General Coding Guidelines

I spent 30 years in industry writing code. Based on that experience, I've put together some guidelines. If you are creating code that you'll never read again, the guidelines aren't so important. However, write-only code is rare. For example, if you need to debug your code, you'll need to read it. Someday, you may need to do something similar so you'll want to reuse part of your code, so you'll need to read it. Or, after a few years, you may need to upgrade your code. Or worse – somebody else may have to pick up your code and it will reflect badly on you if they can't figure out what's going on. So, you should always plan to need to read your code again. Therefore, you should make it as easy on yourself as possible by following these guidelines.

Comments:

Comments are absolutely required. When a variable or constant is declared or assigned for the first time, there should be a comment explaining what it represents. Sometimes the comment will be simple, sometimes it may take a few lines. Get in the habit of writing good comments so that when you update code you haven't looked at in a few years you can quickly be reminded how it works.

A general comment at the top of the program should explain what it does. This should NOT include details of the inner workings, just a general outline. The detailed comments belong with the code which implements those details. That is, prior to a for loop, describe what the loop is doing. Make sure you include what the iterated variables represent. And don't say that you're iterating through all the items as that is obvious. Say why you are iterating through them!

Comments should add value. If you have a line of code that reads, "a = b", don't put in a comment that says, "assign b to a." That's pointless. More interesting is why you are assigning a value to a. So, a better comment might be, "store a copy of b for later reference."

Be warned – something that is obvious to you when you write the code may not be so obvious later. Even the next day you might find yourself wondering what you were thinking! So, if you code something in a clever way, comment it immediately so you don't forget.

Most languages provide the ability to add comments in two ways: single-line comments and multi-line comments. In Python, the **#** symbol starts a single-line comment. The single-line comment can start at the beginning of the line or after some code: any code on the line before the **#** is not part of the comment and will be executed.

```
# This is a comment at the start of the line
mileage = distance / gallons  # This is a comment that starts after some code
```

In Python, multi-line comments start and end with triple double quotes: “””. Multi-line comments are dangerous since it is not always clear where they start and end, and when code appears within multi-line comments it isn't necessarily obvious that it won't execute. Therefore, multi-line comments should not be used. Instead, just start each line with **#.**

```
# Here is a comment
# that is taking up
# more than one line
```

By putting the **#** in front of every line of the comment, it is made obvious that the entire block is a comment.

When there are major sections of code that you want to make stand out, or if you want a banner at the top of your file, you can use a line of **#**s.

```
#############################################################
# This next section implements a really cool feature…
# The important things to know are…
# … and so on until you're done with whatever important information you want to convey
#############################################################
```

White Space:

The use of white space is very important when it comes to reading code. There are two types of white space: horizontal white space and vertical white space. Horizontal white space spreads things out on a line to make them easier to read. Here are two examples:

```
myvar=32+myfunc(var1,var2)/average
```

versus

```
myvar = 32 + myfunc(var1, var2) / average
```

The second is much easier to read!

Vertical white space is the blank line added between lines of code. The blank line should be used to separate functions. The blank line should also be used to separate sections of code. For example, when using a loop, there should be a blank line, followed by any set up required for the loop, followed by the loop. After the loop there should also be a blank line. This makes the loop stand out!

Blank lines should also be inserted around sections of code that perform important tasks. This includes separating lines of code within lops. Blank lines can make your program much easier for you to review. And, it makes it easier for graders too! The example code in your **Computational Physics** text makes excellent use of blank lines. (However, the example code in the text could use more comments…)

Variables:

Variable names should be lowercase. The only exception may be if you want to use initial caps for interior sections of the name. For example, myvarforvelocity is ok, but so is myVarForVelocity. Long variable names are ok as long as they are conveying meaning without making things silly. That is, the prior example would be better as simply velocity, or myVelocity, or even myVel.

Rather than interior initial caps, I prefer the use of underscores. So, the above could by my_velocity. Underscores do make longer names easier to read and avoid the errors described in the next paragraph.

I'd suggest all lowercase for variable names since the names are case sensitive and you may forget which letters were uppercase or lowercase. Depending on how the variable is used, using the wrong case might not generate an error so such bugs are hard to find.

Avoid global variables – they are very dangerous because it's hard to track where they are being used. This can make debugging a nightmare. (It is sometimes required that variables be global, at least as far as a few functions are concerned. However, for the assignments in this class, it will not be necessary unless otherwise noted.)

Constants:

Constant names should be in all caps, e.g. THISVALUE. This makes constants easy to distinguish from variables. Constants should be defined at the top of the file in which they are used if they are used in more than one function or should be defined at the top of the one function where they are used. Note that it is ok, and often wise, to define constants at the top of the file even if they are only used in a single function as you may wind up needing that constant elsewhere.

A constant should NEVER be defined in more than one place. The risk is that you'll need a different value for that constant and won't remember to change it everywhere. For example, you may need to add more significant digits or find that you've been using the wrong value. A neat trick is to use an "easy" value for your constant that will result in an obvious answer to help you debug. (For example, start with the speed of light equal to 1 rather than the actual value.) Once things are working for the easy case, you can assign the real value to the constant.

It is best for two, or more, constants to have the same value if they represent different things or are used in different ways. This allows the different constants to have meaningful names!

While it is occasionally right to use an actual value in your program, it is almost always better to use a constant. That is, don't assign temp = 98.6 when you should define AVG_NORM_TMP = 98.6 and then assign temp = AVG_NORM_TMP. While this may seem like overkill, it does make maintenance easier, especially if you use this value in multiple places and decide to change it from 98.6 to 98.8 due to updated research.

Name Collisions:

When writing software, your program might be combined with code written by others. (This will be the case in Python as we'll be including packages and modules written by others.) As a result, it is possible that the same variable, constant, function, or other names you've chosen have been used in the other code. Care should be taken to ensure that the names you have chosen don't collide with these other names. Most teams have guidelines for naming conventions. That won't be a significant concern for this class. However, be aware that if something isn't working like you'd expect, you might have a naming collision. Python won't warn you about this unless the collision results in an error due to improper usage.

Rules of Precedence:

In mathematics, there are rules which determine the order in which expressions are evaluated. For example, 2**4+8/4 is 18 because exponents are evaluated before multiplication or division, and multiplication and division are evaluated before addition and subtraction. But how sure are we that the author wanted this? And what about all of the other binary operators? Do you really remember all the rules? The answer is to use parentheses to make things clear: ((2**4)+8)/4. In this case, our original interpretation was wrong and, by adding parens we fixed a bug! This was a very simple example using obvious operations in an obvious manner. However, we will be doing a lot of complex math and glancing at a formula won’t necessarily make it obvious the order in which things are, or were supposed to be, evaluated. So, use parentheses to make sure it’s clear! If you get in the habit of using parentheses, you’ll avoid grief. Remember: parens are free!

Also, you should avoid putting too much computation on a single line. When possible, break a very complex formula into pieces and build them up. Here’s a simple example:

```
new_val = ( ( old_val – adjust_val ) ** 2 ) – ( ( older_val – old_val ) ** 2 ) – epsilon
```

could be broken up like this:

```
new_val_first = ( old_val – adjust_val ) ** 2
new_val_second = ( older_val – old_val ) ** 2
new_val = new_val_first – new_val_second – epsilon
```

Warning – do exercise care when breaking down equations so you don’t run into to magnitude issues. That is, make sure operations are done in the same order and that you aren’t losing information when the magnitudes of some operands are much bigger or smaller than others.