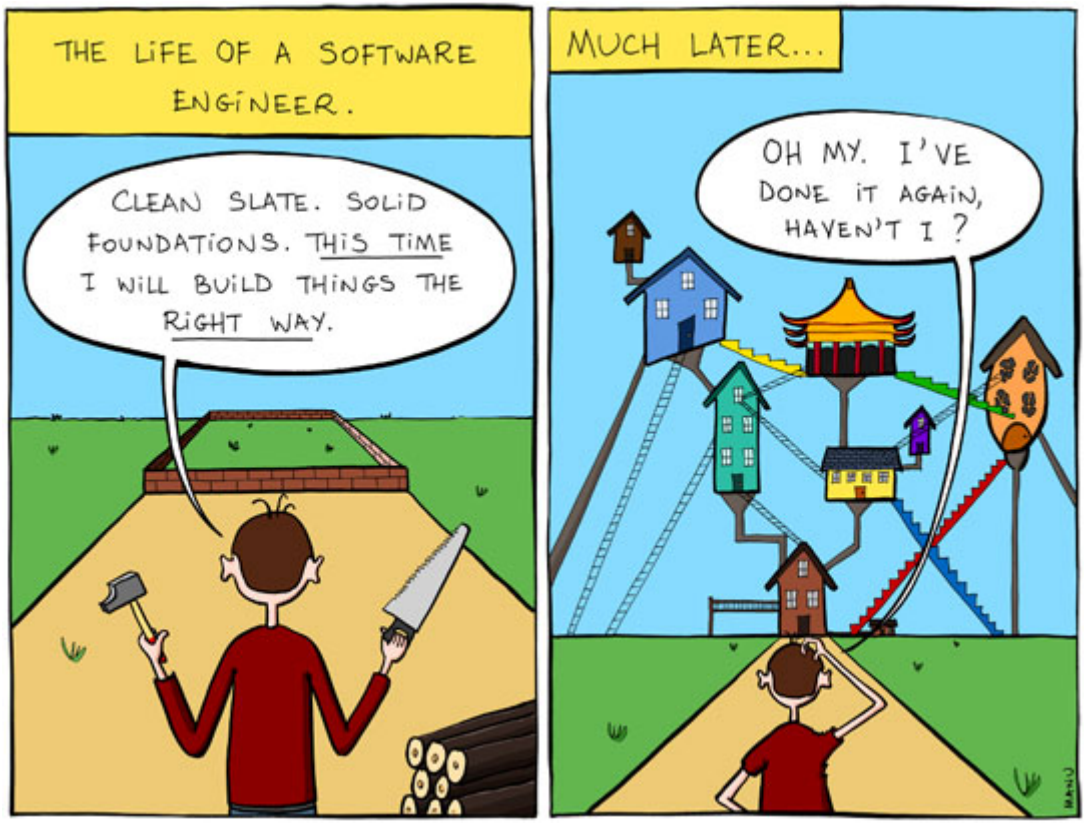
Improving Python Code Structure

or

How not to hate your past self when you look at your code



By Colin Sauze <cos@aber.ac.uk>

For Python Study Group, March 29th 2019

Contents

- Styling code with PEP8
- Static Analysis with Pylint
- Breaking up with functions
- Splitting across multiple files
- Using classes
- Example code and these slides available from: https://github.com/colinsauze/structuring_python

PEP8

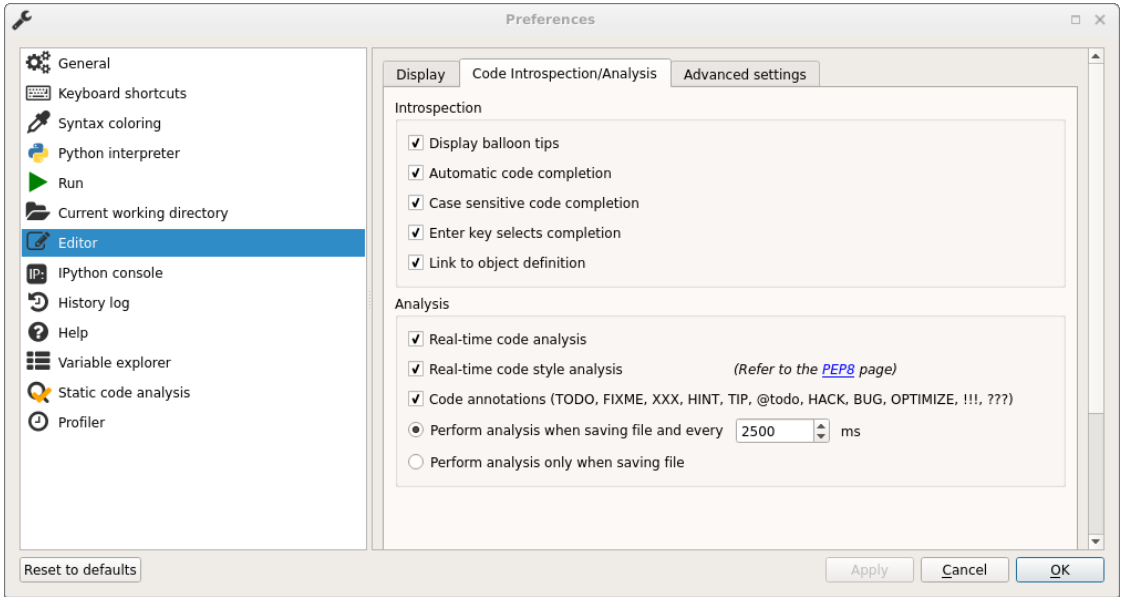
- Python standards document on how Python code should look
  - Where to put spaces, commas, tabs etc
  - Capitalisation conventions
  - Use underscores to separate words in variables/functions
    - e.g. my_variable
- Don’t try to follow it by hand
  - Use automated tools

PEP8 from the command line

- pep8 or pycodestyle command
- `pip install pep8`
- or
- `pip install pycodestyle`
- Both ship with Anaconda
- `pep8 <filename.py>`
- `pycodestyle <filename.py>`

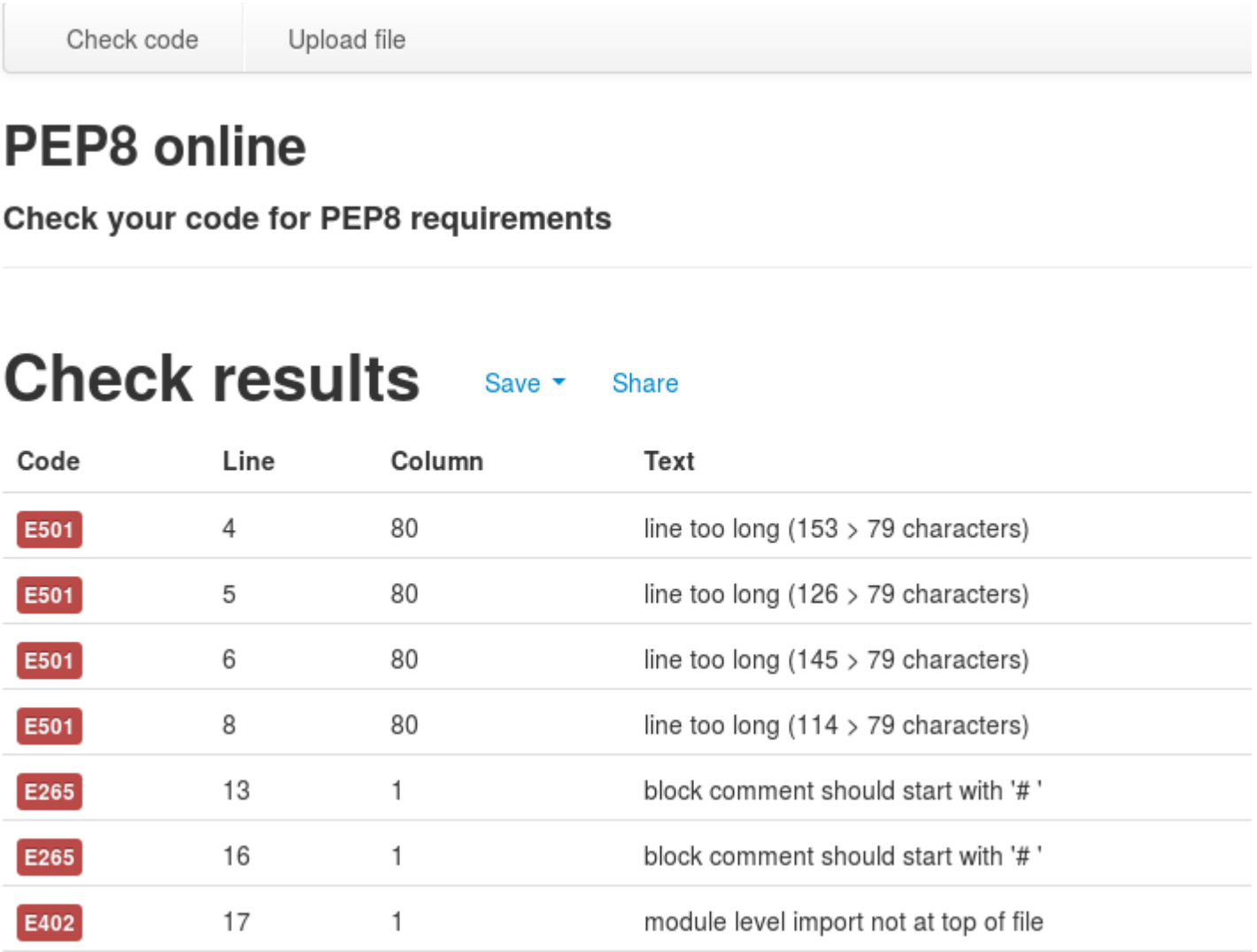
PEP8 in your text editor

- Plugins for many text editors
- In Spyder click
  - Tools Menu, preferences,
  - Editor
  - Code Introspection/Analysis
  - Tick “Real-time code style analysis”



PEP8 checking online

- Paste your code into http://pep8online.com/



Try PEP8 yourself

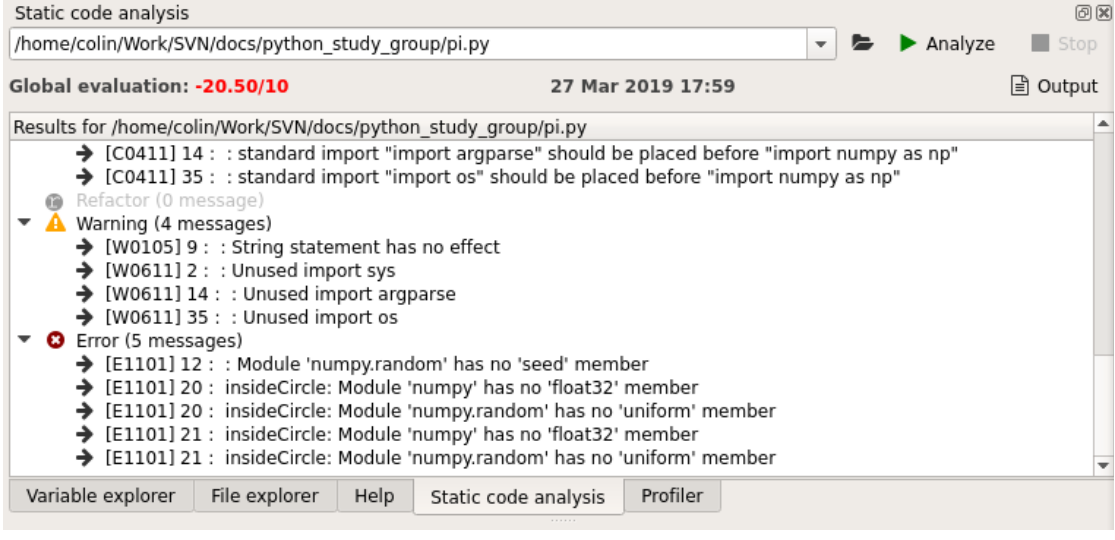
- Download example code from:
  https://github.com/colinsauze/structuring_python/blob/master/pi_messy.py
- Try to fix all the PEP8 errors
- Don’t fix anything else yet
- Look at what issues PEP8 didn’t flag up
- Completed example in pi_pep8.py

Static Analysis with Pylint

- Looks for weaknesses and common mistakes in code
- Overlaps with pep8 but includes more checks
- Very detailed recommendations report
- Very pedantic
- Occasional false positives
- Scores your code out of 10!
  - Negative scores possible!

Running Pylint

- Command line:
  - `pip install pylint`
  - `pylint <filename.py>`
- Spyder:
  - Source menu, Run Static Code Analysis
  - New Static Code Analysis tab opens
  - Click the green analyse button
  - Or press F8



Try Pylint

- Run program from before in Pylint
- Does it pass all checks?
- Can you make it get 10/10?
- Are there still issues with the program that pylint doesn’t flag?
  - Don’t change anything unless pylint requires you to
- Problem with Numpy, messages like: “Module ‘numpy’ has no ‘float32’ member’”
  - At the top of the file add the comment:
  - `# pylint: disable=maybe-no-member`
  - Or
  - `# pylint: disable=no-member`
- “Invalid constant name” errors mean you have a variable outside a function, leave these for now
- Completed example in pi_pylint.py

Refactoring

- Term from Agile Software Development methodologies
  - To improve code readability and quality
  - Remove code “smells”
  - See https://refactoring.guru/smells/

Breaking up code with functions

- Put all code in functions
  - Except a small section to launch the main function
  - Put that inside a `if __name__ == '__main__'` at the bottom of the file.
- Each function should do one thing
- Pass data in as parameters, return the result/modified data
- Document every function

Commenting a function

- Use triple single quotes under the function definition
  - Multi-line comment
  - Use a second triple quote to end the comment
- Describe what the function does
  - :param <paramname>: for each parameter
  - :return: to describe the return.
  - Not used by all document generators.
- Pydoc can build a nice HTML file of these
  - Run pydoc -w <filename> (without .py on the end)
- Alternatives pdoc and sphinx build nicer looking files
  - Pdoc simpler, just pip install pdoc
  - Sphinx requires extra config

Refactor the example code

- Move as much of the code as you can into functions
- Keep functions fairly short (10-20 lines)
- Give each function a single purpose
- Give them a descriptive name
  - Pylint likes names under 30 characters
- Keep pylint and pep8 happy.
- Example in pi_refactored.py

Group functions in files

- When programs get bigger split them across multiple files
- Group common functions together in a single file
- Import them with an import statement
  - `import pi_refactored`
  - `pi_refactored.calculate_pi(10000)`
- Example in pi_refactored_run.py

Using classes

- Object oriented feature
- Put related functions (methods in OO speak) into a class
- Variables inside a class considered internal
  - Python does nothing to enforce this, its just by convention
  - Other OO languages do enforce it
- Class must be instantiated or methods declared static
- All non-static methods in a class take a parameter self
  - self = reference to the current instance

When to use classes?

- You want multiple instances of the same thing
  - Each instance represents one occurrence of something
  - Methods will do something to that data
  - Useful for more complex data structures
- Hierarchy of types
  - Classes can inherit from a parent
  - Copy all the attributes of the parent, add some extras
  - Useful if we want multiple categories of related things
- Larger programs

Using classes properly: Aim for Low Coupling and High Cohesion

- High Cohesion = everything in a class should be related
- Low Coupling = keep interclass dependencies low
- Classes should have some content
  - Java/C++/C# programmers often create classes with almost no content