Code Style

If you ask Python programmers what they like most about Python, they will often cite its high readability. Indeed, a high level of readability is at the heart of the design of the Python language, following the recognized fact that code is read much more often than it is written.

One reason for the high readability of Python code is its relatively complete set of Code Style guidelines and "Pythonic" idioms.

When a veteran Python developer (a Pythonista) calls portions of code not "Pythonic", they usually mean that these lines of code do not follow the common guidelines and fail to express its intent in what is considered the best (hear: most readable) way.

On some border cases, no best way has been agreed upon on how to express an intent in Python code, but these cases are rare.

General concepts

Explicit code

While any kind of black magic is possible with Python, the most explicit and straightforward manner is preferred.

Bad

```
def make_complex(*args):
    x, y = args
    return dict(**locals())
```

Good

```
def make_complex(x, y):
    return {'x': x, 'y': y}
```

In the good code above, x and y are explicitly received from the caller, and an explicit dictionary is returned. The developer using this function knows exactly what to do by reading the first and last lines, which is not the case with the bad example.

One statement per line

While some compound statements such as list comprehensions are allowed and appreciated for their brevity and their expressiveness, it is bad practice to have two disjointed statements on the same line of code.

Bad

```
print('one'); print('two')

if x == 1: print('one')

if <complex comparison> and <other complex comparison>:
    # do something
```

Good

```
print('one')
print('two')

if x == 1:
    print('one')

cond1 = <complex comparison>
cond2 = <other complex comparison>
if cond1 and cond2:
    # do something
```

Function arguments

Arguments can be passed to functions in four different ways.

1. **Positional arguments** are mandatory and have no default values. They are the simplest form of arguments and they can be used for the few function arguments that are fully part of the function's meaning and their order is natural. For instance, in `send(message, recipient)` or `point(x, y)` the user of the function has no difficulty remembering that those two functions require two arguments, and in which order.

In those two cases, it is possible to use argument names when calling the functions and, doing so, it is possible to switch the order of arguments, calling for instance `send(recipient='World', message='Hello')` and `point(y=2, x=1)` but this reduces readability and is unnecessarily verbose, compared to the more straightforward calls to `send('Hello', 'World')` and `point(1, 2)`.

2. **Keyword arguments** are not mandatory and have default values. They are often used for optional parameters sent to the function. When a function has more than two or three positional parameters, its signature is more difficult to remember and using keyword arguments with default values is helpful. For instance, a more complete `send` function could be defined as `send(message, to, cc=None, bcc=None)`. Here `cc` and `bcc` are optional, and evaluate to `None` when they are not passed another value.

Calling a function with keyword arguments can be done in multiple ways in Python; for example, it is possible to follow the order of arguments in the definition without explicitly naming the arguments, like in `send('Hello', 'World', 'Cthulhu', 'God')`, sending a blind carbon copy to God. It would also be possible to name arguments in another order, like in `send('Hello again', 'World', bcc='God', cc='Cthulhu')`. Those two possibilities are better avoided without any strong reason to not follow the syntax that is the closest to the function definition: `send('Hello', 'World', cc='Cthulhu', bcc='God')`.

As a side note, following the YAGNI principle, it is often harder to remove an optional argument (and its logic inside the function) that was added "just in case" and is seemingly never used, than to add a new optional argument and its logic when needed.

3. The **arbitrary argument list** is the third way to pass arguments to a function. If the function intention is better expressed by a signature with an extensible number of positional arguments, it can be defined with the `*args` constructs. In the function body, `args` will be a tuple of all the remaining positional arguments. For example, `send(message, *args)` can be called with each recipient as an argument: `send('Hello', 'God', 'Mom', 'Cthulhu')`, and in the function body `args` will be equal to `('God', 'Mom', 'Cthulhu')`.

However, this construct has some drawbacks and should be used with caution. If a function receives a list of arguments of the same nature, it is often more clear to define it as a function of one argument, that argument being a list or any sequence. Here, if `send` has multiple recipients, it is better to define it explicitly: `send(message, recipients)` and call it with `send('Hello', ['God', 'Mom', 'Cthulhu'])`. This way, the user of the function can manipulate the recipient list as a list beforehand, and it opens the possibility to pass any sequence, including iterators, that cannot be unpacked as other sequences.

4. The **arbitrary keyword argument dictionary** is the last way to pass arguments to functions. If the function requires an undetermined series of named arguments, it is possible to use the `**kwargs` construct. In the function body, `kwargs` will be a dictionary of all the passed named arguments that have not been caught by other keyword arguments in the function signature.

The same caution as in the case of *arbitrary argument list* is necessary, for similar reasons: these powerful techniques are to be used when there is a proven necessity to use them, and they should not be used if the simpler and clearer construct is sufficient to express the function's intention.

It is up to the programmer writing the function to determine which arguments are positional arguments and which are optional keyword arguments, and to decide whether to use the advanced techniques of arbitrary argument passing. If the advice above is followed wisely, it is possible and enjoyable to write Python functions that are:

- easy to read (the name and arguments need no explanations)
- easy to change (adding a new keyword argument does not break other parts of the code)

Avoid the magical wand

A powerful tool for hackers, Python comes with a very rich set of hooks and tools allowing you to do almost any kind of tricky tricks. For instance, it is possible to do each of the following:

- change how objects are created and instantiated
- change how the Python interpreter imports modules
- It is even possible (and recommended if needed) to embed C routines in Python.

However, all these options have many drawbacks and it is always better to use the most straightforward way to achieve your goal. The main drawback is that readability suffers greatly when using these constructs. Many code analysis tools, such as pylint or pyflakes, will be unable to parse this "magic" code.

We consider that a Python developer should know about these nearly infinite possibilities, because it instills confidence that no impassable problem will be on the way. However, knowing how and particularly when **not** to use them is very important.

Like a kung fu master, a Pythonista knows how to kill with a single finger, and never to actually do it.

We are all responsible users

As seen above, Python allows many tricks, and some of them are potentially dangerous. A good example is that any client code can override an object's properties and methods: there is no "private" keyword in Python. This philosophy, very different from highly defensive languages like Java, which give a lot of mechanisms to prevent any misuse, is expressed by the saying: "We are all responsible users".

This doesn't mean that, for example, no properties are considered private, and that no proper encapsulation is possible in Python. Rather, instead of relying on concrete walls erected by the developers between their code and others', the Python community prefers to rely on a set of conventions indicating that these elements should not be accessed directly.

The main convention for private properties and implementation details is to prefix all "internals" with an underscore. If the client code breaks this rule and accesses these marked elements, any misbehavior or problems encountered if the code is modified is the responsibility of the client code.

Using this convention generously is encouraged: any method or property that is not intended to be used by client code should be prefixed with an underscore. This will guarantee a better separation of duties and easier modification of existing code; it will always be possible to publicize a private property, but making a public property private might be a much harder operation.

Returning values

When a function grows in complexity, it is not uncommon to use multiple return statements inside the function's body. However, in order to keep a clear intent and a sustainable readability level, it is preferable to avoid returning meaningful values from many output points in the body.

There are two main cases for returning values in a function: the result of the function return when it has been processed normally, and the error cases that indicate a wrong input parameter or any other reason for the function to not be able to complete its computation or task.

If you do not wish to raise exceptions for the second case, then returning a value, such as None or False, indicating that the function could not perform correctly might be needed. In this case, it is better to return as early as the incorrect context has been detected. It will help to flatten the structure of the function: all the code after the return-because-of-error statement can assume the condition is met to further compute the function's main result. Having multiple such return statements is often necessary.

However, when a function has multiple main exit points for its normal course, it becomes difficult to debug the returned result, so it may be preferable to keep a single exit point. This will also help factoring out some code paths, and the multiple exit points are a probable indication that such a refactoring is needed.

```
def complex_function(a, b, c):
    if not a:
        return None  # Raising an exception might be better
```

```
    if not b:
        return None  # Raising an exception might be better
    # Some complex code trying to compute x from a, b and c
    # Resist temptation to return x if succeeded
    if not x:
        # Some Plan-B computation of x
    return x  # One single exit point for the returned value x will help
              # when maintaining the code.
```

Idioms

A programming idiom, put simply, is a *way* to write code. The notion of programming idioms is discussed amply at c2 and at Stack Overflow.

Idiomatic Python code is often referred to as being *Pythonic*.

Although there usually is one — and preferably only one — obvious way to do it; *the* way to write idiomatic Python code can be non-obvious to Python beginners. So, good idioms must be consciously acquired.

Some common Python idioms follow:

Unpacking

If you know the length of a list or tuple, you can assign names to its elements with unpacking. For example, since `enumerate()` will provide a tuple of two elements for each item in list:

```
for index, item in enumerate(some_list):
    # do something with index and item
```

You can use this to swap variables as well:

```
a, b = b, a
```

Nested unpacking works too:

```
a, (b, c) = 1, (2, 3)
```

In Python 3, a new method of extended unpacking was introduced by **PEP 3132**:

```
a, *rest = [1, 2, 3]
# a = 1, rest = [2, 3]
a, *middle, c = [1, 2, 3, 4]
# a = 1, middle = [2, 3], c = 4
```

Create an ignored variable

If you need to assign something (for instance, in Unpacking) but will not need that variable, use __:

```
filename = 'foobar.txt'
basename, __, ext = filename.rpartition('.')
```

> ### Note:

> Many Python style guides recommend the use of a single underscore "_" for throwaway variables rather than the double underscore "__" recommended here. The issue is that "_" is commonly used as an alias for the `gettext()` function, and is also used at the interactive prompt to hold the value of the last operation. Using a double underscore instead is just as clear and almost as convenient, and eliminates the risk of accidentally interfering with either of these other use cases.

Create a length-N list of the same thing

Use the Python list * operator:

```
four_nones = [None] * 4
```

Create a length-N list of lists

Because lists are mutable, the * operator (as above) will create a list of N references to the *same* list, which is not likely what you want. Instead, use a list comprehension:

```
four_lists = [[] for __ in range(4)]
```

Create a string from a list

A common idiom for creating strings is to use **`str.join()`** on an empty string.

```
letters = ['s', 'p', 'a', 'm']
word = ''.join(letters)
```

This will set the value of the variable *word* to 'spam'. This idiom can be applied to lists and tuples.

Searching for an item in a collection

Sometimes we need to search through a collection of things. Let's look at two options: lists and sets.

Take the following code for example:

```
s = set(['s', 'p', 'a', 'm'])
l = ['s', 'p', 'a', 'm']

def lookup_set(s):
    return 's' in s

def lookup_list(l):
    return 's' in l
```

Even though both functions look identical, because *lookup_set* is utilizing the fact that sets in Python are hashtables, the lookup performance between the two is very different. To determine whether an item is in a list, Python will have to go through each item until it finds a matching item. This is time consuming, especially for long lists. In a set, on the other hand, the hash of the item will tell Python where in the set to look for a matching item. As a result, the search can be done quickly, even if the set is large. Searching in dictionaries works the same way. For more information see this StackOverflow page. For detailed information on the amount of time various common operations take on each of these data structures, see this page.

Because of these differences in performance, it is often a good idea to use sets or dictionaries instead of lists in cases where:

- The collection will contain a large number of items
- You will be repeatedly searching for items in the collection
- You do not have duplicate items.

For small collections, or collections which you will not frequently be searching through, the additional time and memory required to set up the hashtable will often be greater than the time saved by the improved search speed.

Zen of Python

Also known as **PEP 20**, the guiding principles for Python's design.

```
>>> 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!
```

For some examples of good Python style, see these slides from a Python user group.

PEP 8

PEP 8 is the de facto code style guide for Python. A high quality, easy-to-read version of PEP 8 is also available at pep8.org.

This is highly recommended reading. The entire Python community does their best to adhere to the guidelines laid out within this document. Some project may sway from it from time to time, while others may amend its recommendations.

That being said, conforming your Python code to PEP 8 is generally a good idea and helps make code more consistent when working on projects with other developers. There is a command-line program, pycodestyle (previously known as `pep8`), that can check your code for conformance. Install it by running the following command in your terminal:

```
$ pip install pycodestyle
```

Then run it on a file or series of files to get a report of any violations.

```
$ pycodestyle optparse.py
optparse.py:69:11: E401 multiple imports on one line
optparse.py:77:1: E302 expected 2 blank lines, found 1
optparse.py:88:5: E301 expected 1 blank line, found 0
optparse.py:222:34: W602 deprecated form of raising exception
optparse.py:347:31: E211 whitespace before '('
optparse.py:357:17: E201 whitespace after '{'
optparse.py:472:29: E221 multiple spaces before operator
optparse.py:544:21: W601 .has_key() is deprecated, use 'in'
```

Auto-Formatting

There are several auto-formatting tools that can reformat your code, in order to comply with PEP 8.

autopep8

The program autopep8 can be used to automatically reformat code in the PEP 8 style. Install the program with:

```
$ pip install autopep8
```

Use it to format a file in-place with:

```
$ autopep8 --in-place optparse.py
```

Excluding the `--in-place` flag will cause the program to output the modified code directly to the console for review. The `--aggressive` flag will perform more substantial changes and can be applied multiple times for greater effect.

yapf

While autopep8 focuses on solving the PEP 8 violations, yapf tries to improve the format of your code aside from complying with PEP 8. This formatter aims at providing as good looking code as a programmer who writes PEP 8 compliant code. It gets installed with:

```
$ pip install yapf
```

Run the auto-formatting of a file with:

```
$ yapf --in-place optparse.py
```

Similar to autopep8, running the command without the `--in-place` flag will output the diff for review before applying the changes.

black

The auto-formatter black offers an opinionated and deterministic reformatting of your code base. Its main focus lies in providing a uniform code style without the need of configuration throughout its users. Hence, users of black are able to forget about formatting altogether. Also, due to the deterministic approach minimal git diffs with only the relevant changes are guaranteed. You can install the tool as follows:

```
$ pip install black
```

A python file can be formatted with:

```
$ black optparse.py
```

Adding the `--diff` flag provides the code modification for review without direct application.

Conventions

Here are some conventions you should follow to make your code easier to read.

Check if a variable equals a constant

You don't need to explicitly compare a value to True, or None, or 0 – you can just add it to the if statement. See Truth Value Testing for a list of what is considered false.

Bad:

```
if attr == True:
    print('True!')

if attr == None:
    print('attr is None!')
```

Good:

```
# Just check the value
if attr:
    print('attr is truthy!')

# or check for the opposite
if not attr:
    print('attr is falsey!')

# or, since None is considered false, explicitly check for it
if attr is None:
    print('attr is None!')
```

Access a Dictionary Element

Don't use the **`dict.has_key()`** method. Instead, use `x in d` syntax, or pass a default argument to **`dict.get()`**.

Bad:

```
d = {'hello': 'world'}
if d.has_key('hello'):
    print(d['hello'])    # prints 'world'
else:
    print('default_value')
```

Good:

```
d = {'hello': 'world'}

print(d.get('hello', 'default_value')) # prints 'world'
print(d.get('thingy', 'default_value')) # prints 'default_value'

# Or:
if 'hello' in d:
    print(d['hello'])
```

Short Ways to Manipulate Lists

List comprehensions provides a powerful, concise way to work with lists.

Generator expressions follows almost the same syntax as list comprehensions but return a generator instead of a list.

Creating a new list requires more work and uses more memory. If you are just going to loop through the new list, prefer using an iterator instead.

Bad:

```
# needlessly allocates a list of all (gpa, name) entires in memory
valedictorian = max([(student.gpa, student.name) for student in graduates])
```

Good:

```
valedictorian = max((student.gpa, student.name) for student in graduates)
```

Use list comprehensions when you really need to create a second list, for example if you need to use the result multiple times.

If your logic is too complicated for a short list comprehension or generator expression, consider using a generator function instead of returning a list.

Good:

```
def make_batches(items, batch_size):
    """
    >>> list(make_batches([1, 2, 3, 4, 5], batch_size=3))
    [[1, 2, 3], [4, 5]]
    """
    current_batch = []
    for item in items:
        current_batch.append(item)
        if len(current_batch) == batch_size:
            yield current_batch
            current_batch = []
    yield current_batch
```

Never use a list comprehension just for its side effects.

Bad:

```
[print(x) for x in sequence]
```

Good:

```
for x in sequence:
    print(x)
```

Filtering a list

Bad:

Never remove items from a list while you are iterating through it.

```
# Filter elements greater than 4
a = [3, 4, 5]
for i in a:
    if i > 4:
        a.remove(i)
```

Don't make multiple passes through the list.

```
while i in a:
    a.remove(i)
```

Good:

Use a list comprehension or generator expression.

```
# comprehensions create a new list object
filtered_values = [value for value in sequence if value != x]

# generators don't create another list
filtered_values = (value for value in sequence if value != x)
```

Possible side effects of modifying the original list

Modifying the original list can be risky if there are other variables referencing it. But you can use *slice assignment* if you really want to do that.

```
# replace the contents of the original list
sequence[::] = [value for value in sequence if value != x]
```

Modifying the values in a list

Bad:

Remember that assignment never creates a new object. If two or more variables refer to the same list, changing one of them changes them all.

```
# Add three to all list members.
a = [3, 4, 5]
b = a                     # a and b refer to the same list object

for i in range(len(a)):
    a[i] += 3             # b[i] also changes
```

Good:

It's safer to create a new list object and leave the original alone.

```
a = [3, 4, 5]
b = a

# assign the variable "a" to a new list without changing "b"
a = [i + 3 for i in a]
```

Use `enumerate()` keep a count of your place in the list.

```
a = [3, 4, 5]
for i, item in enumerate(a):
    print(i, item)
# prints
# 0 3
# 1 4
# 2 5
```

The `enumerate()` function has better readability than handling a counter manually. Moreover, it is better optimized for iterators.

Read From a File

Use the `with open` syntax to read from files. This will automatically close files for you.

Bad:

```
f = open('file.txt')
a = f.read()
print(a)
f.close()
```

Good:

```
with open('file.txt') as f:
    for line in f:
        print(line)
```

The `with` statement is better because it will ensure you always close the file, even if an exception is raised inside the `with` block.

Line Continuations

When a logical line of code is longer than the accepted limit, you need to split it over multiple physical lines. The Python interpreter will join consecutive lines if the last character of the line is a backslash. This is helpful in some cases, but should usually be avoided because of its fragility: a white space added to the end of the line, after the backslash, will break the code and may have unexpected results.

A better solution is to use parentheses around your elements. Left with an unclosed parenthesis on an end-of-line, the Python interpreter will join the next line until the parentheses are closed. The same behavior holds for curly and square braces.

Bad:

```
my_very_big_string = """For a long time I used to go to bed early. Sometimes, \
    when I had put out my candle, my eyes would close so quickly that I had not even \
    time to say “I’m going to sleep.”"""

from some.deep.module.inside.a.module import a_nice_function, another_nice_function, \
    yet_another_nice_function
```

Good:

```
my_very_big_string = (
    "For a long time I used to go to bed early. Sometimes, "
```

```
        "when I had put out my candle, my eyes would close so quickly "
        "that I had not even time to say “I’m going to sleep.”"
    )

    from some.deep.module.inside.a.module import (
        a_nice_function, another_nice_function, yet_another_nice_function)
```

However, more often than not, having to split a long logical line is a sign that you are trying to do too many things at the same time, which may hinder readability.