The Best of the Best Practices (BOBP) Guide for Python

A "Best of the Best Practices" (BOBP) guide to developing in Python.

In General

Values

- "Build tools for others that you want to be built for you." - Kenneth Reitz
- "Simplicity is alway better than functionality." - Pieter Hintjens
- "Fit the 90% use-case. Ignore the nay sayers." - Kenneth Reitz
- "Beautiful is better than ugly." - PEP 20
- Build for open source (even for closed source projects).

General Development Guidelines

- "Explicit is better than implicit" - PEP 20
- "Readability counts." - PEP 20
- "Anybody can fix anything." - Khan Academy Development Docs
- Fix each broken window (bad design, wrong decision, or poor code) *as soon as it is discovered*.
- "Now is better than never." - PEP 20
- Test ruthlessly. Write docs for new features.
- Even more important that Test-Driven Development--*Human-Driven Development*
- These guidelines may--and probably will--change.

In Particular

Style

Follow PEP 8, when sensible.

Naming

- Variables, functions, methods, packages, modules
  - `lower_case_with_underscores`
- Classes and Exceptions
  - `CapWords`
- Protected methods and internal functions
  - `_single_leading_underscore(self, ...)`
- Private methods
  - `__double_leading_underscore(self, ...)`
- Constants
  - `ALL_CAPS_WITH_UNDERSCORES`

General Naming Guidelines

Avoid one-letter variables (esp. `l`, `O`, `I`).

Exception: In very short blocks, when the meaning is clearly visible from the immediate context

Fine

```
for e in elements:
    e.mutate()
```

Avoid redundant labeling.

Yes

```
import audio

core = audio.Core()
controller = audio.Controller()
```

No

```
import audio

core = audio.AudioCore()
controller = audio.AudioController()
```

Prefer "reverse notation".

Yes

```
elements = ...
elements_active = ...
elements_defunct = ...
```

No

```
elements = ...
active_elements = ...
defunct_elements ...
```

Avoid getter and setter methods.

Yes

```
person.age = 42
```

No

```
person.set_age(42)
```

Indentation

Use 4 spaces--never tabs. Enough said.

Imports

Import entire modules instead of individual symbols within a module. For example, for a top-level module `canteen` that has a file `canteen/sessions.py`,

Yes

```
import canteen
import canteen.sessions
from canteen import sessions
```

No

```
from canteen import get_user  # Symbol from canteen/__init__.py
from canteen.sessions import get_session  # Symbol from canteen/sessions.py
```

Exception: For third-party code where documentation explicitly says to import individual symbols.

Rationale: Avoids circular imports. See here.

Put all imports at the top of the page with three sections, each separated by a blank line, in this order:

1. System imports
2. Third-party imports
3. Local source tree imports

Rationale: Makes it clear where each module is coming from.

Documentation

Follow PEP 257's docstring guidelines. reStructured Text and Sphinx can help to enforce these standards.

Use one-line docstrings for obvious functions.

```
"""Return the pathname of ``foo``."""
```

Multiline docstrings should include

- Summary line
- Use case, if appropriate
- Args
- Return type and semantics, unless `None` is returned

```
"""Train a model to classify Foos and Bars.

Usage::

    >>> import klassify
    >>> data = [("green", "foo"), ("orange", "bar")]
    >>> classifier = klassify.train(data)

:param train_data: A list of tuples of the form ``(color, label)``.
:rtype: A :class:`Classifier <Classifier>`
"""
```

Notes

- Use action words ("Return") rather than descriptions ("Returns").
- Document `__init__` methods in the docstring for the class.

```
class Person(object):
    """A simple representation of a human being.
```

```
    :param name: A string, the person's name.
    :param age: An int, the person's age.
    """
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

On comments

Use them sparingly. Prefer code readability to writing a lot of comments. Often, small methods are more effective than comments.

No

```
# If the sign is a stop sign
if sign.color == 'red' and sign.sides == 8:
    stop()
```

Yes

```
def is_stop_sign(sign):
    return sign.color == 'red' and sign.sides == 8

if is_stop_sign(sign):
    stop()
```

When you do write comments, remember: "Strunk and White apply." - PEP 8

Line lengths

Don't stress over it. 80-100 characters is fine.

Use parentheses for line continuations.

```
wiki = (
    "The Colt Python is a .357 Magnum caliber revolver formerly manufactured "
    "by Colt's Manufacturing Company of Hartford, Connecticut. It is sometimes "
    'referred to as a "Combat Magnum". It was first introduced in 1955, the '
    "same year as Smith & Wesson's M29 .44 Magnum."
)
```

Testing

Strive for 100% code coverage, but don't get obsess over the coverage score.

General testing guidelines

- Use long, descriptive names. This often obviates the need for doctrings in test methods.
- Tests should be isolated. Don't interact with a real database or network. Use a separate test database that gets torn down or use mock objects.
- Prefer factories to fixtures.
- Never let incomplete tests pass, else you run the risk of forgetting about them. Instead, add a placeholder like `assert False, "TODO: finish me"` .

Unit Tests

- Focus on one tiny bit of functionality.

- Should be fast, but a slow test is better than no test.
- It often makes sense to have one testcase class for a single class or model.

```
import unittest
import factories

class PersonTest(unittest.TestCase):
    def setUp(self):
        self.person = factories.PersonFactory()

    def test_has_age_in_dog_years(self):
        self.assertEqual(self.person.dog_years, self.person.age / 7)
```

Functional Tests

Functional tests are higher level tests that are closer to how an end-user would interact with your application. They are typically used for web and GUI applications.

- Write tests as scenarios. Testcase and test method names should read like a scenario description.
- Use comments to write out stories, *before writing the test code*.

```
import unittest

class TestAUser(unittest.TestCase):

    def test_can_write_a_blog_post(self):
        # Goes to the her dashboard
        ...
        # Clicks "New Post"
        ...
        # Fills out the post form
        ...
        # Clicks "Submit"
        ...
        # Can see the new post
        ...
```

Notice how the testcase and test method read together like "Test A User can write a blog post".

Inspired by...

- PEP 20 (The Zen of Python)
- PEP 8 (Style Guide for Python)
- The Hitchiker's Guide to Python
- Khan Academy Development Docs
- Python Best Practice Patterns
- Pythonic Sensibilities
- The Pragmatic Programmer
- and many other bits and bytes