Python Best Practice

Christian Külker

2020-01-18

Contents

This guides follows some ideas from `The Hitchhiker's Guide To Python` and other ideas from the internet and myself. It is a project in progress.

1 PEP (Python Enhanced Proposals)

- `PEP 8`: The Python Style Guide. Read this. All of it. Follow it.
- `PEP 257`: `Docstring` Conventions. Gives guidelines for semantics and conventions associated with Python `docstrings`.

- PEP Index

2 IDE

This sections contains only IDE and editors that I have Python experience with. Be aware that this section is incomplete.

2.1 Vim

Make `vim` `PEP8` compatible. Best if `vim` was compiled with `+python`.

```
set textwidth=79  " lines longer than 79 columns will be broken
set shiftwidth=4  " operation >> indents 4 columns; << unindents 4 columns
set tabstop=4     " a hard TAB displays as 4 columns
set expandtab     " insert spaces when hitting TABs
set softtabstop=4 " insert/delete 4 spaces when hitting a TAB/BACKSPACE
set shiftround    " round indent to multiple of 'shiftwidth'
set autoindent    " align the new line indent with the previous line
```

- If you are using vim also for other language, use plugin ident.
- Improved syntax highlighting.
- `PEP8` compliancepycodestyle
- `PEP8` compliance Pyflakes install vim-flake8 to check PEP8 from within `vim`. Map function `Flake8` to a hotkey.
- Check PEP8 (Pyflake) on every save, edit `.vimrc`: `autocmd BufWritePost*.py call Flake8()`.
- Show PEP8 errors and warnings with syntastic in the quickfix window.

```
set statusline+=%#warningmsg#
set statusline+=%{SyntasticStatuslineFlag()}
set statusline+=%*
let g:syntastic_auto_loc_list=1
let g:syntastic_loc_list_height=5
```

- Use Python-mode: Python code checking with pylint, pyflakes, pycodestyle or mccabe; refactoring, authocompletion with Rope; virtualenv support, search documentation.
- Use SuperTab for code completion with the tab key.

3 Tools

3.1 Pip

Install `pip` if not already done.

3.2 IPython

IPhthon should be used interactively with the features: shell, web-base notebook, data data visualization, GUI toolkits, parallel execution.

```
pip install ipython      # base install only
pip install ipython[all] # notebook qtconsole, tests, ...
```

4 Environment

Use **virtual environments** on a per project base (also with `pip`).

Bash: (for `pip install`, this will make `pip` require a virtual environment)

```
export PIP_REQUIRE_VIRTUALENV=true
```

See this example environment. TODO (/srv/env/python)

5 Project Layout

This is the layout of the project `sample`

```
README.rst
LICENSE             # full license text
Makefile            # not mandatory
setup.py            # package distribution management
requirements.txt    # pip requirements file
sample/__init__.py  # 'sample' is the project name, mostly empty
sample/core.py      # 'sample' is the project name
sample/helpers.py   # 'sample' is the project name
docs/conf.py        # package reference documentation
docs/index.rst      # package reference documentation
tests/test_basic.py
tests/test_advanced.py
```

- use directory `sample` (and replace this with a real name) and do not use `src` or `python`.

6 Makefile

```
init:
    pip install -r requirements.txt
test:
    py.test tests
.PHONY:init test
clean:
    # remove bytecode
    find . -type f -name "*.py[co]" -delete -or -type d -name
    ↪ "__pycache__" -delete
```

7 Testing

- Use a simple and explicit path modification to resolve the package properly
- Individual tests should have import context, create a tests/context.py file:

```
import os
import sys
sys.path.insert(0,os.path.abspath(os.path.join(os.path.dirname(__file__),'..')))

import sample
```

- Within the individual test modules: import the module like so

```
from.context import sample
```

8 Modules

- Module names should be short
- Module names should not contain spaces
- Module names should not contain hyphens

8.1 Import

- Do not use `from module import *`
- Might do `from module import function` might be OK

```
"""Bad:"""
        from MODULE import *
```

```
"""
        FUNCTION(PARAMETER)
"""

"""Better:"""
        from MODULE import FUNCTION
"""
        FUNCTION(PARAMETER)
"""

"""Best:"""
        import MODULE
"""
        MODULE.FUNCTION(PARAMETER)
"""
```

- For nested package use: `import dir1.dir2.module as mod`.

9 Packages

- A directory with an `__init__.py` file is a `package`
- Do not add too many code to `__init__.py`
- It is considered a good practice to leave `__init__.py` emtpy

The statement `import pack.module` will look for the file `pack/__init.py__` and execute all top level statements and than looks for `pack/module.py` and execute all top level statements.

10 Function

Use stateless functions, if possible, because **pure functions** are:

- **deterministic:** the output will always be the same (for same input)
- **easier to change or replace:** if needed
- **easier to test:** with unit tests (less need complex context setup and data cleaning afterwards)
- **easier to manipulate, decorate, and pass around**

11 Decorators

- Use decorator syntax

12 Context Managers

- Use content managers

13 Dynamic Typing

- Do not use same variable for different types of content

14 Mutable (Changeable) and Immutable (Fixed) Types

- Mutable Types should not be used for dictionary keys

```
"""Bad"""
nums = ""
for n in range(20):
    nums += str(n) # slow and inefficient
print nums

"""Good"""
nums = []
for n in range(20):
nums.append(str(n))
print "".join(nums) # much more efficient

"""Better"""
nums = [str(n) for n in range(20)]
print "".join(nums)

"""Best"""
nums = map(str, range(20))
print "".join(nums)
```

```
foo = 'foo'
bar = 'bar'
foobar = foo + bar # This is good
foo += 'ooo' # This is bad, instead you should do:
foo = ''.join([foo, 'ooo'])
```

```
foo = 'foo'
bar = 'bar'
foobar = '%s%s' % (foo, bar) # It is OK
foobar = '{0}{1}'.format(foo, bar) # It is better
foobar = '{foo}{bar}'.format(foo=foo, bar=bar) # It is best
```

15 Explicit Code

```
"""Bad"""
def make_complex( *args):
    x, y = args
    return dict( **locals())
"""Good"""
def make_complex(x, y):
    return {'x': x, 'y': y}
```

16 Statement Per Line

- Use one statement per line

17 Function Parameters (Arguments)

- Easy to read (the name and arguments need no explanations)
- Easy to change (adding a new keyword argument does not break other parts of the code)

17.1 Positional Arguments

- Should be used for simple functions

```
print_at(x,y)
draw_line(x1,y1,x2,y2)
```

17.2 Keyword Arguments

- Often used with optional parameters

```
    send(message, to, cc=None, bcc=None)
```

17.3 Arbitrary Argument List

- Use carefully or avoid it. This can often not be revoked when removing parameters, in a later stage of the application

```
"""Bad"""
send(message, *args)
send('Hello', ['Bilbo', 'Frodo', 'Sauron']
"""Better"""
send(message, recipients)
send('Hello', ['Bilbo', 'Frodo', 'Sauron']
```

17.4 Arbitrary Keyword Argument Dictionary

- Use carefully only if needed.
- **kwargs

18 Magic

- Do not use magic
- Do not change how objects are created and instantiated
- Do not change how the Python interpreter imports modules
- Try not to embed C routines in Python, if possible

19 Be a nice user

- Do not write client code can override an object's properties and methods

20 Return values

- Try to avoid multiple places for return statements

21 Documentation

While there usually one way to do it. It seems when it comes to documentation there are more.

Python makes no exception by suggesting to maintain a `README` file for project documentation and less a `INSTALL` document as ways of installing are usually known and similar. The Python Guide suggests the presence of a `LICENSE` file.

If the project gets bigger and the `README` will get to long (only then) parts of the `README` might be offloaded to a `TODO` file and a `CHANGELOG` file.

For the format `reStructuredText` or `Markdown` is suggested.

Usually this is sufficient for small and medium sized projects. Often the dead of an aspiring open source software project arise when the project reaches the size, that a README is not sufficient any more. Then a wiki or even a home page is suggested and political factions emerge that sometimes fighting severe bike shed color wars. The following short paragraph is for these situations.

`Shpinx` is a popular Python documentation tool utilize `reStructuredText` and spits HTML and PDF out.

For other parts it is advisable to already document the source code. Python can be documented well with `docstrings` (`PEP 0257#specification`) in favor of block comments (`PEP 8#comments`).

A `docstring` comment is basically a multi line comment with three quotes and in some cases they can be used to supplement unit tests.

```
def add_two_numbers(x, y):
    """
    SUM = add_two_numbers(NUMBER_0, NUMBER_1)

    - Combine two NUMBERS via the operation of addition
    - Require a two NUMBERS: NUMBER_0, NUMBER_1
    - Return the sum of 2 NUMBERS
    """
    return x + y
```

Of course in such an obvious case a one line `docstring` comment is apropriate

```
def add_two_numbers(x, y):
    """Add two numbers and return the sum."""
    return x + y
```

22 Bytecode

- Do not commit `*.pyc` files to the source code repository

- Can be disabled with `export PYTHONDONTWRITEBYTECODE=1` (but probably should not)

23 Git

23.1 .gitignore

```
*.pyc
*.pyo
*.pyd
__pycache__
```

Or

```
  # Will match .pyc, .pyo and .pyd files.
*.py[cod]
  # Exclude the whole folder
__pycache__/
```

24 License

- In the US a license file is needed
- Permissive licenses:
  - PSFL(Python Software Foundation License): contributing to Pyhton
  - MIT (X11)/ new BSD/ ISC/ Apache
- Less permissive licenses:
  - LGPL
  - GPL (GPLv2, GPLv3)

25 Package

- guide
- Use pip (and not easy_install)

26 Documentation

26.1 The Hitchhiker's Guide To Python

- Online
- Book

- PDF
- github

27 History

Version	Date	Notes
0.1.1	2022-06-09	Shell->bash, +history, documtation
0.1.0	2020-01-18	Initial release

28 Disclaimer of Warranty

29 Limitation of Liability