Organizing, documenting and distributing code

BCCN Workshop, Summer 2022

Big thanks to Pamela Hathway the original author of the material <3

Motivation

- **Organising** your code in a standardized* way makes it easier to understand and increases **usability** for
  you and **future you** (and **other people**)

* standard in Python, sorry Matlab users!

Contents

- how to organise your code ==> as a package
    - files and folder structure
    - importing and installing your package
- how to make code understandable ==> documentation
- how to handle dependencies ==> virtual environments

(this might not seem extremely relevant now, but if you are to stay in Science, this will help you use code from your old projects and collaborate with other people. And if you end up leaving Academia and work with anything data-related, you will use this daily.)

Discussion

- What makes a good code structure?

? Package structure

Advantage 1

Python package structure

—> know where to find items

e.g. wardrobe

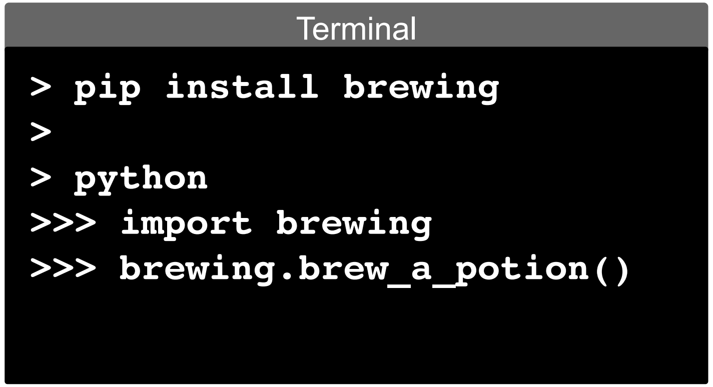
- suit, shirts
- t-shirts
- socks, underwear

same concept applies to code

Advantage 2

Python package structure

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



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

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

Package

- Typically a **package** is a folder (name of folder = name of package) which contains an **`__init__.py`** file and **modules**.

```
brewing
   ┬── __init__.py
   ├── recepies.py
   └── potion_class.py
```

Modules

- A **module** is a .py file consisting of Python code
  e.g. functions and/or classes and/or variables

- its contents can be imported

File: example_module.py

```
""" This is a module. """


some_constant = 3.14


def some_function(x, y):
    return x + y


def ExampleClass():
    def __init__(self):
        self.greeting = “hello”
    def greet(self):
        print(self.greeting)
```

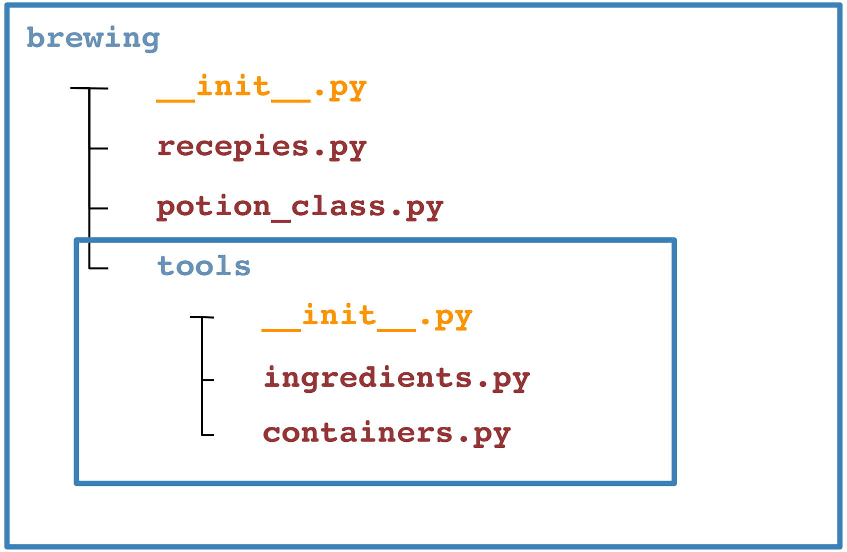
__init__.py

- The **__init__.py** file marks a folder as a package
- can be empty (easiest at the beginning)
- Can be used to control importing

File: __init__.py

Package

- Typically a **package** is a folder (name of folder = name of package) which contains an `__init__.py` file and **modules**.
- A **package** may also contain other **packages**
- The packages you know (numpy, scipy, sklearn, …) follow this structure



```
brewing
  ├─ __init__.py
  ├─ recepies.py
  ├─ potion_class.py
  └─ tools
       ├─ __init__.py
       ├─ ingredients.py
       └─ containers.py
```

?

Importing code from modules

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 <u>all</u> the code in the module (including imports, print statements)

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                           + cooking.eternal_flame
                                            + cooking.fire

2. import cooking as cook                   + cook.eternal_flame
                                            + cook.fire

3. from cooking import eternal_flame        + eternal_flame
                                            X fire

4. from cooking import *                    + eternal_flame
                                            + fire
```

Importing modules

- you can always import code from other modules (.py files) in your ***current directory***
- generally:

```
1. import module-name                  + module-name.object

2. import module-name as abbr          + abbr.object

3. from module-name import object      + object
```

4. ~~from module-name import *~~

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(“But prints when run as script”)
```

?

importing code from a package

Importing a package

- you can always import a package that is located in the directory the script is located in
- Modules in the package are bound to the package name
- If the __init__.py file is empty

```
1. import package                       -

2. import package.module                + package.module.object

3. from package.module import object    + object

4. from package.subpackage.module       + object
   import object
```

Importing

- Follow the instructions in **Exercise 1 Importing.md / .pdf**

?

pip editable installation

Knowledge needed

- what happens when a package is installed?
- what does an editable pip installation do?
- what are the requirements for this?

Available packages

brewing package

- **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

Available packages

brewing package

- **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
```

Knowledge needed

- what happens when a package is installed?
- what does an editable pip installation do?
- what are the requirements for this?

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>
```

Pip editable install

-> installs your project as it would any other package (e.g. numpy) & installs all necessary dependencies (see next section)

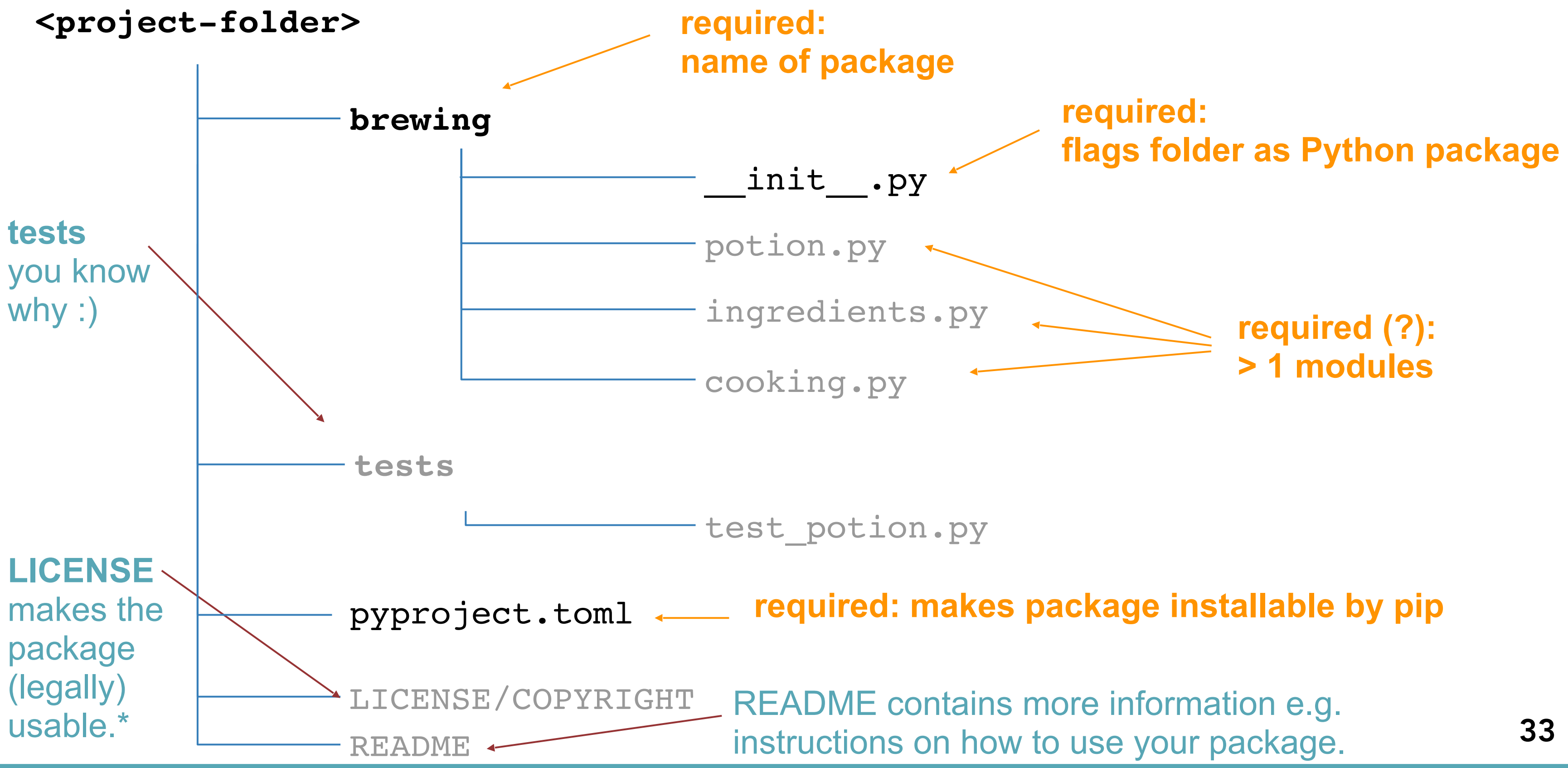
-> pip **adds your directory to the Pythonpath**, so you can import it anywhere on your device (as e.g. numpy). Unlike in the regular install it **doesn’t copy the files**.

-> you use your code as someone else would use it, which forces you to write it in a more usable way

Knowledge needed

- what happens when a package is installed?
- what does an editable pip installation do?
- what are the requirements for this?

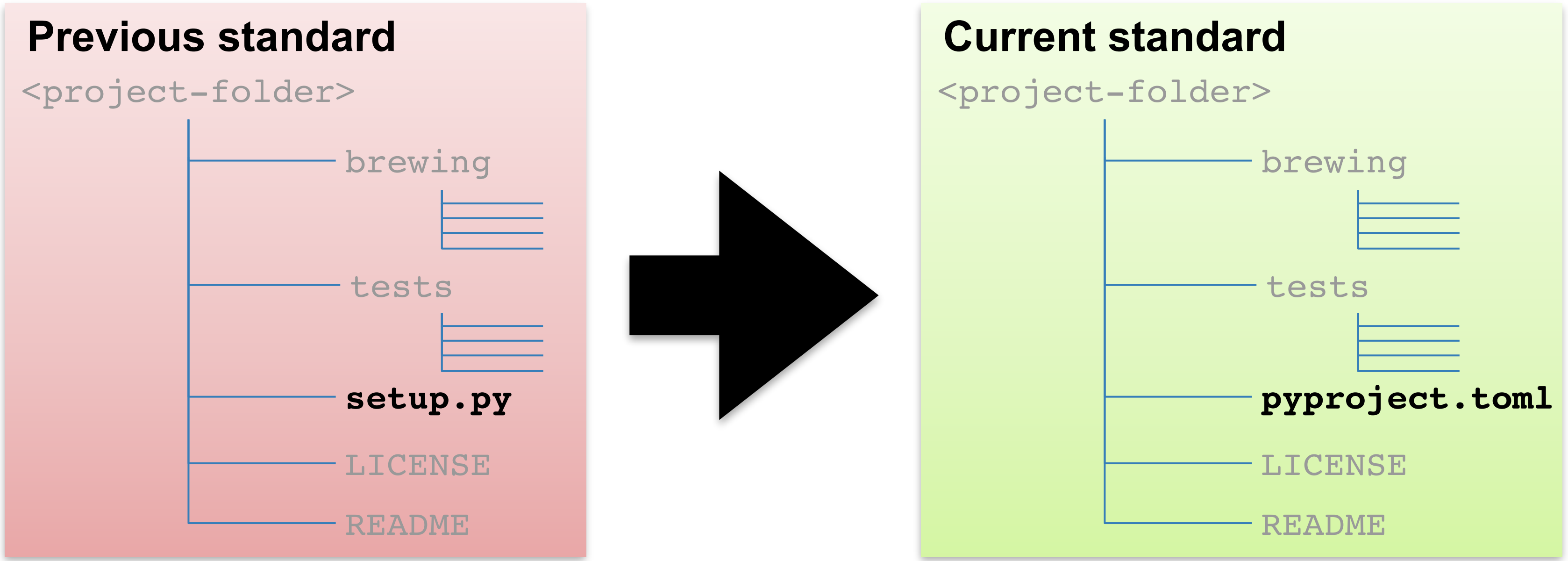
Requirements



<project-folder>
brewing
__init__.py
potion.py
ingredients.py
cooking.py
tests
test_potion.py
pyproject.toml
LICENSE/COPYRIGHT
README
required: name of package
required: flags folder as Python package
required (?): > 1 modules
required: makes package installable by pip
tests you know why :)
LICENSE makes the package (legally) usable.*
README contains more information e.g. instructions on how to use your package.

* pick one from choosealicense.com

Requirements



There have been some recent changes to the recommended way of doing this.

You will still see the old system in a lot of projects!

pyproject.toml

- The pyproject.toml file holds static information about the package its 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

```toml
[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 depend on, not just the full "pip freeze" output)
- 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.

```toml
[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

- "`build-system`" defines which system is used to install the package
- "`setuptools`" is the default, butt there are others, e.g. poetry
- No need to worry about the build system if there are no special requirements- BUT if you do change it, the necessary other keys may change (e.g. `tools.setuptools` key)

```toml
[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"
```

Disclaimer

- The setup with just a `pyproject.toml` is extremely new and isn't supported by all systems with the editable install yet
  - The preliminary fix is to add an empty `setup.cfg` file
- `setup.cfg` was intended to hold all the meta information about the package while `pyproject.toml` holds build information
- In the future hopefully- one unified system

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 2: Editable installation

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

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 analysis/model/… you should try to have it on PyPI to make it easier for others to download and use

?

how to develop code if it’s in a package

Workflow (realistic?)

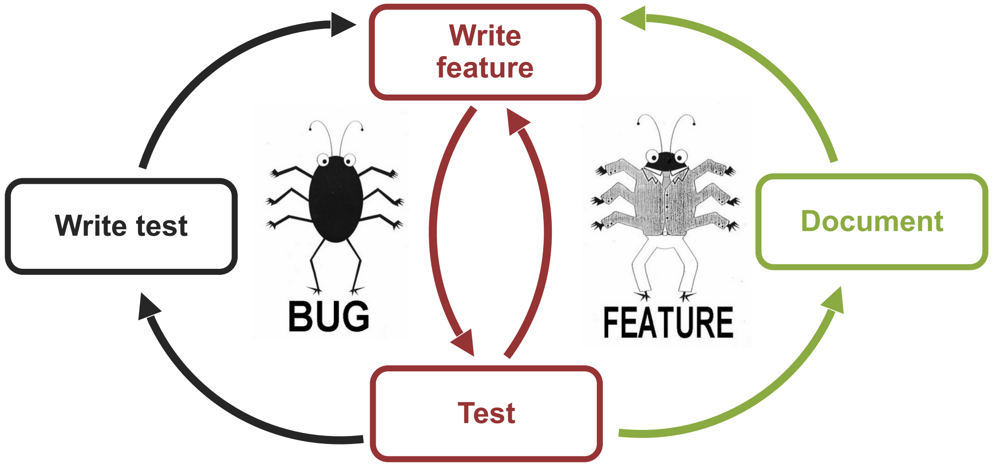
1 Create

Set up folder structure

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

Make installable at this point

2 Build & Test



3 Publish

In `pyproject.toml` update:
`version`
`dependencies`

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 3 Workflow** to write and test a function to make a “Python expert” potion

?

?

documentation

Documentation

- Documenting your code provides a way of making you 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 doctring """

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
- The first line should be in imperative case
  - “add” and not “adds”
- 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

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

def add_points(house, house_points, points=0):
    """ Add 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
```

Typing

- You can declare the type of the function argument
- Types are interpreted like comments- you could still pass something else to the function
- May be useful in combination with IDE type checking features
- 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]]`

```python
""" 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  —> terrible

p = 10  —> just as terrible

poi = 10  —> still terrible

points = 10  —> better, but potentially unspecific

points_add = 10  —> possibly better, possible worse that the one before

points_to_be_added = 10  —> clear, but maybe a bit long
```

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 4 Documentation**.

Keeping track of your 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

virtual environments

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 start over easily (e.g. if you installed a package with specific requirements)

Virtual Environments

- Create and activate a virtual environment following the directions in **Exercise 5 Virtual Environments.md**

- See what changed with regard to the Python interpreter and the installed packages

Summary

Circle back

- **Organising** helps **you** and **future you** (and **other people**)

- following a **package folder structure** makes it easy to find objects
- creating a package will standardise the **import** statements
- doing an **editable install** will enable you to use it as you would do any package (e.g. from any directory) -> as another person would
- **documenting** your code will let anyone read your code more easily
- Using **virtual environments** will isolate your projects from each other and increase your chances of having your code work properly

Mischief Managed

Any questions?

aob

- Advanced Scientific Programming in Python School: https://aspp.school
- Feel free to use the materials + code for you teaching other people what you learned
  If you do, send me an email.
  p.hathway@posteo.de