PEP 8 - Style Guide for Python Code

PEP 8 的英文全称为 Style Guide for Python Code ，即 Python 编码风格指南，其涵盖三个大的方面，代码布局、注释文档以及命名规范。其主要包括以下内容：

- 代码的整体布局，比如缩进、行最大长度、换行与空行、导入语句的位置及组织结构、编码声明、dunder 方法位置等；
- 代码中的引号以及空格、行尾部逗号；
- 复合语句的基本结构；
- 注释编写的基本规范，主要包括块注释、行内注释和文档字符串；
- 针对变量、方法、函数、类等元素的命名规范。

代码的整体布局

代码的整体布局主要囊括了代码在整体结构上应该注意的事项，比如用来区分代码块层次的缩进应使用 4 个空格、代码中函数与类之间应使用两个空行进行分隔等。良好的遵守这些布局规范，有助于代码在整体结构上更加清晰易读，从结构上也有助于对代码逻辑的组织。

缩进与空格 / 制表符

Python 在语法上使用缩进来确定代码块的开始和结束，对于每一级缩进，应为 4 个空格，并且不要混用空格与制表符（在 Python 3 中从语法层面不允许混用空格和制表符，如果混用后会抛出异常：`TabError: inconsistent use of tabs and spaces in indentation`）。需要注意的是，缩进本身是一种语法上的限制，是强制性的，而缩进采用空格还是制表符，采用空格时空格的数量应为多少，这些在语法上是不限制的。

当需要换行时，换行后的内容应垂直对齐被包裹的元素。通常，换行时的缩进有两种建议的方式，一种是使用括号内的隐式换行形式，比如下面这样：

```
def function_name(var_one, var_two,
                  var_three, var_four):
    pass
```

另外一种是像下面这样使用悬挂缩进。需要注意的是，使用悬挂缩进时第一行不应放置元素，并且当元素与下面内容的行相同缩进层次时，可以增加缩进来区分。

```
# 第一行不应放置元素
function_name(
    1, 2,
    3, 4)

# 元素与下面的行相同缩进时，应增加缩进进行区分
def function_name(
        var_one, var_two,
        var_three, var_four):
    print(var_one, var_two, var_three, var_four)
```

当 if 等控制语句的条件部分需要换行时，可以使用括号将条件部分包裹起来，在括号内进行换行。

```
if (condition_one
        and condition_two):
    pass
```

当多行结构在结束时，其右括号可以另起一行与前一行元素的第一个字符对齐或与第一行元素第一个字符对齐。

```
one_list = [
    1, 2, 3,
    4, 5, 6
    ]
two_list = [
    1, 2, 3,
    4, 5, 6
]
```

行最大长度

代码中所有行的最大长度应限制在 79 个字符以内，文档字符串和注释的最大长度应限制在 72 个字符以内。限制最大长度的主要好处在于它使得代码更加紧凑可读，并且有利于代码审阅（code review），可以方便的并排打开两个版本的文件。

当一个代码行的长度超过限制时应进行换行，换行应优先使用括号内隐式换行，其次才是使用反斜杠。

```
# 应优先使用括号内隐式换行
if (condition_one
        and condition_two):
    pass

# 不提倡下面的方式
if condition_one \
        and condition_two:
    pass
```

但在一下场景下，不能使用隐式换行，比如下面 `with` 中有多个语句时，可使用反斜杠。

```
with open(path_one) as file_one, \
        open(path_one) as file_two:
    pass
```

运算符与换行、代码之间的空行

我们可以在二元运算符之前或之后进行换行，推荐的方式是在二元运算符之前换行，像下面这样：

```
result = (var_one
          - var_two
          + (var_three * var_four)
          - var_five)
```

函数与类之间应用两个空行隔开，类中的方法之间应用一个空行隔开，函数或方法中的不同功能的代码块可使用空行隔开。

引号、空格与逗号

单引号还是双引号?

在 Python 中，对单引号和双引号没有特殊的使用要求，它们的作用是相同的。

```
# 当一个字符串需要包含单或双引号时，应使用与最外层不同的引号，来避免进行转义（转义会降低代码的可读性）
my_name = 'my name is "python"'

# 不提倡下面的方式
my_name = 'my name is \'python\''
```

表达式和语句中的空格

在多数场景中不应使用无关的空格

在右括号前、左括号后、逗号、分号、冒号之前都不应使用无关空格：

```
function_name('test', [1, 2, 3])

# 不提倡下面的方式
function_name( 'test' , [1, 2, 3] )
```

在切片操作中，冒号两边的空格数量应相同；在扩展切片（Extended Slices，是指在切片中使用可选的第三个参数 `step`，可称之为步长，默认为 1）中，两个冒号应使用相同的空格数量；当切片中的某个参数被省略时，对应位置不应使用空格。在下面的例子中，大家可能会发现某些 PEP 8 提倡的用法，在 PyCharm 中会被提示 “whitespace before ‘:’ ”，这是由于 PyCharm 默认检查工具 `pycodestyle.py`（`pep8.py`）的问题导致的，详情可以参考该 issue [False positive "E203 whitespace before ‘:’ " on list slice](#) 。

```
my_list[1:6], my_list[var_one + var_two : var_three + var_four]
my_list[1:6:2], my_list[: func_one(var_one) : func_two(var_two)]

# 不提倡下面的方式
my_list[1 :6], my_list[var_one + var_two:var_three + var_four]
my_list[1:6 :2], my_list[ : func_one(var_one) : func_two(var_two)]
```

对于扩展切片的示例如下：

```
# 下面的代码在交互环境中完成（可在 PyCharm 中左下角打开 Python Console 或者使用 IPython 等工具）
>>> my_list = [1, 2, 3, 4, 5, 6, 7, 8]
>>> my_list[1:6]
[2, 3, 4, 5, 6]
# 扩展切片操作，步长值为2，其作用是按照步长值为间隔选取元素，其结果是选取出了具有偶数索引的元素
>>> my_list[1:6:2]
[2, 4, 6]
```

在函数名称与参数列表之间不应使用无关空格；在索引或切片中，括号和字典 / 列表等对象之间不应使用无关空格。

```
function_name(var_name)
my_list[1]
my_dict['key']

# 不提倡下面的方式
function_name (var_name)
my_list [1]
my_dict ['key']
```

在赋值语句中，不应为了对齐赋值语句而使用无关空格。在表达式或语句的尾部不应使用无关空格，尾部的空格会在某些场景下造成问题，比如在使用反斜杠进行换行时，在反斜杠后添加空格，会引发 `SyntaxError: unexpected character after line continuation character` 的异常。

```
var_one = 1
long_var_two = 2

# 不提倡下面的方式
var_one      = 1
long_var_two = 2
```

在函数中，对于形参列表中的默认参数以及函数调用时的关键字参数，其 `=` 两侧不应使用无关空格。

```
def function_one(var_one=True):
    return var_one


function_one(var_one=False)
```

在运算符两侧合理使用空格

在赋值、增强型赋值、比较、布尔等二元运算符两侧应使用空格。

```
var_one = 1
var_two = var_one + 1
var_one += 1
```

在同时使用不同优先级的运算符时，可在最低优先级的运算符两侧使用空格。对于一些比较复杂的场景下，需要我们来自己定夺空格的使用。

```
var_two = var_one*3 + 1
var_five = var_one*var_two + var_three*var_four

# 注意下面的情况，PEP 8 把这种情况交由我们自己定夺，但仍给出了类似下面的示例供我们参考
var_five = (var_one+var_two) * (var_three+var_four)

# 不提倡下面的方式
var_five = (var_one + var_two) * (var_three + var_four)
```

在函数注解中合理使用空格

在函数注解中，冒号前不应使用无关空格，`->` 两侧应使用空格。

```
def function_one(var_one: bool) -> int:
    if var_one:
        ret = 1
    else:
        ret = -1
    return ret
```

当函数注解和默认参数同时存在于同一参数时，在 `=` 两侧应使用空格。

```
def function_one(var_one: bool = True, var_two=False) -> int:
    pass
```

函数注解（Function annotations）是 Python 3 提供的一种可选的语法，可以对函数的参数和返回值进行注解。在该特性在 3.0 中被加入后，官方并未给出其明确的语义，随着社区中相关经验的积累以及官方相关新特性的释放，函数注解目前主要用于类型提示，常结合 `typing` 模块使用，更多细节我们会在后面的小节中讲解。

行尾部的逗号

当你想要获得包含一个元素的元组时，应在表达式尾部添加逗号，并最好将其包裹在圆括号内。

```
var_one = (1,)

# 最好不使用下面的方式
var_one = 1,
```

对于某些包含参数并需要不断扩展的序列，在添加每个参数时，在其尾部也可添加逗号，这样可以规避一些难以发现的问题（请不要小看下方代码所示的问题，当它隐藏在某一段代码中时，你就会感到十分头疼）。

```
# 当在新行中添加参数时，在尾部添加逗号有利于后续扩展，比如可避免漏写上一行结尾处逗号的问题
BACKENDS = (
    'BackendOne',
    'BackendTwo',
    )
# 在扩展时若漏写第二行结尾处的逗号，IDE 中也不会有任何提醒，但会导致无法正常使用该数据
BACKENDS = (
    'BackendOne',
    'BackendTwo'
    'BackThree'
    )
```

复合语句

复合语句是指包含其他语句的语句，比如函数和类的定义、`if`、`while`、`for`、`try`、`with` 等语句，不应将整个复合语句或复合语句中的一部分放置在一行（虽然有时我们仍可在一些场合会看到这种情况）。

```
if var_one == 1:
    pass

# 不提倡下面的方式
if var_one == 1: pass
```

到这里，我们对 Python 编码规范中关于代码的主要结构部分的讲解已经基本完成了，在下半部分中，我们会重点对 Python 中的注释和命名规范进行了解。

注释编写

注释对于代码的阅读、扩展以及维护都非常重要。在 Python 中常用的注释方式有三种，分别为块注释（Block Comments）、行内注释（Inline Comments）、文档字符串（Documentation Strings）。首先我们需要了解的是在注释编写过程中的一些通用原则，这些原则不管针对哪种注释方式，都是值得参考的。

- 应注重注释的可读性和完整性，这样有助于代码后续的扩展和维护；
- 应注重注释的实时性，即代码在修改或扩展过程中，及时更新对应的注释；
- 应优先使用英文编写注释。

以上三点从概括的角度阐述了编写注释时的一些主要思想。而在实际的编写场景中，还有一些通用的建议可以参考，比如：

- 注释应为完整的句子；
- 当注释的开头不为以小写字母开始的标识符时，应大写首字母；
- 在包含多个句子的注释中，在非结尾句的结尾处可以使用两个空格等。

命名规范

命名规范在代码的设计中是一个相当具有扩展性的话题，我们在这里会了解到其中主要的命名规则和 Python 中针对不同元素的常用命名方式。在这些基本的准则基础上，后续我们仍需要对一些高质量开源代码进行阅读，因为编写这些代码的业内大拿们往往在命名背后都有一套能够支撑其选择该命名的理论，在这个过程中我们会不断积累经验，逐步的理解在代码中如何进行一个好的命名。

常见的命名规则

- **匈牙利命名法**：以一至多个小写字母表示其属性、类型，后接首字母大写的一至多个单词表示其作用描述。比如 `m_bCanRun`，其中 `m_` 表示其为成员变量，`b` 表示其类型为布尔值，`CanRun` 表示其代表是否可以检查的含义。
- **驼峰命名法**：以一至多个逻辑单元构成，每个逻辑单元可称为一个识别字，首个识别字首字母小写，其余识别字首字母大写，比如 `canRun`，由于首字母小写，这种形式也被称为小驼峰命名法。
- **帕斯卡命名法**：和驼峰命名法类似，区别为首字母大写，如 `CanRun`，也被称为大驼峰命名法。
- **下划线命名法**：和驼峰命名法类似，区别为所有识别字均小写，识别字之间使用 `_` 连接，比如 `can_run`。

除此之外，还有其他形式的命名规则，比如大写 `CANRUN`、大写结合下划线 `CAN_RUN`、小写 `canrun` 等。另外在 Python 中也有前缀或者后缀（ leading or tailing ）下划线、双下划线等命名方式，在 Linux 代码中也存在对于一组相关方法使用相同的前缀的形式命名形式，比如 `c_run`、`c_stop` 等。

Python 常用的命名方式

通用准则：

- 无论你编写的模块、类、还是函数，其中对外暴露出的公共使用部分的命名应从使用场景出发。比如 `os` 模块中 `makedirs`、`removedirs` 等函数的命名，我们可以清楚的知道这些函数的使用场景，而不用关心其内部实现；
- 尽量使用完整的、准确的、易于理解的、具有明确目的的单词来命名，或者使用下划线将可以表达完整含义的单词（或缩写，但缩写往往会造成不明确）进行连接（哪怕这样会很长，Explicit is better than implicit ）；
- 避免使用容易混淆的字符，比如 l（ L 的小写）、O（ o 的大写）等；
- 避免采用内置名称、关键字和已经使用过的名称，避免采用过于通用的名称，前者会造成原有功能的屏蔽，后者会造成含义过于广泛而失去明确性；
- 与此同时，还有观点表示应避免使用 `tools`、`utils`、`common` 等名称，以及避免使用以 `object`、`manager` / `management`、`handler` 等作为后缀的名称，这些观点中将其称之为反模式，认为这些名称没有起到实际的意义，并且类似 `utils` 的命名反而最终会成为劣质代码的聚集地，并且将这种名称认为是缺乏设计的名称，我们在这里将这些观点当作一种扩展阅读即可；

对于变量（和常量）：

- 变量名可小写，也可使用下划线命名；
- 类型变量名（常用于类型提示）应使用大驼峰命名（关于类型提示，后面的课程中会详细讲解）；
- 常量应使用全大写字母，必要时使用下划线分隔，并且要注意在 Python 中并没有类似其他语言中 `const` 的概念，常量仅仅是一种约定（常量通常放置在代码顶部、或单独的模块、或特定的配置文件中）。对于一组常量，不应使用同一个前缀（这也适用于在同一个类中的方法或者属性），因为这样会和模块名称造成冗余，若多组常量，则针对每一组常量可以使用同一前缀。
- 对于容器类变量，常采用复数名词形式；对于映射类变量，常采用 `key_value` 的形式，其中 `key`、`value` 为键和值的实际含义。
- 对于表示布尔值的变量或者常量（但不仅限于这两者），常采用 `has`、`is` 作为前缀。

对于类（和异常）：

- 类名应使用大驼峰命名（这里不包括一些 Python 内置的类，比如 `int`、`list` ），类和属性常采用名词，方法多采用动词或者包含动词。
- 基类常采用 `Base` 作为前缀，抽象类常采用 `Abstract` 作为前缀。
- 异常名应使用大驼峰命名法，当此异常表示一个错误时，应添加 `Error` 后缀（并不是所有异常都代表代码运行错误，比如 `KeyboardInterrupt`、`SystemExit` ）。

对于函数、方法：

- 函数名、方法名应使用小写，也可使用下划线命名（但我们仍会在一些代码中看到一些方法或者函数名称采用了驼峰命名法，甚至在 Python 的标准库中也存在这样的现象，比如在 `threading` 模块中，因为这些代码的出现往往早于 PEP 8 规范的诞生，同时为了向后兼容而保留，但通常这些模块都会提供相同功能的以小写加下划线命名的方法，我们应该尽可能使用这些新的方法）。
- 实例方法的首个参数名应为 `self`，类方法的首个参数应为 `cls`（`class` 作为关键字不能被使用，常被 `cls` 或 `klass` 替换）；

对于模块和包：

- 模块（modules）名应尽可能使用小写，在必要时可以使用下划线命名（除了 `__init__` 模块以外），当使用 C/C++ 编写扩展模块时，通常需要添加下划线前缀；
- 包（packages）名应使用小写，最好不要使用下划线；

特殊格式：

- 单下划线前缀，比如 `_name`，常称为 "伪私有属性"（也可用于私有的方法、类等），这种命名方式在 Python 中是一种表示私有属性的约定（同时注意 `from ... import *` 时不会导入这种形式的对象，并且这种伪私有变量通常会通过特定的方法来获取或者赋值），私有属性通常没有直接对外的功能，常用于记录内部状态或用于提供一些公共功能的方法内使用；
- 单下划线后缀，比如 `name_`，常用于避免和 Python 的关键字发生冲突；
- 双下划线前缀，比如 `__name`，常用于基类避免和子类中的命名冲突，Python 会通过转换规则将其转换为类似 `_class__name` 的形式，这种方式通常称为命名修饰 name mangling 或 name decoration（这种方式通常用于多继承，应避免将这种形式用在私有属性的命名上）；

  ```
  >>> class MyClass:
  ...     __name = 1
  ...
  >>> MyClass.__name  # 无法直接访问
  Traceback (most recent call last):
    File "<input>", line 1, in <module>
  AttributeError: type object 'MyClass' has no attribute '__name'
  >>> MyClass._MyClass__name
  1
  ```

- 前后双下划线，比如 `__name__`，这是我们之前提到过的 Python 内部常用的 dunder 名称，不应自定义这类命名。