Version 4.0
September 2013

Using Python for Astronomical Data Analysis

This is an unofficial document for internal RIAB training purposes only.

SPACE TELESCOPE SCIENCE INSTITUTE

Space Telescope Science Institute
3700 San Martin Drive
Baltimore, Maryland 21218

Operated by the Association of Universities for Research in Astronomy, Inc., for the National Aeronautics and Space Administration

Revision History

Version	Date	Editor and Contributors
5.0	June 2014	Katie GOSMEYER
Heather GUNNING	Matthew BOURQUE	
4.0	September 2013	Justin ELY, Thompson LEBLANC
3.0	January 2012	Rachel ANDERSON
2.0	October 2009	Sami-Matias NIEMI
1.0	September 2008	Alex VIANA

Send comments or corrections to:
Space Telescope Science Institute
3700 San Martin Drive

Baltimore, Maryland 21218
e-mail: ely@stsci.edu

Contents

1

Basic Python Concepts

1.1 Introduction

Python is a free, open-source, high level interpreted scripting language, similar to PERL. However, Python is also a high-level object-oriented programming language, much like C++, Java, and Ruby.

The intention of this tutorial is to provide the Python basics to understand how to perform simple astronomical data analysis.

We will discuss how to write and run Python scripts in Chapter 7. Until then we will use python interactively, which is best for learning python and for checking your code as you write it.

1.1.1 A Few Notes on Python

- Python is case sensitive. This turns out not to be that difficult as Python is almost always written in lowercase.
- Python does not have 'BEGIN' and 'END' statements. It does however depend on indents. It is best practice to indent with four spaces.

- Comments in code are useful to help you remember what is going on, but also so that others know what you are doing. In Python, comments begin with the # symbol. Multi-line comments can be nested in three quotes.
- The line continuation symbol ' \' can be used to spread a command out over multiple lines. This is not necessary inside parenthesis, brackets, or curly brackets as Python will wait for the closing symbol before assuming the end of the line.
- Python indices start with zero.

1.2 Environment and Set Up

1.2.1 SSB Environment

There are three different versions of the SSB (Science Software Branch) environments available at the institute. They are:

- SSBREL – Updated as needed, standard publicly available release.
- SSBX – Updated weekly, a testing environment. Holds latest bug fixes and updates. Institute only release.
- SSBDEV – Updated daily. Development updates intended for instrument teams to test bug fixes and upgrades. Institute only release.

Note that the version of Python related modules depend on the SSB environment, especially if you are using NFS mounts. To switch to a particular version, type one of the following into your terminal:

```
≫  ur_setup common ssbrel
≫  ur_setup common ssbx
≫  ur_setup common ssbdev
```

1.2.2 Interactive Python Environment

You will need to choose an interactive Python environment. There are two popular options.

- Python:
  To start the Python interactive environment, just type ‘python’ at the prompt in your terminal.

- iPython:
  To use the iPython interactive environment, type ‘ipython’ instead. The advantage of iPython is that besides Python code, you are also able to execute UNIX commands such as ‘ls’ and ‘cp’. Furthermore, you can also log your session. To start the log, type ‘%logstart’ in your iPython shell. This will start logging into a default file named ipython.log.py. The log-file follows a formatting convention of iPython and may not, on a first glance, look readable. However, iPython knows how to repeat the commands in your log-file when ran.

  – iPython Notebook:
    iPython Notebook can be called in the terminal by adding ’notebook’ after ipython. The notebook is an interactive web based interface. It combines code, text, math, plots, and rich media into a single document with all the functionality of ipython.

To exit either, type ‘exit()’. Only once do we use the shell command ‘cp’, for which you would need to use iPython (or just do it in the UNIX shell). Otherwise either environment would work.

1.3 Built-In Data Types

The principal built-in types of Python are numerics, sequences, mappings, files, classes, instances and exceptions. Other data types are available through importing modules (Example: the datetime module).

There are four distinct numeric types in Python: plain integers, long integers, floating point numbers, and complex numbers. In addition, Booleans are a subtype of plain integers. Python fully supports mixed arithmetic: when a binary arithmetic operator has operands of different numeric types, the operand with the “smaller” type is widened to that of the other. Example:

```
python>  a = 2.0
python>  b = 3
python>  a * b
```

Any Python object can be tested for truth value with *in*, if or *while* condition or as an operand of the Boolean operations. Python supports a vast number of truth values, but for simplicity I recommend using *True* and *False*. General Boolean operations *or*, *and*, and *not* are also supported. Comparison operations are supported by all objects. They all have the same priority, which is higher than that of the Boolean operations. In Python, comparisons are done with characters e.g. <, >=, ==, ! = (not like in IDL, where comparisons are done with shortened keywords e.g. *eq*).

Examples of some of the most common data types are below.

- File:
```
python> infile = open('PythonTraining.pdf')
python> infile.name
python> infile.readline()
python> infile.close()
```
- Integer:
```
python> a = 1
python> a == 1
python> a.denominator
```
- Float:
```
python> a = 2.0
```
- String:
```
python> a = 'three'
python> infile = 'infile.fits'
python> outfile = infile.replace('.fits', '_DidWork.fits')
python> 'Outfile:  ', outfile
```
- Tuple: An immutable list; elements can be of any data type.
```
python> a = (1, 2, 3)
python> a = (1, 2.0, 'three')
```
- List: A mutable list; elements can be of any data type.
```
python> a = [1, 2, 3]
python> b = [1, 2.0, 'three']
python> mylist = []
python> mylist.append(3)
python> print mylist
python> mylist = [1, 2, 3, 4]
python> len(mylist)
python> yourlist = [5, 6, 7, 8, 9]
python> mylist + yourlist
python> mylist.append(yourlist)
python> print mylist
```
  Notice that appending a list just adds one more element to 'mylist', and this element is of type list (remember, an element in a list can be of any type).

- Dictionary: A mapping which stores objects by a 'key' and not position. Below we show an example of a dictionary, and a dictionary of dictionaries. We could also create a dictionary of lists.

```
python>  a = {1:'one', 2:'two', 3:'three'}
python>  galaxy_dict = {}
python>  galaxy_dict
python>  galaxy_dict['M31'] = {'RA':'00 42 44.33',
...     'DEC':'+41 16 07.5','Vmag':3.44,'nickname':'Andromeda'}
python>  galaxy_dict
python>  galaxy_dict['M104'] = {'RA':'12 39 59.43185',
...    'DEC':'-11 37 22.9954','Vmag':3.44,'nickname':'Sombrero'}
python>  galaxy_dict.keys()
python>  galaxy_dict['M31'].keys()
python>  galaxy_dict['M31']['Vmag']
```

EXERCISES

Exercise 1 :

Create a dictionary with strings as the keys.

1.4 Objects, Instances, Attributes, Functions, and Methods,

Python is an object oriented programming language and before we go any further, it is important to note the differences and uses of objects, instances, attributes, functions, and methods.

- Objects:
  Data carriers that also carry functions and attributes that work on that data. In Section 1.3, integers, floats, lists, tuples, and dictionaries are all objects.

- Instance:
  In Section 1.3 when we said `galaxy_dict = {}` we created an instance of a dictionary, *galaxy_dict*, while `infile` is an instance of a *file*.

- Attributes:
  denominator is an attribute of an instance of a Python integer. Therefore in Section 1.3 we used `a.denominator`. Attributes do not have the '()' like functions do. Also in Section 1.3 we used the file attribute name when we said `infile.name`.

- Functions:
  A function is a named piece of code that performs an operation and has open and closed

parenthesis at the end (sometimes with variables inside). len() is a function, and in Section 1.3 we pass it the list, *mylist*.

- Methods:
A method is a function attached to an object that then has access to all other methods and attributes of that object. Therefore we usually do not have to pass as much information to methods, and methods are easier to read. All methods are functions, but not all functions are methods. For example, in Section 1.3, after we created the instance *mylist*, we append a number to that list using the method append() with the line `mylist.append(3)`. In Section 1.3 we also used the methods readline() and close() which belong to the file class.

Notice that we use dot notation to call an attribute or a method of a particular instance. As we will see in Section 1.7, dot notation will also be used to call a method or a function from a particular module.

1.5 Double Underscore

In Python you will notice certain names beginning and ending with double underscores. These names are used for attributes and methods that are used or created by the interpreter (for a discussion on the definition of attributes and methods see Section 1.4). Examples include:

- __file__ : an attribute automatically created by the interpreter
- __add__ : an attribute with special meaning to the interpreter
- __init__ : a method implicitly called by the interpreter, defined by the programmer

We will see examples of a few of these in Chapter 7.

1.6 Common Built-In Functions and Statements

For a complete list of Python built-in modules, see `http://docs.python.org/library/index.html`, also listed in Chapter 8. Below we provide descriptions of some of the most useful functions and statements.

1.6.1 print function

USE

The print function performs formatted output.

EXAMPLES

```
python>  a = 3
python>  print a
python>  b = 2
python>  print a + b
python>  print a / b
python>  b = 2.
python>  print a / b
python>  c = 'hi'
python>  print c + ' there'
python>  print 'I brought home {} flowers for \
...     ${} and all she said was {}.'.format(a,b+0.5,c)
```

This last statement is a bit more complicated. The '{}' symbol says to insert a variable from one of the arguments we give at the end. In the brackets we could specify data type, or which variable (0, 1, or 2), but we left it as default, which just takes the next argument in line. Notice the float does not print out exactly as we want. We would rather it say $2.50. Try this:

```
python> print 'I brought home {} flowers for ${:.2f} and all \
...she said was {}.'.format(a, b+0.5, c)
```

In this last example the 'f' stands for 'float', and the '.2' says to include two decimal places. For more information see:
http://docs.python.org/library/string.html#formatstrings.

EXERCISES

Exercise 2 :
Print out your own creative sentence.

1.6.2 help() function

USE

The help() function gives the user information on many aspects of the current Python session. Type 'q' to quit.

EXAMPLES

```
python>  a = 3
python>  help(a)
```

EXERCISES

Exercise 3 :
Type 'help()' and explore the options.

1.6.3 range() function

USE

The range() function creates a list starting at zero of the length you give it. You may also pass the start, stop, and the step you want to use (default of the step is 1).

EXAMPLES

```
python>  range(4)
python>  range(2, 8, 2)
```

SEE ALSO

Other useful similar functions in NumPy (Chapter 2) are numpy.linspace.

1.6.4 dir() function

USE

The dir() function returns the methods and attributes of an object.

EXAMPLES

```
python>  a = range(10)
python>  dir(a)
```

1.6.5 len() function

USE

The len() function returns the number of elements contained in an expression or variable.

EXAMPLES

```
python>  a = range(11)
python>  len(a)
```

SEE ALSO
Other useful similar functions in NumPy (Chapter 2) are numpy.shape, numpy.size.

1.6.6 for loop

USE
The for statement is used to execute one or more statements repeatedly, while incrementing or decrementing a variable with each repetition, until a condition is met.

EXAMPLES

```
python>  a = range(11)
python>  for x in a: print x
python>  c = ['2005', '2006', '2007']
python>  for x in c: print x + '-01'
python>  b = [x*2 for x in a]  #Create a list in one line
```

SEE ALSO
Other useful similar statements are if...elif...else, while.

EXERCISES
Exercise 4 :
Create a vector with 10 values equal to the square of the index.

Exercise 5 :
Create the following sequence using a for *statement: 2001-01-01, 2001-02-01, 2001-03-01, 2001-04-01.*

1.6.7 xrange() function

USE

The xrange() function is similar to range() except that it does not create a new list (saving on memory) but instead acts as a generator of numbers starting at zero up to the stop value you pass it. You may also pass the start, stop, and the step you want to use (default of the step is 1). This is used in iterations.

EXAMPLES

```
python>  for x in xrange(3): print x, x ** 2
```

Within a function, the for statement has the following simple structure:

```
python>  for x in xrange(1, 7):
...     a = x * 3.
...     print a
```

SEE ALSO
Other useful similar statements are range.

Notice that we use range() when we use the actual list created, but xrange() when we just need an iterator.

1.6.8 try ... except statement

USE
The try ... except statement is used as an error handler. It will try whatever we put in the try block, but if whatever you assign as the 'error' occurs, it will go to the except block.

EXAMPLES
Try this code below. Notice the error message you get.

```
python>  a = [1, 2, 3, 4, 5]
python>  for i in xrange(len(a)):
...     a[i + 1] = 100 + a[i]
...     print a[i], a[i + 1]
```

To solve this problem, we put it in a try ... except statement:

```
python>  a = [1, 2, 3, 4, 5]
python>  for i in xrange(len(a)):
...      try:
...          a[i + 1] = 100 + a[i]
```

```
...         print a[i], a[i + 1]
...     except IndexError:
...         print 'This index does not work: ',i + 1
```

EXERCISES

Exercise 6 :

Imagine a situation where your code passes a function a variable, and that variable might be zero. The only problem is that this function divides by that variable.

Write a simple loop which calculates 1/n, where n is what is passed. What happens if $n = 0$*? Write an error handling for this loop (hint: use* try ... except *statement with ZeroDivisionError).*

1.7 Importing Modules, and Common Functions

Python's built-in functions are limited. The diversity of Python's abilities come when we import modules. The following syntax can be used:

```
python>  import <module>
python>  from <module> import <function_in_module>
python>  import <module> as <short_name>
```

1.7.1 math.sqrt() function

USE

If we want to take the square root of a number, we could use the function sqrt() in the module math. The following is two examples of how to do this:

EXAMPLES

```
python>  import math
python>  math.sqrt(100)
```

Notice we did not need the '.py' extension. If we do not need any other math module, and we know we will not name a variable 'sqrt' and overwrite the function, we can do this:

```
python>  from math import sqrt
python>  sqrt(100)
```

1.7.2 glob.glob() function

USE

The glob.glob() function searches for files that match the given path-name. The path-name you give is a string similar to the search strings used for the UNIX/Linux 'ls'.

EXAMPLES

```
python>  import glob
python>  glob.glob('*')
python>  glob.glob('*.pdf')
python>  datadir = '/Users/username/data/'  #insert a usable path
python>  glob.glob(datadir + 'MIR*12-03*ref??.fits')
```

EXERCISES

Exercise 7 :
Search for a set of files on your desktop.

1.7.3 random.random() function

USE

The random.random() function returns a random floating point number in the range [0.0, 1.0).

EXAMPLES

```
python>  import random
python>  random.random()
python>  print 'My random number between 2 and 8: ', \
...     2 + (8-2) * random.random()
```

SEE ALSO

Other useful similar statements are random.uniform(), random.gaus().

1.7.4 re.search() function

USE

The re module is for 'Regular Expression' operations. It is used to work with strings, including sophisticated pattern matching, as in the case of re.search().

EXAMPLES

```
python> import re
python> m = re.search('(?<=_)\d+','MIRI_2011.fits')
python> m.group(0)
```

In the above example we are looking for digits following an underscore, '_'. The '\d' is for digit, and the '+' is for one or more. The '(?<= ...)' matches if the position in the string is proceeded by '...'.

SEE ALSO
Other useful similar statements are re.match().

1.7.5 os.getcwd() and os.chdir() functions

USE
The os module is for 'Operating System' operations. Examples include os.getcwd() which returns the current working directory and os.chdir() which changes the current working directory.

EXAMPLES

```
python>  import os
python>  datadir = '/Users/username/data/'  #insert a usable path
python>  mydir = os.getcwd()
python>  if (mydir != datadir): os.chdir(datadir)
```

SEE ALSO
Other useful similar statements are os.open(), os.close().

1.7.6 sys.exit() function

USE
The sys module is for system-specific parameters and functions. It provides a way to interact with the interpreter. sys.exit() is different than exit() in that it will honor try statements and you can intercept the exit attempt.

EXAMPLES

```
python>  import sys
python>  for i in xrange(10):
...        if i <= 5:
```

```
...             continue
...         else:
...             sys.exit('We do not need a number above 5.')
```

2

NumPy and Data Arrays

2.1 The Uses of NumPy

NumPy is a Python module which adds support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays. NumPy addresses the problems of speed in interpreting languages by providing multi-dimensional arrays and lots of functions and operators that operate on arrays. Any algorithm that can be expressed primarily as operations on arrays and matrices can run almost as fast as the equivalent C code.

2.1.1 NumPy's array vs. Python's built-in list

NumPy introduces new data types, but the most popular, versatile, and useful one is the array. This is similar to arrays in IDL. There are several reasons why you would want to use a NumPy array over Python's built-in list.

- NumPy, PyLab, SciPy, PyFITS and other modules' functions often work with NumPy arrays.

- Every item in a NumPy array is of the same data type. This means there is less information to keep track of which makes array computations faster.
- NumPy arrays act as vectors and therefore we can do things such as element-wise addition and multiplication.

To convert a list to an array, use numpy.array().

2.1.2 numpy.array() function

USE
The numpy.array() function converts a list to a NumPy array.

EXAMPLES

```
python>  import numpy as np
python>  a = [1,2,3,4]      #a python built in list
python>  b = np.array(a)     #converted to a NumPy array
python>  print a
python>  print b
python>  indx = [1,2]
python>  print b[indx]
python>  print b[1:3]       #prints elements 1 to 2, NOT 1 to 3.
python>  print b[:3], b[3:] #prints up to the 3rd element, \
...    and then everything after the 3rd element.
python>  print b[-1],b[-2]
python>  print b[::-1]      #reverses the array.
python>  c = np.array([[1,2,3,4,5],[6,7,8,9,10]])
python>  print c
python>  print c[1,3]       #indices for a multi-dimensional array
python>  print c[1][3]      #this is slower than the previous as it \
...    creates a new array, $c[1]$, and then subscripts that array.
python>  print c[1,:]  #print the first element in the first \
...    dimension, but all in the second dimension
python>  print c[:,1]
```

Notice when we print lists and arrays that the elements in lists are separated by commas while the elements in arrays are only separated by spaces.

EXERCISES
Exercise 8 :
Create a list a and a NumPy array b. Multiply each by 2 and explain what happens. Now add 2 to each array. Again, explain the result.
Exercise 9 :
Create a third list c. Add c to both a and b. Explain the result.

2.2 What a NumPy array really is and a word of caution

A final note about NumPy arrays is that an array is actually an object which points to a block of memory. For example, in the above exercise we created an array *b*. Try the following:

```
python> d = b
```

Now we just created a second array, *d*. Instead of using up twice the memory space, *d* is just a pointer to the memory *b* also points to (remember, we copied an array, and an array is a pointer). Again, try the following:

```
python>  d[2] = 999
python>  print d
python>  print b
```

Notice what happened to *b*. While it saves on memory space, programmers have to be careful. If you know you will want to change one array and not the other, the correct function to use is numpy.copy().

2.2.1 numpy.copy() function

USE
The numpy.copy() function copies the contents of the memory space the array points to.

EXAMPLES
Try the code below and notice the difference in the results from a simple $d = b$ assignment.

```
python>  import numpy as np
python>  a = np.array([1,2,3,4,5,6,7])
python>  b = a.copy()
python>  b[2] = 999
python>  print b
```

```
python>  print a
python>  a.size
python>  a.shape
```

2.3 Other Common NumPy Functions

2.3.1 numpy.arange() function

USE

The numpy.arange() function creates an integer array from zero to the 'stop' parameter given, with a step size of one. The 'start' and 'step' can also be specified. By setting the parameter 'dtype' we can change the data type of the array (i.e. to float).

EXAMPLES

```
python>  import numpy as np
python>  np.arange(10)
python>  1 + np.arange(10, dtype=float) * 4
python>  np.arange(1,40,4,dtype=float)
```

SEE ALSO

A similar function for lists is range().

EXERCISES

Exercise 10 :

Create the sequence 0.1, 0.2,0.3, ... 1.4 using numpy.arange(). *Hint: As noted in the NumPy documentation for numpy.arange(), it is best to use integer step sizes.*

Exercise 11 :

Create the sequence -3.2, -3.0, -2.8, ... -1.0 using numpy.arange(). *See above hint.*

2.3.2 numpy.empty() function

USE

The numpy.empty() function creates a float array of the specified dimensions. Each element of the array is whatever was left in that memory space, therefore it is fast but useful only if you know you will assign each element a meaningful value.

EXAMPLES

```
python>  import numpy as np
python>  np.empty(10)   #pass an argument, which is the dimensions
python>  np.empty((3,4)) #here it is 2D, so the dimensions \
...     we pass is a tuple
```

SEE ALSO
Other useful similar functions are numpy.zeros(), numpy.ones().

2.3.3 numpy.where() function

USE
The numpy.where() function returns an array (or a tuple of arrays) of the indices where the condition is *True*. Otherwise, if you specify substitute values, it will return an array of the same shape as the original with the first value substituted where the condition is *True*, and the second value substituted where the condition is *False*.

EXAMPLES

```
python>  import numpy as np
python>  a = np.arange(11, dtype=float) + 1
python>  b = np.where(a >= 8.)
python>  print b
python>  a[b]
python>  a = np.array([1,2,3,1,2,1,1,1,1,4])
python>  b = np.where(a == 1, 1,0)
python>  print b
```

If we do not need the indices from numpy.where() then we can just use creative indexing for the same effect.

```
python>  a > 5
python>  a[a>5]
```

SEE ALSO
Other useful similar functions are numpy.any(), numpy.all(), numpy.nonzero(), numpy.choose().

EXERCISES
Exercise 12 :
Create a random real 10-element array with numbers between 0 and 1. Select those with counts lower than 0.5.

3

Handeling FITS files and ASCII data tables using Astropy

Astropy is a python library for astronomy developed by professional astronomers and software developers from around the world, some of which work here at STScI in the Science Software Branch. It is under continuous development and is quickly becoming a powerful library, especially for handeling FITS files and ASCII tables. Visit the website listed in Chapter 8 for more information and useful documentation.

The astropy.io.fits module provides an interface to FITS formatted files under the Python scripting language. astropy.io.fits data structures are a subclass of NumPy arrays, which means that they can use NumPy arrays' methods and attributes. The astropy.io.ascii module provides flexible and easy-to-use methods for reading and writing ASCII data tables. In the following sections, we will explore these two modules.

3.1 Opening, Reading, and Closing a FITS File

As an example, we will use data from the *WFC3* instrument located here:

/user/gunning/Python_Training/icft01crq_flt.fits

Please copy this file to your working directory.

Below we show an example of opening a FITS file, getting the data and the header, closing the file, printing out the shape of the data using numpy.shape, printing out header values, and finally making changes to the data.

```
python>  from astropy.io import fits
python>  infile = 'icft01crq_raw.fits'
python>  fits.info(infile)
python>  hdulist = fits.open(infile)
python>  hdr = hdulist[0].header  # Get the primary header
python>  data = hdulist[1].data  # Get the data from the 1st extension
python>  data.shape
python>  print hdr
python>  hdr['FLSHCORR']
python>  hdr['FLSHCORR'] = 'PERFORM'
python>  hdr['FLSHCORR']
python>  print data[-2:]  # Print the last two lines.
python>  data[-1:][0][0] = 0
python>  print data[-1]
```

Notice that *hdr* behaves like a dictionary. We did some bad things to this file, but let's save it anyway to a new file.

```
python>  outfile = 'mybad.fits'
python>  fits.writeto(outfile, data, hdr)
python>  print 'Saved FITS file to: '.format(outfile)
```

Alternatively, if we want to modify the original file directly, we can do the following:

```
python> fits.update(infile, data, 1)
```

As a word of caution, note that astropy.io.fits reads in FITS images as (rows, cols) or (y, x), not (x, y). This is often a 'gotcha' for users who are indexing spacific areas of the array.

3.2 fits.getval() and fits.setval() functions

If you are familiar with IRAF/PyRAF, you are probably familiar with IRAF's hedit function, which allows you to add, delete, and modify keywords in a FITS header.

First, lets take a look of our example file's header using imheader in PyRAF. In PyRAF navigate to the folder where your icft01crq_raw.fits file is located, and try the following:

```
--> imheader icft01crq_raw.fits[0] l+ | page
```

We see that there is a 'DARKFILE' keyword, and it is set to 'iref$y2j13512i_drk.fits.' Say we wanted to recalibrate this file using a different 'DARKFILE'. The value of this keyword can be changed using hedit, as shown here:

```
--> hedit icft01crq_raw.fits[0] DARKFILE 'iref$y2p1831ci_drk.fits'  \
...     verify=no update=yes
```

In the above example we made sure the 'update' parameter was set to 'yes.' We can check that our edit was successful by using imheader again and checking the value of the 'DARKFILE' keyword:

```
--> imheader icft01crq_raw.fits[0] l+ | page
```

Using Python, there is a simple way to do this with astropy.io.fits using fits.getval() and fits.setval(), shown in the example below.

```
python>  from astropy.io import fits
python>  infile = 'icft01crq_raw.fits[0]'
python>  key = 'DARKFILE'
python>  fits.getval(infile, key, 0)
python>  fits.setval(infile, key, value='iref$y2j13512i_drk.fits', ext=0)
python>  fits.getval(infile, key, 0)
```

Now our FITS file is back to its 'initial' state. No harm done.

3.3 Reading and Writing ASCII Data Files

The astropy.io.ascii module provides two robust methods, ascii.read() and ascii.write(), for reading and writing multi-column delimited data tables, respectively.

3.3.1 astropy.io.ascii.read() function

USE

The astropy.io.ascii.read() function reads in a table of data from a specified file and returns an astropy.Table object.

As an example, we will use these files:

/user/gunning/Python_Training/flux_vs_time_A.dat
/user/gunning/Python_Training/flux_vs_time_C.dat

These data are used to plot the flux versus time for a standard star and serves as a monitor of the WFC3/UVIS photometric stability for amps A and C, respectively. In this section, we will read in the data, and in chapter 4, we will use it to produce the plots.

EXAMPLES

```
python>  from astropy.io import ascii
python>  infile = 'flux_vs_time_A.dat'
python>  data = ascii.read('flux_vs_time_A.dat',
                names=['MJD', 'Flux_diff', 'Flux_err', 'Flux_linear_fit'])
python>  print data
python>  print data['MJD']
python>  print data['MJD', 'Flux_diff']
python>  print data['Flux_diff'] * 10
python>  pos_flux = data['Flux_diff'] > 0
python>  print data[pos_flux]
```

EXERCISES

Exercise 13 :
Try reading in the data from 'flux_vs_time_C.dat' using ascii.read()

3.3.2 astropy.io.ascii.write() function

USE

Similar to ascii.read(), the astropy.io.ascii.write() function writes a table of data to a specified file. For fun, lets try taking the MJDs from 'flux_vs_time_A.dat', normalizing them to the first observation date, and writing the new table to a new text file:

EXAMPLES

```
python>  from astropy.io import ascii
```

```
python>  infile = 'flux_vs_time_A.dat'
python>  data = ascii.read('flux_vs_time_A.dat',
              names=['MJD', 'Flux_diff', 'Flux_err', 'Flux_linear_fit'])
python>  first_date = min(data['MJD'])
python>  data['MJD'] = data['MJD'] - first_date
python>  print data
python>  ascii.write(data, 'flux_vs_time_A_mjdnorm.dat')
```

EXERCISES

Exercise 14 :

Try making a MJD-normalized text file for the 'flux_vs_time_C.dat' data using ascii.write()

4

Plotting with Pyplot

4.1 matplotlib.pyplot

Matplotlib and its PyPlot environment is a versatile Python plotting library which produces publication quality figures in a variety of hard-copy formats such as EPS, PDF, and PNG. With PyPlot you can generate scatter and line plots, histograms, power spectra, bar charts, error-charts, pie charts, and many more with just a few lines of code. For the power user, you have full control of line styles, font properties, axes properties, and so on. For useful examples of astronomy plots that can be generated with PyPlot, see Leonardo Ubeda's astroplotlib library at http://astroplotlib.stsci.edu/

4.2 Create a Simple Scatter Plot

We'll start by making a simple scatter plot, demonstrating some of the PyPlot options, and saving our plot in a PDF format.

First, we need to read in some data, as we learned in Chapter [[...]].

```
python>  from astropy.io import ascii
```

```
python>  data_A = ascii.read('flux_vs_time_A.dat', names=['Time',  \
... 'Flux_pcnt_diff', 'Flux_err', 'Flux_linear_fit'])
python>  data_C = ascii.read('flux_vs_time_C.dat', names=['Time',  \
... 'Flux_pcnt_diff', 'Flux_err', 'Flux_linear_fit'])
```

The time is in Modified Julian Date (MJD) and the remaining columns are flux percent differences and are dimensionless. (The fluxes are of a standard white dwarf for a WFC3 monitoring program. The two datasets are from different amplifiers on the WFC3 two-chip mosaic.)

Now we will import PyPlot and make our first plot, using a *figure* object...

```
python>  import matplotlib.pyplot as pyplot
python>  figure, ax = pyplot.subplots()
```

Now we can begin to plot into the *figure* object, via the *ax* axis created in the previous step. Each subsequent call to the inherited *ax.plot* method will update the overall plot. The next two calls plot the two sets of flux differences as scatter plots in blue and red, respectively.

```
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], c='blue')
python>  ax.scatter(data_C['Time'], data_C['Flux_pcnt_diff'], c='red')
python>  figure.show()
```

The *figure.show()* command displays our changes onto the *figure* object.

We should add axis labels...

```
python>  ax.set_xlabel('Time [MJD]', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]',  fontsize=20)
python>  figure.show()
```

Finally let's save the figure. We can save as a PDF, PNG, TIFF, and other file types; we need only to type the appropriate extension.

```
python>  figure.savefig('flux_vs_time_1.pdf')
```

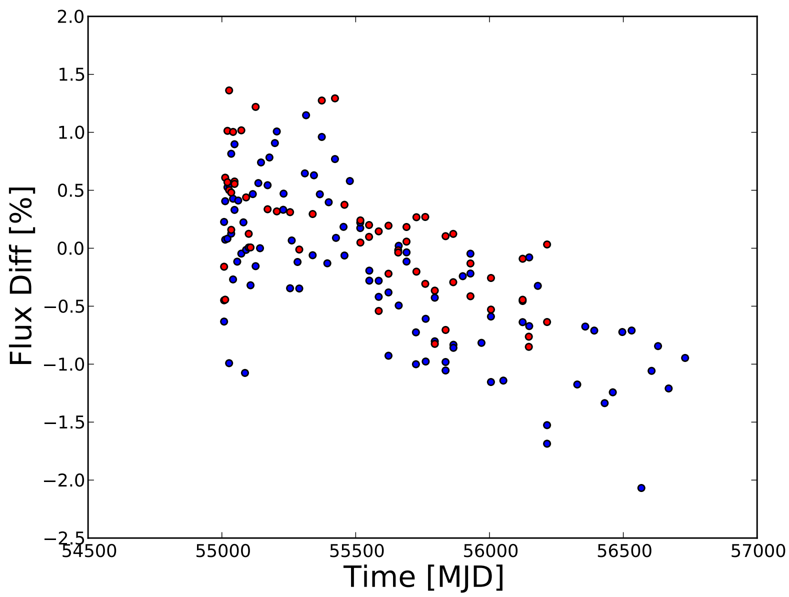

FIGURE 4.1: Our first plot.

4.3 Markers, Lines, and Legends

We have a plot! But we are not yet finished. PyPlot has many, many options available for you to customize your plot. We'll demonstrate just a few for markers, lines, and legends.

We can play with the marker type (*marker*), size (*s*), and transparency (*alpha*) of our of scatter plots' points. We first need to clear the figure with the *ax.clear()* command.

```
python>  ax.clear()
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], \
...   c='blue', marker='x', s=30, alpha=0.75)
python>  ax.scatter(data_C['Time'], data_C['Flux_pcnt_diff'], \
...   c='red', marker='d', s=30, alpha=0.75)
python>  ax.set_xlabel('Time [MJD]', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]', fontsize=20)
python>  figure.show()
```

The plot would be clearer if we added a legend.

```
python>  ax.clear()
```

```
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], \
...    c='blue', marker='x', s=25, alpha=0.75, label='Amp A')
python>  ax.scatter(data_C['Time'], data_C['Flux_pcnt_diff'], \
...    c='red', marker='d', s=25, alpha=0.75, label='Amp C')
python>  ax.set_xlabel('Time [MJD]', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]', fontsize=20)
python>  ax.legend(loc='best', scatterpoints=1)
python>  figure.show()
```

The *ax.legend(loc='best')* command will try to find the least busy section of your plot and stick the legend there.

Next let's plot lines with the linear fits from our data:

```
python>  ax.plot(data_A['Time'], data_A['Flux_linear_fit'], \
...   c='blue', ls='--', linewidth=2, label='Amp A Fit')
python>  ax.plot(data_C['Time'], data_C['Flux_linear_fit'], \
...   c='red', ls=':', linewidth=2, label='Amp C Fit')
python>  ax.legend(loc='best', scatterpoints=1)
python>  figure.show()
```

Suppose we want to denote the 0.0 flux difference with a dashed line and the MJD date 56250.0 with a green line:

```
python>  ax.axhline(0.0, color='k', ls='--', linewidth=1)
python>  ax.axvline(56250.0, color='green', ls='-', linewidth=2)
python>  figure.show()
```

We now have a personalized plot! (It doesn't have to be pretty.) Let's save it.

```
python>  figure.savefig('flux_vs_time_2.pdf')
```

Other options can be found on the Matplotlib PyPlot site listed in Chapter 8.

For different color names, see http://www.w3schools.com/html/html_colornames.asp

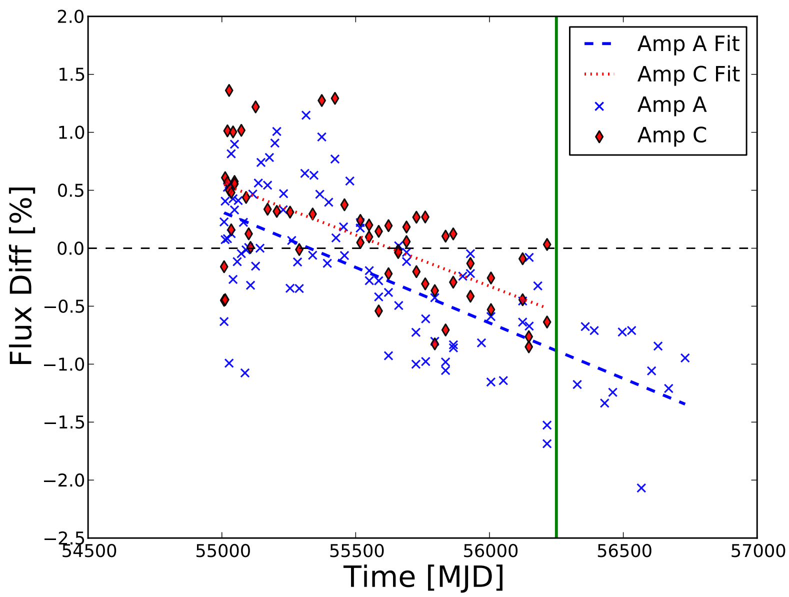

FIGURE 4.2: Our customized plot.

4.4 Error Bars

We have columns giving our error. We can display them on our plot as error bars using the *errorbar* function:

```
python>  ax.clear()
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], c='blue')
python>  ax.errorbar(data_A['Time'], data_A['Flux_pcnt_diff'], \
... yerr=data_A['Flux_err'], c='grey', marker=None, ls='None')
python>  figure.show()
```

4.5 Display Dates on X-axis

MJD is a convenient format for plotting time. But who thinks in MJD? Let's convert MJD to day-month-year and display them on the x-axis. In python this is a little tricky.

```
python>  from astropy.time import Time
python>  time_MJD = [float(item.get_text()) for item in ax.get_xticklabels
```

```
python>  time_convert = Time(time_MJD, format='mjd', scale='utc')
python>  time_ymd_long = time_convert.iso
python>  time_ymd_short = []
python>  for date in time_ymd_long:
...    time_ymd_short.append(date.split(' ')[0])
python>  ax.set_xticklabels(time_ymd_short)
python>  figure.show()
```

See http://astropy.readthedocs.org/en/latest/time/ for more examples.

4.6 Plot Figures Side-by-Side

4.7 Fitting Lines

4.8 Display a FITS Image

4.9 Plot Spectra

4.10 Plot with Pyplot

From Section **??** we learned how to read in data from a file, particularly Gordon2005_Fig16.txt. We will reproduce the slope plot in Figure 16 of the Gordon 2005 paper.

```
python>  import numpy as np
python>  infile = 'Gordon2005_Fig16.txt'
python>  slope, ran_slope_unc, corr_slope_unc, \
...      both_slope_unc, eqn_slope_unc = np.loadtxt(infile,
...      usecols=(0, 1, 2, 3, 4), unpack=True)
```

These arrays have nice descriptive names, but to help make the plotting process clear, we will assign short names.

```
python>  xx = slope
python>  yy1 = ran_slope_unc
```

```
python>  yy2 = corr_slope_unc
python>  yy3 = both_slope_unc
python>  yy4 = eqn_slope_unc
```

Now we will import Pyplot and make our first plot, using a *figure* object...

```
python>  import matplotlib.pyplot as plt
python>  figure, ax = plt.subplots()
```

Now we can begin to plot into the *figure* object, via the *ax* axis created in the previous step. Each subsequent call to the inherited *ax.plot* method will update the overall plot...

```
python>  ax.plot(xx, yy1, ls='--', color='b')
python>  ax.plot(xx, yy2, ls=':', color='r')
python>  ax.plot(xx, yy3, ls='-', color='g')
python>  ax.plot(xx, yy4, ls='-.', color='m')
```

Notice that 'b' is for blue, 'r' is for red, 'g' is for green, and 'm' is for magenta. Furthermore, '-' is for a solid line, '–' is for a dashed line, ':' is for a dotted line, and '-.' is for a dot-dash line. Several other options are available as well and can be found on the Matplotlib Pyplot site listed in Chapter 8.

The lines are pretty faint, and we also need to add a legend and axis labels. Let's start by clearing the *axis* object, replotting, and then force a refresh on our *figure*:

```
python>  ax.clear()
python>  ax.set_xlabel('Slope [e-/s]')
python>  ax.set_ylabel('Slope Uncertainty [e-/s]')
python>  ax.plot(xx, yy1, ls='--', color='b', label='Randon Unc.')
python>  ax.plot(xx, yy2, ls=':', color='r', label='Correlated Unc.')
python>  ax.plot(xx, yy3, ls='-', color='g', label='Both')
python>  ax.plot(xx, yy4, ls='-.', color='m', label='Equation')
python>  ax.legend(loc='best')
python>  figure.show()
```

We can save this plot. The figure will save as a PDF, PNG, TIFF, and others depending on the name given.

```
python>  figure.savefig('fig16.pdf')
python>  figure.clf()
```

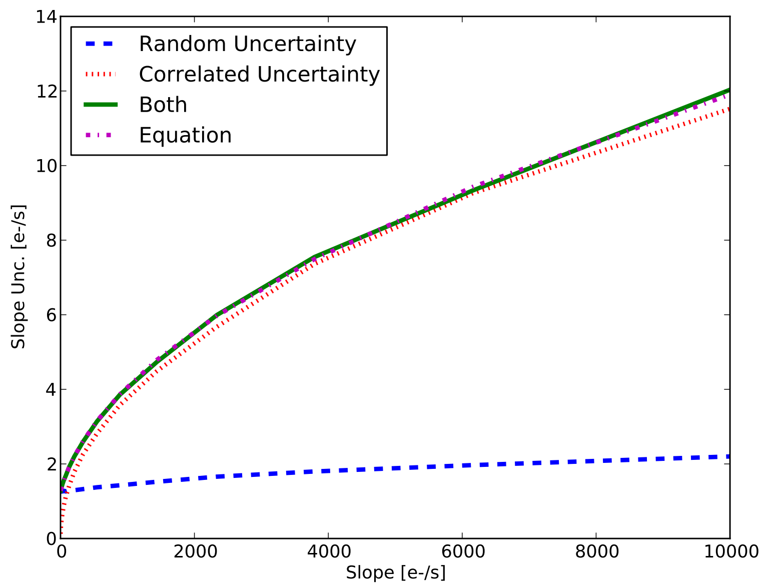

FIGURE 4.3: Our first try at re-creating Figure 16 in Gordon 2005.

Notice that we cleared the figure with *figure.clr()*. This step is not necessary, but is useful to clear a figure object that is no longer needed. Open fig16.pdf and take a look. It looks pretty good, right?

If you look at the paper by Gordon et al. 2005, though, you will see that the key thing we are missing is logarithmic axis. Instead of *matplotlib.pyplot.plot* we will use *matplotlib.pyplot.loglog*:

```
python>  ax.clear()       # Clear the axis...
python>  ax.loglog(xx, yy1, ls='--', lw=3, color='b', label='Random Unc.')
python>  ax.loglog(xx, yy2, ls=':', lw=3, color='r', label='Correlated Unc.')
python>  ax.loglog(xx, yy3, ls='-', lw=3, color='g', label='Both')
python>  ax.loglog(xx, yy4, ls='-.', lw=3, color='m', label='Equation')

python>  ax.set_xlabel('Slope [e-/s]')
python>  ax.set_ylabel('Slope Uncertainty [e-/s]')
python>  ax.legend(loc='best')
python>  figure.show()
```

Again, open the figure you just made. How does that look?

For more plotting options with Matplotlib and Pyplot, check out the matplotlib.org link listed in Chapter 8. Notice that there is a link to NumPy on this page, as well as links to screen-shots,

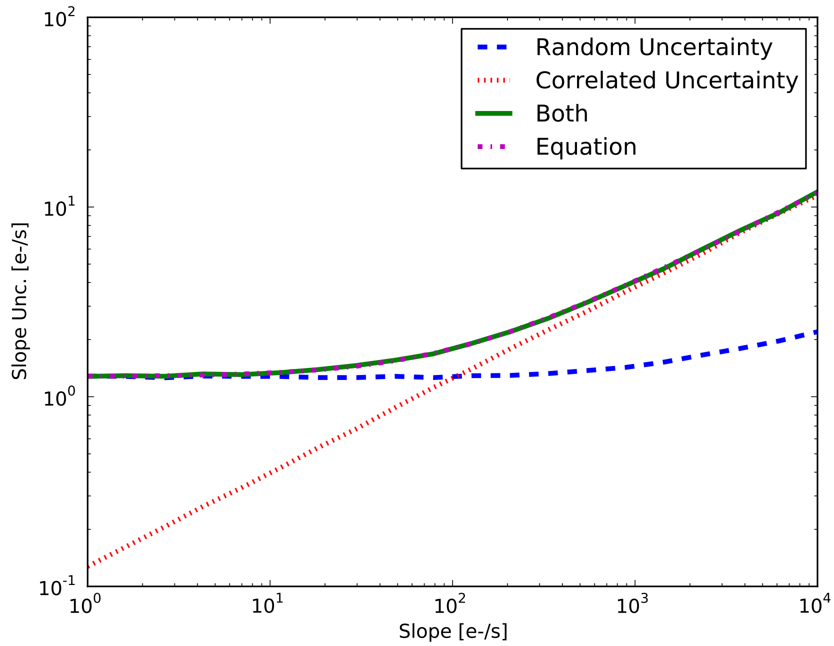

FIGURE 4.4: Our version of Figure 16 in Gordon 2005.

thumbnails, and examples.

Our plot is looking good, however this is getting way too messy. Isn't there a way we can keep this all nicely in a file, where we can type it once correctly and never have to type it again? Sure there is! We need to write a script.

5

Plotting with Pyplot

5.1 matplotlib.pyplot

Matplotlib and its PyPlot environment is a versatile Python plotting library which produces publication quality figures in a variety of hard-copy formats such as EPS, PDF, and PNG. With PyPlot you can generate scatter and line plots, histograms, power spectra, bar charts, error-charts, pie charts, and many more with just a few lines of code. For the power user, you have full control of line styles, font properties, axes properties, and so on. For useful examples of astronomy plots that can be generated with PyPlot, see Leonardo Ubeda's astroplotlib library at http://astroplotlib.stsci.edu/

5.2 Create a Simple Scatter Plot

We'll start by making a simple scatter plot, demonstrating some of the PyPlot options, and saving our plot in a PDF format.

First, we need to read in some data, as we learned in Chapter [[...]].

```
python>  from astropy.io import ascii
```

```
python>  data_A = ascii.read('flux_vs_time_A.dat', names=['Time',  \
... 'Flux_pcnt_diff', 'Flux_err', 'Flux_linear_fit'])
python>  data_C = ascii.read('flux_vs_time_C.dat', names=['Time',  \
... 'Flux_pcnt_diff', 'Flux_err', 'Flux_linear_fit'])
```

The time is in Modified Julian Date (MJD) and the remaining columns are flux percent differences and are dimensionless. (The fluxes are of a standard white dwarf for a WFC3 monitoring program. The two datasets are from different amplifiers on the WFC3 two-chip mosaic.)

Now we will import PyPlot and make our first plot, using a *figure* object...

```
python>  import matplotlib.pyplot as pyplot
python>  figure, ax = pyplot.subplots()
```

Now we can begin to plot into the *figure* object, via the *ax* axis created in the previous step. Each subsequent call to the inherited *ax.plot* method will update the overall plot. The next two calls plot the two sets of flux differences as scatter plots in blue and red, respectively.

```
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], c='blue')
python>  ax.scatter(data_C['Time'], data_C['Flux_pcnt_diff'], c='red')
python>  figure.show()
```

The *figure.show()* command displays our changes onto the *figure* object.

We should add axis labels...

```
python>  ax.set_xlabel('Time [MJD]', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]',  fontsize=20)
python>  figure.show()
```

Finally let's save the figure. We can save as a PDF, PNG, TIFF, and other file types; we need only to type the appropriate extension.

```
python>  figure.savefig('flux_vs_time_1.pdf')
```

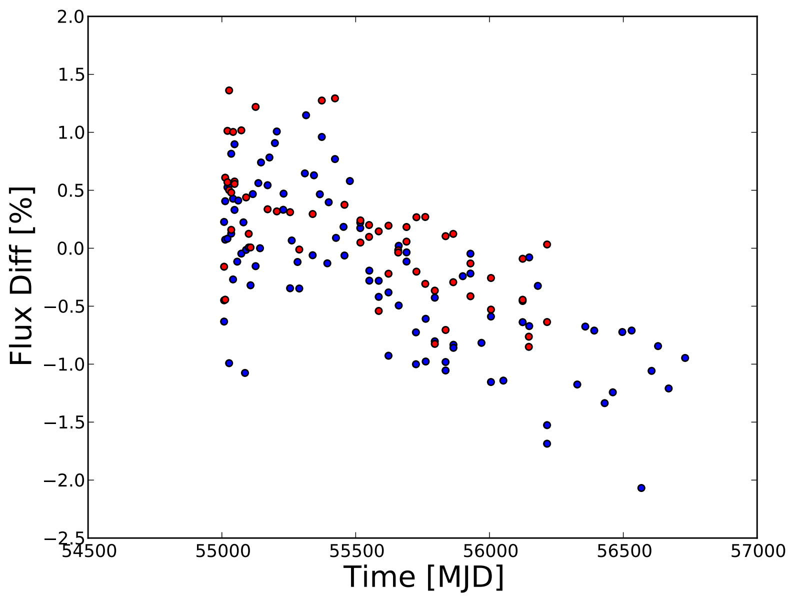

FIGURE 5.1: Our first plot.

5.3 Markers, Lines, and Legends

We have a plot! But we are not yet finished. PyPlot has many, many options available for you to customize your plot. We'll demonstrate just a few for markers, lines, and legends.

We can play with the marker type (*marker*), size (*s*), and transparency (*alpha*) of our of scatter plots' points. We first need to clear the figure with the *ax.clear()* command.

```
python>  ax.clear()
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], \
...    c='blue', marker='x', s=30, alpha=0.75)
python>  ax.scatter(data_C['Time'], data_C['Flux_pcnt_diff'], \
...    c='red', marker='d', s=30, alpha=0.75)
python>  ax.set_xlabel('Time [MJD]', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]', fontsize=20)
python>  figure.show()
```

The plot would be clearer if we added a legend.

```
python>  ax.clear()
```

```
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], \
...    c='blue', marker='x', s=25, alpha=0.75, label='Amp A')
python>  ax.scatter(data_C['Time'], data_C['Flux_pcnt_diff'], \
...    c='red', marker='d', s=25, alpha=0.75, label='Amp C')
python>  ax.set_xlabel('Time [MJD]', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]', fontsize=20)
python>  ax.legend(loc='best', scatterpoints=1)
python>  figure.show()
```

The *ax.legend(loc='best')* command will try to find the least busy section of your plot and stick the legend there.

Next let's plot lines with the linear fits from our data:

```
python>  ax.plot(data_A['Time'], data_A['Flux_linear_fit'], \
...   c='blue', ls='--', linewidth=2, label='Amp A Fit')
python>  ax.plot(data_C['Time'], data_C['Flux_linear_fit'], \
...   c='red', ls=':', linewidth=2, label='Amp C Fit')
python>  ax.legend(loc='best', scatterpoints=1)
python>  figure.show()
```

Suppose we want to denote the 0.0 flux difference with a dashed line and the MJD date 56250.0 with a green line:

```
python>  ax.axhline(0.0, color='k', ls='--', linewidth=1)
python>  ax.axvline(56250.0, color='green', ls='-', linewidth=2)
python>  figure.show()
```

We now have a personalized plot! (It doesn't have to be pretty.) Let's save it.

```
python>  figure.savefig('flux_vs_time_2.pdf')
```

Other options can be found on the Matplotlib PyPlot site listed in Chapter 8.

For different color names, see http://www.w3schools.com/html/html_colornames.asp

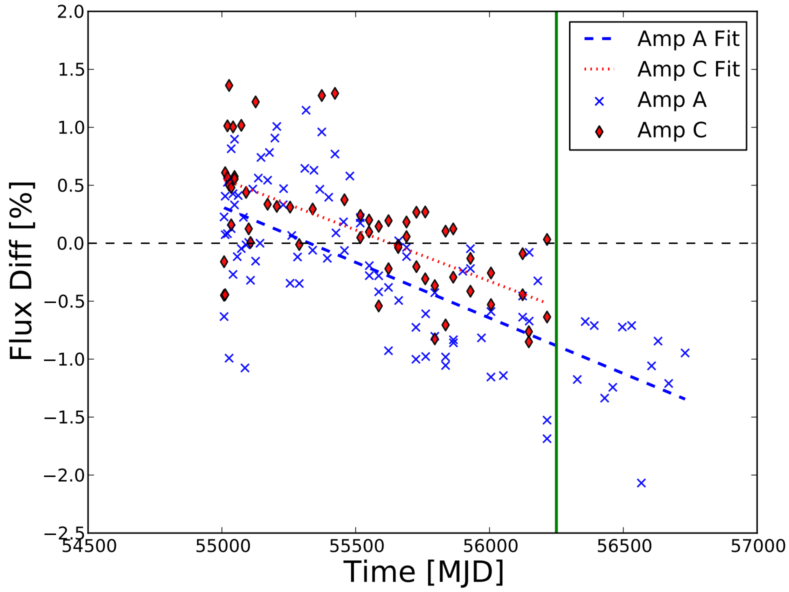

FIGURE 5.2: Our customized plot.

5.4 Error Bars

We have columns giving our error. We can display them on our plot as error bars using the *errorbar* function:

```
python>  ax.clear()
python>  ax.scatter(data_A['Time'], data_A['Flux_pcnt_diff'], c='blue')
python>  ax.errorbar(data_A['Time'], data_A['Flux_pcnt_diff'], \
... yerr=data_A['Flux_err'], c='grey', marker=None, ls='None')
python>  figure.show()
```

5.5 Display Dates on X-axis

MJD is a convenient format for plotting time. But who thinks in MJD? Let's convert MJD to day-month-year and display them on the x-axis. In python this is a little tricky.

```
python>  from astropy.time import Time
python>  time_MJD = [float(item.get_text()) for item in ax.get_xticklabels
```

```
python>  time_convert = Time(time_MJD, format='mjd', scale='utc')
python>  time_ymd_long = time_convert.iso
python>  time_ymd_short = []
python>  for date in time_ymd_long:
...    time_ymd_short.append(date.split(' ')[0])
python>  ax.set_xticklabels(time_ymd_short)
python>  figure.show()
```

See http://astropy.readthedocs.org/en/latest/time/ for more examples.

5.6 Plot Figures Side-by-Side

5.7 Fitting Lines

5.8 Display a FITS Image

5.9 Plot Spectra

5.10 Plot with Pyplot

From Section **??** we learned how to read in data from a file, particularly Gordon2005_Fig16.txt. We will reproduce the slope plot in Figure 16 of the Gordon 2005 paper.

```
python>  import numpy as np
python>  infile = 'Gordon2005_Fig16.txt'
python>  slope, ran_slope_unc, corr_slope_unc, \
...      both_slope_unc, eqn_slope_unc = np.loadtxt(infile,
...      usecols=(0, 1, 2, 3, 4), unpack=True)
```

These arrays have nice descriptive names, but to help make the plotting process clear, we will assign short names.

```
python>  xx = slope
python>  yy1 = ran_slope_unc
```

```
python>  yy2 = corr_slope_unc
python>  yy3 = both_slope_unc
python>  yy4 = eqn_slope_unc
```

Now we will import Pyplot and make our first plot, using a *figure* object...

```
python>  import matplotlib.pyplot as plt
python>  figure, ax = plt.subplots()
```

Now we can begin to plot into the *figure* object, via the *ax* axis created in the previous step. Each subsequent call to the inherited *ax.plot* method will update the overall plot...

```
python>  ax.plot(xx, yy1, ls='--', color='b')
python>  ax.plot(xx, yy2, ls=':', color='r')
python>  ax.plot(xx, yy3, ls='-', color='g')
python>  ax.plot(xx, yy4, ls='-.', color='m')
```

Notice that ‘b’ is for blue, ‘r’ is for red, ‘g’ is for green, and ‘m’ is for magenta. Furthermore, ‘-’ is for a solid line, ‘–’ is for a dashed line, ‘:’ is for a dotted line, and ‘-.’ is for a dot-dash line. Several other options are available as well and can be found on the Matplotlib Pyplot site listed in Chapter 8.

The lines are pretty faint, and we also need to add a legend and axis labels. Let’s start by clearing the *axis* object, replotting, and then force a refresh on our *figure*:

```
python>  ax.clear()
python>  ax.set_xlabel('Slope [e-/s]')
python>  ax.set_ylabel('Slope Uncertainty [e-/s]')
python>  ax.plot(xx, yy1, ls='--', color='b', label='Randon Unc.')
python>  ax.plot(xx, yy2, ls=':', color='r', label='Correlated Unc.')
python>  ax.plot(xx, yy3, ls='-', color='g', label='Both')
python>  ax.plot(xx, yy4, ls='-.', color='m', label='Equation')
python>  ax.legend(loc='best')
python>  figure.show()
```

We can save this plot. The figure will save as a PDF, PNG, TIFF, and others depending on the name given.

```
python>  figure.savefig('fig16.pdf')
python>  figure.clf()
```

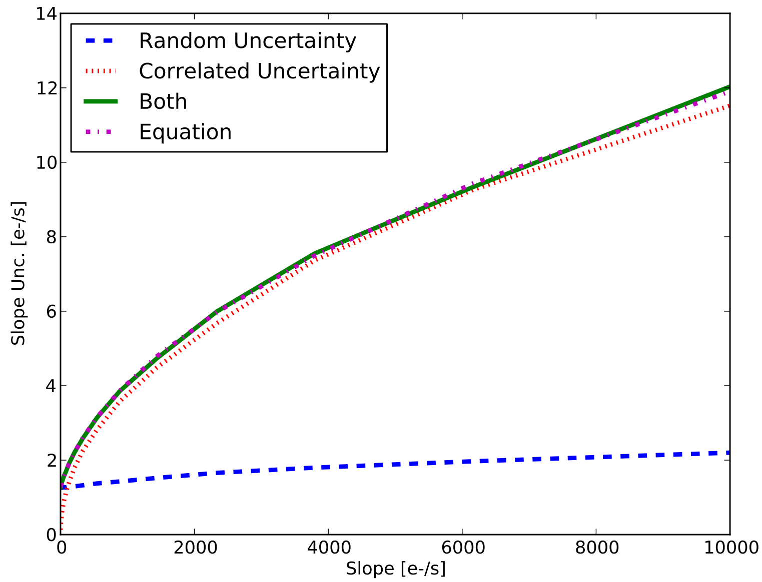

FIGURE 5.3: Our first try at re-creating Figure 16 in Gordon 2005.

Notice that we cleared the figure with *figure.clr()*. This step is not necessary, but is useful to clear a figure object that is no longer needed. Open fig16.pdf and take a look. It looks pretty good, right?

If you look at the paper by Gordon et al. 2005, though, you will see that the key thing we are missing is logarithmic axis. Instead of *matplotlib.pyplot.plot* we will use *matplotlib.pyplot.loglog*:

```
python>  ax.clear()      # Clear the axis...
python>  ax.loglog(xx, yy1, ls='--', lw=3, color='b', label='Random Unc.')
python>  ax.loglog(xx, yy2, ls=':', lw=3, color='r', label='Correlated Unc.')
python>  ax.loglog(xx, yy3, ls='-', lw=3, color='g', label='Both')
python>  ax.loglog(xx, yy4, ls='-.', lw=3, color='m', label='Equation')

python>  ax.set_xlabel('Slope [e-/s]')
python>  ax.set_ylabel('Slope Uncertainty [e-/s]')
python>  ax.legend(loc='best')
python>  figure.show()
```

Again, open the figure you just made. How does that look?

For more plotting options with Matplotlib and Pyplot, check out the matplotlib.org link listed in Chapter 8. Notice that there is a link to NumPy on this page, as well as links to screen-shots,

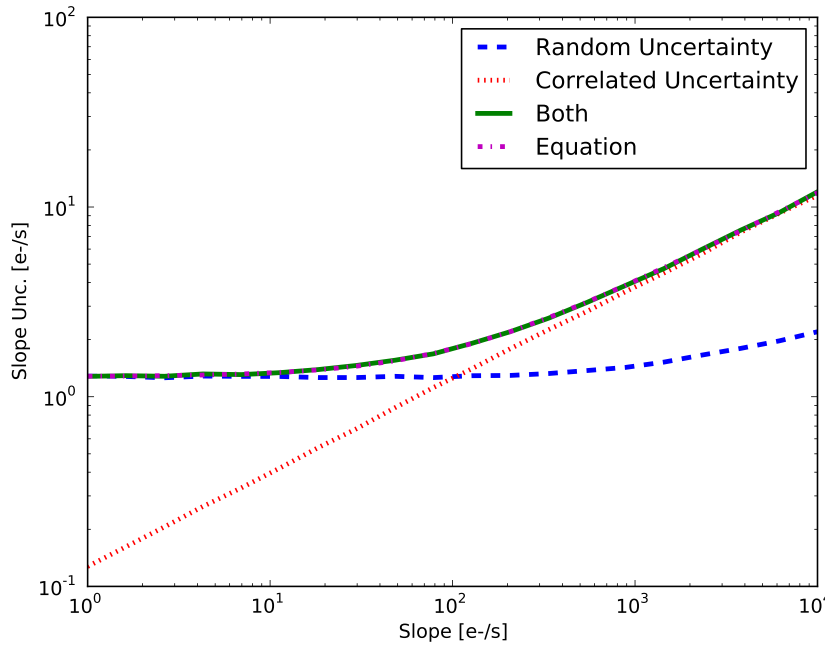

FIGURE 5.4: Our version of Figure 16 in Gordon 2005.

thumbnails, and examples.

Our plot is looking good, however this is getting way too messy. Isn't there a way we can keep this all nicely in a file, where we can type it once correctly and never have to type it again? Sure there is! We need to write a script.

6

PyRAF

6.1 PyRAF Introduction

PyRAF calls IRAF tasks from Python. One of the main motivations for creating PyRAF is to create a version of IRAF that is compatible with a programming language other than CL. CL has many short comings which are more and more apparent in more complicated programs.

PyRAF has its own interactive PyRAF session. To start it, just type ‘pyraf’ in a terminal. This should look familiar to you if you know IRAF. Type ‘.exit’ to exit. The remaining part of this chapter will have you import PyRAF into Python unless otherwise stated.

6.2 A PyRAF Example with iraf.daofind()

Again, create a file in your script directory, and open it in your favorite editor. Add your header. For your ’ABOUT’ section, we will be creating a script showing how python can be used to execute IRAF/PyRAF tasks. We will use the same example data from Chapter **??**, n9vf01hfq_ima.fits.

Import the PyRAF module and from within that module import the IRAF instance. The code for importing these functions looks like this:

```
#Load the packages we need
import pyraf
from pyraf import iraf
from iraf import noao, digiphot, daophot
```

As you can infer 'iraf' is something that lives inside of 'pyraf'. For example, if you want to run phot on some files you need to create a list of coordinates. To do this we will use daofind. Since daofind is an iraf module, we execute it using dot notation, like this:

```
iraf.daofind(parameter1,parameter2,parameter3)
```

The parameters for a PyRAF task executed in the python environment are the same as they would be in the PyRAF environment. Pull up a PyRAF window and open the help file to see what the parameters for iraf.daofind() are. Now are going to run iraf.daofind() on the file n9vf01hfq_ima.fits.

```
#Run daofind on one image
iraf.daofind(image='n9vf01hfq_ima.fits[1]')
```

The '[1]' is for extension '1'. You should be prompted for several parameters, just hit 'enter' to go through the defaults. Type 'ls' in the directory where you ran this – there should now be a file called n9vf01hfq_ima.fits1.coo.1. In PyRAF open up the file and look at the output, then compare this with the image on DS9. Did iraf.daofind() do a good job? You can facilitate this process with the tvmark task.

```
≫  pyraf
pyraf>  images
pyraf>  !ds9 &
pyraf>  display n9vf01hfq_ima.fits[1] 1
```

The last '1' is for which DS9 frame to display in.

```
pyraf>  tv
pyraf>  tvmark 1 n9vf01hfq_ima.fits1.coo.1
```

We can see that we did not to too well, but let's ignore that for now and work on our program.

Go ahead and run this program again, and you will see that it will create a second file called n9vf01hfq_ima.fits1.coo.2. What if we did not want so many files and instead wanted to always write to n9vf01hfq_ima.fits1.coo.1? IRAF might have a problem overwriting this file, so a simple solution would be to remove it. For this we will need to import os. Furthermore, if we don't want to have to press 'enter' at the prompt, we can add to our program the extra parameters it is looking for. Finally, let's write this program so that we can run the process on multiple files. For that we will import glob.

Here is what we have:

```
#! /usr/bin/env python
# HEADER

#Load the packages we need
import pyraf, os, glob
from pyraf import iraf
from iraf import noao, digiphot

#Generate a list of all the fits files
file_list = glob.glob('*_ima.fits')
print file_list

#Loops though all the .fits files
for file in file_list:
    #Test for old files, and remove them if they exist
    file_query = os.access(file + '1.coo.1', os.R_OK)
    if file_query == True:
        os.remove(file + '1.coo.1')
    #Run daofind on one image
    iraf.daofind(
        image = file + '[1]',
        interactive = 'no',
        verify = 'no')
```

EXERCISES

Exercise 15 :
Write your script that uses iraf.daofind().

Anytime you change the default settings to a PyRAF command it is a good idea to change them back. You can do this with the iraf.unlearn() command as shown below.

```
pyraf>  iraf.daofind.unlearn()
```

```
pyraf>  iraf.unlearn('daofind')
```

7

Python Scripts and Functions

Make a directory where you will keep all of your scripts. You will also need to choose a text editor, such as Emacs, TextWrangler, NEdit, or others. You will be using this to edit your scripts.

7.1 MyFirstScript.py

Create a file called MyFirstScript.py in your script directory, and open it in your favorite editor. Comments in scripts are very important, not just to help you remember what exactly you were thinking, but to help others as well should they ever use your code. Therefore, to start off on the right foot, let's create a special header called a docstring that looks something like this:

```
'''
ABOUT:
This is a program that takes this and plots that.

DEPENDS:
Python 2.7.3

AUTHOR:
```

```
M.E. MySelf for STScI, 2011

HISTORY:
2011: Trial program.

USE:
python MyFirstScript.py
'''
```

For the header we show three apostrophes to comment out multiple lines of text. Next thing we will want to have is all of our imports.

```
import numpy as np
import matplotlib.pyplot as plt
```

Now we are ready! Look back through your shell and create your script from Chapter 5. So far you should have something similar to the following:

```
infile = 'Gordon2005_Fig16.txt'
outfile = 'fig16_log.pdf'

slope, ran_slope_unc, corr_slope_unc, \textbackslash
    both_slope_unc, eqn_slope_unc = np.loadtxt(infile,
    usecols=(0, 1, 2, 3, 4), unpack=True)

figure, ax = plt.subplots()

ax.loglog(slope, ran_slope_unc, ls='--', lw=3, color='b',
    label='Random Unc.')
ax.loglog(slope, corr_slope_unc, ls=':', lw=3, color='r',
    label='Correlated Unc.')
ax.loglog(slope, both_slope_unc, ls='-', lw=3, color='g',
    label='Both')
ax.loglog(slope, eqn_slope_unc, ls='-.', lw=3, color='m',
    label='Equation')

ax.set_xlabel('Slope [e-/s]')
ax.set_ylabel('Slope Uncertainty [e-/s]')
ax.legend(loc='best')
figure.show()
```

```
figure.savefig(outfile)
figure.clf()
```

Notice that we listed any variables (or things we might want to change later) at the top of the program, such as *outfile* and *infile*. Also, since we do not have to type the variables over and over, I decided to use the more descriptive names. This will also help if I do not work on this script for awhile and then later come back to it. Now our program is ready to execute.

7.1.1 Executing Python Scripts

There are a few ways to run a Python program. One is to type from your terminal:

≫ `python MyFirstScript.py`

Or, if you are already inside the Python interactive environment, just type:

```
python> import MyFirstScript
```

For the last example, if you want to re-run the script, type:

```
python> reload(MyFirstScript)
```

Another way is to make the program executable and then type:

≫ `MyFirstScript.py`

This is nice. You do not have to type ‘python’, you can run it from anywhere, and you do not have to be in the Python interactive environment. However, we must first do two things.

1. To tell your computer which shell or interpreter should be used for executing this file, at the very top of your file add:

   ```
   #!  /usr/bin/env python
   ```

2. You will need to make your script executable. In the terminal, type:

   ≫ `chmod a+x MyFirstScript.py`

To be able to call your script from anywhere on your computer, add your script directory to your executable path by opening your startup script file (such as .cshrc for tc shell or .bashrc for bash shell, or more generally, .mysetenv or .setenv) and adding the line (with the correct path substituted in):

```
setenv PATH .:/my/script/directory:{$PATH}
```

Next, in your terminal execute the command:

```
≫ source .cshrc
```

EXERCISES
Exercise 16 :
Write your script MyFirstScript.py and execute it.

7.2 Adding Functions

MyFirstScript.py is short and easy to read. However, we can imagine a case where we have such a large program and many repeated tasks that it will become difficult. For example, if we want to create the y-intercept uncertainty plot as well as the slope uncertainty plot we can either repeat several lines of our program (and if we change one copy remember to change the other), or we can create a plotting function and just call that function twice. The general format of declaring a function and then using it is shown below.

```
#! /usr/bin/env python

# Header

def mkplot(data):
    # Make plot here
    return

if __name__=='__main__':
    data = SomeFunctionThatGetsData()
    mkplot(data)
    print 'Now I have a beautiful plot!'
```

Notice the line `if __name__=='__main__':`. It is often useful to start the main program with this statement. In this way you can make the file usable as a script as well as an import-able module, because the code that parses the command line only runs if the module is executed as the 'main' file, i.e. 'MyFirstScript.py'. In this way, from another program (or a Python prompt) we could call

```
from MyFirstScript import mkplot.
```

Let's modify our program to look like this:

```
#! /usr/bin/env python

#Header

__author__ = 'M.E. MySelf'
__version__ = 0.2

import numpy as np
import matplotlib.pyplot as plt

def mkplot(outfile,xx,yy1,yy2,yy3,yy4,ylab='Slope Unc. [e-/s]'):
    figure, ax = plt.subplots()

    ax.set_xlabel('Slope [e-/s]')
    ax.set_ylabel('Slope Uncertainty [e-/s]')
    ax.plot(xx, yy1, ls='--', color='b', label='Randon Unc.')
    ax.plot(xx, yy2, ls=':', color='r', label='Correlated Unc.')
    ax.plot(xx, yy3, ls='-', color='g', label='Both')
    ax.plot(xx, yy4, ls='-.', color='m', label='Equation')
    ax.legend(loc='best')
    figure.show()

    figure.savefig(outfile)
    figure.clf()
    print 'Saved file to: ',outfile
    return

if __name__=='__main__':
    infile = 'Gordon2005_Fig16.txt'
    slope_outfile = 'fig16_slope.pdf'
    yint_outfile = 'fig16_yint.pdf'

    slope, ran_slope_unc, corr_slope_unc, both_slope_unc,  \
        eqn_slope_unc, ran_yint_unc, corr_yint_unc,  \
        both_yint_unc, eqn_yint_unc = np.loadtxt(infile,
        unpack=True)
    mkplot(slope_outfile, slope, ran_slope_unc, corr_slope_unc,
        both_slope_unc, eqn_slope_unc)
    mkplot(yint_outfile, slope, ran_yint_unc, corr_yint_unc,
        both_yint_unc, eqn_yint_unc,
        ylab='Y-Intercept Unc. [e-]')
```

I also added `__author__ = 'M.E. MySelf'` and `__version__ = 0.2`. These lines are not necessary, but they mean I can do the following:

```
python>  import MyFirstScript
python>  MyFirstScript.__version__
python>  MyFirstScript.__author__
```

EXERCISES

Exercise 17 :
What version of NumPy are you using? Does NumPy list an author?

Notice in the definition of our mkplot function that we give the variable 'ylab' a default value. Now we only have to assign a y-axis label when we do not want the default (i.e. the y-intercept case). Finally, mkplot is a function, but not a module.

7.3 Passing Arguments on the Command Line and *argparse*

What if we wanted to run this program, but for other input files? A simple solution would be to just open up our script and edit the file name. However, a more user friendly method would be to allow it to be entered on the command line. *argparse* is a module that makes this process easy. Look at the use of *argparse* in the code below. This code can be run in the following ways:

```
≫  MyFirstScript.py
≫  MyFirstScript.py --help
≫  MyFirstScript.py -f Gordon2005_Fig16.txt
≫  MyFirstScript.py --file Gordon2005_Fig16.txt
```

For more information, check out the link http://docs.python.org/dev/library/argparse.html

```
#! /usr/bin/env python

#Header

__author__ = 'M.E. MySelf'
__version__ = 0.2

import numpy as np
```

```
import matplotlib.pyplot as plt
import argparse

def mkplot(outfile,xx,yy1,yy2,yy3,yy4,ylab='Slope Unc. [e-/s]'):
    figure, ax = plt.subplots()

    ax.set_xlabel('Slope [e-/s]')
    ax.set_ylabel('Slope Uncertainty [e-/s]')
    ax.plot(xx, yy1, ls='--', color='b', label='Randon Unc.')
    ax.plot(xx, yy2, ls=':', color='r', label='Correlated Unc.')
    ax.plot(xx, yy3, ls='-', color='g', label='Both')
    ax.plot(xx, yy4, ls='-.', color='m', label='Equation')
    ax.legend(loc='best')
    figure.show()

    figure.savefig(outfile)
    figure.clf()
    print 'Saved file to: ',outfile
    return

if __name__=='__main__':

    parser = argparse.ArgumentParser(description='Make a plot.')
    parser.add_argument('-f','--file', default='Gordon2005_Fig16.txt',
        type=str, help='Input file.')
    options = parser.parse_args()

    infile = options.file
    slope_outfile = 'fig16_slope.pdf'
    yint_outfile = 'fig16_yint.pdf'

    slope, ran_slope_unc, corr_slope_unc, both_slope_unc,  \
        eqn_slope_unc, ran_yint_unc, corr_yint_unc,  \
        both_yint_unc, eqn_yint_unc = np.loadtxt(infile,
        unpack=True)
    mkplot(slope_outfile, slope, ran_slope_unc, corr_slope_unc,
        both_slope_unc, eqn_slope_unc)
    mkplot(yint_outfile, slope, ran_yint_unc, corr_yint_unc,
        both_yint_unc, eqn_yint_unc,
        ylab='Y-Intercept Unc. [e-]')
```

EXERCISES

Exercise 18 :

Write your script MyFirstScript.py and execute it using all of the examples above. See what happens if you enter something that is not allowed. Edit your mkplot function so that it is more versatile.

7.4 Error Handling and pdb.set_trace()

Debugging can be a difficult and long process, but the pdb module can help. pdb is the Python debugger. There are many ways to use it, but a common method is with pdb.set_trace(). To debug your code using this function, import pdb and insert the line `pdb.set_trace()` into your code before the part you are unsure about. Execute the program. Once the interpreter reaches the pdb.set_trace() mark, it will allow you to interact with the code and print variables as they are defined. Common pdb.set_trace() commands include:

- (n)ext: continue to the next line
- (l)ist: list a few steps
- (b)reak: give line and file to break at
- (s)tep: moves into deeper calls (i.e. a function in NumPy, or our function mkplot).
- (c)ontinue: continue the program like normal.

EXERCISES

Exercise 19 :

Add `import pdb` to your list of imports in your MyFirstScript.py file. Insert the line `pdb.set_trace()`. Execute your code and step through the program, print out variables to make sure they are what they are supposed to be, and step into your mkplot function. Let the program continue.

8

Resources

8.1 Useful Links

Below is a list of links to be used as a reference.

- http://docs.python.org/
- http://legacy.python.org/dev/peps/pep-0008/
- http://www.astropy.org/
- http://wiki.python.org/moin/HowTo/Sorting
- http://ipython.scipy.org/moin/Documentation
- http://matplotlib.sourceforge.net/
- http://www.scipy.org/Numpy_Example_List_With_Doc
- http://docs.scipy.org/doc/
- http://www.scipy.org/Cookbook

The following links are for further training and building of your Python skills.

- http://stsdas.stsci.edu/perry/pydatatut.pdf
- http://www.scipy.org/Additional_Documentation/Astronomy_Tutorial?action=show
- http://python4astronomers.github.com/
- http://code.google.com/edu/languages/google-python-class/
- http://learnpythonthehardway.org/book/
- http://www.pythonchallenge.com/

8.2 Mailing Lists

These python themed STScI e-mail lists are available through MajorDomo at: http://www.stsci.edu/cgi-bin/jDomo.tcl.

- pylunch: A mailing list for a lunch group that presents and discusses python related material.
- python-interested: A mailing list usually used to discuss bugs, fixes, and how to do some outrageous tasks that astronomers come up with.