The grand introduction

When I started to learn Python in 2012 with the aim of doing Astronomy, I didn't find any resources to be imminently helpful. Everything was either too advanced or a computer science style tutorial that went way too slow. It is my hope that after this very fast-paced course you are able to do some Astronomy, with some idea of what's going on under the hood. Don't hesitate to use this as a reference, but do not rely on it. Consult web services like StackOverflow, as they will have up-to-date answers for your questions.

So you want to learn Python?

Before we get to the real meat of the matter I want to take some time to make it clear why you should learn Python. I expect many readers will be reading this for simple reason, "I need to know programming to do research," "Python is the hot new language," or something along those lines. While those may be sufficient motivation, you should know *why* you need to be proficient in programming to be good at Astronomy research and *why* Python is the new language in Astronomy (and most science fields).

The answers to these questions are manyfold, but at the simplest level, programming is required out of necessity. The massive data sets of modern astronomy are unmanagable without at least proficient use of prebuilt tools like the rapidly-fading IRAF. Tools like IRAF will make data processing a tractable problem, but still leave the user with a huge amount of drudge work. It is the removal of such drudgery that make programming skills so valuable. Any problem that a human can solve, a computer can be programmed to solve (this is the basis of theoretical computer science). Therefore, I suggest to the reader that any data processing or data analysis program that must ask the user to intervene beyond some trivial initial settings is unfinished. This is actually the foundation of a common Python tutorial, Automate the Boring Stuff with Python.

So why Python? There are many other languages suitable for data processing, and in some cases an argument could be made that there are better choices. As of 2016 I am horrified that people still think that IDL is in general a better language for Astronomy than Python. There are two reasons Python is so widely used for data analysis: It is a language designed to be beautiful and easy-to-use, and it is very good at incorporating powerful libraries written C and Fortran. Python sometimes gets criticism for being slower than other languages, which it is. Python values programmer time over CPU time, an increasingly good tradeoff as processing power increases. The reason for this difference is that Python is (mostly) interpreted, not compiled like C and Fortran. The reason for this is that Python is interpreted, while fast languages are compiled. An interpreter converts the text you write into machine code that the computer innately understands line-by-line. A compiler on the other hand converts an entire program at once, which lets it make all sorts of optimizations that can result in enormous speedups. This does however come at a cost; compiling takes time. Therefore, interpreted (or scripting) languages are preferred for data analysis because often we only run a program once, then tweak it, and run it again.

Let's set up for some programming

If you have OSX or Ubuntu (probably most other Linux distros too), your computer should already have a version of Python installed. To check if you do, open a terminal (OSX) or command prompt (Windows). Search for and open your OS-appropriate terminal. You should get a prompt that you can type into. Enter the command `python` and press enter. If you don't get an error you at least have a Python interpreter installed. Otherwise you'll need one. I suggest Anaconda Python, a Python distribution produced by Continuum. Hit the Google and install it. You may be offered a choice between Python 2.7 and whatever the most recent version is (3.5 right now). I'll get into the difference later but for now I grudgingly admit that the best choice is probably 2.7, at least until the astro network is updated.

Now you need a text editor. We're not going to type individual commands into the interpreter, we want to write proper programs here. If you're using linux, gedit will suffice, otherwise I suggest Notepad++. If you don't have that or another text editor you have experience with, hit the Google and install it. If Notepad++ is not compatible with your system, I suggest looking for a simple text editor that is. Python comes with a text editor called IDLE which will suffice. Once you are comfortable with Python, I suggest using PyCharm. It has a lot of nice features but possibly too many for the new programmer.

Once you have a Python interpreter and text editor installed you'll be able to learn some of the basics, so let's get started!

Some basic programs

It's a programming tradition for the first program you write to have the computer print `Hello World`. Open up a text file and enter `print('Hello World')`

Save the text file, making note of where you saved it, and open a terminal. Now you need to navigate to wherever you saved the text file and run it. You can use the command `cd` to change directory. To figure out what directory you're in on OSX and Linux use `pwd`, and on Windows `cd ,`. You can change directory to `..` to back up. Once you've navigated to the location of the text file enter the command `python your_text_file.py` to run it. Make sure to state the actual name of the text file you saved after `python`. The file extension (whatever comes after the .) doesn't really matter here, and in some sense it never really does. It's just a hint to your operating system when you tell it to run the file, how it should be run. Since you're passing the file to a Python interpreter here it doesn't matter. So run your file and you should your computer give you a slightly creepy response.

Now that you know some of the basics about running Python files, we can do a bit of math. To assign a value to a variable, use the `=` symbol. So for example, `x = 1` will assign the value `1` to some variable `x`. This lets you manipulate the value at a later time. All your standard mathematical operations apply, use `+` to add values, `-` to subtract, `*` for multiplication, `/` for division, and `**` for exponentiation. As a quick exercise, print the product of 1234 and 5.

When you wrote the Hello World program, the apostrophes you enclosed the text with defined

what's called a string. Loosely, a string is a sequence of characters. It isn't really but that definition is good enough for now. You may find it interesting that the `*` and `+` operators can be used in other contexts. For example, you can print `'Hello World'` multiplied by an integer. You can also use the `+` operator on two strings to concatenate them. This hints at a fundamental detail of programming languages (but Python especially); these operators don't necessarily mean what they look like. Python inherits a lot of notation from mathematics to augment usability. Do you know what set builder notation is? Python supports that.

Face-first into the Astronomy

Opening FITS files

No messing around, it's time to do some things you'll actually have to do in the course of research: Let's open a FITS file. Python doesn't know how to do this on its own, but there is a package called AstroPy that has instructions for Python how how to read them. AstroPy doesn't come with Python by default, if we would say it is part of the standard library which is a set of packages that come with Python. We'll use some of them later. If you installed Anaconda Python you already have AstroPy available. Otherwise you'll need to figure out how to get it. The terminal command `pip install astropy` should do the trick.

Once you have AstroPy ready, we need to introduce ask the Python interpreter to load it at the beginning of our little program. The import statement we want in this case is `from astropy.io import fits`. This is a statement you'll use often. We don't need all of AstroPy here, that would just make things messy so we specify that we only want the part of astropy's io (for input/output) module (packages are made of modules, one or many) that deals with reading and writing FITS files. I have provided you with a FITS file, make sure it is in the same location as the text file you're working with; this will just make things easier. Now we can use the command `some_variable = fits.open('the_fits_file_name')`. Pick your own variable name here, and do not think you're saving space or effort by keeping the name short. If you write a huge program variable names are your only hits at what the variables are. Get into the habit of using descriptive names now.

If you run that program you might notice that your prompt disappears for a brief moment. This is because reading from the hard drive takes a significant amount of time. In this case probably less than a second but if you need to process hundreds of files it adds up quickly. So with your fits file loaded, you can print the variable. You'll notice what you get is only sort of helpful. What AstroPy gives you here is an HDUList. FITS files can hold multiple images, so the most general format for loading one is a list of sorts. In this case, there is only one entry in the list. If you want to access that entry, you need to index the list. In Python this is done by placing square brackets `[]` after the value you want to index, and enclosing an integer in the square brackets. Python is C-based so the first entry is number `0`. Try printing the first entry in the HDUList.

You'll find you still don't get anything that looks too helpful. That is because you now have an HDU (Header Data Unit) which has a header and some data. To access an attribute of something in Python, use `some_thing.the_attribute`. You'll notice we already did this in the import statement.

Your HDU has two useful attributes, data and header. Try printing both of them. If your terminal isn't 80 characters wide, the header will look like a mess. It should default to a width 80 if you didn't resize it. You can skim through the header, which normally contains a lot of useful information about the data. In this case it is only moderately enlightening. You may notice from the header information that this wan't taken at UF. I leave it as an exercise to any Floridians to guess where this was taken.

Displaying images

You may have heard of DS9. If you haven't, it is an ancient program mostly used to open and look at FITS files. I'm now going to show you how to do all the display functionality you get from DS9 using Python. For this, we're going to need at least one other package, matplotlib (same procedure as AstroPy). This is a powerful plotting library modeled after Mathematica, and is how we're going to display the data (the image) stored in the sample FITS file. The import statement we want here is `from matplotlib import pyplot as plt`. This introduces some new syntax; when importing we don't have to stick with the name that the person who wrote the package used, we can substitute our own. In this case shortening the name is a good idea because when plotting you're going to be typing `plt` often and that import statement is pretty much standard.

So let's display the image. Open up the FITS file and pass the first hdu's data to the `imshow` method of matplotlib's pyplot. Using the import statement from above: `plt.imshow(the_image_data)`. You can do this with an intermediate variable or not, it doesn't make a difference in ths case. Now you need to tell pyplot to actually display what you've plotted, so you need to call `plt.show()`. If you have very good eyes, you may be able to recognize this image already. Just in case though, we'll make a few tweaks to it.

Numpy ndarrays

Array conventions

The image data you load from the FITS file is in the form of a numpy object called an ndarray (for n-dimensional array). These ndarrays (or just arrays) are the primary way you will work with Astronomy data, so I'm going to spend some time explaining arrays in general and the particular capabilites of the numpy ndarray. A 1-d array is like a horizontal list, numbered left to right starting at 0. A 2-d array is like a table, where the first index is the row and the second index is the column. Note that the convention for 2-d arrays is backwards of traditional x and y coordinate systems. Just like when we opened a FITS file, arrays are indexed using a number in square brackets, following the name of the array; `some_array[5]` gives you the sixth element in `some_array`, since numbering starts at `0`. These are the only conventions that Python uses which really bother me, but they exist for good reason; most modern programming languages use them too.

Basic indexing and properties

Your image data you have loaded from a fits file is a 2-d array, so think about how you would get just one pixel value from it. The rows are numbered top-down starting at 0, and the columns left-right also starting at 0. You can pull out the first row using `some_array[0]`, then get the first

pixel in that row with `some_array[0][0]`. Thankfully, Python as well as most modern languages supports better syntax for doing this: `some_array[0,0]`. Along with the data stored in the array, numpy ndarrays know a bit about themselves. You can ask them for their size and shape, using `some_array.size` and `some_array.shape`. Print out what the size and shape of the sample data is. You may recognize that the array dimensions are powers of 2; this is very typical of CCD detectors.

Slicing and fancier indexing

So now we know how to get a row or a single pixel value from this image, but what if you want to get a column? The way to think of this is "get the elements in every row, from column `x`." To get the first column you can use `some_array[:,0]`. This `:` is a very nice feature of array slicing. On its own, it just indicates all elements along that axis. So you can even do `some_array[:]` if you want to get what's called a shallow copy of an array (for most cases this is a copy, but sometimes it isn't quite). You can also combine the colon with a number to indicate that you want all elements along that axis, but limited by the number. So `some_array[:10]` gets all rows of your image until but not including the one numbered 10, which means it gives you an array with 10 rows. You can also do `some_array[10:]` which will give you the row numbered 10 onwards. You can also put a number on each side of the colon, `some_array[5:10]`. You can also put in a third number along each axis, which is called the stride (start:stop:stride). This lets you do things like `some_array[::2]`, which pulls the even-numbered entries, and `some_array[1::2]` which pull the odd-numbered entries. Python also supports negative indexes to arrays, where the index `-1` indicates the last element along that axis. This lets you do things like pull out the last some number of elements in an array, regardless of the array's size. Some languages need you to first ask the array what size or shape it is, then compute the location of the last element and pass that to the array. That's unnecessary; an array already knows what size it is and there is no other obvious meaning to a negative index.

The neat application of this is that we can pull out rectangular portions of an array just by indexing. Try displaying the sample image, but only from rows 461 to 562 and columns 375 to 484. You should be able to see a vastly zoomed-in view of the full sample image. Matplotlib will also rescale the image, since the brightest pixels are not in this small region. This image still doesn't look like much though, so we're going to do some math to it so it looks a bit nicer.

Math on arrays

All the math operations I mentioned briefly work on arrays. You can add, subtract, multiply, and divide arrays by constants and arrays by arrays but only so long as numpy can broadcast them together. The most simple case of arrays that can be broadcasted are arrays of the same shape (and therefore the same size). Numpy also provides you with a bunch of tools for doing more complicated operations on arrays, but to get access to all of these we'll need to formally introduce Python to numpy. The typical way to do this is `import numpy as np`, so drop that line into the beginning of your file. You can now take take the square root of every element in an array, using `rescaled = np.sqrt(image_data)`. Take the square root of the data from the FITS file I've given you and `imshow` the new output. You should now be able to see a bit more detail and can probably recognize the image. It's still not very pretty though, so we have a few more corrections to make. First, we can remove an estimate of the sky background. This image was taken from in a suburban

area so the sky is relatively bright. The median of an image sparsely populated with stars is a very good estimate of the overall sky brightness. Use the function `np.median(some_array)` to compute the median of the image data and subtract that. Now to make it really nice we'll need to clip the image. This means we pick a min and max for the image, and every value below the min is set to the min, and values above the max become the max. For this image, take your square-root rescaled and sky-subtracted image and do something like `pretty_image = np.clip(rescaled, 0, 40)`. Now display your pretty image, and take a moment to enjoy it.

The fanciest of indexing

There are two other ways to index numpy ndarrays, both are done by passing arrays to arrays. The most obvious one is by passing arrays of integers. Arrays, including numpy ndarrays, have a data type. They can only store one kind of thing. Computers use a different representation for integer values and non-integer values, called floating point numbers. If you ask for the element at location 3.2 it's unclear which one you actually want or if you're asking for some sort of interpolation. I personally dislike using integer arrays for this purpose because you can run into exotic problems that you won't with the alternative: indexing with boolean arrays. The general idea here is that you're going to construct a boolean array of the same shape as (or one broadcastable to) the array being indexed. To generate a boolean array, you can use the binary comparison operators (binary here means they compare two things). Python supports >, <, ==, >=, <=, !=, &, |, ^. There are others but they're not very important. As a simple exercise, we can draw a crude circle using boolean comparators.

Think about what the definition of a circle is. A circle technically is the set of all points some distance from an orgin. Since we're working in the discrete world with arrays, a single distance will only get you a sparse few points, so we'll draw a filled-in circle. We need to calculate the distance from a set of points in space, then compare them to a cutoff distance with < or >. There is a trick to this that I use, `np.mgrid`. This is the more logical way to do it, though there is a much faster way using array broadcasting. Using `np.mgrid` is a rather special thing, we can index it like an array but it isn't one, but indexing it returns a tuple of arrays.

Tuples: an aside

The tuple is Python's most basic collection (numpy ndarrys are collections too, along with Python's native lists and many others). Tuples can be indexed like arrays, but can only have one dimension and cannot be modified once made. They can however be made from anything and also packed and unpacked. If you just assign a variable to any number of other things separated by commas, you've created a tuple. The syntax `test = x,y` creates a tuple. This lets you do things like `y,x = x,y` to switch variable values, but has a multitude of other uses I'm sure you'll discover. But for now, we can use tuple unpacking to unpack the tuple of arrays that `np.mgrid` returns.

When we call `np.mgrid`, we get a tuple of two ndarrays, so we can unpack it using `y,x = np.mgrid[-50:51,-50:51]`. Print out what `y` and `x` are. You now have the coordinates for positions in a space, so you should be able to use the distance formula to calculate the distance to each point from the origin. With your distances computed, comparing them with some cutoff will give you a

boolean array. Comparing them with two cutoffs then comparing the two boolean arrays with `&` can produce an annulus. You can use matplotlib's imshow again to display your boolean arrays to verify that you have done this correctly.

A final note

I have a few final things to note for this lesson, which are `np.sum` and `np.max` or `np.argmax`. Python provides a very helpful function, `help`. Try printing `help(np.sum)`. This help function will produce very similar information as you will find in online numpy manual, which will be found with a simple Google search. Google and StackOverflow will become your friends as you continue to learn. Most problems are quickly solved with a simple online search.

You are now able to do aperture photometry. I say that in all seriousness; this introduction is intended to be a full-force crash course in Python for astronomy. If you do not read or use any of the other workshops, this alone will prepare you at some level to do research and hopefully leave you equipped with some valuable knowledge. However, if you'd like to learn with some more depth and more reasonable pacing, continue on to data types.

Summary

If you're moving on, the only knowledge I expect you to remember from this lesson is a basic familiarity with Python syntax, and the concept of assigning values to variables, then doing things (specific, I know) with the variables.