EN.540.635 "Software Carpentry"

Python Modules

At this point, we have used Python modules quite extensively within the course. Prime examples being *numpy* and *PIL*. The benefits of using a module are vast, with the simplest argument for them being to allow for a foundation of code so as to not require "reinventing the wheel" each time. In this document, we will discuss how to write our own modules, how to set it up so that Python may "access" them, and, as usual, how to apply proper documentation (via docstrings) so as to not forget how to use them.

Writing a Module

When writing a module, we first make a folder. For this demo, we will write a module called *SoftwareCarpentry* (note, there are no spaces in a module name). As such, we will begin with a folder, somewhere on our computer, named *SoftwareCarpentry*. Within this folder, we must specify to Python that this is a module. This can be done by making the empty file *__init__.py* for now. Next, we can add functionality to our module. We do so by making new files and folders in the *SoftwareCarpentry* folder with necessary code. To begin, let's make a folder called maths, with two files in it: another *__init__.py*, *simple.py*, and *complex.py*. Within *simple.py* we write the following code:

```
'''
This file contains simple math functions as an illustration of how to write
a python module.
'''


def avg(a, b):
    '''
    The average function.  This will return the average of two floats.

    **Parameters**

        a: *float*
            A float to be averaged with another.
        b: *float*
            A float to be averaged with another.

    **Returns**

        c: *float*
            The average of the two input floats.
    '''
    return (float(a) + float(b)) / 2.0
```

Within *complex.py*, we write the following code:

```
'''
This file contains more complex math functions as an illustration of how to
write a python module.
'''


def avg(a, b, weight_a=1.0, weight_b=2.0):
    '''
    The average function.  This will return the average of two floats.
    However, each float is weighted by some value.

    **Parameters**

        a: *float*
            A float to be averaged with another.
        b: *float*
            A float to be averaged with another.
        weight_a: *float, optional*
            A weight for the a parameter.
        weight_b: *float, optional*
            A weight for the b parameter.

    **Returns**

        c: *float*
            The average of the two input floats with appropriate weights.
    '''
    return (float(a) / weight_a + float(b) / weight_b) / 2.0
```

Now, within the *SoftwareCarpentry* folder, we will make a file called *constants.py*. This file will hold the following:

```
PI = 3.14159265359
```

We now have the foundations of the module to test out. However, before doing so we will need to tell our computer how to do so.

Accessing a Module

In all operating systems, there exists something called Environment Variables. Many such variables exist, and they allow a computer to handle low-level operations (mainly, they specify default parameters, as well as folders and file locations). The one of interest to us is the *PYTHONPATH* variable, which allows Python to detect available modules on a computer (outside of the python's installation folder). I will describe how to do this for the various operating systems; however, **be aware that incorrect editting of environment variables can break how an operating system behaves, potentially renduring it unusable if not corrected**.

Linux

If using a Linux machine, one must first identify the terminal's *resource file* that is being *sourced*. Most of the time, shells will be running on *BASH*, and this resource file will be named *.bashrc*. No matter what though, the file will be hidden (as identified by the . starting the file name), and available in the user's *home directory* (which can be identified in a terminal as $\sim$). Thus, changing this can be done via a terminal text editor, such as vim, or by appending to the file directly with the >> operator. The goal now is to update the *PYTHONPATH* variable with the folder containing our new module. Assuming our module path is */home/henry/class/SoftwareCarpentry*, we want to add */home/henry/class* to the *PYTHONPATH* variable. Further, we only want to *append* it to the variable (that is, we do not want to overwrite it). The actual command to do so is:

```
export PYTHONPATH=/home/henry/class:$PYTHONPATH
```

Note the lack of */SoftwareCarpentry* in the path (as, once again, we want to point to the folder holding the module, not the module itself), and also note the *:$PYTHONPATH* at the end. This last part means that we want to keep whatever *PYTHONPATH* has been already assigned to, but pre-append our new path to it.

Now, if you feel comfortable doing so, simply use a terminal text editor to add that to the bottom of your resource file (likely *∼/.bashrc*). Otherwise, you can append the line to the file using a command such as:

```
echo "export PYTHONPATH=/home/henry/class:$PYTHONPATH" >> ~/.bashrc
```

Now, when you are done, you must *source* the updates you have made. This can be done by (1) restarting the computer, (2) closing and opening any applications that you are trying to interface with python, or (3) running the following command in the terminal that you want updated (with the appropriate resource file being specified):

```
source ~/.bashrc
```

MacOS

The approach on MacOS will be very similar to the Linux approach; however, the resource file may either be *∼/.bashrc*, *∼/.bash_profile*, or *∼/.zshrc* depending on what shell you are running on. You must identify which one already exists on your system first before proceeding. When you have identified which one exists, you may then proceed with the explanation laid out in the Linux section.

Windows

If you are using the Windows Subsystem for Linux (WSL), please follow the instructions laid out in the Linux section. Otherwise, if you are running python directly through windows (say, via Sublime, or through the command prompt), then you must update the *PYTHONPATH* another way. This can be done quite easily, thanks to the use of a Graphical User Interface (GUI).

1. Go to the start menu and search "environment". You should see something called "Edit the system environment variables" appear. Click on it.
2. On the bottom right should be a button called "Environment Variables...". Click it.
3. At the top, you should see a list of the User variables. First check to see if *PYTHONPATH* is already there. If so, click it and click *Edit...*. Otherwise, click *New...*. **NOTE - YOU SHOULD AVOID EDITING THE SYSTEM VARIABLES!**
4. **IF YOU CLICKED EDIT -** Windows wants you to separate the folders with a semicolon (;). Thus, add the folder where the *SoftwareCarpentry* folder exists (do NOT point to the SoftwareCarpentry folder itself, but the folder preceeding it). When you have made the appropriate changes, click OK and close.
5. **IF YOU CLICKED NEW -** For *Variable name* put in *PYTHONPATH*, and for the *Variable value* put the folder that holds the SoftwareCarpentry folder. Once again, you are NOT putting in the SoftwareCarpentry folder path, but instead the path of the folder that holds it. When done, click OK and close.
6. At this point, you may either (1) restart your computer or (2) close and re-open any applications that you want to use your module in. This allows the already loaded python to reload with the new path.

Testing

If everything went well, you should now be able to access your own personal module! Try the following:

```
import SoftwareCarpentry as sc
```

If you get no response, then you have succeeded. Otherwise, you may have gotten an error such as *ImportError: No module named SoftwareCarpentry*. If this is the case, go back and make sure you have followed all the instructions properly (have you tried turning it off and on again?).

So, now you have a module and it seems to be working. Try the following two imports out:

```
import SoftwareCarpentry.maths as math
from SoftwareCarpentry import maths as math
```

As you can see, both options appear to work; however, if you try to access *simple* or *complex*, you find things get a bit wonky:

```
import SoftwareCarpentry.maths as math
print(math.simple.avg(3, 6))
```

Weird... you get an *AttributeError* claiming that the module doesn't know what *simple* is. Try this now:

```
import SoftwareCarpentry.maths as math
import SoftwareCarpentry.maths.simple
print(math.simple.avg(3, 6))
```

Now it seems to work! But that's annoying and cumbersome. We can circumvent this requirement with the *__init__.py* files. Specifically, if we wish for the first example to work, we need only add the following line to the *__init__.py* file that is within the maths folder:

```
import SoftwareCarpentry.maths.simple
```

Now, if we try again, the first example works. The *__init__.py* file works similarly to the *__init__* function of a class object; however, instead of running everytime an object is initialized, it runs every time the corresponding module is loaded. In our case, we get two *__init__.py*. So, which one runs when? Well, let's add *print("MAIN")* to the *__init__.py* in the SoftwareCarpentry folder, and add *print("MATHS")* to the *__init__.py* in the maths folder. Now, lets do the following:

```
import SoftwareCarpentry as SC
```

You should see *MAIN* print to the screen. Now, if you do:

```
import SoftwareCarpentry.maths as math
```

You should see *MATH* print to the screen.

Documentation

Now, let's assume we forgot how the *avg* function works. Because we have appropriately handled our docstrings, we can do the following:

```
import SoftwareCarpentry.maths.simple as simple
print(help(simple.avg))
```

And you will get all the details we had written out beforehand! Note, because of how standardized python is in regards to docstrings, this is near universal! Try the following out:

```
from matplotlib import pyplot as plt
print(help(plt.plot))
```