Thankfulness for PEPs

PEP 8 - Style Guide for Python Code
PEP 255 - Simple Generators
and more

PEP 8 - Style Guide for Python

Readability Counts

“One of Guido's key insights is that code is read much more often than it is written.”

Consistency is important

A style guide is about consistency. Consistency with this style guide is important. Consistency within a project is more important. Consistency within one module or function is the most important.

However, know when to be inconsistent -- sometimes style guide recommendations just aren't applicable. When in doubt, use your best judgment.

And don't hesitate to ask!

As a example, single-quoted strings and double-quoted strings are the same. Pick one and stick to it.

Code Layout

Hanging indent

Line break before binary operator

Imports on separate lines

Use white space in a way to assist readability

Module level dunder names (__x__) placed after the module docstring, but before any imports (except from __future__ imports)

Naming Conventions

lowercase or lower_case_with_underscores commonly seen

Class names normally use CapWords convention

Use **self** for first argument to instance methods; use **cls** for first argument to class methods.

One leading underscore for non-public methods or instance variables.

Use two leading underscores for name mangling.

PEP 8 Tools

IDE integrated formatting tool, e.g. PyCharm

autopep8

pep8

Warning: be aware to manage possible version control merge-conflicts, unclear code modification history, code review difficulties

PEP 255 - Simple Generators and more …
PEP 289 Generator Expressions

But first, Iterators … what are they?

An iterator is an object that can be iterated (looped) upon, such as lists, strings, or dictionaries.

A class must implement __iter__ and __next__ methods for iterator functionality.

iter() creates the iterator, while next() steps through the items of an iterator.

Iterators may only be iterated over once.

Next, Simple Generators … what are those??

A simple generator allows an easier path to iterator behavior by producing a sequence of results.

Generators added **yield** keyword to Python.

Some have compared yield to return as a value is returned., But yield saves the state of the function. The next time the function is called, execution continues from the next step.

```
def generate_ints(n):
    for i in range(n):
        yield i
gen = generate_ints(3)    |    next(gen)   # 0    |    next(gen)   # 1    |    next(gen)   # 2
```

Simple Generators …

Calling the generator function allows iteration over the data once.

If there’s a need to iterate more than once, the generator function must be called again.

Remember, this is not like iterating over a list, string, or dictionary, which may be done as many times as one wishes.

Wait, there’s more … Generator Expressions

Or sneaking in PEP 289 just for fun and memory savings.

```
a = [1, 2, 3, 4]
b = (2*x for x in a)
# b is a generator object -- notice the parenthesis use
for i in b:
      print(i, end=" ")
# 2 4 6 8

for x in a:                    # alternatively as a simple generator
      yield(2*x)
```

Generator Expressions Example

From PyCon 2008 “Generator Tricks for System Programmers” by David Beazley
Total bytes transferred from log values (http://www.dabeaz.com/generators/Generators.pdf)

```
with open(“access-log”) as wwwlog:
    byte_column = (line.rsplit(None,1)[1] for line in wwwlog)
    bytes_sent = (int(x) for x in byte_column if x != ‘-’)
    print(“Total “, sum(bytes_sent)
```

Follow code as a processing pipeline.
access-log -> wwwlog -> byte_column -> bytes_sent -> sum() -> total
Each step is using a generator.

Thanks!

https://pep8.org

How - and why - you should use Python Generators
https://medium.freecodecamp.org

http://www.dabeaz.com