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

```
foo = long_function_name(var_one, var_two,
                         var_three, var_four)
```

2. **The closing brace/bracket/parenthesis on multiline constructs may either line up under the first non-whitespace character of the last line of list, as in:**

```
my_list = [
    1, 2, 3,
    4, 5, 6,
    ]
result = some_function_that_takes_arguments(
    'a', 'b', 'c',
    'd', 'e', 'f',
    )
```

3. **Limit all lines to a maximum of 79 characters.**

4. **Backslashes may still be appropriate at times. For example, long, multiple with-statements could not use implicit continuation before Python 3.10, so backslashes were acceptable for that case:**

```
  with open('/path/to/some/file/you/want/to/read') as file_1, \
    open('/path/to/some/file/being/written', 'w') as file_2:
    file_2.write(file_1.read())
```

5. **Should a Line Break Before or After a Binary Operator**

```
income = (gross_wages
          + taxable_interest
          + (dividends - qualified_dividends)
          - ira_deduction
          - student_loan_interest)
```

6. **Code in the core Python distribution should always use UTF-8, and should not have an encoding declaration.**

7. **Imports should usually be on separate lines:**

```
# Correct:
import os
import sys

# Wrong:
import sys, os
```

It's okay to say this though**:**

```
# Correct:
from subprocess import Popen, PIPE
```

8. **Wildcard imports (from <module> import *) should be avoided, as they make it unclear which names are present in the namespace, confusing both readers and many automated tools**.

```
from myclass import MyClass
from foo.bar.yourclass import YourClass
```

9. **Avoid extraneous whitespace in the following situations:**

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

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

10. **Between a trailing comma and a following close parenthesis**

```
# Correct:
foo = (0,)

# Wrong:
bar = (0, )
```

11. **Immediately before a comma, semicolon, or colon:**

```
# Correct:
if x == 4: print(x, y); x, y = y, x

# Wrong:
if x == 4 : print(x , y) ; x , y = y , x
```

12.In an extended slice, both colons must have the same amount of spacing applied. Exception: when a slice parameter is omitted, the space is omitted:

```
# Correct:
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]
ham[: upper_fn(x) : step_fn(x)], ham[:: step_fn(x)]
ham[lower + offset : upper + offset]

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

13.Immediately before the open parenthesis that starts the argument list of a function call

```
# Correct:
spam(1)

# Wrong:
spam (1)
```

14. Immediately before the open parenthesis that starts an indexing or slicing

```
# Correct:
dct['key'] = lst[index]

# Wrong:
dct ['key'] = lst [index]
```

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

```
# Correct:
x = 1
y = 2
long_variable = 3

# Wrong:
x             = 1
y             = 2
long_variable = 3
```

16. Avoid trailing whitespace anywhere.

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

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

17. Function annotations should use the normal rules for colons and always have spaces around the -> arrow if present.

```
# Correct:
def munge(input: AnyStr): ...
def munge() -> PosInt: ...

# Wrong:
def munge(input:AnyStr): ...
def munge()->PosInt: ...
```

18. 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:

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

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

19.When combining an argument annotation with a default value, however, do use spaces around the = sign:

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

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

20. Compound statements (multiple statements on the same line) are generally discouraged

```
# Correct:
if foo == 'blah':
    do_blah_thing()
do_one()
do_two()
do_three()
```

Rather not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
do_one(); do_two(); do_three()
```

21. While sometimes it's okay to put an if/for/while with a small body on the same line, never do this for multi-clause statements. Also avoid folding such long lines!

Rather not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
for x in lst: total += x
while t < 10: t = delay()
```

Definitely not:

```
# Wrong:
if foo == 'blah': do_blah_thing()
else: do_non_blah_thing()

try: something()
finally: cleanup()

do_one(); do_two(); do_three(long, argument,
                             list, like, this)

if foo == 'blah': one(); two(); three()
```

22. Trailing commas are usually optional, except they are mandatory when making a tuple of one element. For clarity, it is recommended to surround the latter in (technically redundant) parentheses:

```
# Correct:
FILES = ('setup.cfg',)
# Wrong:
FILES = 'setup.cfg',
```

```
# Correct:
FILES = [
    'setup.cfg',
    'tox.ini',
    ]
initialize(FILES,
           error=True,
           )
# Wrong:
FILES = ['setup.cfg', 'tox.ini',]
initialize(FILES, error=True,)
```

23. Inline comments should be separated by at least two spaces from the statement

```
x = x + 1                 # Increment x
```

But sometimes, this is useful:

```
x = x + 1                 # Compensate for border
```

23. Documentation Strings

Conventions for writing good documentation strings (a.k.a. “docstrings”) are immortalized in PEP 257.

```
"""Return a foobang

Optional plotz says to frobnicate the bizbaz first.
"""
```

- For one liner docstrings, please keep the closing """ on the same line:

```
"""Return an ex-parrot."""
```

24. There are a lot of different naming styles. It helps to be able to recognize what naming style is being used, independently from what they are used for. The following naming styles are commonly distinguished:

- b (single lowercase letter)
- B (single uppercase letter)
- lowercase
- lower_case_with_underscores
- UPPERCASE
- UPPER_CASE_WITH_UNDERSCORES
- CapitalizedWords (or CapWords, or CamelCase – so named because of the bumpy look of its letters [4]). This is also sometimes known as StudlyCaps.

25. Never use the characters ‘l’ (lowercase letter el), ‘O’ (uppercase letter oh), or ‘I’ (uppercase letter eye) as single character variable names. In some fonts, these characters are indistinguishable from the numerals one and zero. When tempted to use ‘l’, use ‘L’ instead.

26. Modules should have short, all-lowercase names. Underscores can be used in the module name if it improves readability. Python packages should also have short, all-lowercase names, although the use of underscores is discouraged.

27. Class names should normally use the CapWords convention.

28. Always use self for the first argument to instance methods. Always use cls for the first argument to class methods. If a function argument’s name clashes with a reserved keyword, it is generally better to append a single trailing underscore rather than use an abbreviation or spelling corruption. Thus class_ is better than clss. (Perhaps better is to avoid such clashes by using a synonym.)

29. Use the function naming rules: lowercase with words separated by underscores as necessary to improve readability. Use one leading underscore only for non-public methods and instance variables. To avoid name clashes with subclasses, use two leading underscores to invoke Python’s name mangling rules.

30. Constants are usually defined on a module level and written in all capital letters with underscores separating words. Examples include MAX_OVERFLOW and TOTAL.

31. Always decide whether a class’s methods and instance variables (collectively: “attributes”) should be public or non-public. If in doubt, choose non-public; it’s easier to make it public later than to make a public attribute non-public.

32. Always use a def statement instead of an assignment statement that binds a lambda expression directly to an identifier:

```
# Correct:
def f(x): return 2*x
# Wrong:
f = lambda x: 2*x
```

32. Derive exceptions from Exception rather than BaseException. Direct inheritance from BaseException is reserved for exceptions where catching them is almost always the wrong thing to do.

33. When catching exceptions, mention specific exceptions whenever possible instead of using a bare except: clause:

```
try:
    import platform_specific_module
except ImportError:
    platform_specific_module = None
```

A bare except: clause will catch SystemExit and KeyboardInterrupt exceptions, making it harder to interrupt a program with Control-C, and can disguise other problems. If you want to catch all exceptions that signal program errors, use except Exception: (bare except is equivalent to except BaseException:).

```
# Correct:
try:
    value = collection[key]
except KeyError:
    return key_not_found(key)
else:
    return handle_value(value)
# Wrong:
try:
    # Too broad!
    return handle_value(collection[key])
except KeyError:
    # Will also catch KeyError raised by handle_value()
    return key_not_found(key)
```

34. When a resource is local to a particular section of code, use a with statement to ensure it is cleaned up promptly and reliably after use. A try/finally statement is also acceptable.

```
# Correct:
with conn.begin_transaction():
    do_stuff_in_transaction(conn)
# Wrong:
with conn:
    do_stuff_in_transaction(conn)
```

35. Be consistent in return statements. Either all return statements in a function should return an expression, or none of them should.

```
# Correct:

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)
# Wrong:

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

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

36. Use ''.startswith() and ''.endswith() instead of string slicing to check for prefixes or suffixes. startswith() and endswith() are cleaner and less error prone:

```
# Correct:
if foo.startswith('bar'):
# Wrong:
if foo[:3] == 'bar':
```

37. Object type comparisons should always use isinstance() instead of comparing types directly:

```
# Correct:
if isinstance(obj, int):
# Wrong:
if type(obj) is type(1):
```

38. For sequences, (strings, lists, tuples), use the fact that empty sequences are false

```
# Correct:
if not seq:
if seq:
# Wrong:
if len(seq):
if not len(seq):
```

39. Don’t write string literals that rely on significant trailing whitespace. Such trailing whitespace is visually indistinguishable and some editors (or more recently, reindent.py) will trim them. Don’t compare boolean values to True or False using ==

```
# Correct:
if greeting:
# Wrong:
if greeting == True:
```

Worse:

```
# Wrong:
if greeting is True:
```

40. Use of the flow control statements return/break/continue within the finally suite of a try...finally, where the flow control statement would jump outside the finally suite, is discouraged.

```
# Wrong:
def foo():
    try:
        1 / 0
    finally:
        return 42
```

41. If an assignment has a right hand side, then the equality sign should have exactly one space on both sides:

```
# Correct:

code: int

class Point:
    coords: Tuple[int, int]
    label: str = '<unknown>'
# Wrong:

code:int  # No space after colon
code : int  # Space before colon

class Test:
    result: int=0  # No spaces around equality sign
```