2nd ASTERICS-OBELICS International School

4-8 June 2018, Annecy, France.

H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).

Good code practice in Python

Zheng Meyer & Tammo Jan Dijkema
ASTRON

Best Practices for Scientific Computing

ASTRON
Netherlands Institute for Radio Astronomy

Write programs for people, not computers.

a) A program should not require its readers to hold more than a handful of facts in memory at once.
b) Make names consistent, distinctive, and meaningful.
c) Make code style and formatting consistent.

Let the computer do the work.

a) Make the computer repeat tasks.
b) Save recent commands in a file for re-use.
c) Use a build tool to automate workflows.

Make incremental changes.

a) Work in small steps with frequent feedback and course correction.
b) Use a version control system.
c) Put everything that has been created manually in version control.

Don’t repeat yourself (or others).

a) Every piece of data must have a single authoritative representation in the system.
b) Modularize code rather than copying and pasting.
c) Re-use code instead of rewriting it.

Plan for mistakes.

a) Add assertions to programs to check their operation.
b) Use an off-the-shelf unit testing library.
c) Turn bugs into test cases.
d) Use a symbolic debugger.

Optimize software only after it works correctly.

a) Use a profiler to identify bottlenecks.
b) Write code in the highest-level language possible.

Document design and purpose, not mechanics.

a) Document interfaces and reasons, not implementations.
b) Refactor code in preference to explaining how it works.
c) Embed the documentation for a piece of software in that software.

Collaborate.

a) Use pre-merge code reviews.
b) Use pair programming when bringing someone new up to speed and when tackling particularly tricky problems.
c) Use an issue tracking tool.

PEPs (Python Enhancement Proposals)

PEP 8 Style Guide for Python code
PEP 20 The Zen of Python

ASTRON
Netherlands Institute for Radio Astronomy

PEP 8 Style **GUIDE** for Python Code

The guidelines are intended to improve the readability of code
Consistency is the KEY

- Consistency with the style guide is important.
- Consistency within a project is more important.
- Consistency within one module or function is the most important.

A Foolish Consistency is the Hobgoblin of Little Minds

Code Lay-out – Indentation & Line break

Indentation

- Use 4 spaces per indentation level.
- Spaces are the preferred indentation method.
  Should a line break before or after a binary operator?
- Consistency is the key

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

Code Lay-out – Blank Lines

Blank Lines

- Surround top-level function and class definitions with two blank lines.
- Method definitions inside a class are surrounded by a single blank line.

```
from setuptools import setup
from setuptools.command.test import test as TestCommand


class PyTest(TestCommand):
    user_options = [('pytest-args=', 'a', "Arguments to pass into
py.test")]

    def initialize_options(self):
        TestCommand.initialize_options(self)
        self.pytest_args = []
```

ASTRON
Netherlands Institute for Radio Astronomy

Code Lay-out – Imports

Imports should usually be on separate lines, e.g.:

Yes:

```
import os
import sys
```

No:

```
import sys, os
```

Imports are always put at the top of the file

Absolute imports are recommended

```
import mypkg.sibling
from mypkg import sibling
from mypkg.sibling import example
```

Wildcard imports (`from module import *`) should be avoided

Code Lay-out – Comments

Comments that contradict the code are worse than no comments.
Always make a priority of keeping the comments up-to-date when the code changes!
Comments should be complete sentences.
Write your comments in English.

Code Lay-out – Comments Contd.

Block Comments

- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code.
- Each line of a block comment starts with a # and a single space.

```
# Code examples for Good code practice in Python.
```

Inline Comments

- Use inline comments sparingly.
- Inline comments are unnecessary and in fact distracting if they state the obvious. **DON'T** do this:

```
x = x + 1                 # Increment x
```

Code Lay-out – Comments Contd.

Block Comments

- Block comments generally apply to some (or all) code that follows them, and are indented to the same level as that code.
- Each line of a block comment starts with a # and a single space.

```
# Code examples for Good code practice in Python.
```

Inline Comments

- Use inline comments sparingly.
- Inline comments are unnecessary and in fact distracting if they state the obvious. **DON'T** do this:

```
x = x + 1                 # Increment x
```

Code Lay-out – Comments Contd.

Documentation Strings (a.k.a. "docstrings")

- A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition.
- Such a docstring becomes the __doc__ special attribute of that object.
- PEP 257 describes good docstring conventions.

Most importantly, the """ that ends a multiline docstring should be on a line by itself
For one liner docstrings, please keep the closing """ on the same line.

```
"""Return a foobang.

Optional plotz says to frobnicate the bizbaz first.
"""
```

Docstring Versus Block Comments

```
# This function slows down program execution for some reason.
def square_and_rooter(x):
    """Return the square root of self times self."""
    ...
```

The leading comment block is a programmer's note.
The docstring describes the operation of the function or class and will be shown in an interactive Python session when the user types

```
help(square_and_rooter)
```

Self-Documenting Code - Naming

A variable, class, or function name should speak for themselves.

```
decay()
decay_constant()
get_decay_constant()
```

```
p = 100
pressure = 100
```

Self-Documenting Code – Simple functions

Functions must be small to be understandable and testable.
It should do ONLY one thing.

```
import numpy as np

def initial_cond(N, Dim):
    """Generates initial conditions for N unity masses at rest
    starting at random positions in D-dimensional space.
    """
    position0 = np.random.rand(N, Dim)
    velocity0 = np.zeros((N, Dim), dtype=float)
    mass = np.ones(N, dtype=float)
    return position0, velocity0, mass
```

PEP 20 The Zen of Python

By Tim Peters

ASTRON
Netherlands Institute for Radio Astronomy

Beautiful is better than ugly.
Explicit is better than implicit.

The Zen of Python

Explicit is better than implicit

Bad

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

Good

```
def make_complex(x, y):
    return {'x': x, 'y': y}
```

Simple is better than complex.
Complex is better than complicated.
Sparse is better than dense.

The Zen of Python

Make only one statement per line

Bad

```
print('one'); print('two')

if x == 1: print('one')

if <complex comparison> and
<other complex comparison>:
    # do something
```

Good

```
print('one')
print('two')

if x == 1:
    print('one')

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

ASTRON
Netherlands Institute for Radio Astronomy

Errors should never pass silently.
Unless explicitly silenced.

The Zen of Python

There should be one-- and preferably only one – obvious way to do it.
Although that way may not be obvious at first unless you're Dutch.

The Zen of Python

If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.

The Zen of Python

ASTRON
Netherlands Institute for Radio Astronomy

```
>>> import this
```

Want to see the complete list of The Zen of Python?

Argparse – Command line option and argument parsing

ASTRON
Netherlands Institute for Radio Astronomy

The `argparse` module

Added to Python 2.7 as a replacement for `optparse`.
The API for argparse is similar to the one provided by optparse.

The parser class is ArgumentParser.
The constructor takes several arguments to set up the description used in the help text for the program.

`argparse` is a complete argument processing library.

```
import argparse

parser = argparse.ArgumentParser(description='Short sample app')

parser.add_argument('-a', action="store_true", default=False)
parser.add_argument('-b', action="store", dest="b")
parser.add_argument('-c', action="store", dest="c", type=int)

print(parser.parse_args(['-a', '-bval', '-c', '3']))
```

Structuring Your Project

ASTRON
Netherlands Institute for Radio Astronomy

Sample repository by Kenneth Reitz

```
samplemod
├── LICENSE
├── MANIFEST.in
├── Makefile
├── README.rst
├── docs
│   ├── Makefile
│   ├── conf.py
│   ├── index.rst
│   └── make.bat
├── requirements.txt
├── sample
│   ├── __init__.py
│   ├── core.py
│   └── helpers.py
├── setup.py
└── tests
    ├── __init__.py
    ├── context.py
    ├── test_advanced.py
    └── test_basic.py
```

ASTRON
Netherlands Institute for Radio Astronomy

Pitfalls to avoid

- Multiple and messy circular dependencies
- Hidden coupling
  - Modifying code in one class breaks many tests in unrelated test cases
- Heavy use of global state or context
- Spaghetti code
  - Multiple pages of nested `if` clauses and `for` loops with a lot of copy-pasted procedural code and no proper segmentation
- Ravioli code
  - Consists of hundreds of similar little pieces of logic without proper structure

Decorators

Dynamically alter the functionality of a function, method, or class without having to change the source code of the function being decorated
Helps separate business logic from administrative logic

```
from python_toolbox.caching import cache
@cache()
def f(x):
    print('Calculating...')
    return x ** x
```

Dynamic Typing

Avoid using the same variable name for different things
Good discipline: assign a variable only once
Check your code: Pylint, Pyflakes, Flakes8, Pychecker

Bad

```
a = 1
a = 'a string'
def a():
    pass  # Do something

items = 'a b c d'
items = items.split(' ')
items = set(items)
```

Good

```
count = 1
msg = 'a string'
def func():
    pass  # Do something

items_string = 'a b c d'
items_list = items_string.split(' ')
items = set(items_list)
```

ASTRON
Netherlands Institute for Radio Astronomy

References (1)

- Best Practices for Scientific Computing
  Wilson G, Aruliah DA, Brown CT, Chue Hong NP, Davis M, et al. (2014) Best Practices for Scientific Computing. PLOS Biology 12(1): e1001745.https://doi.org/10.1371/journal.pbio.1001745
- Best Practices in Scientific Computing – Software Carpentry
  http://swcarpentry.github.io/slideshows/best-practices/#slide-0
- The Hitchhacker's guide to Python by Kenneth Reitz, Tanya Schlusser. Publisher: O'Reilly Media, Inc.
  http://python-guide-pt-br.readthedocs.io/en/latest/

References (2)

- Transforming Code into Beautiful, Idiomatic Python by Raymond Hettinger – PyCon 2013
  https://www.youtube.com/watch?v=OSGv2VnC0go
- Raymond Hettinger - Beyond PEP 8 -- Best practices for beautiful intelligible code - PyCon 2015
  https://www.youtube.com/watch?v=wf-BqAjZb8M
- Effective Computation in Physics by Anthony Scopatz, Kathryn D. Huff. Publisher: O'Reilly Media, Inc.
  http://physics.codes/

Acknowledgement

- H2020-Astronomy ESFRI and Research Infrastructure Cluster (Grant Agreement number: 653477).