10 useful tips for python coders.

stoworow

stoworow@gmail.com

PyGDA 26 Sep 2016

Outline

PEP 8 - Style Guide for Python Code

What is PEP[1] 8?
Gives coding conventions for the Python code.
Examples of requirements:

- 4 spaces instead of tabs for indentation. (easy with IDE)
- (!) Max 79 characters in line — oh common...
- x = 2 instead of x=2
- camelCase for classes names
- function_name for functions and attributes

[1] PEP - Python Enhancement Proposal

PEP 8 - Style Guide for Python Code

- _some_function - protected attributes/functions/methods
- __some_function - private attributes/functions/methods
- "self" as a first argument in methods
- "cls" as a first argument in class methods
- "if a is not b" instead of "if not a is b"
- all imports always at the beginning
- and dozens of others ...

Exceptions handling

Use try-except-else-finally

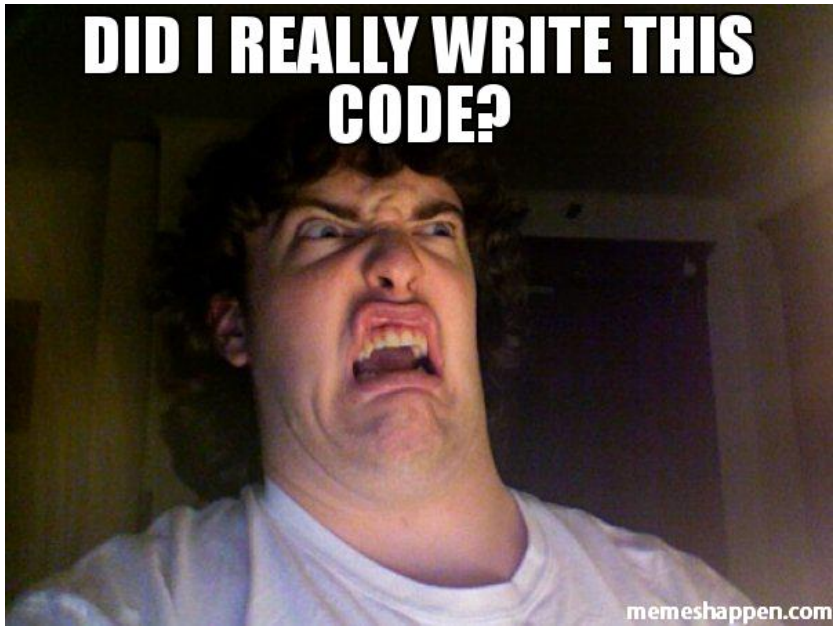
```
try:
        # do work
except IOError:
        # process IO exception
except ValueError:
        # process Value exception
else:
        # job to be done if no error was raised
finally:
        # executed always
```

Exceptions handling

What not to do:

```
try:
        ...
        100 lines of code
        ...
except:
        pass
```

- pass?
- and where actually code fails?



Consider exceptions rather than returning None

By default if function has no return statement it returns None.

Question

Did function return None because it was intended or there is a corner case (in a multi-return function)?

There is also potential issue with processing output of function returned None.

Consider exceptions rather than returning None

Example

```
def divide(a, b):
        try:
                return a/b
        except ZeroDivisionsError:
                return None
```

Watch out on processing the output:

Example

```
result = divide(x, y)
if not result:
        print('Wrong input')
```

"Wrong input" will be printed when $y = 0$ but also when $x = 0$.

Consider exceptions rather than returning None

- I prefer to use construction "If not..." only for boolean values.
- In other cases it is better to explicitly write "If result is None" (explicit is better than implicit).
- It is worth considering raising proper error

Try...except

```
def divide(a, b):
        try:
                return a/b
        except ZeroDivisionsError:
                raise ValueError('Dividing by zero')
```

Consider exceptions rather than returning None

Explicit arguments validation is useful.

Try...except

```
def divide(a, b):
        if b == 0:
                raise ValueError('Dividing by zero')
        return a/b
```

Use super method always

super() - "securely" calls superclass methods

"Old" way of calling super method

```
ThisClassName.__init__(*args, **kwargs)
```

Python 2.x

```
super(ThisClassName, self).__init__(*args, **kwargs)
```

Python 3.x

```
super().__init__(*args, **kwargs)
```

Use super method always

Issues with the "old" way:

- Problem with initialization order.
- Problems with diamond inheritance.

Use super method always

```
class BaseClass(object):
        def __init__(self):
                self.value = 2
class MyClass(BaseClass):
        def __init__(self):
                super(MyClass, self).__init__()
                self.value += 5
class MyClass2(MyClass):
        def __init__(self):
                self.value *= 2
```

Use super method always

```
class MyClass3(MyClass2):
        def __init__(self, val):
                super(MyClass3, self).__init__()
                #BaseClass.__init__(self, val)
                #MyClass2.__init__(self)
                #MyClass.__init__(self)
```

Do you need multilevel inheritance?

Multiple inheritance allows to create complex object oriented constructions. The more complex construction is, the more caution is needed. During design it is worth to consider whether multiple inheritance is really needed (sometimes it is e.g. GUI functionality).

Do you need multilevel inheritance?

Issues with MI:

- Potential crushes between methods implementations.
- Makes it harder to understand the code/architecture.
- Consider to use mixins - object with stateless methods which allows to add new functionality (in shortcut it can be defined as call without __init__ method).

subprocesses - how to

import subprocess

Calling commands without interacting

subprocess.call('chmod 755', shell=True)
or
subprocess.call(['chmod', '755'], shell=Fale)

shell argument - will create new shell process which will launch the command (dangerous).

subprocesses - how to

Differences between shell False/True.

- If shell is set to False, argument is expected to be executable file.
- If you wish to call a command with arguments you have to use list where zero index element is a name of the command and every other element is a argument of command.
- Setting shell to True will allow to pass string to argument directly to system defined shell and shell itself will be responsible for parsing it.

subprocesses - how to

- subprocess.call('ls -l') - this will fail - will search for executable file.
- subprocess.call(['ls', '-l']) - this will work correctly.
- subprocess.call('ls -l', shell=Ture) - correct, equal to running: */bin/sh -c 'ls -l'*
- subprocess.call(['ls', '-l'], shell=True) - will not work properly since it is equal to running command */bin/sh -c ls -l* (not as a string), which is equal to */bin/sh -c ls* (no argument passing).

subprocesses - how to

Calling commands with output interacting

```
proc = subprocess.Popen('some_command', shell=True)
std_out, std_err = proc.communicate()
for line in std_out.split():
        ...
```

Do not use proc.wait() - can deadlock if output is too big.
Hint: Python 3.x provides argument timeout for communicate()

Docstrings are helpful, really!

- Mind creating docstrings with content describing function.
- When a function is doing complex processing you can start with writing function's descriptions first - it helps to clarify what you really want to do.
- The major problem with functions descriptions is that it has to follow all changes in the code (obvious), unfortunately, it not always does...

Docstrings are helpful, really!

```
@classmethod
def setup(cls, *args, **kwargs):
    """
    Allows to modify default parameters of a progress bar.
    Arguments
    ---------
    Only key-worded arguments are accepted.
        len:        int, greater than 0
        style:      str, max length = 1
    """
```

It is good to use reStructuredText format (easy documentation generation with tools like Sphinx).

Let your function name describe functionality

- Writing short functions is a good habit (and that's not only my opinion).
- Functions should do only one certain thing.
- Functionality should be understandable just by reading function's name.

Let your function name describe functionality

Example

```
def process_data(filname):
        fildata = []
        with open(filname, 'r') as f:
                # read data
        for elmn in fildata:
                # some processing
        return fildata
```

This function does 2 things:

- Reads data from file
- Performs some processing

Let your function name describe functionality

Solutions:

- Rename function - e.g. read_data_and_process() - lame...
- Split into 2 functions - read_data() and process_data()

Proper importing

Python allows to import modules in a few different ways.

What not to do

```
from abc import *
```

This will import all names from module, except of private/protected ones. If __all__ was defined it will limit import to only those names. Potential issues in this case:

- ambiguity of namespaces (the same names can exist in different spaces)
- why to bother importing elements we don't need?

Proper importing

better

```
from abc import ABCMeta
class MyABC:
        __metaclass__ = ABCMeta
```

- Still collisions in namespaces might occur.

simplicity is the best

```
import abc
class MyABC:
        __metaclass__ = abc.ABCMeta
```

- Clarity! But readability suffers if module name is long.

Proper importing

For long module names

```
import numpy as np
np.array([1,2,3])
```

Use C implementations over Python ones

(for 2.x, there are rumors that 3.x uses automatically C extensions(unconfirmed by me)).

Prefer C implementations for Python 2.x

```
import pikle as pk
import cPikle as pk
```

Proper importing

Always import at the beginning of a file:

```
#! /bin/python2.7
import os
import sys
import custommodule
...
```

- It may happen that module performs some operation during import (someone forgot __name__ == "__main__" !) if module is imported somewhere in the code it might create false slow down.

Limit debugging with print

"printing" is a popular way to debug problems, however, it can be effective only in case of simple problems. Either you are going to print a lot or you will have to start using logs.

Limit debugging with print

- Get familiar with logging module (if you find it a little confusing search for a different one)
- You can define multiple levels of errors - critical, warnings, infos, debug
- Logs offer easier optimization - e.g you can track functions calls with particular arguments (but remember - "premature optimization [...]")
- If you have a lot of fixed print statements it is sometimes useful to redirect all stdout also to a file (e.g. Python 3.4+ offers special function in contexlib module- redirect_stdout(), in Python 2.x is it a bit tricky but still can be done in 3-4 lines :))

References

- Brett Slatkin - Effective Python
- Robert C. Martin - Clean code (examples in java but the core is the message)

Thank you