5. Intro to Python style, PEP 8

1 Statements

Statements are the basic units of instruction that the Python interpreter parses and processes. In general, the interpreter executes statements sequentially, one after the next as it encounters them.

Python programs are typically organized with **one statement per line**. In other words, each statement occupies a single line, with the end of the statement delimited by the newline character that marks the end of the line.

2 Too long statements

Excessively long lines of code are considered poor practice.

In fact, there is an official **"The Style Guide for Python Code" aka PEP 8** by the Python Software Foundation, and one of its stipulations is that the maximum line length in Python code should be 79 characters.

But, there might be too long statement needed:

```
person1_age = 42
person2_age = 16
person3_age = 71

someone_is_of_working_age = (person1_age >= 18 and person1_age <= 65) or (person2_age >= 18 and per
```

Or lengthy nested lists:

```
a = [[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15], [16, 17, 18, 19, 20], [21, 22, 23, 24
```

In Python a statement can be continued from one line to the next in **2 ways**:

1. Implicit line continuation by **( )**, **[ ]**, **{ }**
This technique is **preferred** according to PEP 8.

```
a = [
    [1, 2, 3, 4, 5],
    [6, 7, 8, 9, 10],
    [11, 12, 13, 14, 15],
    [16, 17, 18, 19, 20],
    [21, 22, 23, 24, 25]
    ]
```

Long expressions can also be continued across multiple lines by **wrapping it in grouping parentheses**. PEP 8 explicitly advocates this way when appropriate:

```
someone_is_of_working_age = (
    (person1_age >= 18 and person1_age <= 65)
    or (person2_age >= 18 and person2_age <= 65)
    or (person3_age >= 18 and person3_age <= 65)
    )
```

2. Explicit line continuation by **backslash**.
To indicate explicit line continuation, use a backslash as the final character on the line (**be careful with whitespaces after it**!).

```
x = 1 + 2 \
    + 3 + 4 \
    + 5 + 6
```

3 Multiple statements per line

If multiple statements may occur on one line, they are separated by a semicolon **;** .
Stylistically, this is generally frowned upon, and **PEP 8 discourages it**.

```
x = 1; y = 2; z = 3
print(x); print(y); print(z)
```

4 Comments

4.1 Single-line comments

Good code explains how, good comments explain why.

Hash symbol # signifies a comment. The interpreter will ignore everything from the hash character through the end of that line:

```
a = ['foo', 'bar', 'baz']  # I am a comment
```

– Comments can be included **within implicit line continuation**:

```
x = (1 + 2   # I am a comment.
     + 3 + 4  # Me too.
     + 5 + 6)
```

4.2 Multi-line comments

Python **doesn't explicitly provide anything** for creating multiline block comments. To create a block comment, you would usually just begin each line with a hash character.
Useful hotkeys:
`Cmd+/` – select and comment out multiple lines.
`Ctrl+/` – select and comment out multiple lines.

However, for code in a script file, there is technically an alternative.

When the interpreter parses code in a script file, it ignores a string literal if it appears as statement by itself. Thus, a **string literal** on a line by itself can serve as a comment. **Since a triple-quoted ' ' ' string can span multiple lines, it can function as a multiline comment**:

```
"""Initialize value for radius of circle.

Then calculate the area of the circle
and display the result to the console.
"""

some code
```

Although this works, **PEP 8 actually recommends against it**. Because of a special Python construct called the **docstrings**.

4.3 How to write comments

(1) One extremely useful way is as an outline for your code i.e. tracking the high-level flow of your program. E.g. **outline a function in pseudo-code**:

```
def get_top_cities(prices):
    top_cities = defaultdict(int)

    # For each price range
    # Get city searches in that price
    # Count num times city was searched
    # Take top 3 cities and add to dict

    return dict(top_cities)
```

(2) Use comments to **define tricky parts of your code**. If you put a project down and come back to it months or years later, you'll spend a lot of time trying to get reacquainted with what you wrote.

(3) Inline comments should be used sparingly **to clear up bits of code that aren't obvious on their own**. Comments should be **DRY** – **'Don't Repeat Yourself'**.

(4) If you have a complicated method or function whose name isn't easily understandable, you may want to include a short comment after the def line to shed some light:

```
def complicated_function(s):
    # This function does something complicated
```

For any public functions, you'll want to **include an associated docstring**. This string will become the .__doc__ attribute of your function and will officially be associated with that specific method.
The **PEP 257 docstring guidelines** will help you to structure your docstring.

```
def sparsity_ratio(x: np.array) -> float:
    """Return a float

       Percentage of values in array that are zero or NaN
    """
```

(5) Obvious naming convention.

Bad example:

```
# A dictionary of families who live in each city
mydict = {"Midtown": ["Powell", "Brantley", "Young"],
              "Norcross": ["Montgomery"],
              "Ackworth": []}

def a(dict):
    # For each city
    for p in dict:
        # If there are no families in the city
            if not mydict[p]:
                # Say that there are no families
                print("None.")
```

Much better:

```
families_by_city = {"Midtown": ["Powell", "Brantley", "Young"],
                        "Norcross": ["Montgomery"],
                        "Ackworth": []}

def no_families(cities):
    for city in cities:
        if not families_by_city[city]:
            print(f"No families in {city}.")
```

(6) Your comments should rarely be longer than the code they support.
If you're spending too much time explaining what you did, then you need to go back and refactor to make your code more clear and concise.

5 Whitespaces

When parsing code, the Python interpreter breaks the input up into **tokens**. Informally, tokens are the elementary language elements: *identifiers*, *keywords*, *literals*, and *operators*. The most common whitespace characters are *space*, *tab*, and *newline*.

Whitespace is mostly ignored, and mostly not required, by the Python interpreter. When it is clear where one token ends and the next one starts, whitespace can be omitted.

PEP 8: Whitespace and other recommendations

Avoid extraneous whitespace in the following situations:

(1) Immediately **inside parentheses, brackets or braces**:

Yes:

```
spam(ham[1], {eggs: 2})
```

No:

```
spam( ham[ 1 ], { eggs: 2 } )
```

(2) Between a **trailing comma** and a following close parenthesis:

Yes:

```
foo = (0,)
```

No:

```
bar = (0, )
```

(3) In a slice the colon acts like a binary operator, and should have equal amounts on either side:

```
ham[1:9], ham[1:9:3], ham[:9:3], ham[1::3], ham[1:9:]
ham[lower:upper], ham[lower:upper:], ham[lower::step]
ham[lower+offset : upper+offset]
```

(4) More than one space around an assignment (or other) operator to align it with another:

Yes:

```
x = 1
y = 2
long_variable = 3
```

No:

```
x             = 1
y             = 2
long_variable = 3
```

(5) Avoid trailing whitespace anywhere. Because it's usually invisible, it can be confusing: e.g. a backslash followed by a space and a newline does not count as a line continuation marker.

(6) Always surround these binary operators with a single space on either side: assignment (=), augmented assignment (+=, −=, etc.), comparisons (==, **!=**, <, >, <=, >=, **in**, **not in**, **is**, **is not**), Booleans (**and**, **or**, **not**).

(7) If **operators with different priorities** are used, consider **adding whitespace around the operators with the lowest priority(ies)**. However, **never use more than one space**.

Yes:

```
i = i + 1
submitted += 1
x = x*2 - 1
hypot2 = x*x + y*y
c = (a+b) * (a-b)
```

No:

```
i=i+1
submitted +=1
x = x * 2 - 1
hypot2 = x * x + y * y
c = (a + b) * (a - b)
```

(8) Don't use spaces around the = sign when used to indicate a keyword argument, or when used to indicate a **default value** for an unannotated function parameter:

```
def complex(real, imag=0.0):
    return magic(r=real, i=imag)
```

But combining an **argument annotation** with a **default value DO use spaces** around = sign:

```
def munge(input: AnyStr, sep: AnyStr = None, limit=1000): ...
```

(9) Imports should usually **be on separate lines**:

Yes:

```
import os
import sys
```

No:

```
import os, sys
```

(10) `return` statements. Either all return statements in a function should return an expression, or none of them should. If any return statement returns an expression, any return statements where no value is returned should **explicitly state this as `return None`**.

Yes:

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)
    else:
        return None

def bar(x):
    if x < 0:
        return None
    return math.sqrt(x)
```

No:

```
def foo(x):
    if x >= 0:
        return math.sqrt(x)

def bar(x):
    if x < 0:
        return
    return math.sqrt(x)
```

(11) Use `.startswith()` and `.endswith()` **instead of string slicing to check for prefixes or suffixes**. Since `startswith()` and `endswith()` are cleaner and less error prone:

Yes:

```
if foo.startswith('bar'):
```

No:

```
if foo[:3] == 'bar':
```

(12) Do not compare booleans with `True` or `False` using ==.

Yes:

```
if greeting:
```

No:

```
if greeting == True: # bad
if greeting is True: # WORSE (?)
```

6 Indentation

In Python, indentation is not ignored. Leading whitespace is used to compute a line's indentation level, which in turn is used to determine grouping of statements.

1. Use **4 spaces per indentation level**.

2. Continuation lines should align wrapped elements either vertically using Python's implicit line joining or using a hanging indent. When using a **hanging indent** the following should be considered; there should be no arguments on the first line and further indentation should be used to clearly distinguish itself as a continuation line.

Yes:

```
# Aligned with opening delimiter.
foo = long_function_name(var_one, var_two,
                         var_three, var_four)

# Add 4 spaces (an extra level of indentation) to distinguish arguments from the rest.
def long_function_name(
        var_one, var_two, var_three,
        var_four):
    print(var_one)

# Hanging indents should add a level.
foo = long_function_name(
    var_one, var_two,
    var_three, var_four)
```

No:

```
# Arguments on first line forbidden when not using vertical alignment.
foo = long_function_name(var_one, var_two,
    var_three, var_four)

# Further indentation required as indentation is not distinguishable.
def long_function_name(
    var_one, var_two, var_three,
    var_four):
    print(var_one)
```