages

Readme.md

pandas = 1.3.0 numpy = 1.22.0

documentation

Paper.pdf

figure.png

Project 2

numba = 1.0.2

Project 1

simulation.py

evaluation.py

code

constants.py

training.py

tests.py

notebooks

better_plot.ipynb

new_plot.ipynb

Notebook.ipynb

Notebook1.ipynb

Organise what?

Project 1

packages

documentation

code

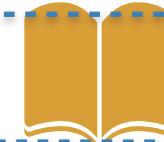
other stuff

packages

documentation

code

other stuff



Organise what?

Project 1

packages

numpy = 1.22.0
pandas = 1.2.4

documentation

Readme.md
figure.png

code

simulation.py
evaluation.py

pip-installable

other stuff

tests/
notebook23.ipynb

Project 2

packages

pandas = 2.0.3
numba = 1.0.2

documentation

Readme.md
figure.png

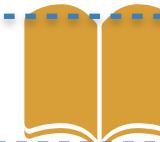
code

constants.py
training.py

pip-installable

other stuff

tests/
notebook23.ipynb



?

defend your code

Virtual Environments

What is a virtual environment?

- A semi-isolated python environment -> you cannot access packages (libraries and their dependencies) installed in other environments.
- packages are installed inside a project-specific virtual environment folder (not added to general python path)
- If you break something, you can delete those folders and start over

Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages

Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages

```
<stdin>:1: FutureWarning: In a future version of pandas all arguments of  
concat except for the argument 'objs' will be keyword-only
```

Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages

```
<stdin>:1: FutureWarning: In a future version of pandas all arguments of  
concat except for the argument 'objs' will be keyword-only
```

—> if you keep updating your python packages, you will run into issues

- code errors
- unexpected results

Previous behavior:

```
In [1]: df.groupby('label1').rolling(1).sum()  
Out[1]:  
          a    b  
label1  
idx1    1.0  2.0
```

New behavior:

```
In [61]: df.groupby('label1').rolling(1).sum()  
Out[61]:  
          a    b  
label1  label1  label2  
idx1   idx1   idx2    1.0  2.0
```

Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages

```
<stdin>:1: FutureWarning: In a future version of pandas all arguments of  
concat except for the argument 'objs' will be keyword-only
```

—> if you keep updating your python packages, you will run into issues

- code errors
- unexpected results

Previous behavior:

```
In [1]: df.groupby('label1').rolling(1).sum()  
Out[1]:  
          a      b  
label1  
          a      b
```

```
DataFrameGroupBy.sum(numeric_only=False, min_count=0, engine=None,  
engine_kwargs=None) #
```

Compute sum of group values.

Parameters: `numeric_only : bool, default False`

Include only float, int, boolean columns.

● *Changed in version 2.0.0:* numeric_only no longer accepts `None`.

Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages

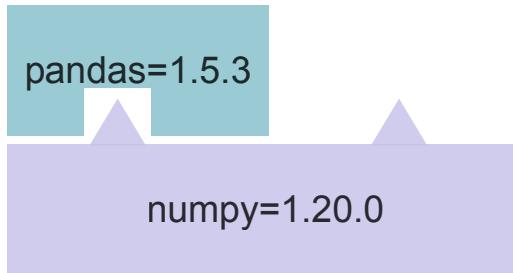
Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages

numpy=1.20.0

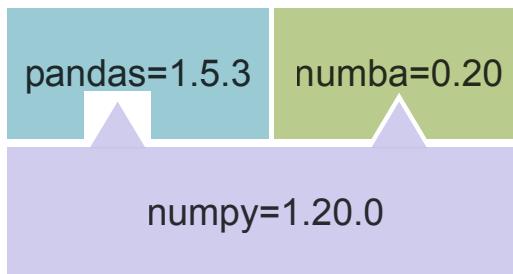
Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages



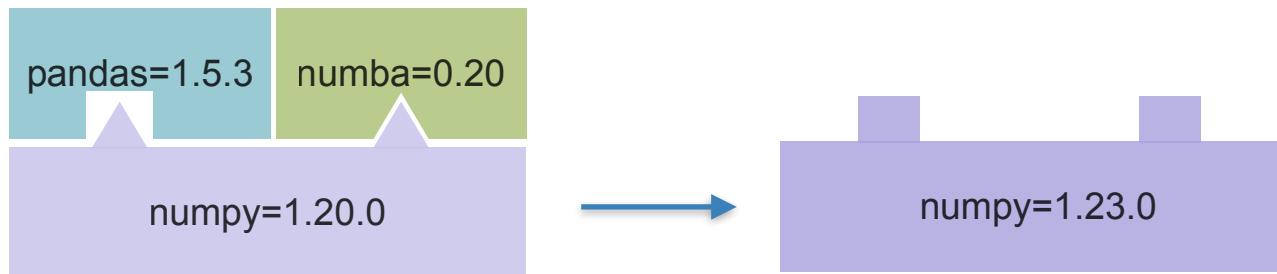
Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages



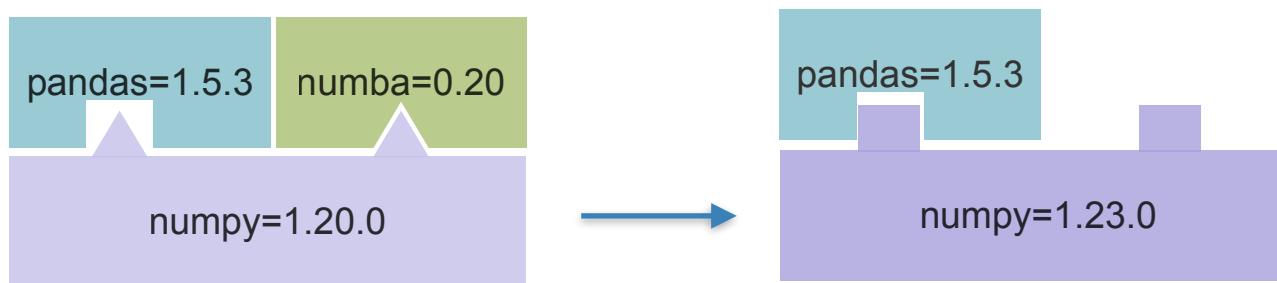
Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages



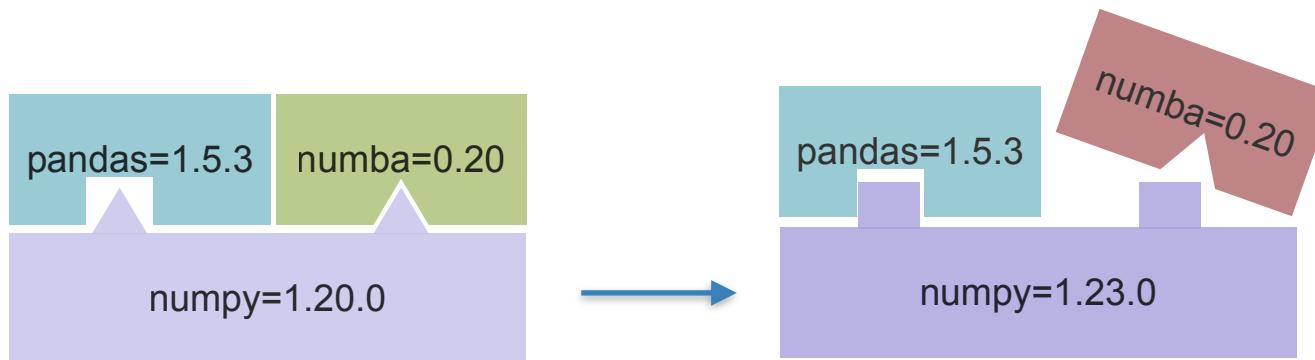
Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages



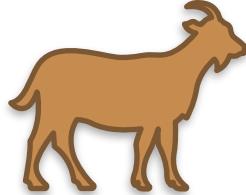
Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages



Why environments?

Project 1
= 1st year
PhD project

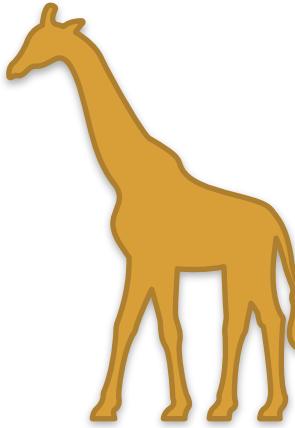
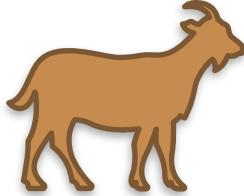


pandas = 1.2.4

dependencies

Why environments?

Project 1
= 1st year
PhD project

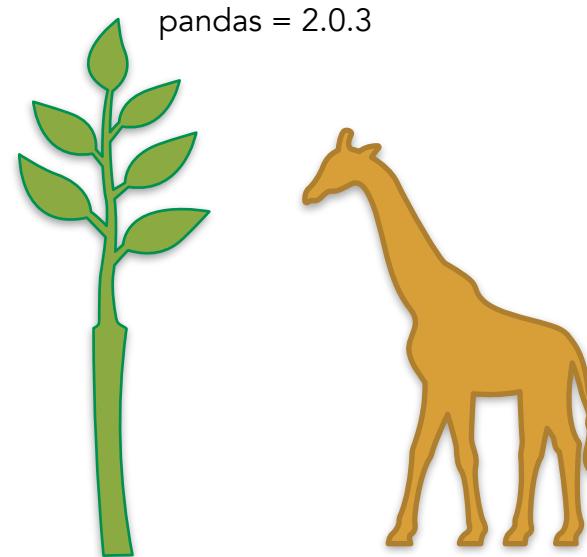
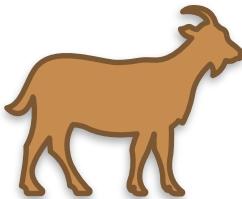


Project 2
= 2nd year
PhD project

pandas = 1.2.4
dependencies

Why environments?

Project 1
= 1st year
PhD project



dependencies

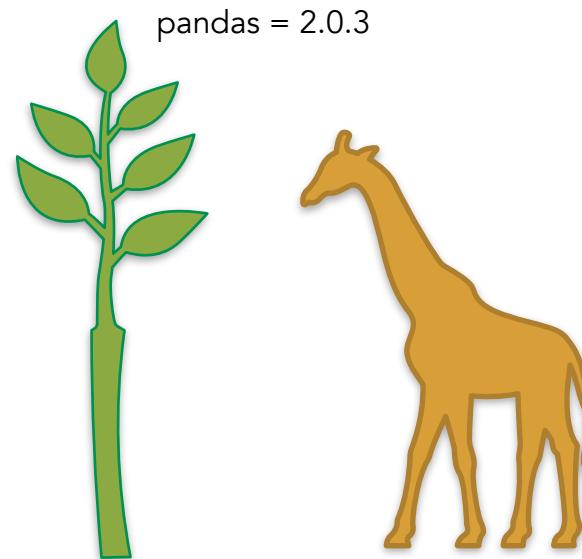
Project 2
= 2nd year
PhD project

Why environments?

Project 1
= 1st year
PhD project



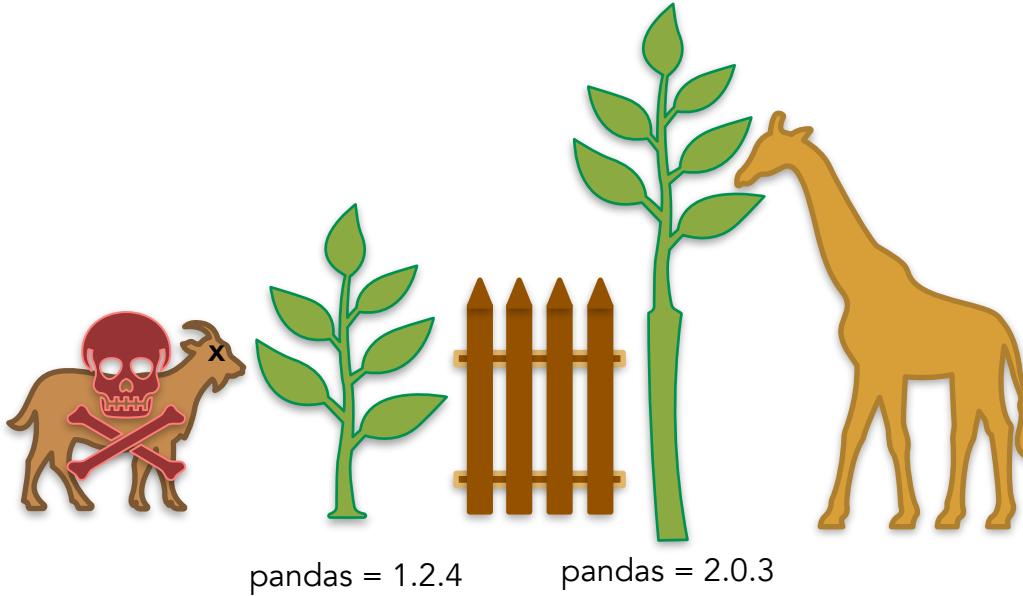
dependencies



Project 2
= 2nd year
PhD project

Why environments?

Project 1
= 1st year
PhD project

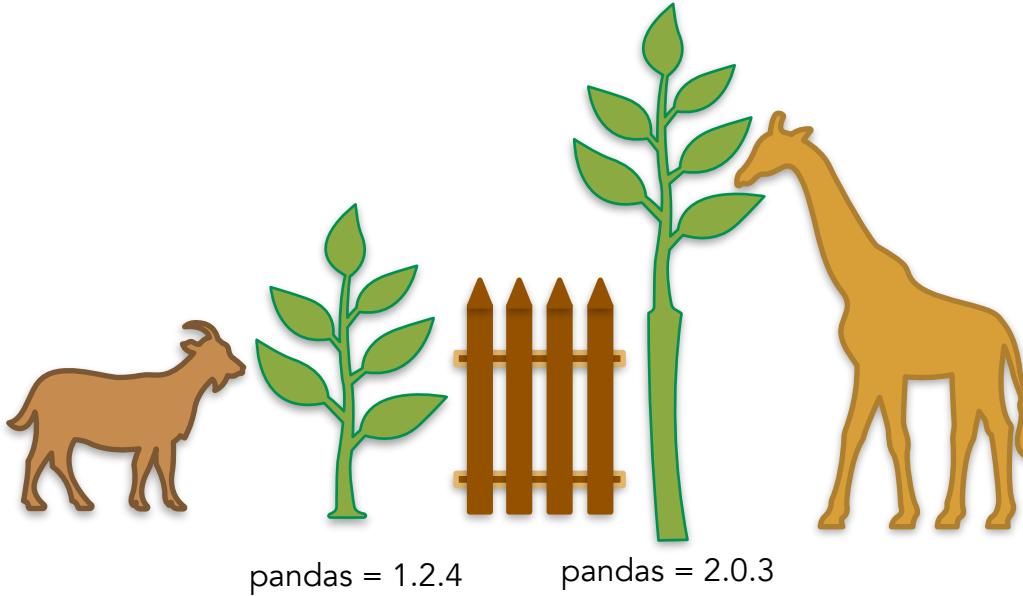


dependencies

Project 2
= 2nd year
PhD project

Why environments?

Project 1
= 1st year
PhD project



dependencies

Project 2
= 2nd year
PhD project

Why environments?

- Avoid importing errors when working on multiple projects / updating your Python packages
- Increased reproducibility: give yourself / other people the exact instructions **and** tools to run your code (cluster, collaboration)



Virtual Environments



- Create and activate a virtual environment following the directions in **Exercise 1 Virtual Environments.md**
- See what changed with regard to the Python interpreter and the installed packages



Additional advantages

- The main advantage is that you can start over if something goes wrong and you have broken nothing!

Environment Managers

- **venv** - current standard recommended by Python
- **poetry** - super useful (if it works o.0).
solves dependency conflicts, creates pyproject.toml for you,
keeps record of all dependency versions in committable file
- **pyenv** - if you need different Python versions in different
projects
- loads of others*: **conda, pipenv, flit, hatch, PDM, ...**

* a description of the chaos:

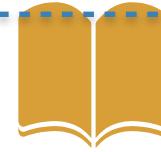
<https://chriswarrick.com/blog/2023/01/15/how-to-improve-python-packaging/> 14

Organise what?

Project 1

packages

packages



Organise what?

Project 1

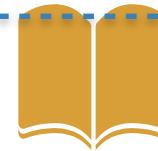
packages

numpy = 1.22.0
pandas = 1.2.4

Project 2

packages

pandas = 2.0.3
numba = 0.21.1



?

Folder structure

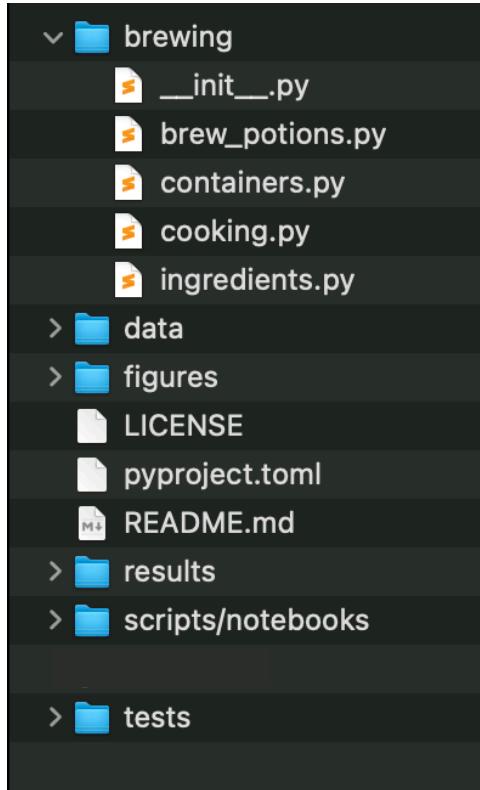
✓ code

- algorithm.py
- calculations.py
- data.npy
- figure (1).png
- figure (2).png
- figure (3).png
- figure.png
- params.npy
- processing.py
- result.h5
- run.ipynb
- run.py
- tests.py
- tools.py

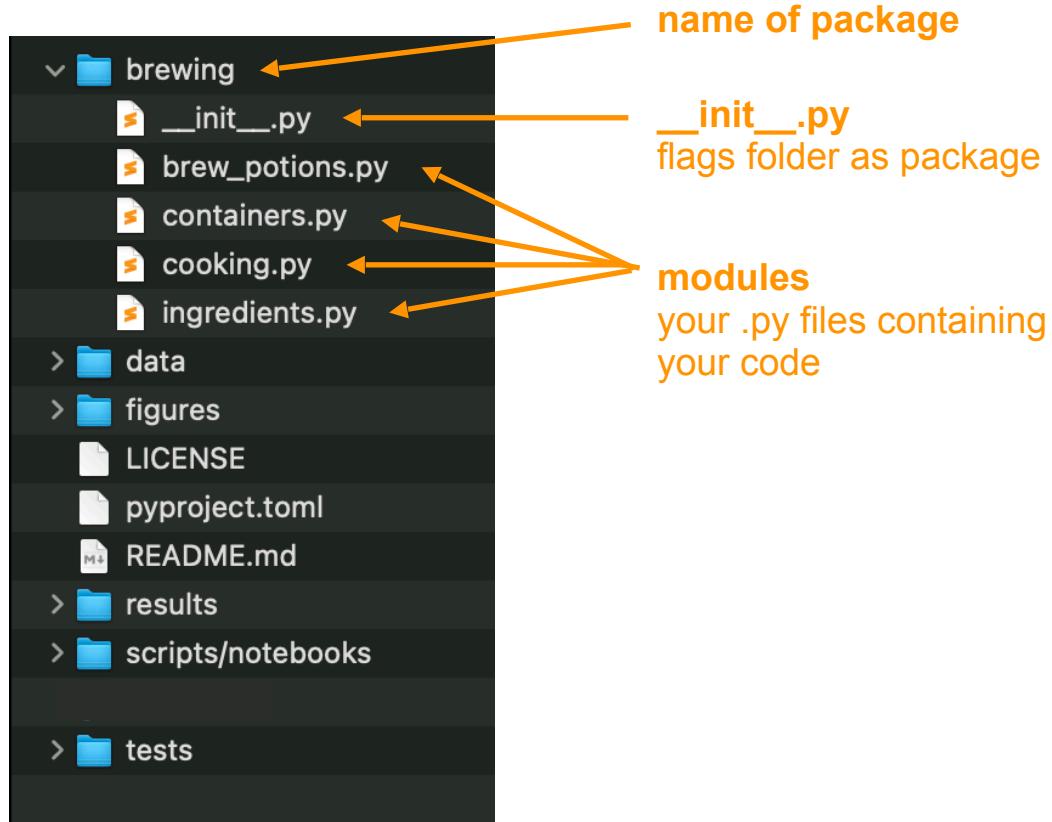
	code	brewing
	algorithm.py	__init__.py
	calculations.py	brew_potions.py
	data.npy	containers.py
	figure (1).png	cooking.py
	figure (2).png	ingredients.py
	figure (3).png	
	figure.png	
	params.npy	
	processing.py	
	result.h5	
	run.ipynb	
	run.py	LICENSE
	tests.py	pyproject.toml
	tools.py	README.md
>		
>		data
>		figures
>		results
>		scripts/notebooks
>		tests

▼	code
algorithm.py	
calculations.py	
data.npy	
figure (1).png	
figure (2).png	
figure (3).png	
figure.png	
params.npy	
processing.py	
result.h5	
run.ipynb	
run.py	
tests.py	
tools.py	
▼	brewing
__init__.py	
brew_potions.py	
containers.py	
cooking.py	
ingredients.py	
▼	data
input_data.npy	
▼	figures
fig_co2_levels.png	
fig_concentration.png	
fig_potion_color.png	
fig_temperature.png	
LICENSE	
pyproject.toml	
README.md	
▼	results
parameters.npy	
potions.h5	
▼	scripts/notebooks
run_brewing.ipynb	
run_brewing.py	
▼	tests
tests.py	

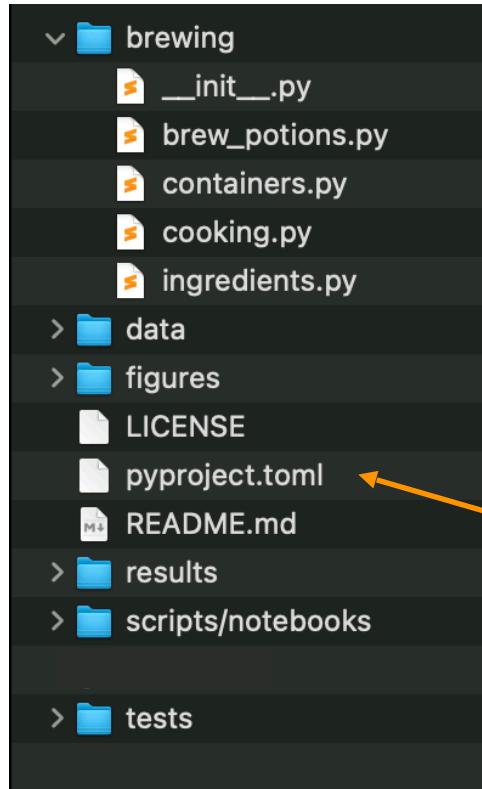
Python package structure



Python package structure



Python package structure



name of package

`__init__.py`
flags folder as package

modules
your .py files containing
your code

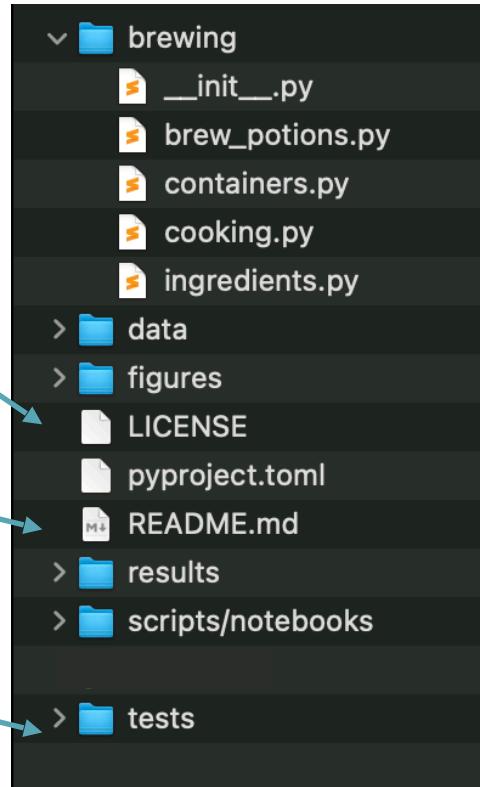
build instructions &
package metadata
will explain more later :)

Python package structure

LICENSE
makes the package
(legally) usable.*

README
contains more information
e.g. instructions on how to
use your package.

tests
you know why :)



name of package

__init__.py
flags folder as package

modules
your .py files containing
your code

**build instructions &
package metadata**
will explain more later :)

Advantage 1

—> know where to find items

e.g. wardrobe

- suit, shirts
- towels
- socks

same concept applies to code

—> use meaningful
file names



Advantage 2

- it makes all of your code **installable***
- which makes all of your code **importable**

```
Terminal  
> pip install brewing  
>  
> python  
>>> import brewing  
>>> brewing.brew_a_potion()
```

* (need a few other changes we will go over)

Advantage 2

- it makes all of your code **installable***
- which makes all of your code **importable**

```
Terminal  
> pip install brewing  
>  
> python  
->>> import brewing  
->>> brewing.brew_a_potion()
```

* (need a few other changes we will go over)

Organise what?

Project 1

packages

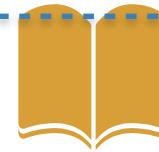
code

other stuff

packages

code

other stuff



Organise what?

Project 1

packages

numpy = 1.22.0
pandas = 1.2.4

code

simulation.py
evaluation.py

other stuff

tests/
notebook23.ipynb

Project 2

packages

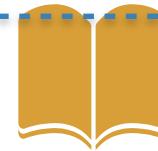
pandas = 2.0.3
numba = 1.0.2

code

constants.py
training.py

other stuff

tests/
notebook23.ipynb



?

Importing

Brewing package

- content of brewing package
 - walk through code
 - run brew_potions.py
 - point out files for exercise

brewing
package

Importing code

- you can always import code from your ***current directory***
 - by calling `import brew_potions`, Python will look for
 - a module called `brew_potions.py` inside the ***current directory***
 - a package called `brew_potions` inside in the ***current directory***
 (= folder called brewing with an `__init__.py` file)
- Importing a module will execute all the code in the module (including imports, print statements)

names & mains

any code running under `if __name__ == "__main__":`

- will be ignored when importing
- will be executed when the module is run as a script

```
if name == "__main__":
    i_will_not_be_imported = True
    print("Does not print when importing")
    print("Prints when run as script")
```

Importing modules

- you can always import code from other modules (.py files) in your ***current directory***
- Options for e.g. importing `eternal_flame`

Importing modules

- you can always import code from other modules (.py files) in your ***current directory***
- Options for e.g. importing `eternal_flame`

1. `import cooking`
2. `import cooking as cook`
3. `from cooking import eternal_flame`
4. `from cooking import *`

Importing modules

- you can always import code from other modules (.py files) in your ***current directory***
- Options for e.g. importing `eternal_flame`

1. <code>import cooking</code>	+ <code>cooking.eternal_flame</code>
2. <code>import cooking as cook</code>	+ <code>cook.eternal_flame</code>
3. <code>from cooking import eternal_flame</code>	+ <code>eternal_flame</code>
4. <code>from cooking import *</code>	+ <code>eternal_flame</code>

Importing modules

- you can always import code from other modules (.py files) in your ***current directory***
- Options for e.g. importing `eternal_flame`

1. <code>import cooking</code>	+ <code>cooking.eternal_flame</code> + <code>cooking.fire</code>
2. <code>import cooking as cook</code>	+ <code>cook.eternal_flame</code> + <code>cook.fire</code>
3. <code>from cooking import eternal_flame</code>	+ <code>eternal_flame</code> x <code>fire</code>
4. <code>from cooking import *</code>	+ <code>eternal_flame</code> + <code>fire</code>

Importing a package

- you can always import a package located in your current directory
- Modules in the package are bound to the package name

1. `import package`
2. `import package.module`
3. `from package.module import object`

Importing a package

- you can always import a package located in your current directory
- Modules in the package are bound to the package name

- *
 1. `import package`
 2. `import package.module`
 3. `from package.module import object`

Importing a package

- you can always import a package located in your current directory
- Modules in the package are bound to the package name

*	1. import package	-
	2. import package.module	+ package.module.object
	3. from package.module import object	+ object

Importing



- Follow the instructions in
Exercise 2 Importing.md

(There is no need to submit a pull request for this exercise)

Importing



- Thought(?) exercise:
Exercise 1 Importing.md

Is there a way to get

- a) any 2
- b) all 3

exercises to work simultaneously?

Importing

- Thought(?) exercise:
Exercise 1 Importing.md

Is there a way to get

- a) any 2
- b) all 3

exercises to work simultaneously?



?

editable installation

Knowledge needed

- **what packages are available?**
- what does an editable pip installation do?
- what are the requirements for this?

Available packages

- **core packages** e.g. time, math, os, ...
(come with Python, no installation needed)
- **installed packages** e.g. numpy, scipy, ...
(packages are downloaded to a system location
e.g. /usr/lib64/python3.11/site-packages/
which is on the Pythonpath => Python can find it)
- **current directory**
- All packages which fall under these categories can be imported

Available packages

- **core packages** e.g. time, math, os, ...
(come with Python, no installation needed)
- **installed packages** e.g. numpy, scipy, ...
(packages are downloaded to a system location
e.g. /Library/Frameworks/Python.framework/Versions/3.9/lib/python3.9/site-packages
which is on the Pythonpath => Python can find it)
- **current directory**
- All packages which fall under these categories can be imported

Installing other packages

- Options to install a package using **pip**

Option 1: if package is included in PyPI

```
pip install numpy
```

Option 2: install from a VCS like git

```
pip install git+https://github.com/<user>/<package-name>.git
```

Installing other packages

- You can install Python packages in your terminal using a package manager

pip

standard package manager for
Python

can install packages from PyPI
(Python Package Index) or from VCS
e.g. github

conda

open source package manager/
environment manager

can install packages which were
reviewed by Anaconda (not all)

Installing other packages

- You can install Python packages in your terminal using a package manager

pip

standard package manager for Python

can install packages from PyPI (Python Package Index) or from VCS
e.g. github



conda

open source package manager/environment manager

can install packages which were reviewed by Anaconda (not all)



Knowledge needed

- what packages are available?
- **what does an editable pip installation do?**
- what are the requirements for this?

Knowledge needed

- what packages are available?
- **what does an editable pip installation do?**
- what are the requirements for this?

Pip editable install

You can import the package you are currently working on as if it were a package you downloaded.

—> This lets you use your own code as any other package you installed

Advantages:

1. you can **import** the objects in the package **from any directory**
(no longer bound to the directory which contains the package)
2. at the same time you can keep your project in your current directory
3. you use your code as someone else would use it, which forces you to write it in a more usable way

Importing own project

- Options to install a package using **pip**

Option 1: if package is included in PyPI

```
pip install numpy
```

Option 2: install from a VCS like git

```
pip install git+https://github.com/<user>/<package-name>.git
```

Option 3: install your package with -e (--editable) option

```
pip install -e <path-to-package>
(cd <path-to-package>; conda develop .)
```

Knowledge needed

- what packages are available?
- what does an editable pip installation do?
- **what are the requirements for this?**

Knowledge needed

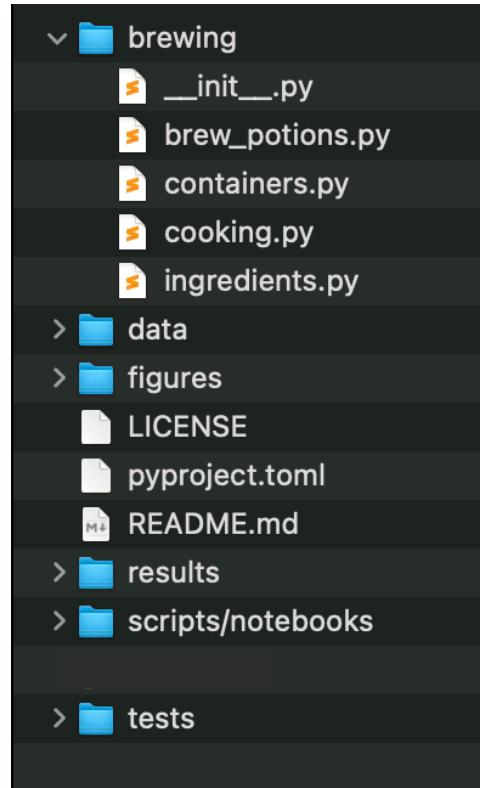
- what packages are available?
- what does an editable pip installation do?
- **what are the requirements for this?**

Python package structure

LICENSE
makes the package
(legally) usable.*

README
contains more information
e.g. instructions on how to
use your package.

tests
you know why :)



name of package

__init__.py
flags folder as package

modules
your .py files containing
your code

**build instructions &
package metadata**
the time has come to
explain this...

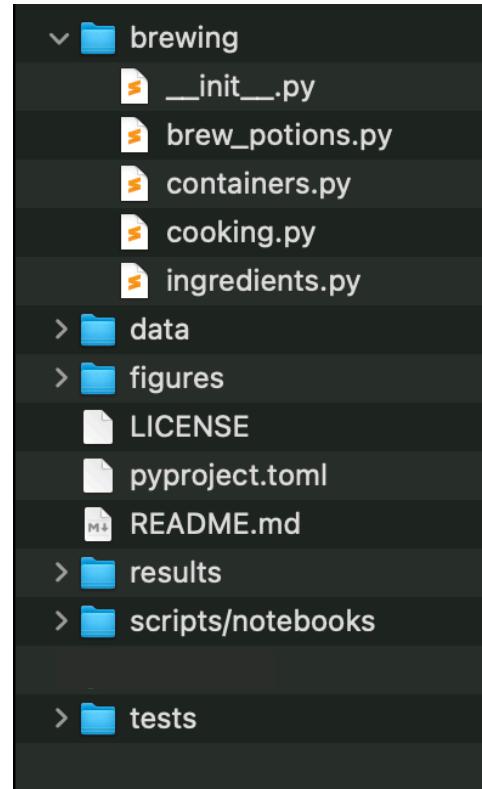
Python package structure

orange files = required in
order to do an editable pip
installation

LICENSE
makes the package
(legally) usable.*

README
contains more information
e.g. instructions on how to
use your package.

tests
you know why :)



name of package

`__init__.py`
flags folder as package

modules
your .py files containing
your code

build instructions &
package metadata
the time has come to
explain this...

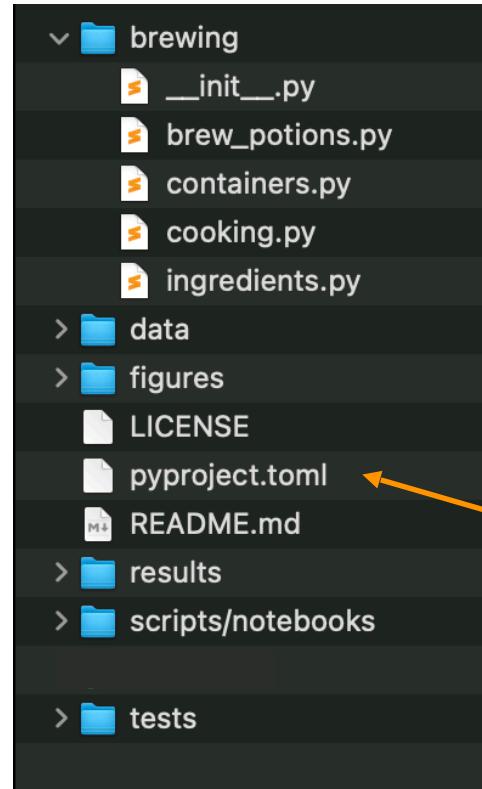
Python package structure

orange files = required in
order to do an editable pip
installation

LICENSE
makes the package
(legally) usable.*

README
contains more information
e.g. instructions on how to
use your package.

tests
you know why :)



name of package

`__init__.py`
flags folder as package

modules
your .py files containing
your code

build instructions &
package metadata
the time has come to
explain this...

pyproject.toml

- The pyproject.toml file holds static information about the package = meta data
- Required entries: name, version, description, authors
- **dependencies** not optional if code relies on other packages to work (go through modules and update regularly, don't just copy '> pip freeze')
-> can also go into separate requirements.txt file

```
[project]
name = "brewing"
version = "0.1.0"
description = "a python package for brewing potions"
authors = [{ name = "H. Granger", email =
"h.granger@hogwarts.ac.uk" }]
license = { file = "LICENSE" }
readme = "README.md"
requires-python = ">=3.7"
dependencies = ["numpy", "matplotlib >= 3.0.0",
"pytest"]
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: BSD License",
    "Operating System :: OS Independent"
]

[tool.setuptools]
packages = ["brewing"]

[build-system]
requires = ["setuptools>=42"]
build-backend = "setuptools.build_meta"
```

pyproject.toml

- The pyproject.toml file holds static information about the package = meta data
- Required entries: name, version, description, authors
- **dependencies** not optional if code relies on other packages to work (go through modules and update regularly, don't just copy '> pip freeze')
-> can also go into separate requirements.txt file

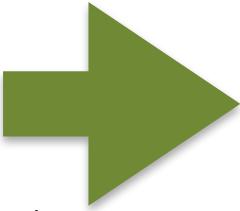
```
[project]
name = "brewing"
version = "0.1.0"
description = "a python package for brewing potions"
authors = [{ name = "H. Granger", email =
"h.granger@hogwarts.ac.uk" }]
license = { file = "LICENSE" }
readme = "README.md"
requires-python = ">=3.7"
dependencies = ["numpy", "matplotlib >= 3.0.0",
"pytest"]
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: BSD License",
    "Operating System :: OS Independent"
]

[tool.setuptools]
packages = ["brewing"]

[build-system]
requires = ["setuptools>=42"]
build-backend = "setuptools.build_meta"
```

pyproject.toml

- dependencies should be kept minimal (only what you actually import in your module files)
- When possible don't depend on a specific version of a package. Conflicting version requirements between packages are annoying to handle as a user.
- When possible don't depend on a specific version of Python. It is usually not necessary.



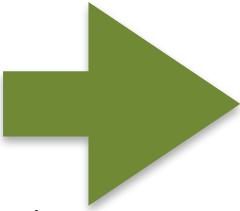
```
[project]
name = "brewing"
version = "0.1.0"
description = "a python package for brewing potions"
authors = [{ name = "H. Granger", email =
"h.granger@hogwarts.ac.uk" }]
license = { file = "LICENSE" }
readme = "README.md"
requires-python = ">=3.7"
dependencies = ["numpy", "matplotlib >= 3.0.0",
"pytest"]
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: BSD License",
    "Operating System :: OS Independent"
]

[tool.setuptools]
packages = ["brewing"]

[build-system]
requires = ["setuptools>=42"]
build-backend = "setuptools.build_meta"
```

pyproject.toml

- dependencies should be kept minimal (only what you actually import in your module files)
- When possible don't depend on a specific version of a package. Conflicting version requirements between packages are annoying to handle as a user.
- When possible don't depend on a specific version of Python. It is usually not necessary.



```
[project]
name = "brewing"
version = "0.1.0"
description = "a python package for brewing potions"
authors = [{ name = "H. Granger", email =
"h.granger@hogwarts.ac.uk" }]
license = { file = "LICENSE" }
readme = "README.md"
requires-python = ">=3.7"
dependencies = ["numpy", "matplotlib >= 3.0.0",
"pytest"]
classifiers = [
    "Programming Language :: Python :: 3",
    "License :: OSI Approved :: BSD License",
    "Operating System :: OS Independent"
]

[tool.setuptools]
packages = ["brewing"]

[build-system]
requires = ["setuptools>=42"]
build-backend = "setuptools.build_meta"
```

Pip editable installation

- `pip install -e <path-to-folder-above-brewing>`

or in the directory above brewing

```
pip install -e •
```

- Follow the instructions in

Exercise 3: Editable installation

(There is no need to submit a pull request for this exercise)



Additional advantages

- If you package your code, in your `pyproject.toml` you will have a record of at least one working combination of the versions of your dependency packages

Additional advantages

- If you package your code, in your `pyproject.toml` you will have a record of at least one working combination of the versions of your dependency packages



Organise what?

Project 1

packages

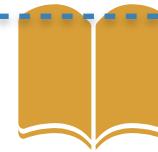
code

other stuff

packages

code

other stuff



Organise what?

Project 1

packages

numpy = 1.22.0
pandas = 1.2.4

code

simulation.py
evaluation.py

pip-installable

other stuff

tests/
notebook23.ipynb

Project 2

packages

pandas = 2.0.3
numba = 1.0.2

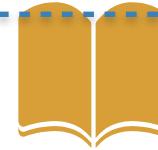
code

constants.py
training.py

pip-installable

other stuff

tests/
notebook23.ipynb



?

how to develop code if it's in a package

Using the editable installation

- You set your imports once and then never worry about them again
- You have not lost any capability, you only gained usability
- If you absolutely must use notebooks, then you can import your code from your modules into your notebook much easier

Workflow (ideal)

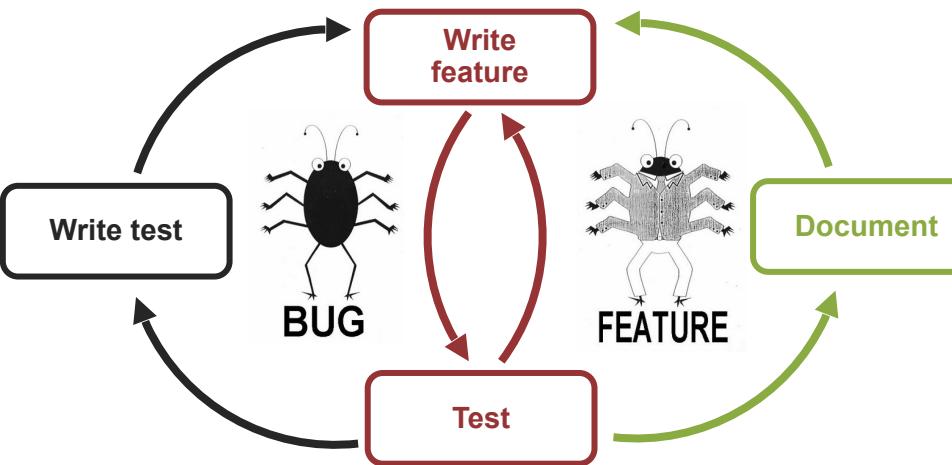
① Create

Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test



③ Publish

In
`pyproject.toml`
update:
`version`
`requirements`
Update `README`

Workflow (realistic?)

① Create

Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test

Write stuff
into ipynb



copy that
stuff to 2nd
ipynb



Write test



change
code ever
so slightly



abandon
test



start new
ipynb



Smugly
write one
comment

③ Publish

In
`pyproject.toml`
update:
`version`
`requirements`
Update `README`

Workflow (realistic?)

① Create

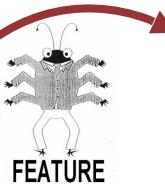
Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test

Write stuff
into ipynb



③ Publish

In
`pyproject.toml`
update:
version
requirements

Update `README`

Workflow (realistic?)

① Create

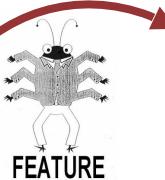
Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test

Write stuff
into ipynb



Move code
to package

③ Publish

In
`pyproject.toml`
update:
version
requirements

Update `README`

Workflow (realistic?)

① Create

Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test

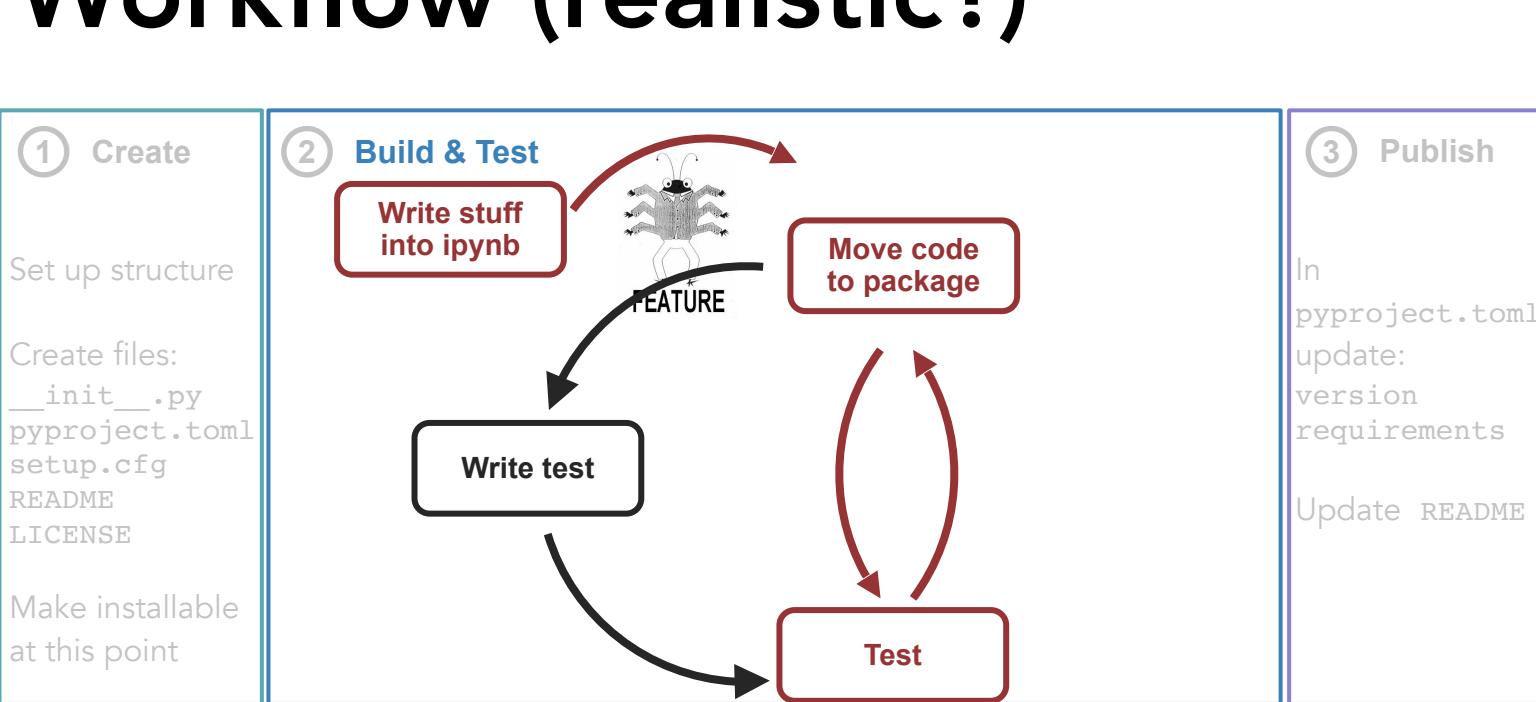
Write stuff
into ipynb



Move code
to package

Write test

Test



Workflow (realistic?)

① Create

Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test

Write stuff
into ipynb



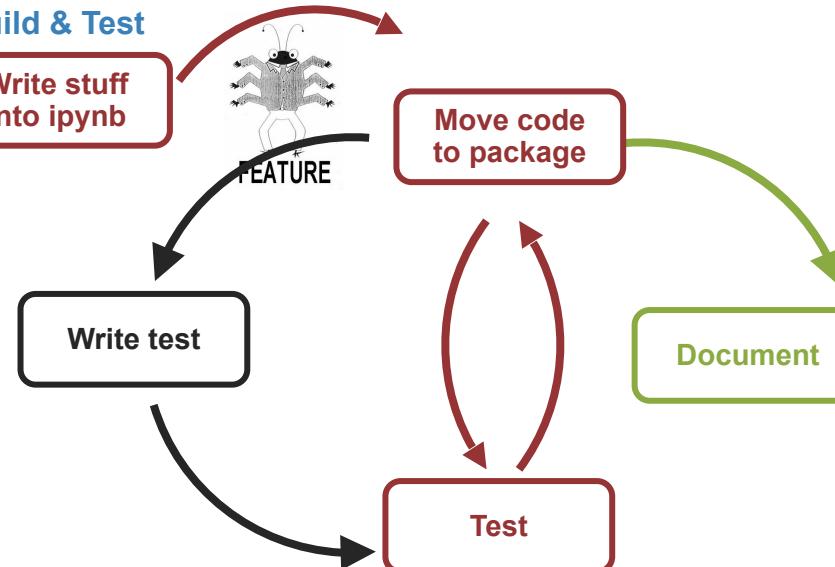
*
FEATURE

Move code
to package

Write test

Document

Test



③ Publish

In
`pyproject.toml`
update:
version
requirements
Update `README`

Workflow (realistic?)

① Create

Set up structure

Create files:
`__init__.py`
`pyproject.toml`
`setup.cfg`
`README`
`LICENSE`

Make installable
at this point

② Build & Test

Write stuff
into ipynb

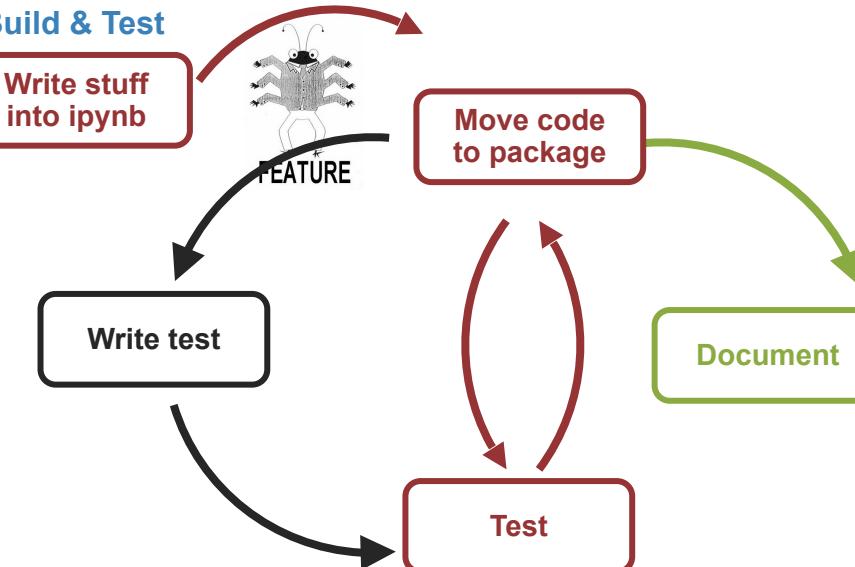


Move code
to package

Write test

Test

Document



③ Publish

In
`pyproject.toml`
update:
`version`
`requirements`
Update `README`

Write your function

- Write the last remaining **potion making function** we need before sharing the package



Exercise:

- Create a branch with a unique name
- Follow the instructions in **Exercise 4 Workflow** to write and test a function to make a “Python expert” potion
- Create a Pull Request

Publishing code

- **Github/Gitlab**
 - perfectly fine for publishing publication code
 - perfectly fine for hosting research group code
- **PyPi: Python Package Index**
 - If you want others to use your library, you must have your code on PyPi to make it easier for others to download and use it

?

readability

Documentation

- Documenting your code provides a way of making your code **usable for future you and others**
 - **Comments (#):** describe what a line (or multiple lines of code do); notes to self
 - **Function/method docstring (""):** purpose of function + params / return
 - **Module docstring (""):** what's in this file

```
""" Module docstring """

def add_points(house_points,
    points=0):
    """ Function docstring """
    # comment
    points += 1000
    return house_points + points
```

NumPy style

- triple double quotes below declaration
- The first line should be a short description
- If more explanation is required, that text should be separated from the first line by a blank line
- Specify Parameters and Returns as
`name : type`
 `description`
(put a line of --- below sections)
- Each line should begin with a capital letter and end with a full stop
- access docs:
`pydoc3 <module>.<object>`

```
""" This module demonstrates docstrings. """

def add_points(house, house_points, points=0):
    """ Adds up points for house cup.

    If the house is Gryffindor, Dumbledore adds
    1000 points no matter what.

    Parameters
    -----
    house_points : int
        Current house cup score.
    points : int, optional
        New points to be added/ subtracted.

    Returns
    -----
    int
    """
    if house == "Gryffindor":
        points += 1000
    return house_points + points
```

NumPy style

- personal suggestion:
if you work with pandas, it is easy to forget the shape of DataFrames.
- Add the format into docstring (and keep up to date!)
OR
Write proper tests, you can always check the DataFrame format there

```
""" This module demonstrates docstrings. """

def some_function(df):
    """ If it helps, you can add a DF example.

    Parameters
    -----
    df : pd.DataFrame
        Historical house cup scores.
                  house      points
        year
        1999    Slytherin       100
        2020    Hufflepuff     2800
    """

    return df
```

Typing

- you can declare the type of the function argument
- the package *mypy* checks whether the types make sense
- Be aware that this might be a pain to maintain if you change your functions often and pass complicated objects...
`tuple[int, dict[str, str]]`

```
""" This module demonstrates docstrings. """

def add_points(house: str,
               house_points: int,
               points: int = 0)
    -> int:
    """ Adds up points for house cup.

    If the house is Gryffindor, Dumbledore adds
    1000 points no matter what.

    Parameters
    -----
    house_points : Current house cup score.
    points : optional; New points to be added
    """
    if house == "Gryffindor":
        points += 1000
    return house_points + points
```

Variable names

- name your variables so that you can later go back and **read** what the code does
(same principle as with module names)

```
x = 10

p = 10

poi = 10

points = 10

points_add = 10

points_to_be_added = 10
```

Variable names

- name your variables so that you can later go back and *read* what the code does
(same principle as with module names)

```
x = 10  -> terrible

p = 10  -> just as terrible

poi = 10 -> still terrible

points = 10 -> better, but potentially unspecific

points_add = 10 -> possibly better, possibly worse than the one before

points_to_be_added = 10 # clear, but maybe a bit long
```

Variable names

Variable names

```
added_points = [10, 5, 1]
# → variable names use underscores

def add_points(house, house_points, points=0):
    if house == "Gryffindor":
        points += 1000
    return house_points + points
# → function names also use underscores

def ScoreKeeper():
    def __init__(self):
        self.house_points = 0
        self._secret_bonus = 5

        def add_points(self, house, points):
            if house == "Gryffindor":
                points += 1000
            return house_points + points
# → Class names use CamelCase
# → private variables (intended for use only within the class) prepend "_"
```

Document your function



- Document the function you just wrote according to the instructions in **Exercise 5 Documentation**.
- Use the same Pull Request



Organise what?

Project 1

packages

documentation

code

other stuff

packages

documentation

code

other stuff



Organise what?

Project 1

packages

numpy = 1.22.0
pandas = 1.2.4

documentation

Readme.md
figure.png

code

simulation.py
evaluation.py

pip-installable

other stuff

tests/
notebook23.ipynb

Project 2

packages

pandas = 2.0.3
numba = 1.0.2

documentation

Readme.md
figure.png

code

constants.py
training.py

pip-installable

other stuff

tests/
notebook23.ipynb



?

Summary

Contents



usability features:

1) separate, individually usable projects

- virtual environments

2) clean folder and file structure

- standard Python package structure

3) error-free importing of code

- editable pip installation

4) readability

- documentation, typing, naming

Contents

usability features:

1) separate, individually usable projects

- virtual environments

2) clean folder and file structure

- standard Python package structure

3) error-free importing of code

- editable pip installation

4) readability

- documentation, typing, naming





Mischief Managed

Any questions?

?

Extra material

Module structure

- constants
- functions
- ...

Keeping track of docstrings

- Most commonly used hosting websites: facilitate building, versioning, and hosting
 - github.io
 - readthedocs.org
- Automate documentation
 - [Sphinx](#): a package to collect docstrings and create a nicely formatted documentation website