venzo

Python Style Guide

Baseline:

- Try to abide the strong recommendations in PEP 8 .
- Use Flake8 for some automated style checks.
- Use for type checking use pylance and typing .
- Use pylint for linting

Table of Contents

1. Introduction
2. Follow the Basics
3. Better Imports
4. Conditionals
5. Container Operations
6. Looping
7. Comments
8. Typing
9. Exception Handling
10. Conclusion

Introduction

The Zen of Python aims to capture the philosophy and design principles that should be focused while writing effective python code.

The Zen of Python

```
In [1]: import this
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
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.
Now is better than never.
Although never is often better than *right* now.
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.
Namespaces are one honking great idea -- let's do more of those!
```

Follow the Basics

Naming conventions

- Give better naming for variables
- Avoid using unclear or cryptic variable names **E.g**:

```
# Don't do  this:
v = 12
name = 'username'
date = date.today()


# Don't do this:
val = 12
client_name = 'username'
current_date = date.today()
```

- Prefer using `_` for these variables which are never used

E.g:

```
min, *_, max = values
iters = [3*3 for _ in range(10)]
```

File Naming

- **Be Descriptive:** Choose names that clearly indicate the purpose or content of the file. A another developer should be able to understand the file's role without opening it.
- **Use Underscores or Hyphens:** Use underscores (_) to separate words in a file name. This enhances readability and makes the name more human-friendly.
- **Avoid Special Characters:** Minimize the use of special characters in file names. Stick to alphanumeric characters, underscores, and hyphens to ensure compatibility across different systems.
- **Follow a Consistent Naming Convention:** Establish a consistent naming convention across your project. Consistency makes it easier for developers to locate files and maintain a clean project structure.
- **Avoid Generic Names:** Steer clear of generic names like "temp," "test," or "new." These names provide little information about the file's purpose and can lead to confusion.

E.g:

```
# Don't do this:
file1.py
file2.py
file3.py

# Do this:
user_operations.py
data_processing.py
utility_functions.py
```

Entities

When dealing with functions and classes, it's crucial to follow clear and consistent naming conventions, ensuring readability and maintainability of your code.

Functions

When naming functions, adhere to the following guidelines:

- **Use Verbs for Function Names:** Choose names that reflect the action or operation performed by the function. This makes it clear what the function does.
- **Follow Snake Case:** Use snake_case for function names, where words are lowercase and separated by underscores.
- **Be Descriptive:** Provide a descriptive name that conveys the purpose of the function. Avoid ambiguous or overly short names.

E.g:

```
# Don't do this:
fn()
calculate()

# Do this:
calculate_total()
validate_user_input()
```

Classes

For naming classes, consider the following recommendations:

- **Follow CamelCase:** Use CamelCase for class names, where each word starts with a capital letter. This convention enhances readability.

- **Include a Constructor Method:** If applicable, provide a meaningful constructor method (often `__init__`) for initializing class instances.
- **Avoid Acronyms or Abbreviations:** Unless widely accepted and understood, avoid using acronyms or abbreviations in class names. Opt for clarity over brevity.
- **Follow the Single Responsibility Principle (SRP):** Ensure that each class has a single responsibility. A class name should indicate its primary responsibility.

E.g:

```
# Don't do this:
c()
data()

# Do this:
UserDataProcessor()
FileReader()
```

Pythonic way

- Assign values to multiple variables in a single line, known as parallel assignment. This is especially useful when the assigned values are related.

  ```
  # Don't do this:
  x = 1
  y = 2
  z = 3

  # Do this:
  x, y, z = 1, 2, 3
  ```

- Chain assignments when multiple variables share the same value. This can be more efficient and reduces redundancy.

  ```
  # Don't do this:
  a = 10
  b = 10
  c = 10

  # Do this:
  a = b = c = 10
  ```

- pack and unpack arguments in single line

  ```
  # don't do this
  val1 , val2 , val3 = myfunc()[0] , myfunc()[1] ,myfunc()[2]


  # do this
  val , val2 , val3 = myfunc()
  ```

Better Imports

- When managing imports in your Python code, it's essential to maintain clarity and avoid cluttering the namespace. Follow these guidelines for better import statements:
- Use import statements for packages and modules only, not for individual types, classes, or functions.

E.g:

```
# Don't do this:
# Do this:
import datetime
import math
from datetime import date
from math import floor, sqrt

current_date = datetime.date.today()
square_root = math.sqrt(25)
rounded_value = math.floor(8.7)
```

- Import what exactly needed

```
n't do this:
    from module import *

# Do this:
    import module
```

- use isort for sort the imports

```
   pip install isort

  isort myfile.py
```

Conditionals

- Simplify conditions by using the truthy or falsy nature of values when appropriate.

```
 # Don't do this:
if x != 0:
    # do something

# Do this:
if x:
    # do something
```

- Use Ternary Operators effectively .

```
 # Don't do this:
if x > 0:
    y = 1
else:
    y = 0

# Do this:
y = 1 if x > 0 else 0
```

- Avoid Redundant Conditions
- Use `in` for Membership Testing

```
 # Don't do this:
if value == 'apple' or value == 'orange' or value == 'banana':
    # do something

# Do this:
if value in ['apple', 'orange', 'banana']:
    # do something
```

- Always safe exit

```
 # don't do this
if val:
    do_some()
else:
    return None

# do this instead
if val is None:
    return None
else:
    do_some()
```

- Avoid writing lengthy if else conditions, better to use switch statements .
- Aviod writing one liner condition

```
 # don't do this
if "some" in val : do_something()
```

Container Operations

Container operations in Python involve manipulating and working with iterable objects like lists, tuples, and strings. Employ these practices for effective container operations.

- **List Comprehensions:** Utilize list comprehensions for concise and efficient creation of lists.

```
 # don't do this
squares = []
for x in range(5):
    squares.append(x**2)

# do this instead
squares = [x**2 for x in range(5)]
```

- **Dictionary Comprehension:** Use dictionary comprehension to create a dictionary from an iterable.

```
 # don't do this
squares_dict = {}
for x in range(5):
    squares_dict[x] = x**2

# do this instead
squares_dict = {x: x**2 for x in range(5)}
```

Looping

- Use the `range` function to generate a sequence of numbers .
- Use `break` and `continue` effectively .
- Use the `zip` function to iterate over multiple iterables simultaneously .
- Use `Enumerate` for Index and Value based iteration if index required .

```
 data = [12,12,1,2,1212]

for index , val  in enumerate(data) :
    print(val)
```

Comments

- Use right style and comments for function , classes and methods
- Try running pydoc on your module to see how it looks
- Use `__doc__` attribute to access the docs for a object .

```

class ChildClass(Parent):
  """ this is a child class """

val = ChildClass()
print(val.__doc__)
```

- Comments should be as readable as narrative text, with proper capitalization and punctuation .

```
  def cal_avg(amount : int ) -> float
 """ calculates average value for the records """
```

Typing

- Override the methods in child when required
- You can declare aliases of complex types using `TypeAlias`
- Add typing variables if necessary

```
 val : Typeclass = SomeOtherClass().main()
```

- A common predefined type variable in the typing module is AnyStr

```
from typing import AnyStr
def check_some(x: AnyStr) -> AnyStr:
  pass
```

- Define what the file intakes and returns

```
def another_fun(a : str) -> Tuple(str , str)
    return a , a+"some_val"
```

- For any type of method add `Any`

```
from typing import Any as anytype
```

Exception Handling

- Make use of built-in exception classes first .
- Utilize error in exception like `ValueError` , `TypeError` so on .
- Never catch all exception instead leave a `else` at the end .

```
  try:
    some()
  except:
    some2()
  except:
    some2.1()
  else:
    some3()
```

- Minimize the amount of try/except block
- Introduce contextlib for shorthand syntax .

```
with contextlib.suppress(Exception):
  some()
```

Conclusion

The goal is not just to make the code work, but to make it work well and be easily understood by others.