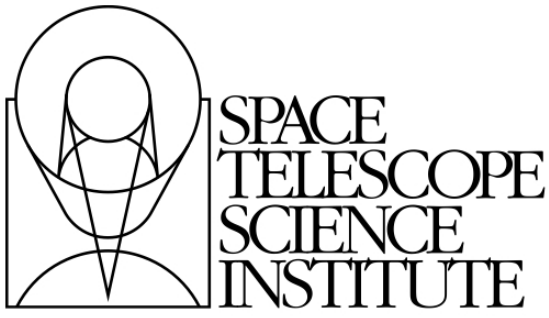
Version 4.0
September 2013

Using Python for Astronomical Data Analysis

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



Space Telescope Science Institute
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: bourque@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 6. 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>  print 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 6.

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.

EXAMPLES

```
python>  a = range(11)
python>  for x in a: print x
python>  b = ['2005', '2006', '2007']
python>  for x in b: print x + '-01'
```

One can create a list with a for loop in one line. This is known as list comprehension.

```
python>   c = [x*2 for x in range(11)]
```

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

EXERCISES
Exercise 4 :
Create a list 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 * 3.0
...     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 a number in range(11). What happens when 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'. A list is returned containing any matching files.

EXAMPLES

```
python>  import glob
python>  glob.glob('*')
python>  glob.glob('*.pdf')
python>  datadir = '/Users/gunning/Python_Training/'  #insert a usable path
python>  glob.glob(datadir + 'flux_vs_time_?.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 interpreted 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_raw.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>  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'
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(infile,
                 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 3, using the two files 'flux_vs_time_A.dat' and 'flux_vs_time_C.dat' found in /user/gunning/Python_Training/.

```
python>  from astropy.io import ascii
python>  data_A = ascii.read('flux_vs_time_A.dat', names=['Time',  \
... 'Flux_diff', 'Flux_err', 'Flux_linear_fit'])
python>  data_C = ascii.read('flux_vs_time_C.dat', names=['Time',  \
... 'Flux_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.

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_diff'], c='blue')
python>  ax.scatter(data_C['Time'], data_C['Flux_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')
```

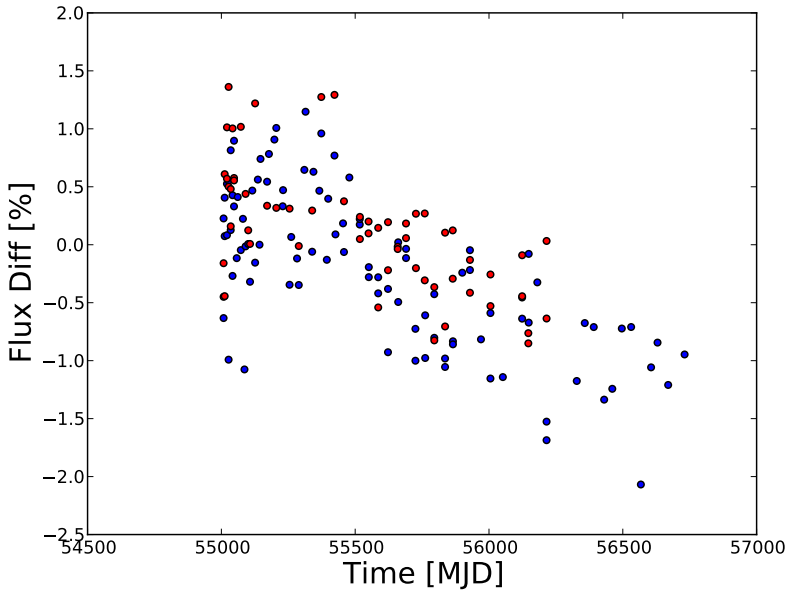


FIGURE 4.1: Our first plot.

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 *axis* object with the *ax.clear()* command.

```
python>  ax.clear()
python>  ax.scatter(data_A['Time'], data_A['Flux_diff'], \
...   c='blue', marker='x', s=30, alpha=0.75)
python>  ax.scatter(data_C['Time'], data_C['Flux_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 cleaner if we added a legend.

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

```
python>  ax.scatter(data_A['Time'], data_A['Flux_diff'], \
...   c='blue', marker='x', s=25, alpha=0.75, label='Amp A')
python>  ax.scatter(data_C['Time'], data_C['Flux_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()
```

Notice the *ls* option is how we select our line type. 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

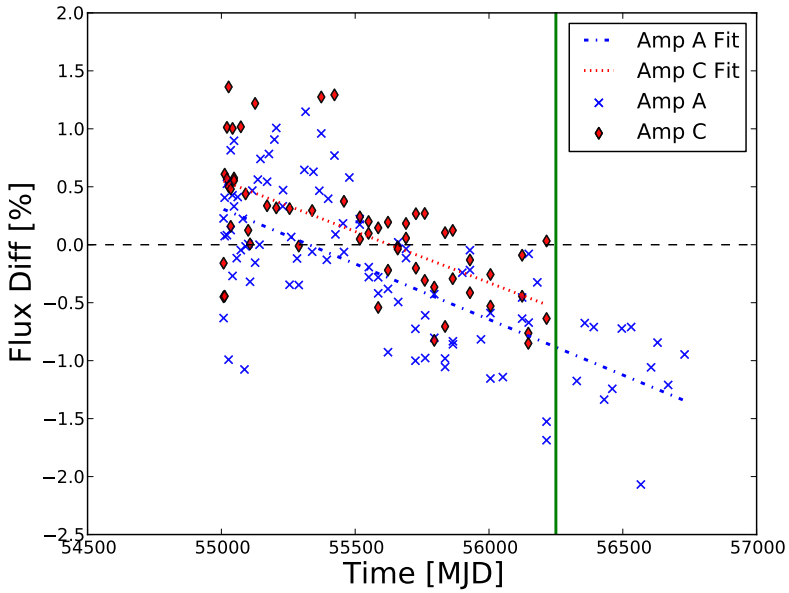


FIGURE 4.2: Our customized plot.

4.4 Error Bars

We have columns giving our error in flux percent difference. 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_diff'], \
... c='blue', s=10)
python>  ax.errorbar(data_A['Time'], data_A['Flux_diff'], \
... yerr=data_A['Flux_err'], c='k', marker=None, ls='None')
python>  figure.show()
```

4.5 Display Year-Month-Day Dates as Tick Labels

MJD is a convenient format for plotting time. But who thinks in MJD? Let's convert MJD to year-month-day and display them on the x-axis. In Python this is a little tricky. We will show one way to do it; you may be able to find a better way.

Let's continue working on the plot we began in the Error Bars section. We first need to import

Time from *astropy*, and then save into a list the MJD dates that are currently displayed as tick labels (converting them into floats as we do so).

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

We next need to convert the MJD date list into a *Time* object.

```
python>  time_convert = Time(time_MJD, format='mjd', scale='utc')
```

Now we can convert the MJD dates into year-month-day dates. Because the conversion also includes minutes and seconds, which we do not want in this example, we will use the *split()* method to extract just the year-month-days and append them into a new list.

```
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])
```

Finally we can display our new tick labels on the x-axis and save the figure.

```
python>  ax.set_xticklabels(time_ymd_short)
python>  ax.set_xlabel('Time', fontsize=20)
python>  ax.set_ylabel('Flux Diff [%]', fontsize=20)
python>  figure.show()
python>  figure.savefig('flux_vs_time_3.pdf')
```

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

4.6 Display Plots Side-by-Side

As with most plotting matters in Python, there's more than one way to display multiple plots on the same figure. We will demonstrate making a simple 2×1 multiple plot with just the *subplot()* method.

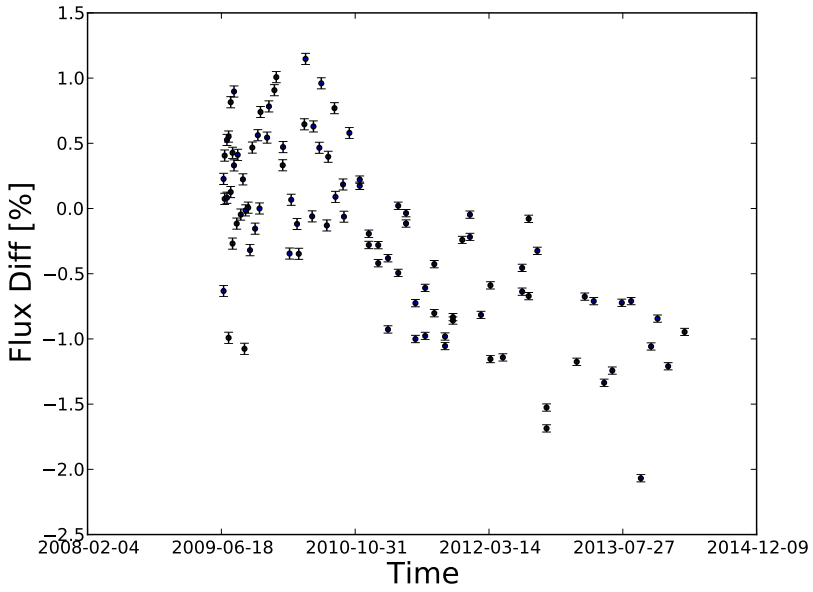


FIGURE 4.3: Our error bar plot with x-axis tick labels converted from MJD to year-month-day.

Let's make a new figure, resizing it to 10×10 inches so we can better see our plots. Then we'll place the first plot at position 211. The first number (2) denotes how many plots are to be placed vertically; the second number (1) denotes how many plots are to be placed horizontally; and the third number (1) denotes where the current plot is to be placed in this grid (in this case, the first position).

```
python>  figure = pyplot.figure(figsize = (10,10))
python>  ax1 = pyplot.subplot(211, title='Amp A')
python>  ax1.scatter(data_A['Time'], data_A['Flux_diff'], c='blue')
```

We can make the top plot's x-axis invisible...

```
python>  pyplot.setp(ax1.get_xticklabels(), visible=False)
```

We next place the second plot at position 212. We use the *sharex* and *sharey* options to force the second plot's axes to match the first's.

```
python>  ax2 = pyplot.subplot(212, title='Amp C', sharex=ax1, sharey=ax1)
python>  ax2.scatter(data_C['Time'], data_C['Flux_diff'], c='red')
```

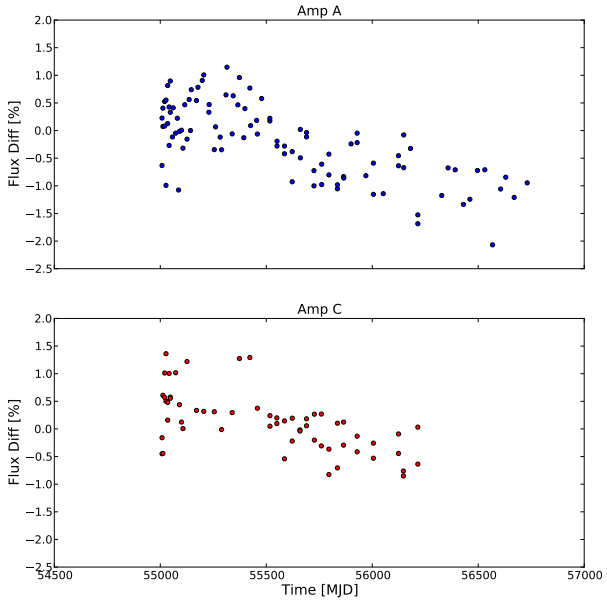


FIGURE 4.4: Our 2×1 multiple plot.

We'll finish by setting the x and y axes labels and saving the figure.

```
python>  ax1.set_ylabel('Flux Diff [%]', fontsize=15)
python>  ax2.set_ylabel('Flux Diff [%]', fontsize=15)
python>  ax2.set_xlabel('Time [MJD]', fontsize=15)
python>  figure.show()
python>  figure.savefig('flux_vs_time_4.pdf')
```

4.7 Display a FITS Image

Copy the FITS file 'ibsa01fpq_flt.fits' from /user/gunning/Python_Training/ to your current working directory. It is a WFC3/IR image of a corner of the galaxy M51.

We will use astropy's *fits* package to open the file, as we learned in Chapter 3, and extract the image data from the SCI extension.

```
python>  import astropy.io.fits as fits
```

```
python>  image = fits.open('ibsa01fpq_flt.fits')
python>  sci_data = image['sci'].data
python>  pylplot.clf()
python>  pyplot.gray()
python>  pyplot.imshow(sci_data, vmin=0, vmax=60)
python>  pyplot.show()
python>  image.close()
```

How does that look? Mess with the *vmin* and *vmax* to see if you can improve the scaling. The *pyplot.gray()* command sets the plot to greyscale. Take a look at the plot without it.

We can add labels:

```
python>  pyplot.title('M51', fontsize=20)
python>  pyplot.xlabel('x pixels', fontsize=15)
python>  pyplot.ylabel('y pixels', fontsize=15)
```

Before saving, we can try annotating features in our image. In this example, we point to a star near pixel coordinates (640, 810) with an arrow.

```
python>  pyplot.annotate('star', xy=(640,810),  xytext=(520,770), \
... color='white', arrowprops=dict(facecolor='white', width=3.5))
python>  pyplot.savefig('M51.pdf')
```

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, thumbnails, and examples.

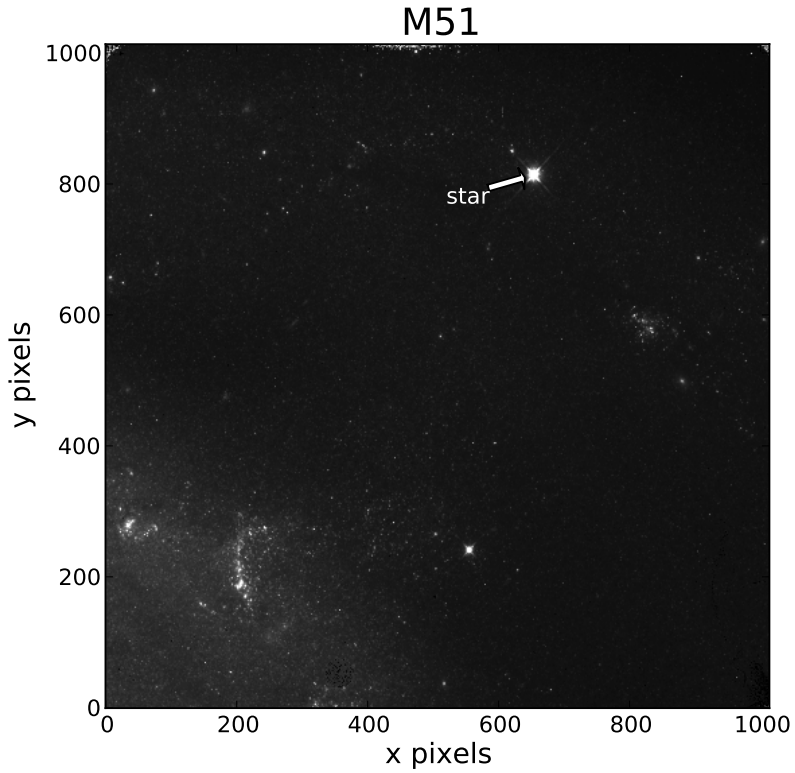


FIGURE 4.5: A corner of the galaxy M51, displayed in greyscale using *pyplot.imshow()*.

5

IPython Notebook

5.1 Getting Started

IPython Notebook is an interactive web-based tool which combines code execution, mathematics, and rich media output.

You can find a plethora of information at http://ipython.org/ipython-doc/dev/notebook/index.html

Let's get started. The data needed for this chapter can be found in /user/gunning/Python_Training/.

To open a notebook window, open a terminal and initiate Ureka:

```
≫  ur_setup common ssbx
≫  ipython notebook
```

Opening the notebook this way will create a new tab in your browser. All subsequent interactions will now be through the browser and the terminal will be running the server that the browser interacts with.

IPython Notebook will look for notebook files (.ipynb) in the directory where you started the notebook in. Once in the notebook window, to execute a cell you must hit shift+enter.

5.2 Plotting Inline

A unique feature that the notebook has is the ability to plot inline.

You can initiate inline plotting two ways.

From the terminal:

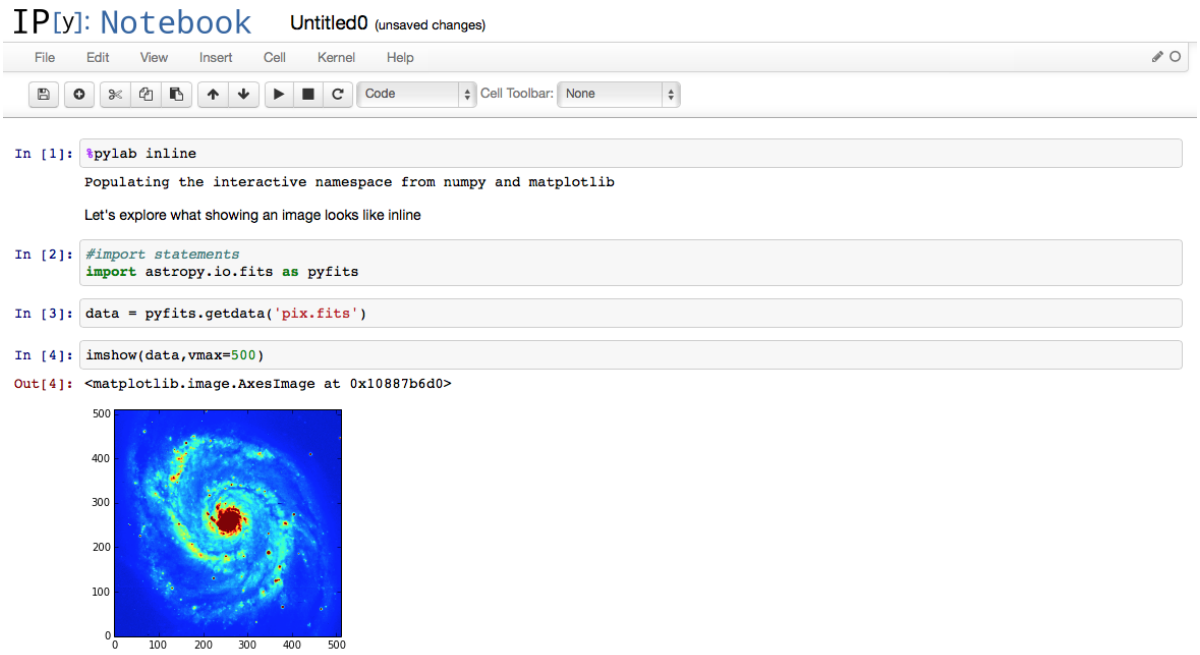
```
≫  ur_setup common ssbx
≫  ipython notebook --pylab inline
```

From the notebook window:

```
python>  %pylab inline
```

Without telling the notebook to plot inline, plots will pop up in a separate window just as they normally would plotting in the python terminal.

5.3 Notebook Examples



IP[y]: Notebook Untitled0 (autosaved)

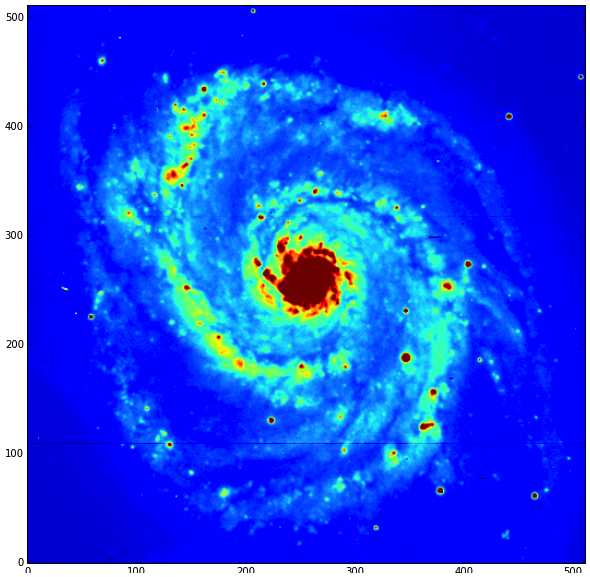
File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

You can change the size of the inline image with figsize

In [7]:
```
figsize(10,10)
imshow(data,vmax=500)
```

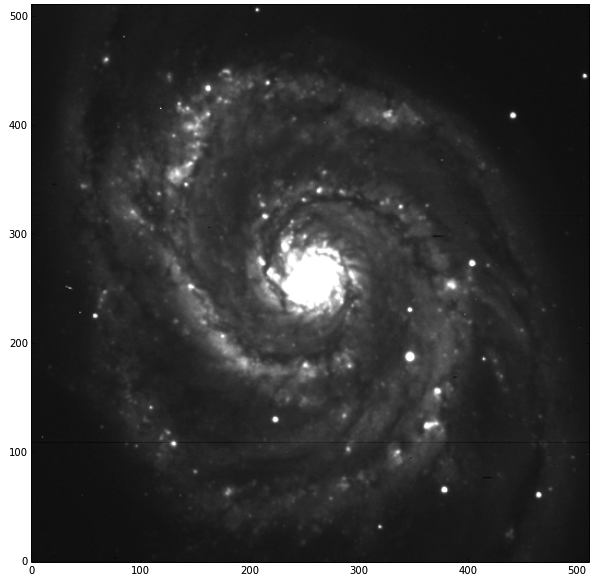
Out[7]: <matplotlib.image.AxesImage at 0x10a0f3a10>



You can also change the colors of displayed images using colormaps.

In [9]:
```
imshow(data,vmax=500,cmap=cm.gray)
```

Out[9]: <matplotlib.image.AxesImage at 0x10a0d6790>



Since figsize was set not attributed to a specific plot or image, it will apply to subsequent cells.

IP[y]: Notebook Untitled0 (autosaved)

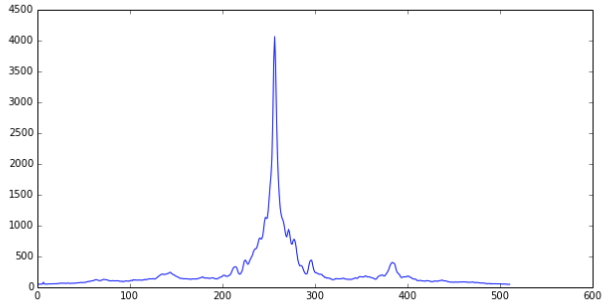
File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

```
In [16]: figsize(10,5)

In [17]: plot(data[256])

Out[17]: [<matplotlib.lines.Line2D at 0x10a132a10>]
```



This is a basic line plot of the 256th row in the image

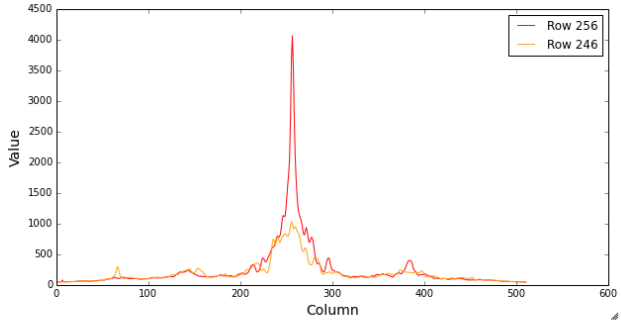
IP[y]: Notebook Untitled0 (autosaved)

File Edit View Insert Cell Kernel Help

Code Cell Toolbar: None

```
In [20]: plot(data[256],color='red',label='Row 256')
         plot(data[246],color='orange',label='Row 246')
         xlabel('Column',fontsize=14)
         ylabel('Value',fontsize=14)
         legend()

Out[20]: <matplotlib.legend.Legend at 0x101ccc650>
```



A bit fancier of a figure, but still quite basic. All the functionality of iPython works in the notebook as well.

```
In [22]: #sum of all the values in data
         data.sum()

Out[22]: 28394234

In [23]: #max value in data
         data.max()

Out[23]: 19936
```

if you type data. then hit tab in a cell it will give you options for data.

6

Python Scripts and Functions

In this chapter, we will create a python script that performs some basic calibration/monitoring tasks for WFC3/UVIS "hot pixels" found in DARK images. The goal is to create a module that serves as simple yet practical example of a script that one might write for work on an instrument team (generally speaking of course – if this script were to be used as part of a real calibration, it would likely be expanded and reconfigured to be more robust). In addition, we will explore some of python's "best practices" that promote code readability, consistency, reusability, etc. Our example will use standard conventions, docstrings, and minimal redundancy. By no means should this style be used as a hard rule or guideline, but rather a suggestion for producing high-level, easy-to-use python code.

Please choose a text editor for editing your script, such as SublimeText, Emacs, TextWrangler, NEdit, or others.

6.1 hotpix_monitor.py

Our task at hand is to create a script that will construct a plot of the number of hot pixels found in a collection of WFC3/UVIS DARK images versus time. To determine if a pixel is "hot" or not, we will implement a threshold that the user of the script can supply via a command-line argument; if a pixel's value is greater than this threshold, it is considered to be a hot pixel. The

dark images are located in /user/gunning/Python_Training/uvis_darks/, and like all full-frame UVIS images, they contain a primary header (extension 0), two SCI extensions (extensions 1 and 4), two ERR extensions (extensions 2 and 5), and two DQ extensions (extensions 3 and 6). However, we will only concern ourselves with extensions 0, 1, and 3 in this example.

Create a file called hotpix_monitor.py in your script directory, and open it in your favorite editor. It is best practice to name scripts (or modules) with all-lowercase, using underscores to separate words. The name of your script should describe what the script does. In this case, we chose hotpix_monitor.py because our script will monitor hot pixels!

The first thing we will put in our script is this line:

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

This tells the computer that the python interpreter should be used for executing this file.

Comments, or "docstrings" 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 docstring. For our module docstring for our module. We will include (1) A description of what the script does, (2) The author(s) of the script, (3) how the script is used, and (4) a descrption of any output products. Our docstring should look something like this:

```
"""Monitors the hot pixel evolution of WFC3/UVIS darks.

Authors:
    First Last, MMM YYYY

Use:

Outputs:

"""
```

We don't quite know how the script will be used yet, or what the output products will be, so we will fill those sections in after we have written the script. It is best practice to use three quotes (""") at the beginning and end of your docstring, and to keep the lines in the docstrings to maximum of 72 characters long.

Next, after the module docstring, we will import the various modules that we will need:

```
import argparse
from astropy.io import fits
import glob
import numpy as np
import matplotlib.pyplot as plt
```

Notice that the import statements are in alphebetical order. This is yet another "best practice"; each import should be on its own line and in alphabetical order.

Now we need to make our script useable from the command line. To do this, we will add the following line after the import statements:

```
if __name__ == '__main__':
    # Code goes here.
```

This allows any code placed under the if __name__ == '__main__' statement to be executed when the script is called from the command line. Technically, we could put all of our code that we will type under this code block, and it would be executed when the script is called from the command line. However, it is better practice to make the code more flexible by adding separate functions, each of which perform a separate task. By having separate functions, not only could our module be executed from the command line, but its individual functions could also be imported by other python modules and/or used within the python environment itself. This could prove to be useful if we ever want to use the functionality that this module has to offer in other scripts. The following shows an example of how to declare a function:

```
def some_function():
    # Some code
```

Thus, for our script, we want our code to follow this generic workflow:

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

"""
Module docstring
"""

def main_function():
    """
    Function docstring
```

```
    """
    # some code

if __name__=='__main__':
    main_function()
```

In this way, the code contained in the main function will be executed when the script is called from the command line *and also* can be imported by other modules.

Before we start adding code to find hot pixels and make plots, let's add a function that implements the threshold command line argument. We will do this using the argparse module. Add the following code block between your import statements and the if __name__ == '__main__' block:

```
def parse_args():
    """Parse command line arguments.

    Parameters:
        nothing

    Returns:
        args : argparse.Namespace object
            An argparse object containing all of the added arguments.

    Outputs:
        nothing
    """

    parser = argparse.ArgumentParser()
    parser.add_argument('-t --threshold',
        dest='threshold',
        action='store',
        type=float,
        required=False,
        help='Pixel value to be used as hot pixel threshold (in counts)',
        default=9.0)
    args = parser.parse_args()

    return args
```

Notice that our function docstrings are a little different from our module docstring. Typi-

cally, users are not concerned with how individual functions are used, but rather what parameters/variables are needed or returned by the function. Thus, in the function docstring, we include descriptions of (1) what the function does, (2) the parameters that the function requires, (3) the object(s) returned by the function, and (4) any output(s) that the function produces upon execution. Also notice that we split the parser.add_argument command into several lines. This is because it is best practice to limit non-docstring lines to 79 characters.

By grabbing what is returned from the parse_args() function, we can now utilize the threshold parameter. Add a call to the parse_args() function, as such:

```
if __name__=='__main__':
    args = parse_args()
```

This should leave our script looking something like this:

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

"""Monitors the hot pixel evolution of WFC3/UVIS darks.

Authors:
    First Last, MMM YYYY

Use:

Outputs:

"""

import argparse
from astropy.io import fits
import glob
import numpy as np
import matplotlib.pyplot as plt


def parse_args():
    """Parse command line arguments.

    Parameters:
        nothing
```

```
    Returns:
        args : argparse.Namespace object
            An argparse object containing all of the added arguments.

    Outputs:
        nothing
    """

    parser = argparse.ArgumentParser()
    parser.add_argument('-t --threshold',
        dest='threshold',
        action='store',
        type=float,
        required=False,
        help='Pixel value to be used as hot pixel threshold (in counts)',
        default=9.0)
    args = parser.parse_args()

    return args


if __name__=='__main__':
    args = parse_args()
```

We know that the script must calculate the number of hot pixels in a several images, so adding another function that will take an image and the threshold as parameters and return the number of hot pixels and image's observation date would be useful; we could then simply call this function for each image in our image list and obtain the information that we need to produce the plot. For convenience, we will calculate the percentage of the detector occupied by hot pixels, instead of just returning the total number of hot pixels. Thus, let's call our new function get_percentage_hotpix(), and place it into our script:

```
def get_percentage_hotpix(image, threshold):
    """Determine the observation date and number of hot pixels for the
    image.

    Parameters:
        image : string
```

```
            The path to the image to process.
        threshold : float
            The threshold above which pixels will be considered hot
            pixels.

    Returns:
        mjd : float
            The Modified Julian Date of the image.
        percentage_hotpix : float
            The percentage of hot pixels in SCI extension of the image.

    Outputs:
        nothing
    """

    # Open the file
    hdulist = fits.open(image)

    # Get data from the SCI and DQ extensions
    sci = hdulist[1].data
    dq = hdulist[3].data

    # Get the MJD of observation
    mjd = hdulist[0].header['EXPSTART']

    # Determine which pixels are hot pixels
    good_data = sci[np.where(dq == 0)]
    hotpix = good_data[np.where(good_data > threshold)]

    # Calculate the percentage of hot pixels
    num_pix = float(sci.size)
    num_hotpix = float(hotpix.size)
    percentage_hotpix = (num_hotpix / num_pix) * 100.

    return mjd, percentage_hotpix
```

Now our script should look something like this:

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

```
"""Monitors the hot pixel evolution of WFC3/UVIS darks.

Authors:
    First Last, MMM YYYY

Use:

Outputs:

"""

import argparse
from astropy.io import fits
import glob
import numpy as np
import matplotlib.pyplot as plt


def get_percentage_hotpix(image, threshold):
    """Determine the observation date and number of hot pixels for the
    image.

    Parameters:
        image : string
            The path to the image to process.
        threshold : float
            The threshold above which pixels will be considered hot
            pixels.

    Returns:
        mjd : float
            The Modified Julian Date of the image.
        percentage_hotpix : float
            The percentage of hot pixels in SCI extension of the image.

    Outputs:
        nothing
    """

    # Open the file
    hdulist = fits.open(image)
```

```
    # Get data from the SCI and DQ extensions
    sci = hdulist[1].data
    dq = hdulist[3].data

    # Get the MJD of observation
    mjd = hdulist[0].header['EXPSTART']

    # Determine which pixels are hot pixels
    good_data = sci[np.where(dq == 0)]
    hotpix = good_data[np.where(good_data > threshold)]

    # Calculate the percentage of hot pixels
    num_pix = float(sci.size)
    num_hotpix = float(hotpix.size)
    percentage_hotpix = (num_hotpix / num_pix) * 100.

    return mjd, percentage_hotpix


def parse_args():
    """Parse command line arguments.

    Parameters:
        nothing

    Returns:
        args : argparse.Namespace object
            An argparse object containing all of the added arguments.

    Outputs:
        nothing
    """

    parser = argparse.ArgumentParser()
    parser.add_argument('-t --threshold',
        dest='threshold',
        action='store',
        type=float,
        required=False,
        help='Pixel value to be used as hot pixel threshold (in counts)',
```

```
        default=9.0)
    args = parser.parse_args()

    return args


if __name__=='__main__':
    args = parse_args()
```

Notice that the get_percentage_hotpix() function is placed before the parse_args() function, even though the parse_args() function is executed before get_percentage_hotpix(). This is because it is good practice to put functions in alphabetical order for readability purposes. Similarly, it is best practice to leave the if __name__=='__main__' codeblock at the end of the program.

Now we are ready to implement the core functionality; gathering hot pixel statistics and plotting them versus time. Again, we will do this in a separate function. Let's call it plot_hot_pixels(). In the function, we will have to (1) define the list of DARK images to process, (2) find the percentage of hot pixels for each image using the threshold parameter, (3) plot the percentage of hot pixels versus time, and (4) save the figure. For fun, we will also plot a vertical line representing when a UVIS anneal occurred, in which the detector was warmed up to reduce the number of hot pixels. Our function should look something like this:

```
def plot_hot_pixels(threshold):
    """Plots WFC3/UVIS hot pixel evolution.

    Parameters:
        threshold : float
            The threshold above which pixels will be considered hot
            pixels.

    Returns:
        nothing

    Outputs:
        hotpix.png - A plot showing the number of hot pixels over time
    """

    # Define image list
    dark_files = glob.glob('/user/gunning/Python_Training/uvis_darks/*.fits')
```

```
    # Initialize plot
    figure, ax = plt.subplots()
    ax.grid()
    ax.minorticks_on()
    ax.set_title('WFC3/UVIS Hot Pixel Evolution')
    ax.set_xlabel('Days from anneal')
    ax.set_ylabel('Number of Hot Pixels (% of Chip)')

    # Set the anneal date
    anneal = 56687.4698727

    # Plot the percentage of hot pixels
    for dark_file in dark_files:

        print 'Processing {}'.format(dark_file)

        # Determine observation time and number of hot pixels
        time, hotpix = get_percentage_hotpix(dark_file, threshold)

        # Plot the number of hot pixels vs time
        ax.scatter(time - anneal, hotpix, s=50, c='k', marker='+')

    # Plot the anneal
    ax.axvline(x=0, c='r', ls='--', lw=2, label='Anneal')

    # Place the legend
    ax.legend(loc='best')

    # Save the figure
    savefile = 'hotpix_{}.png'.format(str(threshold))
    plt.savefig(savefile)
    print 'Saved figure to {}'.format(savefile)
```

Great! Now we just need to call this function in our __main__ code block with our threshold parameter:

```
if __name__ == '__main__':

    args = parse_args()
    plot_hot_pixels(args.threshold)
```

We now have all of the pieces in place, and know what our script does. Thus, we can update our module docstrings to provide futher explainiation:

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

"""Monitors the hot pixel evolution of WFC3/UVIS darks.

Authors:
    First Last, MMM YYYY

Use:
    This script is inteded to be executed from the command line as
    such:

        >>> python dark_monitor.py

    The hot pixel threshold (in counts) can be specified with the -t
    or --threshold argument.  For example:

        >>> python dark_monitor.py -t 8.5

    If no threshold is specified, the default value of 9.0 counts is
    used.

Outputs:

    hotpix_<threshold>.png -- A plot showing the number of hot pixels
                              in each dark as a function of MJD,
                              placed in the current working directory.

References:

    The Python RIAB Training Document
    (https://confluence.stsci.edu/display/INSRIA/RIA+training)
"""
```

So, our entire script should look something like this:

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

```
"""Monitors the hot pixel evolution of WFC3/UVIS darks.

Authors:
    First Last, MMM YYYY

Use:
    This script is inteded to be executed from the command line as
    such:

        >>> python dark_monitor.py

    The hot pixel threshold (in counts) can be specified with the -t
    or --threshold argument.  For example:

        >>> python dark_monitor.py -t 8.5

    If no threshold is specified, the default value of 9.0 counts is
    used.

Outputs:

    hotpix_<threshold>.png -- A plot showing the number of hot pixels
                              in each dark as a function of MJD,
                              placed in the current working directory.

References:

    The Python RIAB Training Document
    (https://confluence.stsci.edu/display/INSRIA/RIA+training)
"""

import argparse
from astropy.io import fits
import glob
import numpy as np
import matplotlib.pyplot as plt


def get_percentage_hotpix(image, threshold):
    """Determine the observation date and number of hot pixels for the
    image.
```

```
    Parameters:
        image : string
            The path to the image to process.
        threshold : float
            The threshold above which pixels will be considered hot
            pixels.

    Returns:
        mjd : float
            The Modified Julian Date of the image.
        percentage_hotpix : float
            The percentage of hot pixels in SCI extension of the image.

    Outputs:
        nothing
    """

    # Open the file
    hdulist = fits.open(image)

    # Get data from the SCI and DQ extensions
    sci = hdulist[1].data
    dq = hdulist[3].data

    # Get the MJD of observation
    mjd = hdulist[0].header['EXPSTART']

    # Determine which pixels are hot pixels
    good_data = sci[np.where(dq == 0)]
    hotpix = good_data[np.where(good_data > threshold)]

    # Calculate the percentage of hot pixels
    num_pix = float(sci.size)
    num_hotpix = float(hotpix.size)
    percentage_hotpix = (num_hotpix / num_pix) * 100.

    return mjd, percentage_hotpix


def parse_args():
```

```
    """Parse command line arguments.

    Parameters:
        nothing

    Returns:
        args : argparse.Namespace object
            An argparse object containing all of the added arguments.

    Outputs:
        nothing
    """

    parser = argparse.ArgumentParser()
    parser.add_argument('-t --threshold',
        dest='threshold',
        action='store',
        type=float,
        required=False,
        help='Pixel value to be used as hot pixel threshold (in counts)',
        default=9.0)
    args = parser.parse_args()

    return args


def plot_hot_pixels(threshold):
    """Plots WFC3/UVIS hot pixel evolution.

    Parameters:
        threshold : float
            The threshold above which pixels will be considered hot
            pixels.

    Returns:
        nothing

    Outputs:
        hotpix.png - A plot showing the number of hot pixels over time
    """
```

```
    # Define image list
    dark_files = glob.glob('/user/gunning/Python_Training/uvis_darks/*.fits')

    # Initialize plot
    figure, ax = plt.subplots()
    ax.grid()
    ax.minorticks_on()
    ax.set_title('WFC3/UVIS Hot Pixel Evolution')
    ax.set_xlabel('Days from anneal')
    ax.set_ylabel('Number of Hot Pixels (% of Chip)')

    # Set the anneal date
    anneal = 56687.4698727

    # Plot the percentage of hot pixels
    for dark_file in dark_files:

        print 'Processing {}'.format(dark_file)

        # Determine observation time and number of hot pixels
        time, hotpix = get_percentage_hotpix(dark_file, threshold)

        # Plot the # of hot pixels vs time
        ax.scatter(time - anneal, hotpix, s=50, c='k', marker='+')

    # Plot the anneal
    ax.axvline(x=0, c='r', ls='--', lw=2, label='Anneal')

    # Place the legend
    ax.legend(loc='best')

    # Save the figure
    savefile = 'hotpix_{}.png'.format(str(threshold))
    plt.savefig(savefile)
    print 'Saved figure to {}'.format(savefile)


if __name__ == '__main__':

    args = parse_args()
    plot_hot_pixels(args.threshold)
```

6.2 Executing Python Scripts

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

```
≫ python hotpix_monitor.py
```

Or, if you are already inside the Python interactive environment, just import the module and call its main function:

```
python> import hotpix_monitor
python> hotpix_monitor.plot_hot_pixels(9.0)
```

Lastly, since we put #/usr/bin/env python at the top of our program, we can execute it directly:

```
≫ hotpix_monitor.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. To be able to call your script from anywhere on your computer, add your script directory to your executable PATH:

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

EXERCISES
Exercise 15 :
Write the script hotpix_monitor.py *and execute it.*
Exercise 16 :
Try executing hotpix_monitor.py *with a different threshold. Did the plot change?*
Exercise 17 :
Add another command line argument called -s *or* –savefile*, which stores the plot saving location. Use this new argument to replace the hard-coded save location. Then, try executing the script and saving to a location that you pass as an argument.*

6.3 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 18 :
Add `import pdb` *to your list of imports in your* hotpix_monitor.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. If they are, let the program continue.*

6.4 More about Python Coding Styles and Best Practices

In this chapter, we have touched on some of the "best practices" that are recommended when coding in Python. Two good resources for learning about these best practices are PEP-8 (http://legacy.python.org/dev/peps/pep-0008/), which covers general conventions, and PEP-257 (http://legacy.python.org/dev/peps/pep-0257/), which discusses docstrings. If you have an interest in scripting in python, please take the time to review these two documents.

7

PyRAF

7.1 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.

7.2 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 some JWST/MIRI data, n9vf01hfq_ima.fits, which can be found in /user/gunning/Python_Training/.

Import the PyRAF module and from within that module import the IRAF instance. The code for

importing these functions looks like this:

```
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.

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

EXERCISES

Exercise 19 :
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')
```

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 the Outlook Web App (OWA):

- 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.