Good code practice in Python

Zheng Meyer-Zhao
SURFsara

Outline

Best practices for scientific computing
PEP 8 & PEP 20
Conventions and Idioms
Structuring your project
Testing your code

Best Practices for Scientific Computing

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

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 = []
```

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.

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

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
```

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

```
>>> import this
```

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

Conventions and Idioms

Alternatives to checking for equality

Bad

```
if attr == True:
    print('True!')

if attr == None:
    print('attr is None!')
```

Good

```
# Just check the value
if attr:
    print('attr is truthy!')

# or check for the opposite
if not attr:
    print('attr is falsey!')

# or, since None is considered false,
explicitly check for it
if attr is None:
    print('attr is None!')
```

Accessing dictionary elements

Bad (also removed in Python 3.x)

```
d = {'hello': 'world'}

if d.has_key('hello'):
    print(d['hello'])
else:
    print('default_value')
```

Good

```
d = {'hello': 'world'}

print(d.get('hello', 'default_value'))
print(d.get('thingy', 'default_value'))

# Or:
if 'hello' in d:
    print(d['hello'])
```

Looping over dictionary keys

Bad (This doen't work in Python 3.x any more)

```
d = {'matthew': 'blue', 'rachel': 'green', 'raymond': 'red'}

for k in d:
    print(k)

for k in d.keys():
    if k.startswith('r'):
        del d[k]
```

Good

```
d = {k: d[k] for k in d if not k.startswith('r')}
print(d)
```

Manipulating lists

Bad

```
# Filter elements greater than 4
a = [3, 4, 5]
b = []
for i in a:
    if i > 4:
        b.append(i)
```

Good

```
# List comprehension
a = [3, 4, 5]
b = [i for i in a if i > 4]

# Or:
b = filter(lambda x: x > 4, a)
```

Manipulating lists Contd.

Bad

```
# Add three to all list members.
a = [3, 4, 5]
for i in range(len(a)):
    a[i] += 3
```

Good

```
# List comprehension
a = [3, 4, 5]
a = [i + 3 for i in a]

# Or:
a = map(lambda i: i + 3, a)
```

Looping over a collection and indices

What people normally do

```
colors = ['red', 'green', 'blue', 'yellow']

for i in range(len(colors)):
    print(i, '--->', colors[i])
```

Better

```
for i, color in enumerate(colors):
    print(i, '--->', color)
```

Distinguishing multiple exit points in loops

What people normally do

```
def find(seq, target):
    found = False
    for i, value in enumerate(seq):
        if value == target:
            found = True
            break
    if not found:
        return -1
    return i
```

Better

```
def find(seq, target):
    for i, value in enumerate(seq):
        if value == target:
            break
    else:
        return -1
    return i
```

Unpacking sequences

What people normally do

```
p = 'Raymond', 'Hettinger', 0x30, 'python@example.com'

fname = p[0]
lname = p[1]
age = p[2]
email = p[3]
```

Better

```
fname, lname, age, email = p
```

Updating multiple state variables

What people normally do

```
def fibonacci(n):
    x = 0
    y = 1
    for i in range(n):
        print(x)
        t = y
        y = x + y
        x = t
```

Better

```
def fibonacci(n):
    x, y = 0, 1
    for i in range(n):
        print(x)
        x, y = y, x+y
```

Concatenating strings

What people normally do

```
names = ['raymond', 'rachel', 'matthew', 'roger',
         'betty', 'melissa', 'judith', 'charlie']

s = names[0]
for name in names[1:]:
    s += ', ' + name

print(s)
```

Better

```
print(', '.join(names))
```

How to open and close files

What people normally do

```
f = open('data.txt')
try:
    data = f.read()
finally:
    f.close()
```

Better

```
with open('data.txt') as f:
    data = f.read()
```

Structuring Your Project

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
```

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)
```

Virtual environment

Keeps dependencies required by different projects in separate places
Keeps your global site-packages directory clean
Very handy when you need a specific version of a package for certain projects
Easy to setup the environment thanks to `requirements.txt`

```
$ cd projectname
$ virtualenv –p python2 venv
$ source venv/bin/activate
$ source venv/bin/activate
$ (venv)$ pip install –r requirements.txt
```

Virtual environment – Use requirements.txt

Method 1:

```
$ pip freeze > requirements.txt
$ pip install –r requirements.txt
```

Method 2:
Use pipreqs: generate requirements.txt based on imports

```
$ pip install pipreqs
$ pipreqs /path/to/project
$ pip install –r requirements.txt
```

Testing your code

Why, when and where?

It is important

- Check whether your code works correctly
- Save time in debugging

Always test your code

- When you start, and again when you finish
- Test-Driven Development
- Continuous integration

Separate code from tests as much as possible

- Have a top-level `tests/` directory in your project

Good tests should be

Automated
Fast
Reliable
Informative

- Test functions should have names starting with `test_`.
- Name of test functions should describe what the test does, it can be very long if needed.

Focused – test just one thing per test

What and how to test?

Test what is important

- Compare expected outputs versus observed outputs for known inputs
- Should cover behavior from the common to the extreme, but not every single value within those bounds
- Test edge/corner cases

Test frameworks

- unittest – included in Python standard library
- pytest
- nose2

Test coverage

- Can be used to see which part of the code is tested
- Though caution needs to be taken when interpreting the results

Write tests with `unittest`

```
import unittest


class BasicTestSuite(unittest.TestCase):
    """Basic test cases."""

    def test_absolute_truth_and_meaning(self):
        assert True


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

Running tests

```
[MLT0093:samplemod zhengm$ nose2
hmmm...
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s

OK
[MLT0093:samplemod zhengm$ py.test
============================= test session starts ==============================
platform darwin -- Python 2.7.13, pytest-3.1.1, py-1.4.33, pluggy-0.4.0
rootdir: /Users/zhengm/src/play/python/samplemod, inifile:
collected 2 items

tests/test_advanced.py .
tests/test_basic.py .

=========================== 2 passed in 0.04 seconds ===========================
```

Write tests with nose

```
from nose.tools import assert_equal
from myproject.fibonacci import fibonacci

def test_fibonacci_0():
    # test edge 0
    obs = fibonacci(0)
    assert_equal(0, obs)

def test_fibonacci_1():
    # test edge 1
    obs = fibonacci(1)
    assert_equal(1, obs)
```

Write tests with pytest (1)

```
import pytest
from myproject.wallet import Wallet, InsufficientAmount

@pytest.fixture
def empty_wallet():
    """Returns a Wallet instance with zero balance"""
    return Wallet()

@pytest.fixture
def wallet():
    """Returns a Wallet instance with a balance of 10"""
    return Wallet(10)
```

Write tests with pytest (2)

```
@pytest.fixture
def my_wallet():
    """Returns a Wallet instance with a balance of 20"""
    return Wallet(20)

def test_default_initial_amount(empty_wallet):
    assert empty_wallet.balance == 0

def test_setting_initial_amount(wallet):
    assert wallet.balance == 10

def test_wallet_add_cash(wallet):
    wallet.add_cash(90)
    assert wallet.balance == 100
```

Write tests with pytest (3)

```
def test_wallet_spend_cash(wallet):
    wallet.spend_cash(10)
    assert wallet.balance == 0

def
test_wallet_spend_cash_raises_exception_on_insufficient_amount(empty_
wallet):
    with pytest.raises(InsufficientAmount):
        empty_wallet.spend_cash(100)
```

Write tests with pytest (4)

```
@pytest.mark.parametrize("earned, spent, expected", [
    (30, 10, 40),
    (20, 2, 38),
])
def test_transactions(my_wallet, earned, spent, expected):
    my_wallet.add_cash(earned)
    my_wallet.spend_cash(spent)
    assert my_wallet.balance == expected
```

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/
- Transforming Code into Beautiful, Idiomatic Python by Raymond Hettinger – PyCon 2013
  https://www.youtube.com/watch?v=OSGv2VnC0go

References (2)

- 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/
- Ned Batchelder: Getting Started Testing - PyCon 2014
  https://www.youtube.com/watch?v=FxSsnHeWQBY
- A nice pytest tutorial
  https://semaphoreci.com/community/tutorials/testing-python-applications-with-pytest
- About context managers
  https://jeffknupp.com/blog/2016/03/07/python-with-context-managers/