Python Style Guide

CS 5001 Intensive Foundations of Computer Science

When writing code, it is important to keep in mind that every program has two target audiences: (1) the machine that will execute the code and (2) the programmers who will read, write, debug, and maintain the code. For the machine, **syntax** describes the set of rules defining what is possible to write in the language. For the programmer, **style** (or coding standards) describes the set of rules that should be followed to help the humans reading the code to understand it.

In this document, we will set forth a set of simple rules, presented in a short, and easy to understand format. The idea is to provide clear standards to follow that will make your code easier to read and understand.

No standard is perfect and no standard is universally applicable. Sometimes you find yourself in a situation where you need to deviate from these standards. Before you decide to ignore a rule, you should first make sure you understand why the rule exists and what the consequences are if it is not applied. If you decide you must violate a rule, then document why you have done so. This is the prime directive.

Classification of Style Guidelines

Before we begin to enumerate the rules, we present five categories for our rules:

Formatting

Many professional programmers have to navigate millions of lines of code on a daily basis and have to switch from one part of the code base to another -- instantly understanding what the code they are looking at is doing. This is only possible with consistent, predictable formatting.

Formatting is so important, that it has been the center of very long-running, counterproductive "flamewars" in which programmers have lost time, focus and community spirit by pitting themselves against each other claiming there can only be one way of writing code. The animosity is motivated by the fact that when the formatting style has a large degree of freedom, it leaves people to develop their own style and preference, and to become used to it. If you are used to one formatting style, it can be very distracting to read or write fluently in another formatting style.

For this reason, some languages, like Python, have mandated strict formatting guidelines on indentation and nesting. For other languages, technology, in the form of *code reformatters*, have helped provide consistency across the contributions of many people.

In this course, we will use Wing 101 that has the capability to reformat the code according to specific guidelines of indentation, spacing, etc. Even if you are not using an IDE that can do this, you are responsible for writing code that follows guidelines determined by the industry standard style checked called `pycodestyle`[...]. This tool analyzes code and indicates what formatting errors (and a couple other errors) that exist in the code. Each submission will be run against this style checker. You should strive to submit code that does not generate any format errors.

Commenting

Comments are meant to be minimal but informative. They serve to help others (and ourselves) in understanding what the code is intended to do quickly.

While it may seem that the early assignments are relatively easy to understand without comments -- especially while you are in the process of writing them and they are fresh in your mind -- it is good to get into the habit of commenting as you go before things get more complicated.

Comments should be:

- **Concise** -- Any comment you write should only be as long as necessary to communicate relevant information
- **Reader-helpful** -- Add comments only if they add value to the code itself. Unless the code is exceedingly complicated, try not to write comments that simply repeats the code. You should assume that the reader understands Python.
- **Used to break code into logical units** -- Using minimal, concise comments to break code into logical units helps the reader understand exactly what each code block is doing. It will take some practice to decide where to break up code and some of it will come down to personal style. Try to find the right balance of aiding understanding without being too excessive or too minimal

Commenting can be used as part of an effective code-writing strategy as well. Instead of commenting after the code is written, try commenting before writing the code. By breaking down your program logically into smaller chunks and then working on those small chunks individually you can avoid some bugs and logical errors much more effectively.

Naming

The naming of variables, constants, functions, and classes are so central to understanding code that a bad name can be the difference between a fellow developer understanding what everything does at first glance and not having any clue of where to begin.

Try hard to come up with accurate, concise names that define what the entity is or what the entity does. Names should be:

... https://pypi.org/project/pycodestyle/

- **Clear** -- Many programmers will sacrifice clarity for speed. It is important to understand that using names that save a handful of keystrokes instead of using a truly accurate name is just bad programming. Good programmers spend very little of their time typing (maybe 10-15%) and spend most of their time thinking! Using clear names can ultimately make the thinking clearer.
- **Concise** -- Consider how you can clearly name a variable without using excessively long names. For example, if you are storing the name of a user's favorite color, you might not want to call the variable `the_users_ favorite_color` even someone who is not computer savvy would know what you were talking about if you used `favorite_color`
- **Be Consistent** -- different languages use different rules when it comes to naming conventions: PascalCase, camelCase, snake_case, lowercase, or hyphen-case. Your job as a programmer is to adopt the naming conventions of the language in use in your project. In the case of Python, you should use usually snake_case

The naming of entities in programming is central to what we do and they need to be taken seriously. But coming up with clear, concise, and consistent variable names, you can make communication between you and other developers (and your instructors) more efficient and easier to understand.

Maintainability

Writing clean, concise, and maintainable code can be tricky when it is your first time thinking about a problem. If it's your first time, then just getting code working is your first consideration. Your next step, however, should be making code readable using good organization and naming practices that increase your ability to debug and maintain your own code. It will also make it easier if you are working with others or pass your code on to someone else in the future for them to maintain your code. The thing to keep in mind even then is that the code you are writing is not really for yourself. The code you write now is for someone else to read, and that someone else can even be your future self three months from now

Efficiency

As a programmer, it is important to not only write code that works, but also code that works quickly and uses resources effectively. For processing small amounts of data, inefficient code may run quickly, but as the amount of data you are processing increases, the runtime can quickly deteriorate. Later in this course, and as you move on to future CS classes, you will study the efficiency of different algorithms and data types more in depth. For your current purposes, try to write code that doesn't take unnecessary steps

Coding Standards for CS 5001

The rules covered in this document are relevant to the Python programming language Some of the rules can be checked automatically, others cannot. We will do both in this class.

Automatic style checks is accomplished using the industry standard style checked called `pycodestyle`[1]. This tool analyzes code and indicates what formatting errors (and a couple other errors) that exist in the code. Each submission will be run against this style checker. You should strive to submit code that does not generate any format errors.

For those things that cannot be checked automatically, we will review your code to see how it adheres to the standards set for this course. These standards are defined module-by-module. You are responsible for writing code that follows these guidelines:

Module 1 -- Getting Started

- **[commenting] Every file should have a file comment.**
  A file comment serves to identify the where, when, and why a file was written. In this course, your file comment should contain the name(s) of the person(s) who wrote it, the course name and semester, and the assignment that the file is associated with. It is also a good idea to include a short description of the code in the file. For example:

```
'''
Joe Schmeggeggie
CS 5001, Spring 2020
Lab 1 -- Getting started

This is a basic hello world program.
'''
```

- **[maintainability] Programs all start with `main()`**
  In many languages, execution begins with a function called `main()`. While Python is a bit more flexible than this, writing programs in this manner makes it easy to understand

1 https://pypi.org/project/pycodestyle/

where your program begins. As such, we will require that all programs begin with writing a `main()` function:

```
def main():
   print("Welcome to Align Online!"

if __name__ == "__main__"
   main()
```

Module 2 -- Variables & Arithmetic Operators

- **[naming] Use snake_case for variable names**
  The syntax of Python requires that variables be named with any combination of alpha-numeric characters as long as it starts with a letter. Style says that variable names should use snake_case -- be named with only lowercase letters, using a single underscore to separate different elements.

```
balance = 100.00
interest_rate = 0.06
total_payment = balance * ( + interest_rate)
```

- **[naming] Use uppercase with underscores for constant names**
  When the value of a variable should never change, use only uppercase letters and separate words with underscores.

```
INTEREST_RATE = 0.06
DAYS_PER_WEEK =
```

- **[naming] Use meaningful names**
  When you name a variable or constant, use a name that is, and will remain, meaningful to those programmers who must eventually read your code. Since variables represent things, it is a good idea to *use singular nouns* to name a variable. Avoid using single-character or generic names that do little to define the purpose of the entities they name. Using meaningful words can make your code a lot easier to understand. For example, in the following code, the purpose for the variables "x", "y", and "z" is unclear:

```
x = 100.00
y = 0.06
z = x * ( - y)
```

The code is much easier to understand when meaningful names are used:

```
balance = 100.00
INTEREST_RATE = 0.06
```

```
total_payment = balance * ( + interest_rate)
```

- **[naming] Avoid abbreviations**
  Abbreviations reduce the readability of your code and introduce ambiguity if more than one meaningful name reduces to the same abbreviations. For example, "temp" could represent a temporary value, a temperature, or even information about Temple, Arizona. It is much better practice to avoid such ambiguity in variable names.

  Do not attempt to shorten names by removing vowels. This practice reduces the readability of your code and introduces ambiguity if more than one meaningful name reduces to the same consonants.

- **[formating] Add whitespace around operators**
  White space is the area of a page devoid of visible characters. Code with too little white space is difficult to read and understand, so use plenty of white space to delineate comments, code blocks, and expression clearly. One place where whitespace makes code easier to read is when it is used in expressions that use operators. Use a single white space character between any operator and the variables or literals on either side of it.

  For example, consider the following expression:

```
value = x1+-2*x2+y1--3*y2
```

  Adding white space around the operators makes this code easier to read:

```
value = x1 + -2 * x2 + y1 - -3 * y2
```

- **[maintainability] Clarify the order of operations with parentheses**
  The order of operations in a mathematical expression is not always obvious. Even if you are certain as to the order, you may safely assume others will not be so sure.

```
width = ((buffer * offset) / pixel_width) + gap
```

Module 3 -- Boolean Expressions & Conditionals

- **[formatting] Indent nested code.**
  Use indentation of exactly four spaces (never tabs) to indicate nested code. In Python, mixing tabs and spaces in your code is an error. For this reason, most Python IDEs will insert four spaces whenever you hit the tab key.

Module 4 -- Functions & Testing

- **[naming] Use snake_case for function names**
  The syntax of Python requires that functions be named with any combination of alpha-numeric characters as long as it starts with a letter. Style says that variable names should use snake_case -- be named with only lowercase letters, using a single underscore to separate different elements.

```
def payment_calculator(balance, interest_rate):
```

- **[naming] Use meaningful names**
  When you name a function, use a name that is, and will remain, meaningful to those programmers who must eventually read your code. Since functions do stuff, *use a verb* to name a function. Avoid using abbreviations, single-character or generic names that do little to define the purpose of the entities they name. Using meaningful words can make your code a lot easier to understand

- **[commenting] Document all functions**
  The first line inside a function definition should be a *docstring* -- a comment delimited by three single-quotes that explains how to use the function. The first line of a docstring (the summary) should be a phrase that ends with a period and describes the functions effects as a command. After the summary, a docstring should explain the meaning of each parameter (by name) and the return value:

```
def normalize(values, limit):
    '''
    Function -- normalize
        Scales the values in a list so that all the values fall within
        the range 0-limit
    Parameters:
        value -- the original list of values, will not be modified
        limit -- the largest absolute value that value can have after
                 scaling
    Returns a list containing each of the values, scaled such that their
            ratios with each other are the same but that the largest
            absolute value is the limit
    '''
```

- **[commenting] Keep comments and code in sync**
  When you modify code, make sure you also update the related comments. The code and documentation together form a software product, to treat each with equal importance.

- **[maintainability] Define small functions that do only one thing**
  Smaller functions that do only one thing are easier to design, code, test, document, read,

understand, and use than larger ones. When a function is getting too large spanning several screens in length, find a way to factor a meaningful part of it out into a separate helper function.

- **[maintainability] Replace repeated nontrivial expression with equivalent functions**
  Do not duplicate code. It is often the case that we have code that needs to be called from multiple places. Rather than copying code from one place in your code and pasting it in another and possibly copying an error, move the code into its own function and call it from both places. This means that future changes are localized, so maintenance is easier and testing effort is reduced. For example, consider the following code that draws a pattern:

```
def main():
    print("+=*=*=*=*=*=*=*+"
    print("|../\..../\..|"
    print("|./\/\../\/\.|"
    print("|/\/\/\/\/\/\|"
    print("|\/\/\/\/\/\/|"
    print("|.\/\/..\/\/.|"
    print("|..\/....\/..}"
    print("+=*=*=*=*=*=*=*+"
    print("|\/\/\/\/\/\/|"
    print("|.\/\/..\/\/.|"
    print("|..\/....\/..}"
    print("|../\..../\..|"
    print("|./\/\../\/\.|"
    print("|/\/\/\/\/\/\|"
    print("+=*=*=*=*=*=*=*+"
```

Breaking the repetitive portions of this pattern into functions localizes each piece of the pattern so that any modifications to the pattern is localized making it easier to modify the code when the need arises:

```
def vertical():
    print("+=*=*=*=*=*=*=*+"

def up_arrows():
    print("|../\..../\..|"
    print("|./\/\../\/\.|"
    print("|/\/\/\/\/\/\|"

def down_arrows():
    print("|\/\/\/\/\/\/|"
    print("|.\/\/..\/\/.|"
```

```
    print("|..\/....\/..}"

def main():
    vertical()
    up_arrows()
    down_arrows()
    vertical()
    down_arrows()
    up_arrows()
    vertical()
```

Module 5 -- while Loops

- **[maintainability] Always use the loop condition to terminate a loop**
  Avoid writing loops that require early termination or skipping, Consider the following loop that correctly prints the numbers from 1 to 5 (inclusive):

```
number =
whileTrue
    print(number)
    if number ==
        break
    number +=
```

  This is a bad example of how to write a loop because using a loop condition instead of a break would make the code easier to understand:

```
number =
while number <=
    print(number)
    number +=
```

Module 6 -- Strings & Lists

- **[naming] Use meaningful names**
  When naming list variables, *use plural nouns* to indicate that the variable holds more than one value. For example, a list containing all of the grades earned by a student over the course of a semester "`grades`" rather than "`grade`". Otherwise, use all of the naming conventions for variables.

- **[maintainability] Always use a tuple for an immutable list.**
  If your intention for the content of a list is that it should never change, then you should

use a tuple instead. This ensures that it will not change and makes your code more maintainable.

Module 7 -- for Loops

- **[maintainability] Always use a while loop for indefinite iteration**
  The `for` loop is a **definite** loop and that means that the number of iterations is determined when the loop starts. We can't use a definite loop unless we know the number of iterations ahead of time. The **indefinite** loop keeps iterating until certain conditions are met -- this is a `while` loop.

Module 9 -- Recursion

- No new rules.

Module 10 -- Exception Handling

- **[maintainability] Do not silently absorb a run-time or error exception**
  Do not allow your program to ignore that something bad happened. Doing so makes it hard to debug because information is lost. For example, in the following code, the `FileNotFoundError` is ignored.

```
filename = input("Enter filename: "
try
   file = open(filename)
   ...
except FileNotFoundError:
   # leaving this blank causes information to be lost
   # and makes your program difficult to debug
   pass
...
```

Instead, setup your code so that if you catch the FileNotFoundError, your program does something useful with it:

```
done = False
whilenot done:
    filename = input("Enter filename: "
    try
        file = open(filename)
        ...
        file.close()
```

```
        done = True
    except FileNotFoundError:
        print(filename + " was not found, try again"
    ...
```

- **[maintainability] Always catch the most specific error possible**
  When writing code to handle unexpected errors by adding a try/except block, it is important that we only handle the specific errors that we know what to do with. Consider the following code:

```
try
    # code that can generate ValueError and ZeroDivisionError

except
    print("An error occured"
```

  There are two problems with this code. The first is obvious. By using the catch-all except block, we have no way of printing a useful message since we do not know which error was raised. But worse, the catch-all except block will catch errors that we do not know how to handle. For instance, when a user types Ctrl-C, an exception is raised to stop the execution of a program. This exception will be caught by the catch-all except block!

  The better way to write this code is to catch the specific errors that can be raised by the code:

```
try
    # code that can generate ValueError and ZeroDivisionError

except ValueError:
    print("Invalid value was used"
except ZeroDivisionError:
    print("Tried to divide by zero"
```

Module 11 -- Dictionaries & Sets

- **[maintainability] Use the appropriate abstract data type for the problem**
  It is always possible to use an abstract data type that is not appropriate for the problem, for instance using parallel lists to maintain sets of data that could more accurately be maintained in a dictionary, or using a list that allows duplicates when a set will eliminate those duplicates. Learning to use the right abstract data type for a particular problem can be tricky in the beginning but with practice you will get better at it.

Module 12 -- Classes

- **[naming] Use PascalCase for class names**
  The syntax of Python requires that classes be named with any combination of alpha-numeric characters as long as it starts with a letter. Style says that class names should use PascalCase -- be named using upper and lower case such that each element of the name begins with a single upper case letter.

- **[naming] Use meaningful names**
  When you name a class, use a name that is, and will remain, meaningful to those programmers who must eventually read your code. Since classes represent things, it is a good idea to *use singular nouns* to name a class. Avoid using abbreviations, single-character or generic names that do little to define the purpose of the entities they name. Using meaningful words can make your code a lot easier to understand. For example, a class named "C" does not indicate what it represents. Additionally naming a class that represents student grades "Grades" is less clear than naming it "Gradebook".

- **[maintainability] Always save a class in its own file**
  Placing classes in their own file makes them more modular.

- **[maintainability] Always write a constructor**
  All objects have some data associated with them called *attributes.* Whenever we write a class, we must also write code that initializes those attributes. This is the purpose of the constructor. In Python, the constructor is implemented in the `__init__` method:

```
class Complex
   def__init__(self, real_part, imaginary_part):
       self.r = real_part
       self.i = imaginary_part

   ...
```

- **[maintainability] Always write the __str__ method**
  Suppose that you had the `Complex` class that is defined above. At some point, while using this class, you will have declared a `Complex` object and then tried to see what the object contains by printing it:

```
c = Complex(,
...
print(c)
```

  This produces the following output:

```
<__main__.Complex> object at 0x0419B150
```

This is how the computer sees the object and that's just fine, but it is not friendly for us humans. For this reason, you should always implement the `__str__` method when implementing a class. The `__str__` method is used to convert an object to a string representation so that it can be presented in a human readable form. It's a good idea when implementing this method that you are as descriptive as possible:

```
    def__str__(self):
        return"Complex: {:.2f} + {:.2f}i".format(self.r, self.i)
```

- **[commenting] Document the class**
  The first line inside a class should document the purpose of the class.

```
class Button
    '''
    A Button is a labeled rectangle in a window.  It is activated
    or deactivated with the activate() and deactivate() methods.
    The clicked(p) method returns True if the button is active
    and p is inside it.
    '''
```

- **[commenting] Document all methods of the class**
  Similar to how we documented a function on the first line inside the function using a docstring, we also want to add a docstring to each method of a class. The first line of a docstring (the summary) should be a phrase that ends with a period and describes the functions effects as a command. After the summary, a docstring should explain the meaning of each parameter (by name) and the return value.

Module 13 -- Stacks & Queues

- No new rules.

Module 14 -- Efficiency, Searching and Sorting

- No new rules.

Getting Started Writing Tests

When we first start writing code, one

Appendix A -- pycodestyle Error Codes

`pycodestyle` is a tool that determines how well your code follows the industry standard style for formatting. It analyzes your code and checks if it adheres to the standards generating messages in a variety of messages indicating the problem with your representing format errors in the code. Our job is to learn to interpret these messages. For example,

```
first_program.py:32: W291 trailing whitespace
```

- **`first_program.py`** is the name of the file.
- represents the line in the file where the style issue occurs
- **`32`** represents the column on the line where the style issue occurs
- **`W291 trailing whitespace`** is the message provided by pycodestyle. Refer to the Python Style Guide that we have linked to the course to find out what this message means.

For the rest of this document, I will list the different errors that `pycodestyle` generates.

Indentation

Error Code	Description
E101	Indentation contains mixed spaces and tabs
E111	Indentation is not a multiple of four
E112	Expected an indented block
E113	Unexpected indentation
E114	Indentation not a multiple of four in a comment
E115	Expected an indented block in a comment
E116	Unexpected indentation in a comment
E117	Over-indented
E122	Continuation line missing indentation or outdented
E124	Closing bracket does not match visual indentation
E125	Continuation line with same indent as next logical line
E127	Continuation line over-indented for visual indent
E128	Continuation line under-indented for visual indent

E129	Visually indented line with same indent as next logical line
E131	Continuation line unaligned for hanging indent
W191	Indentation contains tabs

Whitespace

E201	Whitespace after '('
E202	Whitespace before ')'
E203	Whitespace before ':'
E211	Whitespace before '('
E221	Multiple spaces before operator
E222	Multiple spaces after operator
E223	Tab before operator
E224	Tab after operator
E225	Missing whitespace around operator
E226	Missing whitespace around arithmetic operator
E227	Missing whitespace around bitwise or shift operator
E228	Missing whitespace around modulo operator
E231	Missing whitespace after ',', ';', or ':'
E251	Unexpected spaces around keyword / parameter equals
E261	At least two spaces before inline comment
E262	Inline comment should start with "# "
E265	Block comment should start with "# "
E266	Too many leading "#" for block comment
E271	Multiple spaces after keyword
E272	Multiple spaces before keyword
E273	Tab after keyword

E274	Tab before keyword
E275	Missing whitespace after keyword
W291	Trailing whitespace
W292	No newline at end of file
W293	Blank line contains whitespace

Blank lines

E301	Expected 1 blank line, found 0
E302	Expected 2 blank lines, found 0
E303	Too many blank lines (3)
E304	Blank lines found after function decorator
E305	Expected 2 blank lines after end of function or class
E306	Expected 1 blank line before a nested definition
W391	Blank line at end of file

Imports

E401	Multiple imports on one line
E402	Module level import not at top of file

Line & Line Breaks

E501	Line too long (82 > 79 characters)
E502	The backslash is redundant between brackets
W504	Line break after binary operator
W505	Doc line too long (82 > 79 characters)

Statements

E701	Multiple statements on one line (colon)
E702	Multiple statements on one line (semicolon)
E703	Statement ends with a semicolon
E711	Comparison to None should be ‘if cond is None:’
E712	Comparison to True should be ‘if cond is True:’ or ‘if cond:’
E713	Test for membership should be "not in"
E714	Test for object identity should be "is not"
E721	Do not compare types, use "isinstance()
E722	Do not use bare except, specify exception instead
E731	Do not assign a lambda expression, use a def
E741	Do not use variables named 'l', 'O', or 'I'
E742	Do not define classes named 'l', 'O', or 'I'
E742	Do not define functions named 'l', 'O', or 'I'

Runtime

E901	SyntaxError or IndentationError
E902	IOError

Deprecation warning

W601	.has_key() is deprecated, use "in"
W602	Deprecated form of raising exception
W603	"<>" is deprecated, use "!="
W604	Backticks are deprecated, use "repr()"
W605	Invalid escape sequence
W606	"Async" and "await" are reserved keywords starting with Python 3.7

References

- van Rossum, Guido and Coghlan, Nick. PEP 8 -- Style Guide for Python Code. Version August 1, 2013.
- Concrete style guidelines for COS126, Princeton University, Spring 2020