?

packages and editable installation

Importable code

Code that we can import

- □ **current directory**
- □ **core packages** e.g. time, math, os, …
- □ **installed packages**
  e.g. numpy, scipy, … installed via pip / conda / …
  (saved in system location e.g. /usr/lib64/python3.11/site-packages/
  on Pythonpath => Python can find it)

Pip editable installation

Navigate into the 2024-Heraklion-ODD folder (terminal)

Run **`pip list`**. What do you see?

Run **`pip install -e .`** (full stop = this directory)

Run **`pip list`** again. What has changed?

Run main.py again from `scripts/`, which imports the `make_example_potion` function. Does it work now?

Pip editable install

—> An editable installation 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 and all changes are immediately available (no re-install required)
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**

Editable installation: install your package with -e (--editable) option

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

Other option: if package is included in PyPI

```
pip install numpy
```

Other option: 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)

?

how to develop code with editable install

Changes to your workflow

None*

* for developing code if you are used to working with .py files.
(you won’t be able to use this if you only develop in jupyter)

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

Create a Pull Request

Notes

Changes to your workflow II

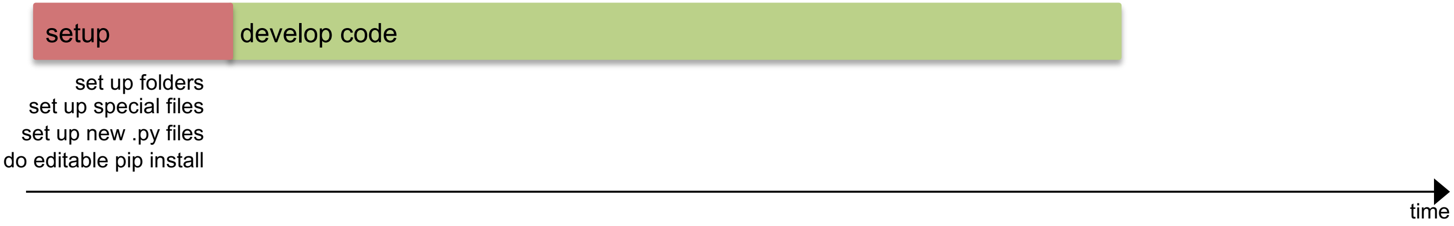
None* **

* for developing code if you are used to working with .py files. (you won’t be able to use this if you only develop in jupyter)

** you will have to do some setup steps at the start and regular updates

Short projects

start with package setup



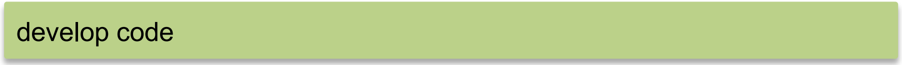
setup

develop code

set up folders
set up special files
set up new .py files
do editable pip install

time

start without package setup

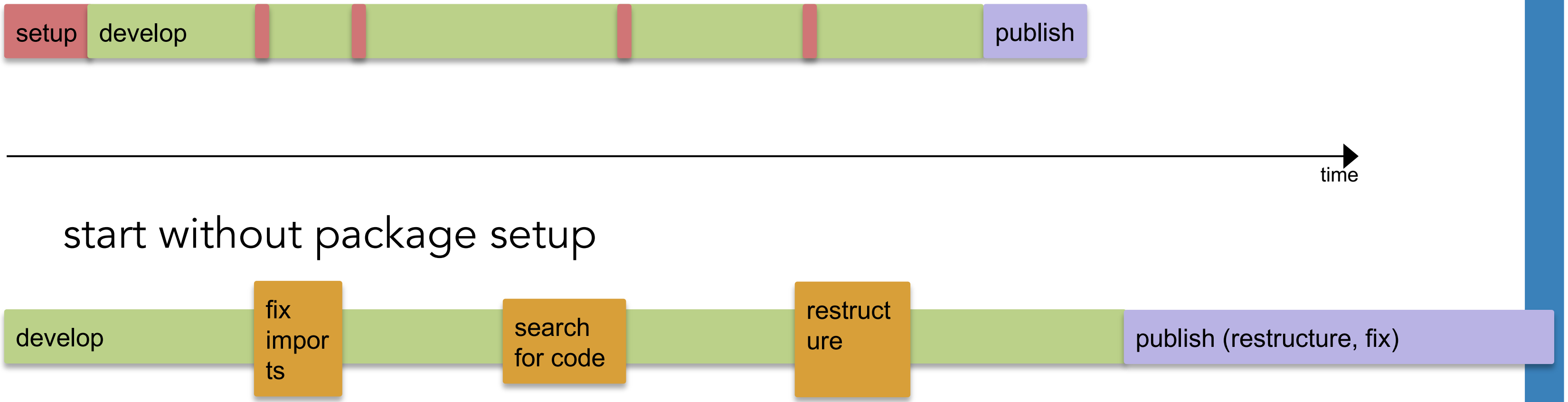


develop code

→ for a small, short project, a package setup might take longer (but it will still be much better to pick back up later)

Longer (=research) projects

start with package setup



start without package setup

→ for a longer project, having a structure from the start will pay off! especially when you want to **share or publish your code**

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 break now?

?

package structure – required files

Package structure

Here is an example of a Python package structure

- □ What do you notice about the files and the structure?
- □ What is familiar / unfamiliar?

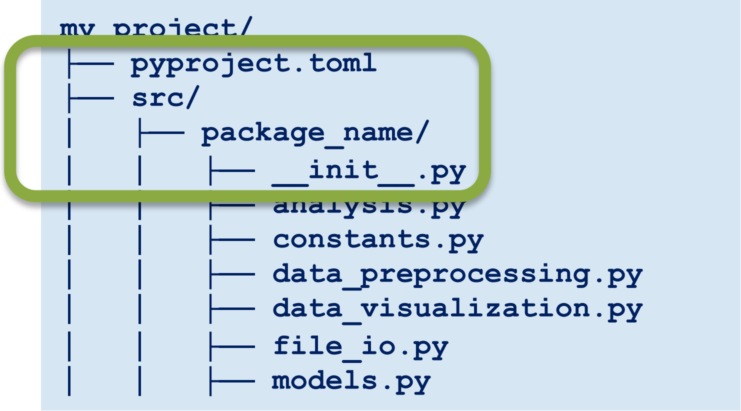
```
my_project/
├── pyproject.toml
├── src/
│   ├── package_name/
│   │   ├── __init__.py
│   │   ├── analysis.py
│   │   ├── constants.py
│   │   ├── data_preprocessing.py
│   │   ├── data_visualization.py
│   │   ├── file_io.py
│   │   ├── models.py
```

Notes

Python package structure

Files that make a package (for your own use)

- □ src / <name-of-package>
  - □ __init__.py
  - □ Modules
- □ pyproject.toml



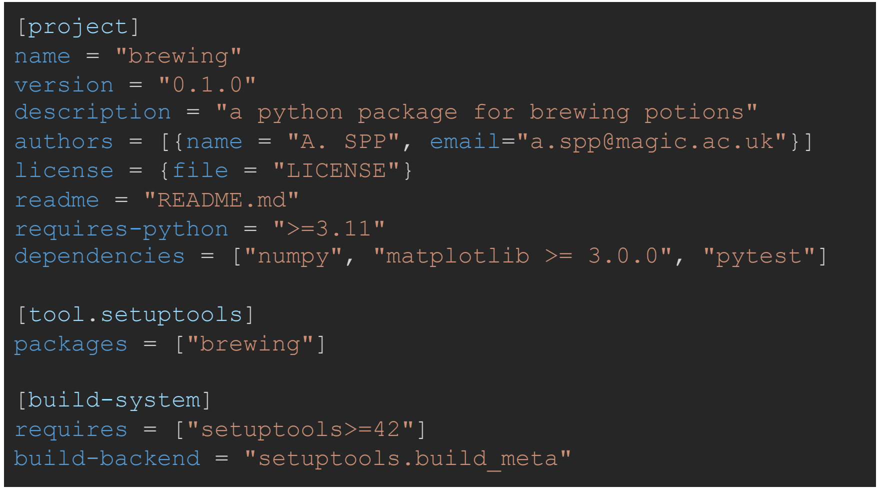
```
my_project/
├── pyproject.toml
├── src/
│   ├── package_name/
│   │   ├── __init__.py
│   │   ├── analysis.py
│   │   ├── constants.py
│   │   ├── data_preprocessing.py
│   │   ├── data_visualization.py
│   │   ├── file_io.py
│   │   ├── models.py
```

* pick one from choosealicense.com

pyproject.toml

The pyproject.toml file holds static information (meta data) about the package

- □ general information
- □ build information
- □ dependencies



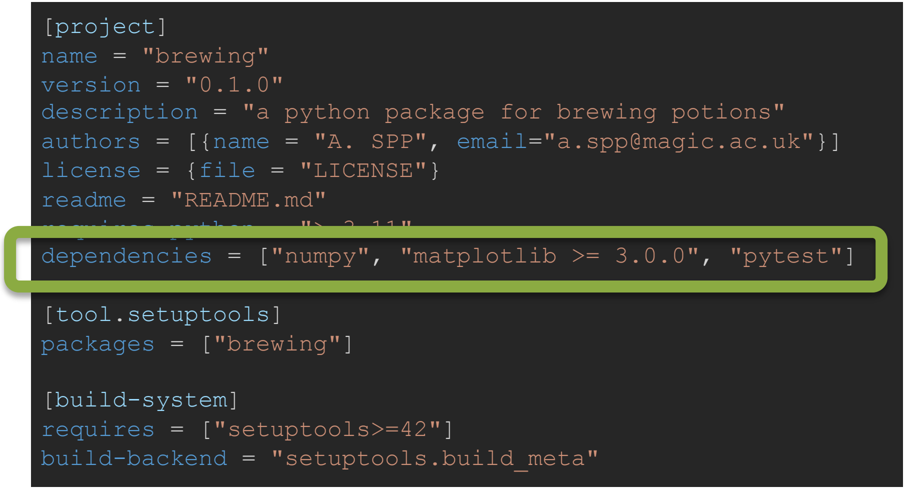
```toml
[project]
name = "brewing"
version = "0.1.0"
description = "a python package for brewing potions"
authors = [{name = "A. SPP", email="a.spp@magic.ac.uk"}]
license = {file = "LICENSE"}
readme = "README.md"
requires-python = ">=3.11"
dependencies = ["numpy", "matplotlib >= 3.0.0", "pytest"]

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

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

pyproject.toml

The pyproject.toml file holds static information (meta data) about the package

Dependencies:

- □ Declare what you import in the code → it will not work in other places otherwise!
- □ don't just copy „`pip list`"!
- □ Whenever you add a new package, add it to the requirements
- □ 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 = "A. SPP", email="a.spp@magic.ac.uk"}]
license = {file = "LICENSE"}
readme = "README.md"
requires-python = ">= 3.11"
dependencies = ["numpy", "matplotlib >= 3.0.0", "pytest"]

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

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

src and __init__.py

src folder holds your code

`__init__.py` designates your folder with .py files are a package for pyproject.toml

contents of __init__.py file →

* pick one from choosealicense.com

Organising file contents

Like in many other areas in life,
code is often organized ***by purpose or thematically***

Separate code into files/folders by

- □ ***purpose / theme*** – data handling, preprocessing, plotting, …
- □ ***type*** – i/o config, parameters, functions

Single responsibility principle

```
my_project/
├── pyproject.toml
├── src/
│    ├── package_name/
│    │    ├── __init__.py
│    │    ├── analysis.py
│    │    ├── constants.py
│    │    ├── data_preprocessing.py
│    │    ├── data_visualization.py
│    │    ├── file_io.py
│    │    ├── models.py
```

Organising file contents

Within-module standard order:

1. imports

- standard Python library
- installed packages
- local modules

2. constants

```
DATA_DIR = "/path/to/data"
```

3. classes and functions

```
class ClassName:
def func_name():
```

4. main execution block

```
if __name__ == "__main__":
```

All the advantages

The setup steps only take time at the start

- □ Set up the project structure, then never worry about it again
- □ Set your imports, then never worry about them again
- □ The more projects you set up like this, the easier it will become. In the end it's faster than solving a single import error.

You unlock so many abilities with only a little effort

- □ sharing code, publishing, endlessly looking for functions or files, avoiding import errors when moving files, …

If you continue coding after inside but especially outside academia, this will be the standard you will encounter.

Our goal

1. Local importing

   → review and best practices

2. Packages and editable installatio

   → avoid importing errors

3. Repo structure

   → organize folders and files in a standardized way

4. Environments

   → avoid and alleviate package installation problems

5. Accessibility

   → make code more readable, understandable and usable

break now?

?

repo structure