Coding I: Python Style Guide

Bear Creek High School, 2023-24

> "Programming is best regarded as the process of creating works of literature, which are meant to be read."
>
> - Donald Knuth

1 Introduction

Style is just as important as substance when writing software. Style helps to ensure that your code can be read and maintained by others, and it conveys to the reader that you are serious and care about the quality of your software.

Python is, by design, an easy-to-read language. It lacks a lot of the confusing syntactic features of languages like Java or C, and enforces consistent spacing on its programmers. This alone practically makes Python look like standard, written text, and as a result Python is pretty inherently well styled.

That said, there are still some conventions you must stick to in your programming for the sake of legibility and clarity beyond the rules that Python enforces automatically. Many of these rules are taken directly from Google's Python style guide and from the style guide that Python has for itself - that is to say, these conventions are widely agreed upon by real-world Python programmers, and adhering to them will help your code exist alongside software written by professionals.

Again, **adhering to this style guide is important.** It is imperative that people other than you be able to read and understand your code, and that will be nearly impossible without a consistent format. You will see throughout this course just how frustrating it is to read another person's code when they don't follow stylistic rules.

Throughout this course, your code will be graded not just on its function but also on its adherence to this style guide.

2 Naming

Naming is crucial in programming, as it allows others to quickly get an idea of what a file, function, or variable do. For this reason, you will be asked to practice consistent naming in your code.

The following rules apply to naming when writing Python programs for this course:

- When in Python, use `snake_case` only. More specifically, programs, variables, and file names *must* be in lower-case only, and any names that contain multiple words must have those words separated by underscores.
- All Python file names must end with the extension `.py`
- *All* programmer-chosen names within a file, including variable names, must be relevant to the program. This will take practice, but here are some general guidelines to be aware of:
  - **Function names** usually involve verbs, like `count_items()` or `populate_table()`.
  - **Variable names** are usually nouns, like `index` or `user_count`.
  - When naming something, consider: what does it do? What does it represent? How does it fit into the context of the overall program?

- That said, extremely long and wordy variable names become detrimental to a program's legibility. *Always try to keep variable names descriptive but short.*
- The only exception to this rule is that short variable names are allowed when their contexts are very limited, like when naming a counter variable in a `for` loop (i.e. it's OK to say `for i in range(10)` instead of `for counter in range(10)`.

3 Formatting

Python specifically is picky about indentation, and will refuse to run a program that is not properly indented. There are some drawbacks to this, but generally it is a very good thing. Python insists that all *code blocks* be indented with four *spaces* (i.e., not tabs). Content within a code block that should be indented include things like:

- Function implementations
- Content to be executed by an `if` statement
- Content to be executed within a loop

A general rule is that anywhere you see a colon (`:`) in your code, the line(s) underneath it should be indented four spaces deeper until the block is complete. For example:

```
def my_function(username):
    if (username == 'kercheval'):
        print('hey, you're me!')
        start_teacher_mode()
    elif (username == 'student'):
        if (username.arrival_time().is_late()):
            print('L8!')
        print('welcome to class!')
    else:
        print('what are you doing here??')
```

The rules above are enforced by Python, meaning your program will not run if you do not follow them. The rules below are enforced by me, meaning your program will not receive full credit if you do not follow them. They may seem tedious to maintain at times, but it is critical you adhere to them so that your code is readable:

- **Your program must never be more than 80 columns wide.** This is a fairly standard, accepted convention across many programming languages. It ensures that your code can be read on many types of editor and in terminals, and nothing breaks a train of thought quicker than having to scroll horizontally when reading someone else's code.
  - Where to break a line of code that exceeds 80 columns is generally up to the programmer, but it usually is best done before operators, with the new line indented in such a way that it is vertically aligned with variables or statements of the same context. For example:

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

  - When using operations on multiple variables, chain the additional content on multiple lines with the operation leading the next element, instead of following the previous one. For example, do this:

    ```
    the_raven = "Once upon a midnight dreary, "
                + "while I pondered, weak and weary, "
                + "over many a quaint and curious volume "
                + "of forgotten lore"
    ```

and not this:

```
the_raven = "Once upon a midnight dreary, " +
            "while I pondered, weak and weary, " +
            "over many a quaint and curious volume " +
            "of forgotten lore"
```

- **Your program's functions must never be more than 50 lines long.** This helps prevent the reader from getting lost, and it helps you keep your functions concise and on-task.
- Use spaces to pad mathematical operators. This makes complex operations significantly more legible. For example, write `nine = 4 + 5` instead of `nine=4+5`.

4 Bad Habits

Like glitter, bad programming habits are easy to pick up and hard to get rid of. Some languages are very permissive of these habits, including Python. Indeed, many of these habits make programming easier, since they reduce the burden on you as the programmer. Don't be tempted. The dangerous programming practices listed below, while perhaps legal according to Python, will cause debugging mayhem and hideous code for you down the line if you depend on them, and in this class they will result in a reduction of credit on assignments.

- **No global variables! Never!** Global variables are a very easy solution to the classic programming question of "where do I put this variable and how do I pass it around?" However, they make it practically impossible to debug which process might be editing them, and heaven forbid you accidentally pick a variable name in a function that you've already defined globally. I will show you how global variables work in the name of giving you the truth, but I will be vocally upset if you use them in a program.
- **No `while (True)`!** It's is not only lazy, it's a one-way ticket to infinite loop town. Always use a condition to quantify your `while` loops' endings. They make your code safer, more predictable, and easier to read.
- **Be careful with conditionals.** Large blocks of labyrinth-like nested `if/elif/else` statements are a telltale sign of a program that has not been well-designed. Always consider ways to be concise with your conditionals. This is as simple as asking yourself questions like "do I really need `if/elif` in this case, or can I just use `if/else`? Do I even need an `else`?"

5 Idioms and Syntactic Sugar

Programming is full of *idioms*, which are language-specific constructs for representing different actions. It is also full of *syntactic sugar*, which are complex or tedious operations wrapped in simple syntax. Python is a language famous for being particularly "sugary," which is part of the reason why it's so popular. Both idioms and syntactic sugar make writing code quicker and usually more concise, but they can quickly make your code impossible to decipher to anyone who doesn't know them. For that reason, we will generally avoid using idioms in this course, and we will learn the syntactically "sour" equivalent of all sugary expressions first. Some of these expressions, however, are so common in programming that they come up more often than their idiom- or sugar-free counterparts. Acceptable idioms and sweetened expressions in the scope of this course, then, include:

- `n += 3`
  - This represents `n = n + 3`. Writing the operation either way is acceptable in this course.
- `k /= 2`
  - This represents `k = k / 2`. Writing the operation either way is acceptable in this course.

- `if (condition)`
  - This is an acceptable way to check whether `condition` is true, and is in fact significantly preferred to writing `if (condition == true)`, which is redundant.
- `if (not condition)`
  - This is an acceptable way to check whether `condition` is false, but writing `if (condition == false)` is fine as well, since it is not redundant.
- Sugary array processing
  - These are very common in Python programs, and they can make certain programming situations way easier. For example:

    ```
    data = [1, 2, 3, 10, 20, 30]
    for item in data:
        item += 5
    ```

    Is a sugary way of saying:

    ```
    data = [1, 2, 3, 10, 20, 30]
    for i in range(len(data)):
        data[i] += 5
    ```

    You should be familiar with both ways of writing a `for` loop, since there are logical drawbacks to the sugary one, as well as for the sake of being able to read code in other languages, which tend to follow the latter example's syntax (or something even more complex).
- Sugary array evaluation
  - There are a handful of syntactic sugar constructs in Python that are very helpful array evaluation tools, meaning they tell you something about the array's contents. A famous one is the word `in`, which evaluates whether or not a given element is in an array:

    ```
    menu = [ ... some food choices here ... ]
    if 'pizza' in menu:
        print('hooray! Pizza!')
    elif 'pizza' not in menu:
        print('aw man, guess I will have the chocolate cake then')
    ```

    You can generally use these at will. They'll save you a lot of time!

6 Comments

Appropriate use of comments (or *documentation*) is *the* way to help others read your code. Good commenting will ensure that your code is thought out well, that it's maintainable in the future, and that you yourself can read your old programs and understand why you made the design choices you did.

Quality comments are required for the code you write in this class. Lack of documentation in code will be considered a bug, and code that is not properly documented will not receive full credit.

Comments in Python begin with a pound sign (`#`), and every new line in the comment must have a pound in front of it. The syntax is the same for *block comments*, which describe programs, functions, or big ideas, and *in-line comments*, which add a small amount of context to the line of code they're on. For example:

```
# This is a block comment. The code below shows two ways
# to arrive at the number six. Curious minds could probably
# come up with more.
# Below, there are two examples of in-line comments.
six_one_way = 3 + 3             # Addition
six_another_way = 3 * 2         # Multiplication
```

Note that the comments are capitalized and punctuated, and that the in-line comments are a healthy distance away from the code they comment. I expect your comments to look the same. Don't be lazy!

When and where to leave comments is generally up to the programmer, but in this course (and in enterprise software development) we will impose some conventions. Those conventions are:

- All `.py` files define a program. At the top of each file, you must include information about the program defined there, in the following format:

  ```
  # program_name
  # Author: [your name]
  # Date of most recent edit
  #
  # Brief description of what the program is/represents, how it fits within
  #     the project as a whole
  #
  # Function signatures:
  # Signatures (but not descriptions) of class's methods, for example:
  # set_value(new_value)
  # another_function()
  # is_equal_to(thing_to_compare)
  ```

- Similarly, all functions defined within a file must be commented. A function's documentation should be in the following format:

  ```
  # function_name
  # Description of what the function does, *not* how it does it, including
  #     contracts it upholds, assumptions it makes
  #
  # Parameters:
  # parameter 1, brief description for context, if needed
  # parameter 2, brief description for context, if needed
  #
  # Returns:
  # type of and context for any returned values
  #     (say none for void methods)
  ```

- Along with class and method comments, in-line comments are welcome for new "paragraphs" in your code, that is, places in your program where your process changes direction, as well as for places in your code where you perform an operation or calculation that may not be immediately obvious to the reader, where you want to help them to understand your thought process.

Effective commenting, especially in-line commenting, is a delicate art, and it requires you to be able to tell when your comments are useful and when they are clutter. You can generally consider comments to be "the more the merrier" for this course, but if you find yourself constantly leaving comments explaining your work, reflect on whether or not you can change your code's algorithms and structure so that it's understandable without an excess of comments.

7 Notes on linguistic precision

You will find throughout your experience with technology that computer scientists tend to be nitpicky, pedantic grammar freaks. While sometimes annoying and counterproductive (see the Linux vs GNU/Linux debate), this quality is important. Typos, a lack of precision, and incorrect word choice might be inconsequential for small projects, but on large-scale programming tasks, they can seriously hinder a user's ability to understand what's going on, as well as your ability as a programmer to effectively communicate about your work. In this course, therefore, you are expected to be as precise with your English as you are with your Python. This is a broad expectation, but some specifics include:

- Run `aspell check [filename]` on your program prior to submission to catch typos.
- An equals sign by itself is called the "assignment operator" and is pronounced "gets," not "equals." `x = 5` is pronounced "x gets five."
- Two equals signs represents a check for equality and is pronounced "equals." `thing1 == thing2` is pronounced "thing1 equals thing2."
- Using brackets to access an array element is pronounced "sub." `my_list[3]` is pronounced "my_list sub three."