A Python Style Guide

Duane A. Bailey

[Draft of September 16, 2020]

Implementations of algorithms—programs—can be *beautiful*. We make programs beautiful because they will be appreciated, later, by some reader. That may be you. If you're a tool-builder, users will view your code by what they read. Good programming languages make it easier to cast our abstract ideas as programs, but they also form an effective medium for human communication. Python is no exception.

Part of what makes a beautiful algorithm implementation is just its ability to leverage the syntax of the programming language to convey meaning. When an algorithm is written well, the language's presentation *facilitates*, rather than impedes understanding. Part of the process of learning a language is to learn the common practices and coding idioms. Programming is a social discourse and its idioms efficiently convey not only program meaning, but programmer intention.

This document is about coding practices—"pythonic" approaches—to writing code that facilitates understanding. These are cast as a series of rules, to focus the attention. Straying from these guidelines can have the effect of making your code harder for the reader to understand.

Still, Orwell might have proposed: *break any rule should it lead to a design not worth defending.*

Rule 1 USE ONLY PYTHON VERSION 3.

Python 2 does not include features we depend on to write scalable and correct programs.

This section discusses how we name things or *objects* in Python. Often, the name given to a value is our only hint at its meaning. Good names make good programs.

Rule 2.1 LIMIT NAMES TO ALPHABETIC AND NUMERIC CHARACTERS.

This restriction makes your programs more readable and maintainable.

DO: `pi`, `rstrip`, `reader`, `atan2`

Rule 2.2 USE MEANINGFUL NAMES. SINGLE CHARACTER *are* ALLOWED IN `for` AND `with`.

Thoughtful naming conveys meaning.

DO: `answer`, `position`, `opponent`, `wordList`

DON'T: `a`, `p`, `o`, `aFile`

Rule 2.3 USE "CAMEL CASE" TO IMPROVE READABILITY OF MULTIWORD IDENTIFIERS.

Start successive words within a name with uppercase letters. A consistent and compact approach makes multiword identifiers more readable and understandable.

DO: `mammothMascotImage`, `purpleCowMotto`, `diskRead`

DON'T: `input_file`, `dread`

Rule 2.4 ONLY USE UNDERSCORES TO INDICATE PRIVACY OR SPECIAL VARIABLES.

Underscores, while legal characters within names, convey specific meanings. Underscores should be avoided except at the beginning of private identifiers and, used in pairs, surrounding special names. The lone underscore (`_`) is reserved to mean "this value is unimportant".

DO: `__name__`, `p._x`, `_`

DON'T: `life_as_usual`, `yourAnswerIs____`

This is version 1.0 of *A Python Style Guide*, found at `http://www.cs.williams.edu/~bailey/pyStyle.pdf`.

Spaces, tabs, and end-of-line marks (collectively called, *whitespace*) are used, as in writing, to delimit words and phrases. In Python, whitespace also plays an important role in *structuring* programs.

Rule 3.1 INDENT CONSISTENTLY USING ONLY SPACES.

Python uses consistent indentation to identify *suites* of related statements. Python treats tab characters as subtly different than spaces. Some editors may attempt to substitute several spaces with a single tab. This worries Python, which will warn of inconsistent indentation.

TIP: `emacs`: set the variable `indent-tabs-mode` to `nil` to ensure correct behavior.

Rule 3.2 USE BLANK LINES TO SEPARATE DEFINITIONS. USE BLANK LINES, SPARINGLY, WITHIN DEFINITIONS.

Blank lines help the reader identify the start of a new definition. Unused lines can also break longer suites of statements into logical parts. Multiple, adjacent blank lines, however, tend to make definitions appear scattered.

Rule 3.3 USE SPACES AS YOU WOULD WITH HUMAN LANGUAGE PUNCTUATION.

Spaces and punctuation in programs serve a similar purpose to punctuation in human language. Use single spaces to separate conceptual units of the program, but do not use spaces where you would not find them in a human language. Some feel that using spaces around operators in expressions can make the “mathematics” clearer. Your choice to use spaces to emphasize operators—or not—should be consistent throughout your code.

DO:

```
def fibo(n, a=1, b=1):
    """Compute the n-th value in the Fibonacci sequence:
    fibo(0) = a, fibo(1) = b, fibo(n) = fibo(n-2)+fibo(n-1)."""
    for _ in range(n):
        a, b = b, a+b
    return a
```

DON’T:

```
def gcd(a,b) :
    """ Compute the greatest common integer divisor of a and b. """
    if a> b:
        result = gcd( b , a)
    elif a ==0 :
        result = b
    else
        result=gcd(b%a,a)
    return result
```

Rule 3.4 PLACE EACH STATEMENT ON A DEDICATED LINE. DO NOT USE SEMICOLONS.

Python prefers one statement per line. While semicolons (`;`) can be used to separate statements, it tends to make it appear statements are to be executed concurrently, when, in fact, they are executed sequentially. Avoid semicolons.

DO:

```
ratio = a//b
remainder = a%b
```

DON’T:

```
a = b; b = a       # *not* the same as a,b = b,a
```

Part of Python's feeling of expressiveness is its *succinctness*. Still, succinctness can lead to a feeling of *density*; we need to be careful to make sure our Python statements are manageable, consumable chunks of logic.

Rule 4.1 Limit lines to 80 characters. Use parentheses to delimit longer expressions.

Because Python's statements are line-oriented, it is important to make sure that readers are not confused by long lines wrapped by editors or printers. Either break up long lines into several shorter statements or manually break lines to make reading more straightforward.

Rule 4.2 Consistently use quotes (`'hello'`) or quotations (`"hello"`) around strings.

Python has two alternative ways to delimit strings: single apostrophes (or *quotes*) and quotation marks. Pick one style of delimiting strings, and use it consistently throughout your code.

Do:

```
    print("So, "+name+", do you want to play again?")
```

Don't:

```
    print("The lazy fox jumps over the quick "+dogColor+' dog.')
```

Rule 4.3 Use parentheses only when necessary.

Parentheses in expressions (`()`) indicate the desired order of evaluation. Wrapping entire expressions in parentheses is typically unnecessary. In particular, `if`, `while`, `return`, and `yield` statements do not require parentheses around their expressions. Excessive use of parentheses can impact program readability.

Do:

```
    if b*b >= 4*a*c:
        radical = (b*b - 4*a*c)**0.5
    else:
        print("Roots are imaginary.")
```

Don't:

```
    def printBinary(n):
        if (n < 2):
            print((n%2))
        else:
            printBinary((n//2))
            printBinary((n%2))
```

These rules govern how we *document* programs. Documentation helps readers understand what code does and how it's used. In Python, good documentation can support testing.

Rule 5.1 Every module, class, and function should contain an initial docstring.

When the statement in a function, class, or module is an unused string, Python interprets that string as documentation, called a *docstring*. Python documentation is available through the `pydoc3` command. Try, for example,

```
pydoc3 math
```

Inside the interpreter, documentation about an object is easily retrieved using the `help` method:

```
>>> help(print)
```

Programmers who build useful tools describe how they are used.

One important exception is the documentation of the short, one-off *lambda functions*. These functions are short enough to be easily understood. (If not, they should be declared using `def`.)

Do:

```
def randint(low, high):
    """Pick a random integer between low and high (inclusive)."""
    return low + abs(random()) % (high-low+1) # includes high
```

Documentation for definitions of identifiers that begin with underscores are *not* implicitly volunteered by `pydoc3`. Programmers should still provide docstrings for these definitions because (1) their documentation may still be explicitly requested and (2) making a definition public, later, should not expose undocumented features.

Rule 5.2 Use triple-quotations (`"""`) to delimit docstrings.

Because docstrings typically span multiple lines, it is universal practice to use three quotation marks (`"""`) to delimit this text, even if it is currently a single line.

Rule 5.3 Assume programs are read from top to bottom.

Comments beginning with the hash mark (`#`) describe code on a particular line, or, if standalone, on lines that follow. Where code is necessarily obscure or complex, it is helpful to provide a comment to facilitate the reader's understanding. Comments that appear on a line with code describe the line. Comments that appear on dedicated lines describe the code that *follows*.

Do:

```
# compute the real roots of the polynomial a*x**2+b*x+c:
radical = (b*b-4*a*c)**0.5     # using multiplication to compute b**2
root1 = (-b - radical)/(2*a)   # left root
root2 = (-b + radical)/(2*a)   # right root
```

Don't:

```
# Is "year" a leap year?
mult4 = (year % 4) == 0
# year is divisible by 4
century = (year % 100) == 0
mult400 = (year % 400) == 0
isLeap = mult4 and ((not century) or mult400)
```

```
    # ie.  a year is a leap year if it is divisible
    # by 4, typically.  years divisible by 100
    # are not leap years, UNLESS they are divisible by 400
    # e.g. 2000 was a leap year, but 2100 won't be.
```

Rule 5.4 In docstrings, demonstrate example usage using doctest format.

Python supports a wide variety of testing strategies, including *unit tests*. The best docstrings include examples of function or method use:

```
def gcd(a, b):
    """Compute the greatest common divisor of integers a and b.

    >>> gcd(0, 0)
    0
    >>> gcd(0, 99)
    99
    >>> gcd(10, -15)
    5
    """
```

Python can optionally verify these examples with the `doctest` package.

Tip: (1) Always place a space after the prompt ("`>>> `"). (2) When a result is a string, delimit the expected text with single quotes. (3) Execute doctests as part of the script's main suite of statements:

```
if __name__ == "__main__":
    import doctest
    doctest.testmod()
```

Rule 5.5 Use correct spelling, punctuation, and grammar to improve comment readability.

Documentation is provided for humans to read. You can help them by writing informative documentation in language they can readily understand.

Do:

```
def randWord(self):
    """Pick a random word from the dictionary."""
    return random.choice(self.wordList)
```

Don't:

```
def meanLength(wordList):
    """Compute da average....another silly method IMHO"""
    if wordList:
        result = sum([len(word) for word in wordList])/len(wordList)
    else:
        result = 0
    return result
```

This section—about meanings or *semantics*—gathers together miscellaneous guidelines about choice in design. Python supports many approaches to solving problems. Some decisions are problematic because (1) they lead to code that is hard to maintain, or (2) they lead to code that works but is unnecessarily difficult to understand.

Rule 6.1 Avoid the use of `pass`.

The use of the word `pass` (or the equivalent `...`) is a do-nothing statement used where code is required. Its use suggests incomplete code or contorted program structure. We *will* use the `pass` statement to suggest areas where students should focus, but the expectation is that all `pass` statements will be replaced by working code.

The use of `pass` as a stand-in for a branch of an `if` statement can always be simplified.

Rule 6.2 Manage files with the `with` statement, if possible.

Files should be closed when you are finished. When writing to a file, failing to close the file has the potential to lead to incomplete writes. Python's `with` statement provides a mechanism for opening a file for access by subordinate statements. When the suite is finished, Python will ensure the file is automatically closed.

Do:

```
with open('census.csv') as popFile:
    popRows = csv.reader(popFile)
    towns = [row[0] for row in popRows]
```

Don't:

```
wordFile = open('/usr/share/dict/words')
wordList = [word.strip() for word in wordFile]
```

Rule 6.3 Avoid unnecessary global variables.

While Python functions allow references to *global* variables, the practice should be avoided.

By contrast, top-level definition of *constants* (e.g. `math.pi`) increase readability and maintainability of code.

Do:

```
SystemDictionary = '/usr/share/dict/words'
def readDict(dictName = SystemDictionary):
    with open(dictName) as dictFile:
        words = [word.strip() for word in dictFile]
    return words
```

Don't:

```
base = 2                # ok: a constant
convertedNumber = ''    # not ok: will change
def binary(n):
    global convertedNumber
    if n == 0:
        convertedNumber = '0'
    else:
        while n:
            digit = '0' if n%base == 0 else '1'
            convertedNumber = digit + convertedNumber
            n //= base
```

Rule 6.4 Imports from separate modules should appear as separate imports.

While Python allows symbols to be imported from separate modules in a single `import`, the practice should be avoided. Import symbols from distinct modules using distinct import statements. This makes the dependencies of a program more understandable and improves maintainability of code.

Do:

```
from math import pi, e           # natural math constants
import matplotlib.pyplot as plt   # for ploting data in polar coordinates
```

Don't:

```
import csv, random, doctest      # random utilities we might need
```

Rule 6.5 Return function values once, as the last statement.

Functions must always return values. It is always possible to a single `return` statement at the end of the function. Best practice avoids the use of multiple returns. Because every path through a function must produce a value, having a return only at the end guarantees that behavior. In general, the multiple points of return may (1) make it more difficult to reason about function behavior and (2) make it difficult to maintain the such behavior as the code matures.

Do:

```
def maxDefault(nums, default = 0.0):
    """Compute maximum of nums, or default if empty nums."""
    mean = default
    if nums:
        mean = max(nums)
    return mean
```

Don't:

```
def add(vals):
    """Compute sum of vals."""
    if vals:
        return vals[0] + add(vals[1:])
    return 0
```

In some cases (e.g. recursion), multiple points of return can improve readability:

```
def hex(n):
    """Convert a value to a hexadecimal string."""
    if n < 16:
        return "0123456789abcdef"[n]    # single digits
    else:
        return hex(n//16)+hex(n%16)     # many digits
```

Rule 6.6 Only use comprehensions when they are short and simple.

Python provides compact mechanisms for building containers, called *comprehensions*. These are idiomatic forms that replace simple, common `for` statements. When the comprehension approaches the length of a line is ceases to be easily understood. In those cases, use the `for` statement.

Do:

```
# compute average word length in (nontrivial) story
with open('story.txt') as f:
    lines = [line for line in f]
# gather a list of words
words = []
for line in lines:
    words.extend(line.split())
mean = sum([len(word) for word in words])/len(words)
```

DON'T:

```
# extract palindromes from 'story.tex'
with open('story.txt') as f:
    pals = {word for line in f for word in line.split() if word == word[::-1]}
```

Similarly, generator expressions should be kept as simple as possible.

Rule 6.8 USE ONLY THE SIMPLEST CONDITIONAL EXPRESSIONS. NEVER NEST THEM.

A conditional *expression* replaces a simple `if` *statement* whose branches select between related values of the same type. Use them only in the simplest cases, and never nest them.

DO: The following code handles printing plurals:

```
print('Tell us {} {}!!'.format(n, 'story' if n == 1 else 'stories'))
```

DON'T: This code is harder to understand or maintain:

```
# np is the number of people standing in line
print('I see '+('no one' if np == 0
                else 'a person' if np == 1
                else 'a couple' if np == 2
                else 'some people')+' waiting for tickets.')
```

Rule 6.7 LIMIT USE OF `lambda` FUNCTIONS TO SIMPLE, USE-ONCE SITUATIONS.

Lambda functions are nameless definitions that should be used to describe simple, single-use, single parameter functions. Typically, they are used to map values to sorting keys. Their anonymity and abbreviated syntax can obscure the meaning of important code.

DO:

```
# sort cities by decreasing population
cities.sort(key=lambda city: city.census, reverse=True)
```

DON'T:

```
# compute product of numbers
prod = lambda *args : 1 if not args else args[0] * prod(*args[1:])
```

Rule 6.9 USE ONLY IMMUTABLE VALUES FOR DEFAULT ARGUMENTS.

Function arguments with default values can reduce redundancy in code and improve code reuse. It is important, though, to make sure the default value is immutable. If the default is mutable, the default value can be changed.

DO: This updates an entry in a dictionary, possibly creating one if necessary.

```
def include(key, value, table = None):  # None is immutable
    if table is None:
        table = dict()   # this default value can (and will) be changed
    table[key] = value
    return table
```

Don't: This code attempts to do something similar, but will only create one dictionary with all the of the key-value pairs:

```
def include(key, value, table = dict()):  # one dict() is created at def
    table[key] = value
    return table
```

Rule 6.10 Use implicit boolean interpretations of values.

"Zero-like" values: numeric versions of zero or containers holding zero values are interpreted as `False`, and all others an interpreted as `True`. Use this fact to meaningfully simplify conditional expressions.

Do:

```
# l is a list to be destructively shuffled
shuffled = []
while l:
   selected = randint(0,len(l)-1)
   shuffled.append(l.pop(selected))
```

Don't:

```
while l != []:
    first,*rest = l
    process(first)
    l = rest
```

(Though remember Rule 6.12: When checking for an object reference, use `is` or `is not None`.)

Rule 6.11 Avoid the use of `True` and `False` in boolean computations.

Expressions involving explicit boolean constants (`True` and `False`) can always be improved. Simpler expressions are more readable.

Do:

```
while ballWentFoul:
   pitchToBatter()
   ...
```

Don't:

```
if ballCaught == False:  # more readable: if not ballCaught:
    waitToRun()
```

Rule 6.12 When testing validity of object references, use `is None`.

In Python, the unique value `None` means *no object*. Variables that currently do not reference any object—sometimes referred to as a *null*-reference—have value `None`. Functions that do not return a value, return `None`. To test for a value use `is not None`. Similarly `is None` checks for a lack of value. While `None` is interpreted as the boolean `False`, so are other reasonable referenced *values*.

Do:

```
    def combine(aStream, bStream, combiner = None):
        if combiner is None:                   # idiom for complex default arg
            combiner = lambda a,b : a+b
        while aStream and bStream:
            yield combiner(a.pop(0), b.pop(0))
```

Don't:

```
    def __init__(self, name, alergies):
        if not alergies:                        # wrong, use: alergies is None
            self.reactions = "<Unknown>"
```

Rule 6.13 Distinguish script and module behaviors.

Python scripts often mature into small collections of tools. Avoid executing script-only code during import with the Python idiom:

```
   # end of script definitions
   if __name__ == "__main__":
       # beginning of script execution
```

Everything in Python is an object. Python 3 supports the careful development of new classes of objects. We describe, here, an approach to designing classes that *limits* decisions that lead to “brittle” code that violates our understanding of how to build common data abstractions.

Rule 7.1 Declare a class’s attributes in the `__slots__` variable.

Organizing the valid names of attributes in a list of strings called `__slots__` guards against rogue, object-specific attributes. Typically, these attributes are considered `private`, and start with one or more underscores.

Rule 7.2 Use properties to simulate attributes.

Property annotations allow the class designer to carefully control access to private attributes through accessors and mutators.

Do:

```
def pt(object):
    """A class for managing 2D coordinates."""
    __slots__ = ["_x", "_y"]
    def __init__(self, x, y):
        self._x = x
        self._y = y

    @property
    def x(self):
        return self._x

    @property
    def y(self):
        return self._y

    @property
    def radius(self):
        return (self.x**2+self.y**2)**0.5

    @property
    def angle(self):
        return math.atan2(self.y,self.x)

    def __str__(self):
        return "<a point r={}, a={}>".format(self.radius, self.angle)
```