EN.540.635 “Software Carpentry”

Functions and Classes

When programming, simplifying code is an unspoken “must”. It is quite common for new programmers to simply copy-paste large segments of code multiple times to save re-typing them, or to use long if-elif-else chains to handle complex logic. Both of these approaches are prone to a wide variety of issues, but the most common of which are as follows:

1. Copy-paste errors can occur in which one segment of code that is to be repeated may have a typo. This can be difficult to debug, as there may be many instances of the same lines of code, with a simple error (such as an accidentally indented statement) being hard to isolate.
2. If the logic of your code changes, and you had chosen to instead copy-paste segments of code a large number of times, this means each instance must also be updated. It is easy to overlook one instance, and have an incorrect code running.

The easiest fix to this is simply to use a *function*. Functions are blocks of code that can be accessed repeatedly, and as such can greatly simplify one’s workflow and minimize errors.

Functions

While we have already seen and used functions quite a lot, we review them here. The simplest definition of a function is a segment of code that, together, performs some task (such as calculating a value, or processing data). The easiest example of this is to look at math. In mathematics we can define a line as $y = mx + b$. If we wish to define a function $f(x)$ that will return the corresponding y given an x, we could do so (in Python) as follows:

```python
def f(x, m=3, b=0):
    '''
    The corresponding y to an x for the standard line (y = mx + b).
    In this case, we default m and b to 3 and 0 respectively.

    **Parameters**

        x: *float*
            The x for which we should calculate y.
        m: *float, optional*
            The slope of the line.
        b: *float, optional*
            The y-intercept of the line.

    **Returns**

        y: *float*
            The y value corresponding to the given x, m, and b.
    '''
    x, m, b = float(x), float(m), float(b)
    return m * x + b
```

Now, there are many things to dissect here! First and foremost, we will discuss the function definition. Python functions all begin with *def*. This simply signifies to the python interpreter that what follows next will be a function. Afterwards, we name our function (f), and specify the arguments it expects. This is where things get interesting! By default, we require the user to specify a minimum of one *argument*, and up to two additional *keyword arguments*. An argument is simply the input to a function, while a keyword argument means that a default exists for said argument, and that it does not need to be specified (assuming the default value is okay). The benefits of keyword arguments are many, but one is that they allow you to handle many situations within a single function. In this case, if you simply want to offset the line, you could do the following:

```
y = f(3.0, b=10.0)
```

Notice how I specified b, but not x. This is because of two things: (1) b is a keyword argument and (2) b does not follow x in the order of the function definition. To understand this a little better, let's instead change the slope of the line, but leave the y-intercept the same:

```
# One approach
y = f(3.0, m=5.0)
# Another approach
y = f(3.0, 5.0)
```

In this case, because our function definition has m follow after x, we can either specify or not specify that m is a keyword! In general, if you are assigning a value to a keyword I would recommend you to specify it. This is because if the function is ever edited such that the keyword order changes, then if the keywords are not specified, you would be incorrectly assigning variables. This brings me to my last statement on keywords, the following two are equivalent (due to specification of keywords):

```
# Both of these do the same thing!
y = f(3.0, m=5.0, b=10.0)
y = f(3.0, b=10.0, m=5.0)
```

Now, it is important to note the *docstring* I have added to my function. The docstring is the large comment at the start of my function (wrapped in a triple quote). It specifies the function's purpose, any additional things of note (such as paper references, or failings of the function), the parameters the function expects, and what the function will return. This is important, as we even specify the data types of our function! Further, because of how helpful python is, we can access this information at any time by calling the *help* function! This help function also exists for things like numpy functions. Try out the following (if you are doing this in the Python interpreter, you do not need to use the print statement):

```
import numpy as np
print(help(np.std))
```

Finally, the function will always *return* something when it is done running. In fact, you can prematurely end a function by calling the return statement. In our example, we return the value of interest, making it such that we can use our function to calculate values and have them return. In other situations though, we may not need to print anything. For example, imagine the following function:

```
def printAuthors():
    '''
    This function will simply print information on the authors.

    **Returns**

        None
    '''
    print("Author: hherbol")
    print("Date: Feb 25, 2019")


printAuthors()
```

Notice how I do not even specify a return statement in this example. That is because there always exists an implicit return of *None* if no return statement is encountered when a function ends. This is also why I do not assign the return of the function to a variable, as in essence I would simply be assigning *None* to the variable.

One final thing to note - you can have as many return statements within your function as you want! However, as soon as one is run, the function will end.

Classes

Classes are objects that greatly simplify the life of a programmer. They stem from an idea called Object Oriented Programming (OOP), and come with tennants that are ideally followed:

1. Abstraction - The idea that the logic need not be known. An example of this is importing the function *Image.open* from *PIL*. We do not care about the details behind decoding images, we simply want the end results.
2. Polymorphism - The idea that our approach must be robust and extensible when it can be. For example, using *Image.open* we were able to open .png files, .jpeg files, and more! All within the same object.
3. Encapsulation - The idea that our object is well "contained" and can be easily setup and moved. Essentially, a well encapsulated object can be readily installed and used without the need to compile a variety of dependencies alongside it. Further, encapsulation has the added benefit of being more "idiot-proof", hiding away the sensitive variables from the grubby paws of an end-user.
4. Inheritance - The idea that we can easily extend our objects, and make the en masse. This is best understood by appreciating the video game Minecraft, in which the many different "blocks" in the game can be comprised of a single object with slightly different properties.

It shouldn't be hard to appreciate the power of a class and OOP, and as such I will focus less on convincing you of its usefulness, and more on implementing one in Python. See the example on the next page. First, every python class starts with the *class* statement. This simply tells python that, what follows (and is indented) is part of a class object. We then name our class MathyMath. Note, it is common for classes to be named in **C**amel**C**ase (ex. **M**athy**M**ath). Afterwards, we acknowledge that we have docstrings! Specifically, each function (sometimes called a *method* when it is part of a class object) that needs one has one, and the docstring for *__init__* follows the class initialization. Touching on the subject, classes have special functions and variables, commonly written with two underscores before and after the names, that give them default features (you may have noticed I skipped the "object" part, I will come back to this later). Some common default features of a class are:

- *__init__* - The init function, which is run on each initialization of an object. That is, every time you make a new object (in this case by doing *newObj = MathyMath([], 0.0, 0)*), the init function is run.
- *__repr__* - The representation of the object. This is more for debugging, and as such should be as detailed as possible!
- *__str__* - The string representation of the object. This is for displaying, and should be "pretty"/"useful" to a user.
- *__lt__* - A function that will let you know if this object is less than another object. For example, assuming you have two objects named *A* and *B*, you could do *A* ¡ *B* and the *__lt__* function will decide if this is *True* or *False*. Note, others like this exist, I will not write them all out here.
- *__del__* - A function to elaborate on what should be done when this object is deleted.
- *__hash__* - A way of uniquely describing this object. This is useful to have, as you can then do things like *if A in [X, Y, Z]* (that is, check if an equivalent object exists in a list).

When using a class object, you will notice the repeated use of *self*. Simply put, you can save functions and variables to your object, and access it elsewhere using *self*. However, *self* is only used within the object itself. When accessing the variables and functions from outside the object, you simply call the object by name (in this case, the named objected is initialized as poly). For example:

```
# Generate a new polynomial object
poly = MathyMath([3.0, 2.0, 1.0], 0.0, 3)
# Edit the default_coeff variable
poly.default_coeff = 2.0
```

```python
class MathyMath(object):
    '''
    This is an example class that does some maths.

    **Parameters**

        coeffs: *list, float*
            The list of coefficients for some the n-dimensional polynomial.
        default_coeff: *float*
            A default coefficient for values outside of n-dimensions.
        dim: *int*
            The dimensionality of this polynomial
    '''

    def __init__(self, coeffs, default_coeff, dim):
        self.dim = dim
        # Appropriately handle default_coeff
        if len(coeffs) < dim:
            fix_len = dim - len(coeffs)
            coeffs = coeffs + [default_coeff] * fix_len
        self.coeffs = coeffs
        self.default_coeff = default_coeff
        self.powers = list(range(self.dim))[::-1]

    def __repr__(self):
        '''
        The debugger's representation of the object.
        '''
        s1 = "coeffs = " + str(self.coeffs)
        s2 = "default_coeff = " + str(self.default_coeff)
        s3 = "dim = " + str(self.dim)
        return '\n'.join([s1, s2, s3])

    def __str__(self):
        '''
        The pretty representation of the object.
        '''
        return " + ".join(["%.2f * x**%d" % (c, i) for i, c in zip(self.powers, self.coeffs)])

    def f(self, x):
        '''
        Given an x, return the corresponding value for the polynomial.
        '''
        return sum([c * x**i for i, c in zip(self.powers, self.coeffs)])


if __name__ == "__main__":
    # Make an instance of the MathyMath object, and save it in a variable
    # called poly
    poly = MathyMath([3.0, 2.0, 1.0], 0.0, 3)
    # This will access the __str__ function
    print(poly)
    print(poly.f(4))
```