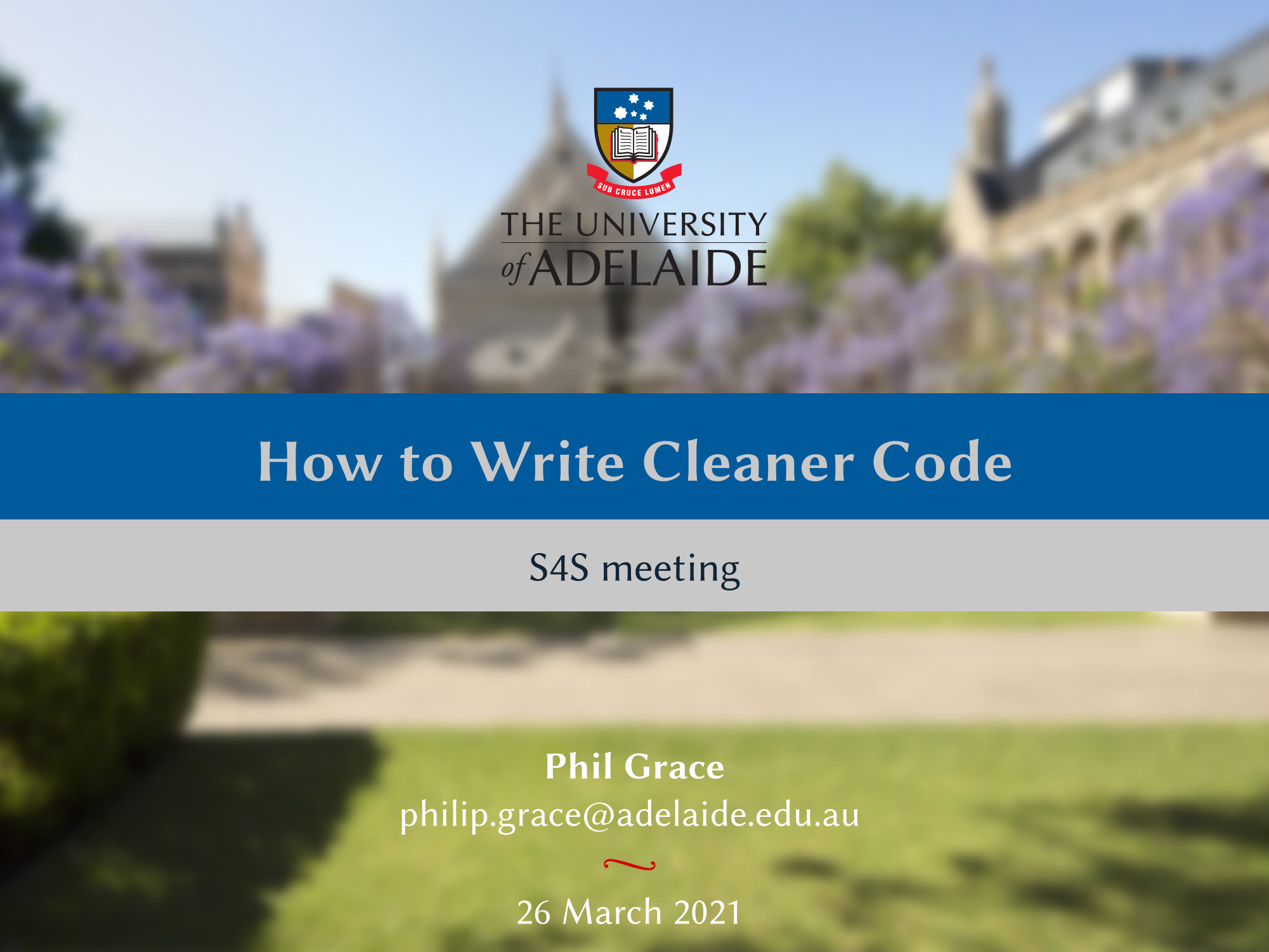


How to Write Cleaner Code

S4S meeting

Phil Grace
philip.grace@adelaide.edu.au

26 March 2021

Why follow software best practices?

- Most physicists never do any formal software engineering study.
- Other people have thought about software design much more than us, and we can benefit from that.
- Please think of the poor student who has to read your code in a few years. This poor student might even be you.

Why follow software best practices?

- Most physicists never do any formal software engineering study.
- Other people have thought about software design much more than us, and we can benefit from that.
- Please think of the poor student who has to read your code in a few years. This poor student might even be you.

An appeal to authority (original author of `keras`)

François Chollet 
@fchollet

Replying to @fchollet

Buggy code is bad science. Poorly tuned benchmarks are bad science. Poorly factored code is bad science (hinders reproducibility, increases chances of a mistake). If your field is all about empirical validation, then your code *is* a large part of your scientific output.

4:56 PM · Jul 15, 2018 · Twitter Web Client

44 Retweets **2** Quote Tweets **174** Likes

A note about this talk

- The latter half of this talk is more geared toward Python. However, some of the advice may transfer to other languages.
- The project structure section of the talk is not intended to be The One Way To Structure Your Code, but is instead meant to provide guidelines.
- This talk isn't "How To Write Good Python Code" (that might be a later talk).

Use git (seriously...)

Why should you use git?

Two bad coding practices among physicists:

- Keeping old versions of your code lying around, with titles like `make-plots.py`, `make-plots-working.py`, `make-plots-working-FINAL.py`, `make-plots-working-OLD.py`
- Commenting out unused code rather than deleting it, resulting in files that are majority comments.

Proper `git` usage solves both of these:

- Records snapshots of your code as it is developed.
- Gives you the peace of mind to make sweeping changes (including deletions) to your code.

If you delete something and realise later you actually need it, it's still there in your `git` history.

Why should you use git?

Two bad coding practices among physicists:

- Keeping old versions of your code lying around, with titles like `make-plots.py`, `make-plots-working.py`, `make-plots-working-FINAL.py`, `make-plots-working-OLD.py`
- Commenting out unused code rather than deleting it, resulting in files that are majority comments.

Proper `git` usage solves both of these:

- Records snapshots of your code as it is developed.
- Gives you the peace of mind to make sweeping changes (including deletions) to your code.

If you delete something and realise later you actually need it, it's still there in your `git` history.

Why don't people use git?

- Steep learning curve.
- Beginners often track the wrong things.
- It's hard to version control a poorly structured project. (More on this later.)

Getting started with git

If you just learn the basic `git` commands without learning a bit about how `git` actually **works**, you're going to get confused very quickly.

Good place to start: chapters 1 & 2 of the Pro Git book (available for free in multiple formats). Should take about an afternoon, and you'll be in a much better position to start using `git`.

General git "best practices"

- Commit early and often.
- Each commit should have a **single purpose**. This means only including files with a logical grouping of changes. (No "it's the end of the day, better commit everything" commits!)
- Write meaningful commit messages. This is easier to do if you follow the second dot point.
- Don't commit broken code: **test your changes** before committing.

Following these best practices will make your git history more understandable and useful for other people and for future you.

General git "best practices"

- Commit early and often.
- Each commit should have a **single purpose**. This means only including files with a logical grouping of changes. (No "it's the end of the day, better commit everything" commits!)
- Write meaningful commit messages. This is easier to do if you follow the second dot point.
- Don't commit broken code: **test your changes** before committing.

Following these best practices will make your git history more understandable and useful for other people and for future you.

Track the right things

A `git` repo should track **a single project**.

What should you track in a repo?

Yes	No
Source code	Build directories & compiled objects
Documentation	Figures
	Data
	Temporary files
	OS files
	Application files
	Dependencies

Rule of thumb: track **project files**; don't track files generated through the running of your code.

Track the right things

A `git` repo should track **a single project**.

What should you track in a repo?

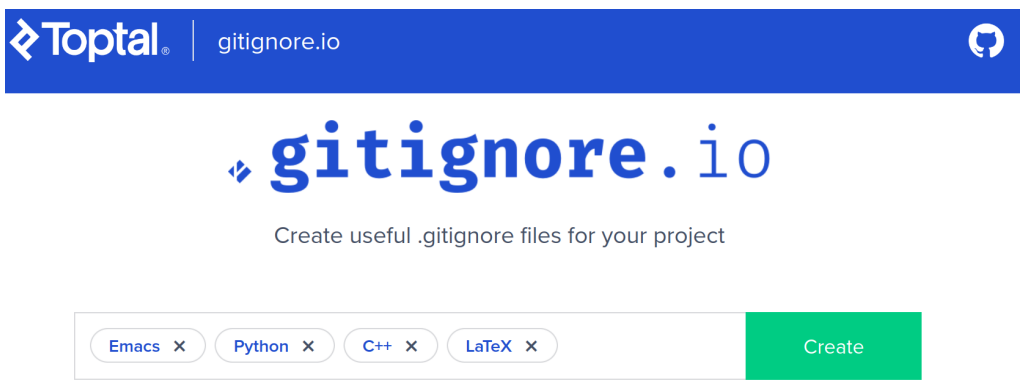
Yes	**No**
Source code	Build directories & compiled objects
Documentation	Figures
	Data
	Temporary files
	OS files
	Application files
	Dependencies

Rule of thumb: track **project files**; don't track files generated through the running of your code.

.gitignore templates

Other people have already thought of all the files you don't want to track.

gitignore.io contains hundreds of templates for common editors, languages, and OS's. Select the ones you need, and paste the results into `.gitignore` in the top level of your project.



From there, you can then add anything else you don't want to track (like a `log` directory or `*.root` files).

Pre-commit hooks

If you are using `git`, then you can benefit from "`git` hooks": quick scripts that are run before letting you commit any changes.

Examples:

- Run a **style checker** over your files (like `autopep8`).
- Run a **static code analyser** (like `flake8`) to check for obvious syntax errors.
- Delete trailing whitespace from code files.

The Python package `pre-commit` (pre-commit.com) makes `git` hooks easy. And you can use `pre-commit` even if your project doesn't contain any Python.

Setting up pre-commit

First define the set of hooks you want in `.pre-commit-config.yaml`:

```yaml
repos:
  - repo: https://github.com/pre-commit/pre-commit-hooks.git
    rev: v3.4.0
    hooks:
      - id: check-yaml
        name: Check YAML files
      - id: trailing-whitespace
        name: Check trailing whitespace
    - id: end-of-file-fixer
      name: Check files end in single new line
  - repo: https://gitlab.com/pycqa/flake8.git
    rev: 3.8.4
    hooks:
      - id: flake8
        name: Run flake8 static code checker
```

Running pre-commit hooks

Install the `pre-commit` package, then install your hooks in your repo. The hooks will run whenever you commit.

```
$ pip install pre-commit
$ pre-commit install
$ git add .; git commit  # or 'pre-commit run' to just run hooks
Check YAML files.....(no files to check)[SKIPPED]
Check trailing whitespace.....[PASSED]
Check files end in single new line.....[PASSED]
Run flake8 static code checker.....[FAILED]
- hook id: flake8
- exit code: 1
scripts/plot.py:10:1: F401 'numpy' imported but unused
```

If a hook fails, you cannot commit until you fix the problem and `git add` your changes. Using this will maintain a **minimum standard of quality** in your code.

Available hooks

Some of my favourite hooks:

- Check all `YAML`, `TOML`, `JSON`, `XML` files are parseable.
- Code style checkers and autoformatters for C++, Python, FORTRAN, and more.
- black: The "uncompromising code formatter" for Python.
- Static code analysers for C++, Python, and more.

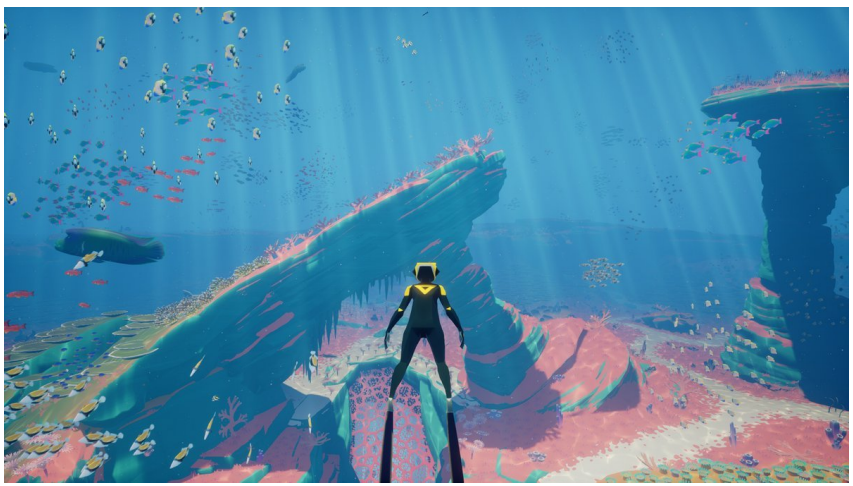
Some of these will fix the warnings for you, others will require you to fix them yourself.

Many more listed on pre-commit.com/hooks.html

Game Corner: ABZÛ

- Length: 2 hours
- Skill required: none
- Genre: Fish & good vibes

Very chill game that just makes you realise how cool ocean wildlife is.



Give your project a structure

Why?

The most common project structure for physicists is “bunch of scripts in a folder.”

You can make your code readable & understandable for others (and yourself) if you maintain a proper structure from the beginning. This is a form of “**self-documentation**.”

There are a few online guides to project structures for data science:

- Structuring Your Project (python-guide.org)
- Cookiecutter Data Science
- How to organise your Python data science project

I’ll go over the important points from these, but remember that these are only guidelines, not mandates.

Why?

The most common project structure for physicists is "bunch of scripts in a folder."

You can make your code readable & understandable for others (and yourself) if you maintain a proper structure from the beginning. This is a form of "**self-documentation**."

There are a few online guides to project structures for data science:

- Structuring Your Project (python-guide.org)
- Cookiecutter Data Science
- How to organise your Python data science project

I'll go over the important points from these, but remember that these are only guidelines, not mandates.

Overview of a basic structure

```
~/project/
├── analysisproject/
│   ├── __init__.py
│   ├── definitions.py
│   └── util.py
├── data/
├── fig/
├── log/
├── .gitignore
├── .pre-commit-config.yaml
├── README.md
├── setup.cfg
└── setup.py
```

In the next few slides, I'll address each part of this.

Top-level files

- `README.md`: The "landing page" for newcomers.
  - Explain why the project exists, how things are organised, and where to go for more information.
  - Don't try to document your entire project interface here; there are other places to do that.
- `setup.py` & `setup.cfg`: The setup script and configuration for when you `pip install` your project. If your project depends on particular Python packages, then you can specify this here and they will be automatically installed.
- `.gitignore` & `.pre-commit-config.yaml`: Already explained these.

Top-level directories

- `data`, `log`, `fig` (should all be ignored by `.gitignore`):
  - `fig` could include timestamped directories (see backup slides).
  - `log` is useful if you run batch jobs.

Cookiecutter Data Science recommends separate subdirectories in `data` for the different steps of data processing: `raw`, `cleaned`, `processed`, etc.

> *Don't ever edit your raw data, especially not manually, and especially not in Excel. Don't overwrite your raw data. Don't save multiple versions of the raw data. Treat the data (and its format) as immutable. The code you write should move the raw data through a pipeline to your final analysis.*

Top-level directories

- `data`, `log`, `fig` (should all be ignored by `.gitignore`):
  - `fig` could include timestamped directories (see backup slides).
  - `log` is useful if you run batch jobs.

Cookiecutter Data Science recommends separate subdirectories in `data` for the different steps of data processing: `raw`, `cleaned`, `processed`, etc.

> *Don't ever edit your raw data, especially not manually, and especially not in Excel. Don't overwrite your raw data. Don't save multiple versions of the raw data. Treat the data (and its format) as immutable. The code you write should move the raw data through a pipeline to your final analysis.*

Top-level directories

- `analysisproject`: A directory for your source code. Whatever name you give this folder will be the package name you import from.
- `scripts`: executable scripts to "do things." The scripts here should primarily import code from `analysisproject`.
- `tests`: A suite of scripts to test that individual parts of your code in `analysisproject` work as expected.

Top-level directories

- `notebooks`: Jupyter notebooks, if that's your thing.

Notebooks are nice, but they tend to encourage bad coding practices: in particular, huge chunks of code copied between cells or notebooks.

Anything that you use multiple times should be refactored into `analysisproject`. This will allow you to use that same code in data processing scripts.

The following notebook trick will auto-reload imports whenever your module is used, so changes in the module are immediately seen in your notebook.

```
%load_ext autoreload
%autoreload 2
from analysisproject import util
```

Top-level directories

- `notebooks`: Jupyter notebooks, if that's your thing.

Notebooks are nice, but they tend to encourage bad coding practices: in particular, huge chunks of code copied between cells or notebooks.

Anything that you use multiple times should be refactored into `analysisproject`. This will allow you to use that same code in data processing scripts.

The following notebook trick will auto-reload imports whenever your module is used, so changes in the module are immediately seen in your notebook.

```
%load_ext autoreload
%autoreload 2
from analysisproject import util
```

`pip install` your own code

With some simple steps, you can `pip install` your project. This will allow you to reuse code by importing it.

1. Write a `setup.py` in the top level of your project, containing the lines:

```
from setuptools import setup
setup(install_requires=["natpy>=0.0.6"])
```

2. Write a `setup.cfg` in the top level of your project, which will give `pip` the project metadata.

```
[metadata]
name = cool-analysis-project
version = 0.0.1
description = My analysis project
author = Firstname Lastname
author_email = firstname.lastname@adelaide.edu.au
```

pip install your own code

3. Create a directory in your project, and create an empty `__init__.py` (so Python knows it is a package).

```
$ mkdir analysisproject
$ touch analysisproject/__init__.py
```

4. In the top level of your directory, run the following to install your project.

```
$ pip install --editable .
Installing collected packages: cool-analysis-project
  Running setup.py develop for cool-analysis-project
Successfully installed cool-analysis-project
$ pip show cool-analysis-project
Name: cool-analysis-project
Version: 0.0.1
Summary: My analysis project
Author: Firstname Lastname
Author-email: firstname.lastname@adelaide.edu.au
```

pip install your own code

Now your code is installed, and you can import it from any other script:

```
>>> import analysisproject
>>> analysisproject
<module 'analysisproject' from '~/project/analysisproject/__init__.py'>
```

If your directory structure is...

```
~/project/
├── analysisproject/
│   ├── __init__.py
│   ├── definitions.py
│   ├── util.py
│   └── study/
│       ├── steering.py
│       └── tasks.py
├── setup.cfg
└── setup.py
```

then subdirectories are automatically submodules:

```
>>> import analysisproject.util
>>> from analysisproject.study
 ↪  import tasks
```

This makes it really easy to keep your code organised.

pip install your own code

Now your code is installed, and you can import it from any other script:

```
>>> import analysisproject
>>> analysisproject
<module 'analysisproject' from '~/project/analysisproject/__init__.py'>
```

If your directory structure is...

```
~/project/
├── analysisproject/
│   ├── __init__.py
│   ├── definitions.py
│   ├── util.py
│   └── study/
│       ├── steering.py
│       └── tasks.py
├── setup.cfg
└── setup.py
```

then subdirectories are automatically submodules:

```
>>> import analysisproject.util
>>> from analysisproject.study
 ↪  import tasks
```

This makes it really easy to keep your code organised.

Game Corner: SOMA

- Length: 10 hours
- Skill required: none
- Genre: Sci-Fi, Psychological Horror

You go in for a brain scan, and wake up in an abandoned underwater facility.

Find out what happened and enjoy soul-destroying epiphanies about your own existence.

Avoid spoilers for this game.

The anti-ABZÛ: have an **awful** time underwater.

- Length: 10 hours
- Skill required: none
- Genre: Sci-Fi, Psychological Horror

You go in for a brain scan, and wake up in an abandoned underwater facility.

Find out what happened and enjoy soul-destroying epiphanies about your own existence.

Avoid spoilers for this game.

The anti-ABZÛ: have an **awful** time underwater.

- Length: 10 hours
- Skill required: none
- Genre: Sci-Fi, Psychological Horror

You go in for a brain scan, and wake up in an abandoned underwater facility.

Find out what happened and enjoy soul-destroying epiphanies about your own existence.

Avoid spoilers for this game.

The anti-ABZÛ: have an **awful** time underwater.

Writing good module code

Organise your code into functions

Most common type of bad physicist code: hundreds of lines of procedural code. Even worse when hundreds of lines are contained in an `if` clause.

```
if SomeBool:
    # O(100) lines
else:
    # O(100) lines
```

Problems:

- Hard to read.
- Hard to test.
- To change one small thing, you need to understand the entire file.
- Impossible to reuse code without copying huge chunks, and then commenting out random bits.

Organise your code into functions

It is much better practice to make your code **modular** and **reusable**. You can achieve this by factoring code into functions and classes.

In Python, scripts can be both modules and executables.

```
def double_number(x):
    return 2*x

if __name__ == "__main__":
    x = 3
    print(f"Twice of {x} is {double_number(x)}")
```

If this script is executed, it will run the code in the `if __name__ == "__main__":` block. However, you can also import this script, and reuse `double_number` elsewhere (rather than copying and pasting).

Organise your code into functions

It is much better practice to make your code **modular** and **reusable**. You can achieve this by factoring code into functions and classes.

In Python, scripts can be both modules and executables.

```
def double_number(x):
    return 2*x

if __name__ == "__main__":
    x = 3
    print(f"Twice of {x} is {double_number(x)}")
```

If this script is executed, it will run the code in the `if __name__ == "__main__":` block. However, you can also import this script, and reuse `double_number` elsewhere (rather than copying and pasting).

Organise your code into (pure) functions

It's considered good practice to write "**pure functions**."

- No global state dependence—only depends on variables that are passed to it.
- No side-effects—no changes to variables it wasn't passed.

```
def print_value():
    global value
    print(value)

value = 3
print_value()  # bad!
```

```
def print_value(value):
    print(value)

value = 3
print_value(value)  # better :)
```

Benefits of writing pure functions:

- Avoids weird bugs due to a global variables being changed without your knowledge.
- They're easier to test and debug.

Organise your code into (short) functions

Another common bad practice: when physicists do write functions, they make them do too much!

A function should do one thing. Functions that do one thing have many benefits:

- easier to write a docstring—should be possible to give a one-sentence explanation,
- easier to give it a descriptive name,
- can be tested in isolation,
- to modify a function, you only need to understand the function.

Organise your code into (short) functions

Signs that your code is in need of refactoring:

- Deeply nested `if`'s and `for`'s.
- Functions are too long.
  - No hard rule, but I try to keep my functions less than 40 lines, with a typical length of 10–20 lines.
- Running almost identical procedures with only a few parameters changed.
  - Factor this into functions which take those parameters as arguments.

Module recommendations from PEP8

Submodules should be logical collections of code. If you've written code for a particular study, put it into a submodule.

- Scope by submodule, not by underscores. That is:

```
# Bad!
~/project/
└── analysisproject/
    ├── study_structures.py
    └── study_models.py
```

```
# Better :)
~/project/
└── analysisproject/
    └── study/
        ├── structures.py
        └── models.py
```

- The Python style guide (PEP 8) recommends modules should have short, all-lowercase names.

Utilising your project structure

One difficulty in maintaining your project structure is accessing particular directories from inside scripts. Typical methods look like:

```
fig.savefig("../../../fig/SomePlot.png")
fig.savefig("/home/philgrace/PhD/analysisproject/fig/SomePlot.png")
```

Both of these are fragile ways of finding the `fig` directory.

Instead, use `analysisproject/definitions.py` to define project-level variables, including the project directory:

```
from pathlib import Path

# Get project directory relative to definitions.py.
# WARNING: Do not move definitions.py without redefining this.
PROJECT_BASE_DIRECTORY = Path(__file__).parent.parent
```

One difficulty in maintaining your project structure is accessing particular directories from inside scripts. Typical methods look like:

```
fig.savefig("../../../fig/SomePlot.png")
fig.savefig("/home/philgrace/PhD/analysisproject/fig/SomePlot.png")
```

Both of these are fragile ways of finding the `fig` directory.

Instead, use `analysisproject/definitions.py` to define project-level variables, including the project directory:

```
from pathlib import Path

# Get project directory relative to definitions.py.
# WARNING: Do not move definitions.py without redefining this.
PROJECT_BASE_DIRECTORY = Path(__file__).parent.parent
```

In `analysisproject/utils.py`, you can then use this to create a convenience function to interact with your project directories:

```python
from pathlib import Path  # pathlib is great!

from analysisproject.definitions import PROJECT_BASE_DIRECTORY

def find_file(*args, **kwargs):
    """Return file path relative to project directory."""
    return Path(PROJECT_BASE_DIRECTORY, *args, **kwargs)
```

Seeing it in action:

```
>>> find_file()
PosixPath('~/project')
>>> find_file("data", "hello", "world.txt")
PosixPath('~/project/data/hello/world.txt')
```

Central configs

If there are settings you use throughout your project, you may wish to factor these out into a **central configuration**. Two benefits:

- increases transparency,
- gives you the ability to change your settings globally (reducing risk of error).

Either put these in:

- a Python script like `analysisproject/definitions.py`, or
- YAML file/s in a `config` directory, which are read in.

Central configs: code for reading YAML configs

For example, if you had a file `~/project/config/datasets.yaml` containing:

```yaml
DatasetLabel: cool_data
InputDatasets:
  - /tape/DataProd/file1.root
  - /tape/DataProd/file2.root
  - /tape/DataProd/file3.root
```

Then you could write a helper function in `utils.py` to read it and return the contents as a `dict`:

```python
import yaml

def read_config(name):
    ConfigFile = find_file("config", name).with_suffix(".yaml")
    with open(ConfigFile) as f:
        config = yaml.safe_load(f)
    return config
```

Central configs: code for reading YAML configs

In action,

```
>>> read_config("datasets")
{
    "DatasetLabel": "cool_data",
    "InputDatasets": [
        "/tape/DataProd/file1.root",
        "/tape/DataProd/file2.root",
        "/tape/DataProd/file3.root",
    ],
}
>>> config = read_config("datasets")
>>> config["DatasetLabel"]
'cool_data'
```

Documentation

Nobody likes writing documentation. But if your documentation is bad, others simply won't use your code.

Think about the "audience" you are writing for:

- new users trying to apply your code or build on it,
- others trying to validate/replicate the work you did,
- yourself in three years time.

Good 30min talk from PyCon AU: What nobody tells you about documentation. Breaks down documentation into four types: tutorials, how-to guides, technical reference, and explanation.

Documentation

Nobody likes writing documentation. But if your documentation is bad, others simply won't use your code.

Think about the "audience" you are writing for:

- new users trying to apply your code or build on it,
- others trying to validate/replicate the work you did,
- yourself in three years time.

Good 30min talk from PyCon AU: What nobody tells you about documentation. Breaks down documentation into four types: tutorials, how-to guides, technical reference, and explanation.

Places where documentation exists:

- `README.md`: Introduce new users, installation instructions, directions to more extensive documentation.
- "Self-documentation" aspects:
  - consistency in variable/function naming convention (often "verb-noun" for functions),
  - consistency in project structure.
- Interface documentation: function and class docstrings. This is often the bulk of the documentation.

Documentation: general tips

- **Write documentation while you write code.**
  - In Python, there are two easy places to do this: docstrings at the top of files, and class/function docstrings.

```
"""
This is a module-level docstring.
It explains the purpose of entire module/script.
"""

def find_file(*args, **kwargs):
    """
    This is a function docstring.
    It explains action function performs, and
    expected arguments & returned values.
    """
    return Path(PROJECT_BASE_DIRECTORY, *args, **kwargs)
```

- The **why** is often more important than the **what** (the user can get the "what" by reading your code).

Unit testing

"Unit tests" run small bits of your code to ensure the individual pieces work. This is easier to do if you've factored your code into functions.

They will typically run a single function, and make some assertion about the expected outcome.

```
import unittest

class TestUtilFunctions(unittest.TestCase):
    def test_double_number(self):
        number = 4.2
        self.assertEqual(double_number(number), 8.4)

if __name__ == "__main__":
    unittest.main()
```

If you unit test your code, then you can immediately find out if a change you've made has broken something in your code.

Unit testing

Almost every language has packages for writing suites of unit tests.

- Python: `unittest` in the standard library
- C++: Google Test
- FORTRAN: FUnit

If you're writing a full-fledged application (like `natpy`), you should aim for full "coverage" (every line of code is tested).

Even if you think full unit testing is overkill, you should at the very least have simple tests to make sure your code works before submitting 2000 batch jobs.

Almost every language has packages for writing suites of unit tests.

- Python: `unittest` in the standard library
- C++: Google Test
- FORTRAN: FUnit

If you're writing a full-fledged application (like `natpy`), you should aim for full "coverage" (every line of code is tested).

Even if you think full unit testing is overkill, you should at the very least have simple tests to make sure your code works before submitting 2000 batch jobs.

Game Corner: Outer Wilds

Not this game: 

- Length: 20 hours
- Skill required: very little
- Genre: Sci-Fi

You're an alien exploring their solar system, unravelling ancient mysteries.

This game was made for physicists (lot of astrophysics and quantum mechanics).

The entire point of this game is discovery, so **anything I could say would be a spoiler.**

Not this game: 

- Length: 20 hours
- Skill required: very little
- Genre: Sci-Fi

You're an alien exploring their solar system, unravelling ancient mysteries.

This game was made for physicists (lot of astrophysics and quantum mechanics).

The entire point of this game is discovery, so **anything I could say would be a spoiler**.

`git`:

- Use it. Please.
- Commit early and often.
- Use pre-commit hooks to do frequent, basic code checks.

Project structure:

- Lay out a project structure in advance.
- Make your project a package, so you can reuse code.

Writing good code:

- Organise your code into small functions.
- Organise those functions into logically-grouped modules.
- Write documentation while you write code.

Backup slides

Embedding metadata in plots

Very common for physicists to include the description of what a figure is and when/how it was made in the **filename**. Better practice is to put this into the **metadata**.

The `Figure.savefig` function in `matplotlib` has a `metadata` option, which expects a `dict`:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 0], [0, 1])
fig.savefig(
    "line.png",
    metadata={"Author": "Phil Grace", "Creator": __file__}
)
```

Valid keys for PDFs: `Title`, `Author`, `Subject`, `Keywords`, `Creator`, `Producer`, `CreationDate`, `ModDate`, `Trapped`.

Embedding metadata in plots

Very common for physicists to include the description of what a figure is and when/how it was made in the **filename**. Better practice is to put this into the **metadata**.

The `Figure.savefig` function in `matplotlib` has a `metadata` option, which expects a `dict`:

```
import matplotlib.pyplot as plt

fig, ax = plt.subplots()
ax.plot([1, 0], [0, 1])
fig.savefig(
    "line.png",
    metadata={"Author": "Phil Grace", "Creator": __file__}
)
```

Valid keys for PDFs: `Title`, `Author`, `Subject`, `Keywords`, `Creator`, `Producer`, `CreationDate`, `ModDate`, `Trapped`.

Viewing image metadata

Most PDF viewers let you easily view the metadata. For PNG images, use the `identify` tool from `imagemagick`:

```
$ identify -verbose line.png
Image: /path/to/line.png
  Format: PNG (Portable Network Graphics)
  Mime type: image/png
  ...
  Properties:
    Author: Phil Grace
    Creator: plot-line.py
    date:create: 2021-03-18T12:19:14+11:00
    date:modify: 2021-03-18T12:19:14+11:00
    ...
    Software: Matplotlib version3.3.3
```

Embedding git info in plots

With git, you have a full history of your code development, and you can use this to record which code version was used to generate plots.

We'll start off by writing a function to run git commands on your project directory:

```
import subprocess
from pathlib import Path

from analysisproject.util import find_file

def run_command(command):
    """Run shell command in project directory and capture output."""
    process = subprocess.run(
        command, cwd=find_file(), stdout=subprocess.PIPE
    )
    return process.stdout.decode("utf8").strip()
```

Embedding git info in plots

```
def get_git_hash():
    """Get the identifier of the last commit in the repo."""
    return run_command(["git", "rev-parse", "HEAD"])

def get_last_commit_date():
    """Get the date of the last commit in the repo."""
    return run_command(["git", "log", "-1", "--format=%cd"])

def make_description():
    """Create a string including the script name and git info."""
    import __main__
    script = Path(__main__.__name__).resolve()  # get script name
    git_hash = get_git_hash()
    date = get_last_commit_date()
    return f"{script} @ {git_hash} (last commit date: {date})"
```

And now putting this into the figure metadata:

```
>>> fig.savefig("line.png", metadata={"Creator": make_description()})
```

Fancy save_plot function

If you want to get very fancy, you can write a wrapper to save as PNG and PDF, and include the metadata in both:

```python
def save_plot(fig, title, *, subdir=".", formats=(".png", ".pdf")):
    """Save figure in multiple formats to fig directory."""

    # Create directory, and don't complain if it already exists
    directory = find_file("fig", subdir)
    directory.mkdir(parents=True, exist_ok=True)

    metadata = {"Creator": make_description()}
    for fmt in formats:
        filename = (directory / title).with_suffix(fmt)
        fig.savefig(filename, metadata=metadata)
        print(f"Wrote {filename}")
```

```
>>> save_plot(fig, "a_great_plot", subdir="better_plots")
Wrote ~/project/fig/better_plots/a_great_plot.png
Wrote ~/project/fig/better_plots/a_great_plot.pdf
```

Even fancier save_plot function

Finally, you can extend your save_plot function to automatically create a timestamped subdirectory.

```
from datetime import datetime as dt

def save_plot_with_date(*args, **kwargs):
    today = dt.strftime(dt.today(), "%Y%m%d")
    save_plot(*args, subdir=today, **kwargs)
```

```
>>> save_plot_with_date(fig, "a_great_plot")
Wrote ~/project/fig/20210326/a_great_plot.png
Wrote ~/project/fig/20210326/a_great_plot.pdf
```