Lillian Wright *May 3, 2020*

Beginner's Guide to Coding: No Experience Necessary

Abstract

This tutorial consists of a brief overview of some basic computer knowledge and jargon, followed by a tutorial for creating a plot of star formation rate vs stellar mass taken from galaxies in the 3D-HST Treasury program. In addition to this, I am hoping to tackle some of the most common mistakes and obstacles encountered by undergraduate students with no background in computer science or coding language whatsoever. I start by covering the basic skills, such as orienting oneself in a computer system using a terminal, defining and using different files and directories, and knowing the difference between a terminal and a python shell. I will then demonstrate different simple plotting techniques in a Jupyter notebook, with an example plot which contains a sub-panel with color-coded data.

Introduction

Learning to code may seem like a daunting and overwhelming task to tackle head-on (which, let's be honest, it sometimes is). Granted, it is a frustrating process with a steep learning curve and you will inevitably make many mistakes. But the reward when you are able to execute code on your own is extremely gratifying. In this tutorial, I will attempt to help you navigate some of the more common mistakes and misconceptions that first-time coders experience. The purpose of this tutorial is to walk you through some of the basics step-by-step to leave no room for misconception.

First things first: the best way to learn is by doing! Let's start by locating and opening a terminal. Depending on the computer, this could be in a couple of different places so I will go over two separate scenarios (Mac vs Windows):

<u>Mac:</u>

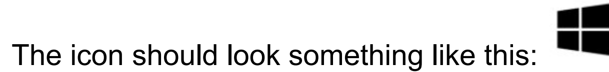
The terminal on my own macbook is typically located at the bottom right of my dock and looks like a little winky face missing an eye >_<

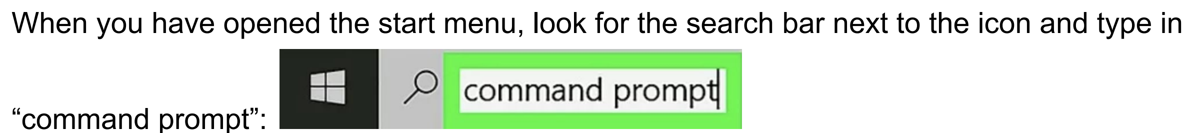
If you have *never* opened a terminal on your computer, you may have to open up your launchpad which should be at the bottom left of your computer screen and is depicted with this symbol:

When you have opened your launchpad, type “terminal” into the search bar at the very top: your terminal icon should be the first to pop up:

Windows:

First go ahead and find your computer’s start menu at the bottom left of your desktop.

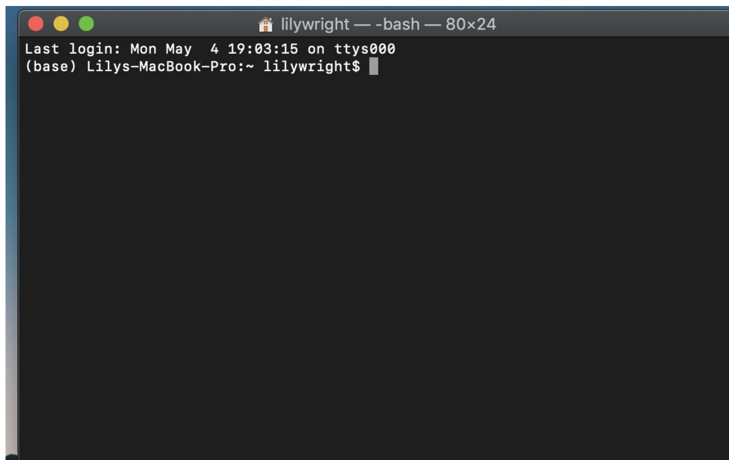
The icon should look something like this: 

When you have opened the start menu, look for the search bar next to the icon and type in “command prompt”: 

The first item that comes up is your terminal icon, which for windows looks like this:

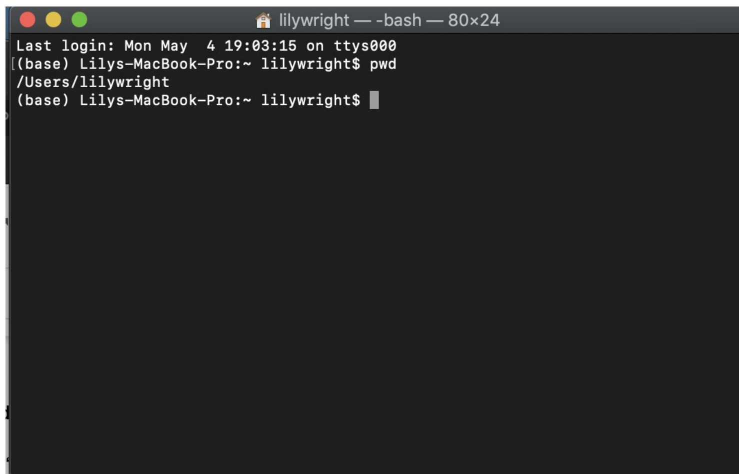
Now that you have located your terminal, let's go ahead and use it!
One important thing you can use your terminal for is orienting yourself within your own computer.
In other words: you can directly ask your computer “Where am I?”
To do this, simply type `pwd` into the command line and hit enter (this stands for “print working directory” and will tell you exactly where you are)

For example, when I open up my terminal by clicking on the icon, it looks like this:



Notice there is a text cursor next to the second line titled (base)
This is known as the *command line* and is exactly what it sounds like: *where you type in your commands.*

When I ask my computer to “print working directory” It looks like this:



My computer then answered my question to “where am I?” on the line directly below the original command: “/Users/lilywright” .
This tells me that I am currently in a folder titled “lilywright” in a directory titled “Users” in my computer- pretty neat huh? This is also known as my *home directory.*

As you become more experienced with using your terminal, you can ask more complex questions or even navigate through your computer by creating different files and directories. Think of your terminal not only as a compass but also a means of transportation to get to wherever in your own computer you want to be. All you need to know is your destination “address” (i.e., the file path name/location).

For example, say you have downloaded a set of data that you want to use to create a graph. Naturally, this data set is now in your “Downloads” folder. But if you are not located in your downloads folder when you attempt to read your data, your computer won’t be able to find it. Think of it like trying to find your bed in your kitchen, or a sink in the living room- it doesn't make sense right?

For beginners, it might be a little easier to figure out exactly where you are in your computer and where you want to go using a Jupyter notebook. We will talk about Jupyter Notebook in more detail briefly, but for now always keep in mind that your terminal is available at any time to help you orient yourself.

Where can I learn to “speak” computer?

For beginners with no background in computer language, I recommend checking out *Code Academy* and *Data Camp*:

- Code Academy: https://www.codecademy.com/
- Data Camp: https://www.datacamp.com/

Both of these sites offer a limited number of *free coding lessons*- yay!
These lessons will really help you get the feel of how to interact with your computer through the coding language known as *Python*.
These courses will also help you gain some basic knowledge and confidence as you will be guided through all of the lessons in a python shell. It is good to get familiar with the shell format as this is what you will be working on once you are ready to download the software

You can also find several Python coding tutorials similar to the one you are currently reading in the *Astronomy Tutorial Repository* created by the Whitaker Research Group. These tutorials cover topics such as writing a Python script, debugging your code, gaussian fit, and much more.
Astronomy Tutorial Repository:
https://www.astrowhit.com/astronomy-research-tutorial-repository

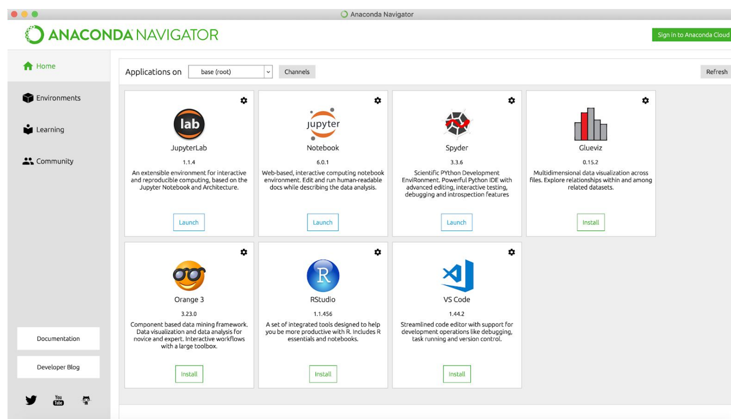
Downloading Anaconda

After you have taken some online classes, the next step would be to download Anaconda.

***An Important Distinction*: *Anaconda* is the name of a software that is a Python distribution platform that you are about to download. *Python* is the name of the language you are learning to code in. There are many other coding languages you may have heard of, such as Java and C. We will be using Python because it is user-friendly as well as being most convenient for our coding purposes. Moreover, in Python there is a large library of astronomy-specific codes that you may find broadly helpful for your own research.

To Download Anaconda: https://www.anaconda.com/products/individual

1) Follow the installation instructions on the website and you will see a little green icon in your "Applications" folder.
2) Go ahead and click on it, you will then be brought to the "Anaconda Navigator" which presents many different options to choose from.



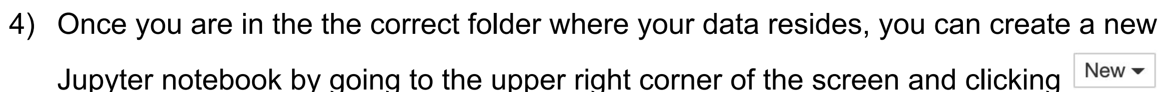
Today we will be working in *Jupyter Notebook* but it is important to note that there are a lot of other useful programs included in this package, including *Spyder* (this program combines a terminal with a text editor so that you can do all of your coding in one place) as well as *Glueviz* (this program is typically used to create complex plots).
**Note*: To see a nice tutorial about using *Glueviz* to create some nice U-V/V-J rest-frame color diagrams, click here

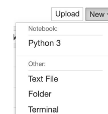
3) For now, click on the *Jupyter Notebook* "Launch" button to launch the program!

The first thing you will see is the Jupyter "Home Page"

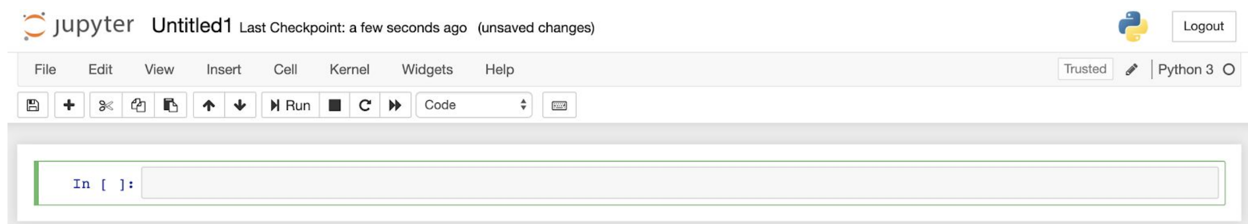
This home page lets you choose where in your computer you want to open a notebook. Remember earlier I mentioned that it might be a little easier for beginners to orient themselves in Jupyter rather than with a terminal? This menu makes it a little easier for you to determine where you are and make sure you open a notebook where you want to.

If, for example, you have downloaded a data set that you want to create a plot with- naturally it will be in your "downloads" folder as listed above.
Because you cannot access your bed while standing in your kitchen, you will need to go into the "Downloads" folder to access the data you want to use.
That being said, it is generally good practice to move the data into a new more permanent location within a research directory. I recommend taking a few moments to organize your project and data by setting up a directory structure. For example, I have defined a folder named "Research" where I keep all of my research data. This will save you time later, I promise.

4) Once you are in the the correct folder where your data resides, you can create a new Jupyter notebook by going to the upper right corner of the screen and clicking 

then selecting "Python 3" from the dropdown menu 

. Your new notebook should look something like this:



**Note:* Don't forget to name your notebook so you don't lose it!

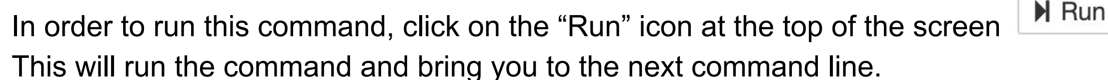
Importing Packages

Let's import a few packages that will help us read in our data and make some nice graphs. Some common packages you will be importing almost every time you open a notebook are `numpy` and `matplotlib`.
`numpy` is a package which helps processing number arrays and `matplotlib` is handy for making plots.
Just like how your computer cannot locate your data if you don't tell it exactly where to look, you also need to load and define packages to be able to use them.
If ever you get an error that it doesn't know what package you are trying to use, your first line of defense is to make sure you have loaded/defined this package correctly.

To import the `numpy` and `matplotlib` functions: simply type in your command line:

```
In [ ]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline
```

In order to run this command, click on the "Run" icon at the top of the screen



This will run the command and bring you to the next command line.

Reading Data

Now that you have imported these packages, you can start by "reading in" your data.

**Note*: Reading data incorrectly is one of the more common problems that may cause difficulty later on in the Python script. It is imperative you take a little bit of extra time to ensure that this step is completed correctly.
A great tutorial that goes into more detail about the process of reading in data can be found here, but for now I'll just give you the brief overview.

You will first want to identify which type of file you are reading in (this will typically be something along the lines of an ascii, txt, fits, or .csv files)

For the purposes of this tutorial, let's say you are trying to read an *ascii* file (otherwise known as a text or *txt* file because the data is typed out and stored in a text format).

You can determine file type by looking at the last three letters after the period of the file name (assuming it was named appropriately).
For example, the file we will be looking at in this tutorial (which can be accessed here) is named "3dhst_whitaker14_sfr.dat.txt" which is a *txt* or *ascii* file because it ends with .txt
This may seem self explanatory, but it is really important that you know which type of file you are trying to read, otherwise your computer will either read your data incorrectly if at all.

In the case where we are reading an ascii file, we will need to import one more package from *astropy*. To do this, type in the command line:

```
In [2]: from astropy.io import ascii
```

Now, let's tell our computer where to actually locate our file.
Remember, for this example we know that we saved our data in our "Downloads" folder, so to access that data we have to tell our computer to look in that folder. We do this by defining a *path* for our computer to take to reach our data, which in this case would be "Downloads"
If you have already opened your notebook in your "Downloads" folder you just need to define your current location:
To do this, simply type:

```
In [5]: catalog_path="./"
```

If for some reason you are not located in the folder where your data resides, you can define the full pathname:

```
In [14]: catalog_path="Users/lilywright/Downloads"
```

To specify even further, we must also define the name of the file we are trying to locate.
At the same time, we will use the *ascii* feature we imported from *astropy* to read our data as a table by defining three factors: *data start, header start, and delimiter.*

```
In [4]: whitaker = ascii.read(catalog_path+'3dhst_whitaker14_sfr.dat.txt', data_start=1,header_start=0,delimiter='\t')
```

There are a few important things to notice about this command:

- First, I defined my entire data set as "whitaker" to save time and typing.
- Second, I told my computer exactly where to find the file by typing *catalog_path* (which we have already defined as our "Downloads" folder) + "3dhst_whitaker14_sfr.dat.txt" (which is our file name)

- Third, I indicated where and how to read the data by defining the *data start, header start and delimiter.* The *data start* defines the line data actually starts on (these files often have written comments in the first few lines). Similarly, *header start* defines where the header information is located. Finally, *delimiter* is the spacer between data columns (see below for definitions).

To better understand how to define these three variables, let's open up our file and take a look:

```
jupyter 3dhst_whitaker14_sfr.dat.txt 02/26/2020                                   Logout
File  Edit  View  Language                                                    Plain Text

 1 zmin    zmax    lmass   SFR eSFR    L_IR    eL_IR   L_UV    eL_UV   L_IR_avg    eL_IR_avg   L_UV_avg    eL_UV_avg   x   xerr
   beta
 2 # from Table 2 in K.E. Whitaker et al. (2014)
 3 #
 4 # All star formation rates and luminosities are logarithmic, with units solar mass per year and
 5 # solar luminosities, respectively.  L_IR and L_UV are measured from median stacks, whereas
 6 # L_IR_avg and L_UV_avg are measured from an average stack.  beta is the UV continuum slope.
 7 # Please see Whitaker et al. (2014b) for more details on the stacking analyses and assumptions
 8 # when deriving these measured UV+IR SFRs.
 9 #
10 0.5 1.0 8.4 -0.60   0.50    7.69    0.63    9.01    0.01    -99 -99 -99 -99 9.04    0.01    -1.76
11 0.5 1.0 8.7 -0.38   0.11    8.83    0.09    9.15    0.01    -0.37   0.11    8.75    0.11    9.18    0.01    -1.74
12 0.5 1.0 8.9     -0.20   0.05    9.09    0.05    9.32    0.01    -0.19   0.05    9.05    0.05    9.34    0.01    -1.73
13 0.5 1.0 9.1 0.05    0.03    9.59    0.03    9.47    0.01    0.09    0.03    9.64    0.02    9.49    0.01    -1.67
14 0.5 1.0 9.3  0.23   0.02    9.85    0.02    9.59    0.01    0.27    0.02    9.92    0.01    9.61    0.01    -1.49
15 0.5 1.0 9.5 0.45    0.02    10.17   0.01    9.70    0.01    0.53    0.02    10.28   0.01    9.73    0.01    -1.32
16 0.5 1.0 9.7 0.65    0.02    10.45   0.02    9.76    0.01    0.75    0.02    10.57   0.01    9.81    0.01    -1.08
```

As you can see, our *header* names for our columns begin on line 1, however, if you were to open this up in a terminal, *this line would present as line 0*
*******Rule of thumb*: subtract 1 from the row number that is listed in Jupyter Notebook.

You may now logically assume: "I can see that the data points start at line 10, so shouldn't you define datastart=9 *(10-1)*?"
This is logical thinking, however, notice that whoever created the file put a # symbol on each line of written commentary (lines 2-9)

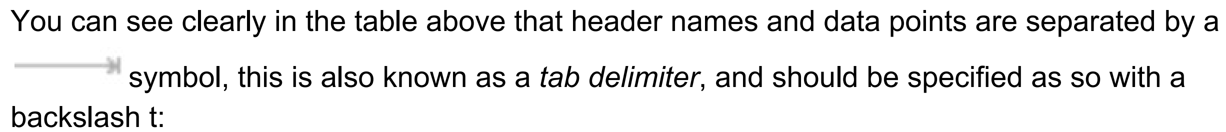
Putting this # symbol in front of any line of code automatically tells Python *not to run that line of code*. For this reason, the # symbol is very useful for leaving written commentary on your code for yourself and others to view.
However, this also means that your computer will read line 10 as line 2 (because there are # on lines 2-9).

If we refer back to our *Rule of thumb* we know to define our datastart as *2-1 = 1.*

Delimiter Types

The delimiter can be a source of frustration. Basically defining your delimiter tells your computer what symbol to look for to separate your data points.

You can see clearly in the table above that header names and data points are separated by a



symbol, this is also known as a *tab delimiter*, and should be specified as so with a backslash t:

```
delimiter='\t'
```

Another common delimiter is the space bar: `delimiter=' '`

You may also see commas (,), semi-colons(;), quotes(“, ‘), braces({ }), slashes(/ \), or pipes (|). Just make sure that whatever separates your header values and data points is what you define to be your delimiter, otherwise your computer will not correctly read your data.

If you are able to run this line of code with no errors, the best practice is to then *print* your data table to make sure that it was read correctly. To do this, simply type `print(`name of your data structure here`)`.

Your computer should give you something like this:

```
In [5]: print(whitaker)

zmin zmax lmass  SFR  eSFR  L_IR ... L_UV_avg eL_UV_avg   x   xerr  beta
---- ---- ----- ----- ---- ----- ... -------- --------- ----- ---- -----
 0.5  1.0   8.4  -0.6  0.5  7.69 ...    -99.0     -99.0  9.04 0.01 -1.76
 0.5  1.0   8.7 -0.38 0.11  8.83 ...     8.75      0.11  9.18 0.01 -1.74
 0.5  1.0   8.9  -0.2 0.05  9.09 ...     9.05      0.05  9.34 0.01 -1.73
 0.5  1.0   9.1  0.05 0.03  9.59 ...     9.64      0.02  9.49 0.01 -1.67
 0.5  1.0   9.3  0.23 0.02  9.85 ...     9.92      0.01  9.61 0.01 -1.49
 0.5  1.0   9.5  0.45 0.02 10.17 ...    10.28      0.01  9.73 0.01 -1.32
 0.5  1.0   9.7  0.65 0.02 10.45 ...    10.57      0.01  9.81 0.01 -1.08
 0.5  1.0   9.9  0.78 0.03 10.63 ...    10.75      0.01  9.81 0.02 -0.84
 0.5  1.0  10.1  0.99 0.07 10.89 ...    11.01      0.01  9.82 0.05 -0.55
 0.5  1.0  10.3  1.06 0.05 10.96 ...    11.09      0.01  9.83 0.01 -0.42
 ...  ...   ...   ...  ...   ... ...      ...       ...   ...  ...   ...
 2.0  2.5   9.3  0.82 0.06 10.44 ...    10.35      0.06  10.2 0.01 -1.58
 2.0  2.5   9.6  1.05 0.03 10.77 ...    10.76      0.03 10.33 0.01 -1.46
 2.0  2.5   9.8  1.26 0.03 11.04 ...     11.1      0.02 10.41 0.01 -1.29
 2.0  2.5  10.0  1.46 0.03 11.33 ...    11.38      0.01 10.44 0.02 -1.02
 2.0  2.5  10.3  1.64 0.03 11.55 ...    11.61      0.01  10.4 0.02 -0.84
 2.0  2.5  10.5  1.86 0.05 11.79 ...    11.89      0.02 10.34 0.04 -0.62
 2.0  2.5  10.7  1.95 0.08 11.89 ...     12.0      0.02 10.25 0.05 -0.36
 2.0  2.5  10.9  2.07 0.06 12.02 ...    12.08      0.03 10.23 0.02 -0.28
 2.0  2.5  11.1   2.2  0.1 12.15 ...    12.28      0.03 10.19 0.06 -0.07
 2.0  2.5  11.3  2.32 0.12 12.27 ...    12.38      0.05 10.23 0.04  0.16
 2.0  2.5  11.5  2.39 0.17 12.35 ...    12.56      0.07 10.24 0.08 -0.25
Length = 50 rows
```

If you have read your data in correctly, you should see the *correct data values* listed under the *correct header names*, and the *Length* at the bottom should be equal to the amount of rows in your data set. For this example, we know our data has 50 rows so this is what a successful attempt would look like! If these values don’t match up correctly, something went wrong and you might need to do some investigative work.

Making your first plot

Now that we have successfully read our data table, let’s get to the fun part and start making some plots!

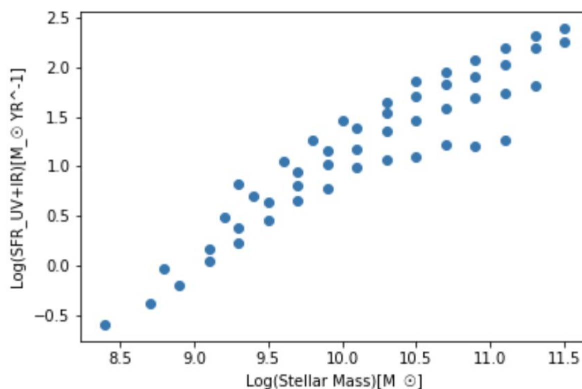
The first thing you want to do is figure out what to plot. Our data set contains lots of information about the galaxies used to create it: *stellar mass, redshift, star formation rate, luminosity* etc. When you are exploring new data sets, it can be helpful to plot various columns and look for patterns and trends within the data. For the purposes of this tutorial, let's look at *star formation rate* as a function of *stellar mass*. What does this relation tell us physically? Let's plot it and find out!

We will first define what we want to call star formation rate and stellar mass. In the catalogues, star formation rate is listed as "SFR" and stellar mass is listed as "lmass". Note that the stellar mass as well as the star formation rate are logarithmic base-10 measurements. In order to avoid confusion, we will define these variables as "lmass" and "lsfr" like so:
Information about the log/linear nature of data can be located in the written comments section of the file

```
In [6]: lmass=whitaker['lmass']
        lsfr=whitaker['SFR']
```

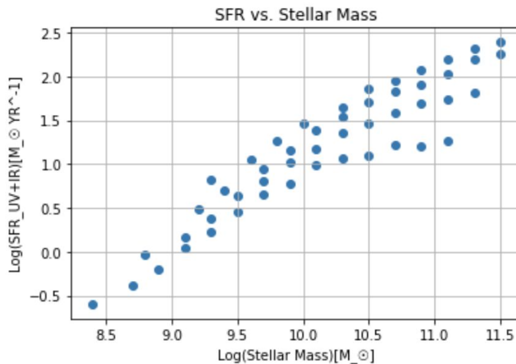
Remember when we imported `matplotlib` (used for graphing) as `plt`? Well now that we are plotting we will use the shortcut `plt` before every command so that our computer knows that we want to make a plot. If we were to make a scatter plot of star formation rate as a function of stellar mass, the code would look something like this:

```
In [11]: plt.scatter(lmass, lsfr)
         plt.xlabel('Log(Stellar Mass)[M_⊙]')
         plt.ylabel('Log(SFR_UV+IR)[M_⊙ YR^-1]')
         plt.show()
```



Notice that you defined you wanted a *scatter plot*, with the x-axis labeled *"Log(Stellar Mass)[M_⊙]"* and the y-axis labeled *"Log(SFR_UV+IR)[M_⊙ YR^-1]"*.
This graph looks good but let's add some more detail, like a grid and title.
The commands for title and grid lines are the following:

```
In [7]: plt.scatter(lmass, lsfr)
        plt.xlabel('Log(Stellar Mass)[M_⊙]')
        plt.ylabel('Log(SFR_UV+IR)[M_⊙ YR^-1]')
        plt.title('SFR vs. Stellar Mass')
        plt.grid()
        plt.show()
```



**Note*: Make sure that every time you are labeling something (such as an axis or title name) you put the words in “” or ‘‘. This will tell your computer to treat the code as *text* and not a command.

To better understand what we have plotted, let’s take a look at the original paper where this data was derived: https://iopscience.iop.org/article/10.1088/2041-8205/783/2/L30

You’ll notice that we see the same trends but the data in the paper is divided into four bins of redshift/lookback time. To display our data even better, we want to color code these galaxies by redshift.

Adding Layers to Plots

First, we will define redshift as the value in between `zmin` and `zmax` listed for each data point so that each point falls into a separate redshift bin. Define `zmin`, `zmax`, and `redshift` as the following:

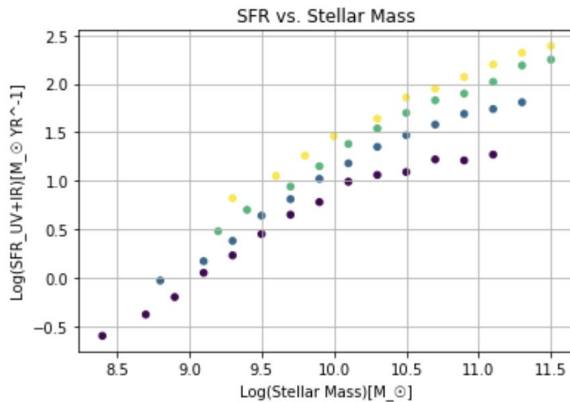
```
In [12]: zmin=whitaker['zmin']
         zmax=whitaker['zmax']
         redshift=(zmin+zmax)/2
```

Now we will create a second plot that displays three separate variables simultaneously: *stellar mass*, *star formation rate*, and *redshift*.
To do this, you will have to use the `ax.` command to define a separate *subplot.*

```
In [12]: fig=plt.figure()
         ax=fig.add_subplot(111)
         ax.scatter(lmass, lsfr, s=20, c=redshift, marker='o')
         ax.set_xlabel('Log(Stellar Mass)[M_⊙]')
         ax.set_ylabel('Log(SFR_UV+IR)[M_⊙ YR^-1]')
         ax.set_title('SFR vs. Stellar Mass')
         ax.grid()

         plt.show()
```

Notice that we have replaced every `plt` with an `ax.` except for `plt.show`

We are still creating a scatter plot of *stellar mass vs SFR* but we have defined a third variable *redshift* (*c=redshift)* which adds the color layer. Normally sub-plot implies separate panels, but in this context it is just overplotting data on the same original panel*s.* This graph should present as the same scatter plot above with additional color coding by redshift:

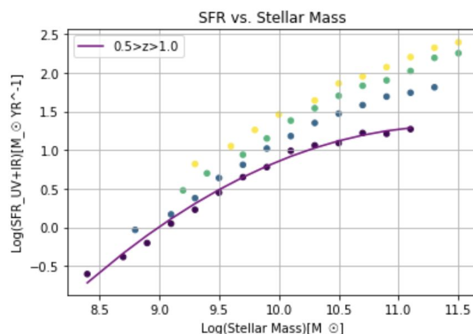


This looks much better.

Adding Functions to Plots

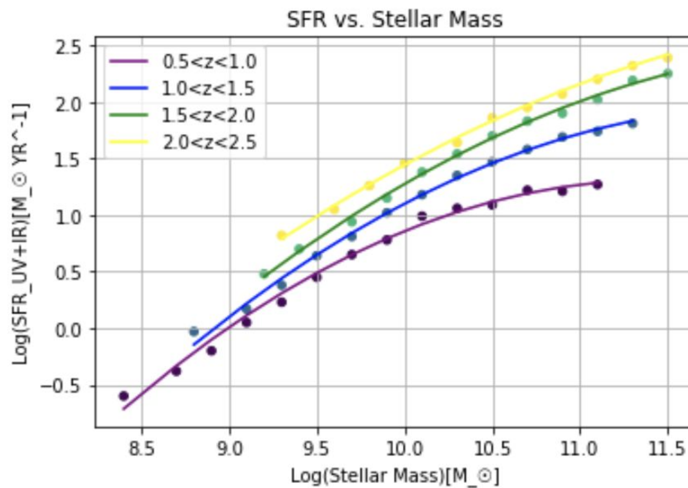
Whitaker et al. (2014) calculated functions of best fit for each redshift bin- how do you plot this and label it on your graph?

```
In [14]: fig=plt.figure()
         ax=fig.add_subplot(111)
         ax.scatter(lmass, lsfr, s=20, c=redshift, marker='o')
         ax.set_xlabel('Log(Stellar Mass)[M_⊙]')
         ax.set_ylabel('Log(SFR_UV+IR)[M_⊙ YR^-1]')
         ax.set_title('SFR vs. Stellar Mass')
         ax.grid()

         x1=lmass[(redshift>0.5)&(redshift<1.0)]
         y1=-27.4020+5.02150*x1-0.219550*x1**2
         plt.plot(x1, y1, color='purple', label='0.5>z>1.0')

         plt.legend()

         plt.show()
```



Notice that we clearly defined variables `x1` and `y1` as redshift points within the defined category and best fit function. We used the `plt.plot` command to plot the function, and told our computer we wanted the color to be “purple” and the label for this function to be “0.5<z<1.0”

When you use the command `plt.legend()`, the label is then shown in the *legend* in the upper left hand corner of the graph. You can do this separately for each redshift category and end up with a final graph that looks like:



Concluding Remarks

Now that you have successfully plotted your first graph with Python (hooray!) I hope that you are feeling a bit more comfortable and less intimidated when it comes to coding. This tutorial was intended to walk you through some of the more common obstacles that many undergraduate astronomy majors face when first starting out. (And hopefully I was able to save you some time and frustration as well)

As daunting as it may at first seem (and as frustrating as it will *always* be) learning to code with Python is an imperative skill for the modern astronomer. It is an extremely versatile computer language that will allow you to analyze, display, and interpret data through your own research. You will inevitably face more obstacles on this steep learning curve so don’t be afraid to ask for help from your teachers, advisors, peers, *comp-sci majors*, GOOGLE, etc.

The key to success is practice so if you have successfully completed this tutorial a good next step would be to try and follow some more advanced tutorials that can be found in the *Undergraduate Research Tutorial Repository:*
https://www.astrowhit.com/astronomy-research-tutorial-repository

Good Luck and Happy Coding!!! :)