Python4Cardiff Documentation

Release 1.0

Tom Aldcroft, Brian Refsdal, Tom Robitaille

April 26, 2013

CONTENTS

Note: Please bring your laptop to the workshop!

The scope of the workshop is not to give a complete introduction to Python, but instead to make you familiar with the basics of Python, numerical analysis, plotting, and Astronomy packages, so that you can directly use it in your work/research, and hopefully learn more! The functionality is presented in such a way that beginners in Python should be able to follow, but please ensure that you have a functional Python distribution installed prior to the workshop.

CHAPTER

ONE

SCHEDULE

Time	Topic
10:00am	Why use Python?
10:30am	Python installation clinic
11:00am	Introduction to Scientific Python
1:00pm	Lunch
2:00pm	Python in Astronomy (hands-on)
3:00pm	Coffee
5:00pm	Pub!

CHAPTER

TWO

TOPICS

2.1 Optional: Emergency Installation

If you already have a Scientific Python distribution, but not Astropy and APLpy, just do:

```
pip install astropy
pip install aplpy
```

If you have *not* previously installed a Scientific Python distribution, use the following instructions for an EMERGENCY scientific Python install :-)

1. Go to the Anaconda CE Downloads page.
2. Download the file for your platform
3. On Linux or Mac, do:

   ```
   bash <downloaded_file>
   ```

   and answer the questions with the default by just pressing `<enter>`. At the end of the install, you will get a message like:

   ```
   export PATH=/Users/tom/anaconda/bin:$PATH
   ```

   Follow these instructions, rehash or open a new terminal, and then test that if you type `python`, you get a prompt similar to:

   ```
   Python 2.7.3 |AnacondaCE 1.3.1 (x86_64)| (default, Jan 10 2013, 12:10:41)
   [GCC 4.0.1 (Apple Inc. build 5493)] on darwin
   Type "help", "copyright", "credits" or "license" for more information.
   >>>
   ```

   with `AnacondaCE` in the first line.

   On Windows, double click the .exe file to install.
4. Install Astropy with:

   ```
   conda install astropy
   pip install aplpy
   ```

2.2 Introduction to Scientific Python

2.2.1 Running Python code

Before we learn about the actual Python language, we need to know where to enter it. Python can be used either interactively, or from scripts. There is not one right way to do it, and it depends whether you are interested in an interactive exploratory analysis, or running or re-running a more complex program.

Interactive use

To run Python code interactively, one can use the standard Python prompt, which can be launched by typing `python` in your standard shell:

```
$ python
Python 2.7.2 (default, Nov  5 2011, 20:09:20)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The >>> indicates that Python is ready to accept commands. If you type `a=1` then press enter, this will assign the value `1` to `a`. If you then type `a` you will see the value of `a` (this is equivalent to `print a`):

```
>>> a = 1
>>> a
1
```

The Python shell can execute any Python code, even multi-line statements, though it is often more convenient to use Python non-interactively for such cases.

The default Python shell is limited, and in practice, you will want instead to use the IPython (or interactive Python) shell. This is an add-on package that adds many features to the default Python shell, including the ability to edit and navigate the history of previous commands, as well as the ability to tab-complete variable and function names. To start up IPython, type:

```
$ ipython
Python 2.7.2 (default, Nov  5 2011, 20:09:20)
Type "copyright", "credits" or "license" for more information.

IPython 0.11 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]:
```

The >>> symbols are now replaced by `In [x]`, and output, when present, is prepended with `Out [x]`. If we now type the same commands as before, we get:

```
In [1]: a = 1

In [2]: a
Out[2]: 1
```

If you now type the up arrow twice, you will get back to `a = 1`.

Note: IPython is much more than just a Python shell with a few improvements compared to the default shell. See the IPython documentation for a full overview of its capabilities!

Running scripts

While the interactive Python mode is very useful to exploring and trying out code, you will eventually want to write a script to record and reproduce what you did, or to do things that are too complex to type in interactively (defining functions, classes, etc.). To write a Python script, just use your favorite code editor to put the code in a file with a `.py` extension. For example, we can create a file called `test.py` containing:

```
a = 1
print a
```

We can then run the script on the command-line with:

```
$ python test.py
1
```

Note: The `print` statement is necessary, because typing `a` on its own will only print out the value in interactive mode. In scripts, the printing has to be explicitly requested with the print command. To print multiple variables, just separate them with a comma after the print command: `print a, 1.5, "spam"`. To print variable within strings use the following syntax: `print ("This is a integer: %d, this is a float: %f, and is a string: %s" % (5, 3.141, "spam"))`

Executable Scripts

It is also possible to make Python scripts executable. Simply add `#!/usr/bin/env python` on the first line of your `test.py` script and change the file permission to make it executable with `chmod +x test.py`. Now the script can be run without the preceeding `python` command; instead you can just type `./test.py` in the command line. Note that this will only work on Linux and Macs, not on Windows.

Combining interactive and non-interactive use

It can sometimes be useful to run a script to set things up, and to continue in interactive mode. This can be done using the `%run` IPython command to run the script, which then gets executed. The IPython session then has access to the last state of the variables from the script:

```
$ ipython
Python 2.7.2 (default, Nov  5 2011, 20:09:20)
Type "copyright", "credits" or "license" for more information.

IPython 0.11 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??' for extra details.

In [1]: %run test.py
1

In [2]: a + 1
Out[2]: 2
```

2.2.2 Python Types and Operations

Python supports a number of built-in types and operations. This tutorial covers the most common types, but information about additional types is available here

Basic numeric types

The basic data numeric types are similar to those found in other languages, including:

- Integers (`int`):

  ```
  >>> i = 1
  ```

- Floating point values (`float`):

  ```
  >>> f = 4.3
  ```

- Complex values (`complex`):

  ```
  >>> c = complex(4., -1.)
  ```

Manipulating these behaves the way you would expect, so an operation (`+`, `-`, `/`, `*`, `**`, etc.) on two values of the same type produces another value of the same type, while an operation on two values with different types produces a value of the more 'advanced' type:

- Adding two integers gives an integer:

  ```
  >>> 1 + 3
  4
  ```

- Multiplying two floats gives a float:

  ```
  >>> 3. * 2.
  6.0
  ```

- Subtracting two complex numbers gives a complex number:

  ```
  >>> complex(2.,4.) - complex(1.,6.)
  (1-2j)
  ```

- Multiplying an integer with a float gives a float:

  ```
  >>> 3 * 9.2
  27.599999999999998  # int * float = float
  ```

- Multiplying a float with a complex number gives a complex number:

  ```
  >>> 2. * complex(-1.,3.)
  (-2+6j)  # float * complex = complex
  ```

- Multiplying an integer and a complex number gives a complex number:

  ```
  >>> 8 * complex(-3.3,1)
  (-26.4+8j)  # int * complex = complex
  ```

However, there is one case where this happens but is not desirable, and that you should be aware of, which is the division of two integer numbers:

```
>>> 3 / 2
1
```

This behavior is widely regarded as a huge design mistake and Python 3.x has been fixed to behave like you would expect (more on Python 3.x later). A way to prevent this is to cast at least one of the integers in the division to a `float`:

```
>>> 3 / float(2)
1.5
```

or:

```
>>> 3 / 2.
1.5
```

Lists

Lists are sequences that can contain inhomogeneous data types:

```
>>> l = [4, 5.5, "spam"]
>>> l[0]
4
>>> l[1]
5.5
>>> l[2]
'spam'
```

One useful operation with lists is +, which can be used for concatenation:

```
>>> [1,2,3] + [4,5,6]
[1, 2, 3, 4, 5, 6]

>>> ('spam', 'egg') + ('more spam','!')
('spam', 'egg', 'more spam', '!')
```

Lists can be *sliced*, meaning that we extract a chunk from the list:

```
>>> a = ['spam', 'egg', 'bacon']
>>> a[0:2]
['spam', 'egg']
```

Appending items to a list is also easy:

```
>>> a = [1, 4, 3]
>>> a.append(5)
>>> a
[1, 4, 3, 5]
```

Strings

Strings (`str`) will be familiar from other programming languages:

```
>>> s = "Spam egg spam spam"
```

You can use either single quotes (`'`), double quotes (`"`), or triple quotes (`"'`) to enclose a string (the last one is used for multi-line strings). To include single or double quotes inside a string, you can either use the opposite quote to enclose the string:

```
>>> "I'm"
"I'm"
```

```
>>> '"hello"'
'"hello"'
```

or you can *escape* them:

```
>>> 'I\'m'
"I'm"

>>> "\"hello\""
'"hello"'
```

You can access individual characters or chunks of characters:

```
>>> s[5]
'e'

>>> s[9:13]
'spam'
```

Note that strings are immutable, that is you cannot change the value of certain characters without creating a new string:

```
>>> s[5] = 'r'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

As for lists, concatenation is done with +:

```
>>> "hello," + " " + "world!"
'hello, world!'
```

Finally, strings have many methods associated with them, here are a few examples:

```
>>> s.upper()
'SPAM EGG SPAM SPAM'   # An uppercase version of the string

>>> s.index('egg')
5  # An integer giving the position of the sub-string

>>> s.split()
['Spam', 'egg', 'spam', 'spam']  # A list of strings
```

Dictionaries

One of the remaining types are dictionaries (`dict`) which you can think of as look-up tables:

```
>>> d = {'name':'m31', 'ra':10.68, 'dec':41.27}
>>> d['name']
'm31'
>>> d['flux'] = 4.5
>>> d
{'flux': 4.5, 'dec': 41.27, 'name': 'm31', 'ra': 10.68}
```

A note on Python objects

Most things in Python are objects. But what is an object?

Every variable or function in Python is actually a object with a type and associated attributes and methods. An *attribute* is a property of the object that you get or set by giving the <object_name> + dot + <attribute_name>, for example `img.shape`. A *method* is a function that the object provides, for example `img.argmax(axis=0)` or `img.min()`.

Use tab completion in IPython to inspect objects and start to understand attributes and methods. To start off create a list of 4 numbers:

```
l = [3, 1, 2, 1]
l.<TAB>
```

This will show the available attributes and methods for the Python list `l`. **Using <TAB>-completion and help is a very efficient way to learn and later remember object methods!**

```
In [17]: l.<TAB>
l.append   l.extend   l.insert   l.remove   l.sort
l.count    l.index    l.pop      l.reverse
```

You can then use `?` to find out more about a specific method:

```
In [2]: l.append?
Type:       builtin_function_or_method
String Form:<built-in method append of list object at 0x101a2d758>
Docstring:  L.append(object) -- append object to end
```

2.2.3 Control Flow Statements

Setting and modifying variables will only get you so far before you need to use control flow statements such as if statements, or loops. This section describes a few of the most basic control flow statements that you will need to get started.

`if` statements

The basic syntax for an if-statement is the following:

```
if condition:
    # do something
elif condition:
    # do something else
else:
    # do yet something else
```

Notice that there is no statement to end the if statement. Notice also the presence of a colon (`:`) after each control flow statement. Python relies on indentation and colons to determine whether it is in a specific block of code. For example, in the following example:

```
if a == 1:
    print "a is 1, changing to 2"
    a = 2
print "finished"
```

The first print statement, and the `a = 2` statement only get executed if `a` is 1. On the other hand, `print "finished"` gets executed regardless, once Python exits the if statement.

Note: Indentation is very important in Python, and the convention is to use four spaces (not tabs) for each level of indent.

Back to the if-statements, the conditions in the statements can be anything that returns a boolean value. For example, `a == 1`, `b != 4`, and `c <= 5` are valid conditions because they return either `True` or `False` depending on whether the statements are true or not. Standard comparisons can be used (== for equal, `!=` for not equal, <= for less or equal, >= for greater or equal, < for less than, and > for greater than), as well as logical operators (`and`, `or`, `not`). Parentheses can be used to isolate different parts of conditions, to make clear in what order the comparisons should be executed, for example:

```
if (a == 1 and b <= 3) or c > 3:
    # do something
```

More generally, any function or expression that ultimately returns `True` or `False` can be used.

Note: In the previous examples we have included comments. All lines starting with the `#` character are ignored by Python. If you would like to comment out a section of code, you can enclose it in trip quotes: `"'commented out code"'`.

`for` loops

The most common type of loop is the `for` loop. In its most basic form, it is straightforward:

```
for value in iterable:
    # do things
```

The *iterable* can be any Python object that can be iterated over. This includes lists, tuples, dictionaries, strings. Try the following in IPython:

```
In [1]: for x in [3, 1.2, 'a']:
   ...:     print x
   ...:
3
1.2
'a'
```

Note that by putting the colon at the end of the first line, IPython automatically knows to indent the next line, so you don't need to indent it yourself. After typing the `print` statement, you need to press enter twice to tell IPython you are finished writing code.

A common type of for loop is one where the value should go between two integers with a specific set size. To do this, we can use the `range` function. If given a single value, it will give a list ranging from 0 to the value minus 1:

```
In [2]: range(10)
Out[2]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

If given two values, these will be the starting value, and one plus the ending value:

```
In [3]: range(3, 12)
Out[3]: [3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Finally, if a third number is specified, this is taken to be the step size:

```
In [4]: range(2, 20, 2)
Out[4]: [2, 4, 6, 8, 10, 12, 14, 16, 18]
```

The `range` function can be used as the iterable in a `for` loop.

Exercise

Write a for loop that prints out the integers 1 to 9, but not 5 and 7.

```
In [1]: for x in range(10):
   ...:    if x != 5 and x != 7:
   ...:        print x
   ...:
0
1
2
3
4
6
8
9
```

`while` loops

Similarly to other programming languages, Python also provides a `while` loop which is similar to a `for` loop, but where the number of iterations is defined by a condition rather than an iterator:

```
while condition:
    # do something
```

For example, in the following example:

```
In [1]: a = 0

In [2]: while a < 10:
   ...:     print a
   ...:     a += 1
   ...:
0
1
2
3
4
5
6
7
8
9
```

the loop is executed until `a` is equal to or exceeds 10.

Exercise

Write a while loop to print out the Fibonacci numbers below 100.

```
In [1]: a = 0

In [2]: b = 1

In [3]: while a < 100:
   ...:     print a
   ...:     c = a + b
   ...:     a = b
   ...:     b = c
   ...:
0
1
1
2
3
5
8
13
21
34
55
89
```

2.2.4 Numpy basics

Numpy is a package that allows us to create multi-dimensional arrays, and a large fraction of Numpy is written in C, providing very fast array operations within Python.

Making arrays

Arrays can be created in different ways:

```
>>> import numpy as np

>>> a = np.array([10, 20, 30, 40])  # create an array from a list of values
>>> a
array([10, 20, 30, 40]

>>> b = np.arange(4)  # create an array of 4 integers, from 0 to 3
>>> b
array([0, 1, 2, 3]),

>>> np.arange(0.0, 10.0, 0.1)  # create a float array from 0 to 100 stepping by 0.1
array([ 0. ,  0.1,  0.2,  0.3,  0.4,  0.5,  0.6,  0.7,  0.8,  0.9,  1. ,
        1.1,  1.2,  1.3,  1.4,  1.5,  1.6,  1.7,  1.8,  1.9,  2. ,  2.1,
        2.2,  2.3,  2.4,  2.5,  2.6,  2.7,  2.8,  2.9,  3. ,  3.1,  3.2,
        3.3,  3.4,  3.5,  3.6,  3.7,  3.8,  3.9,  4. ,  4.1,  4.2,  4.3,
        4.4,  4.5,  4.6,  4.7,  4.8,  4.9,  5. ,  5.1,  5.2,  5.3,  5.4,
        5.5,  5.6,  5.7,  5.8,  5.9,  6. ,  6.1,  6.2,  6.3,  6.4,  6.5,
        6.6,  6.7,  6.8,  6.9,  7. ,  7.1,  7.2,  7.3,  7.4,  7.5,  7.6,
        7.7,  7.8,  7.9,  8. ,  8.1,  8.2,  8.3,  8.4,  8.5,  8.6,  8.7,
        8.8,  8.9,  9. ,  9.1,  9.2,  9.3,  9.4,  9.5,  9.6,  9.7,  9.8,
        9.9]),

>>> np.linspace(-np.pi, np.pi, 5)  # create an array of 5 evenly spaced samples from -pi to pi
array([-3.14159265, -1.57079633,  0.        ,  1.57079633,  3.14159265]))
```

New arrays can be obtained by operating with existing arrays:

```
>>> a + b**2  # elementwise operations
array([10, 21, 34, 49])
```

Arrays may have more than one dimension:

```
>>> f = np.ones([3, 4])  # 3 x 4 float array of ones
>>> f
array([[ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.],
       [ 1.,  1.,  1.,  1.]]),

>>> g = np.zeros([2, 3, 4], dtype=int)  # 3 x 4 x 5 int array of zeros
array([[[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]],
       [[0, 0, 0, 0],
        [0, 0, 0, 0],
        [0, 0, 0, 0]]]),

>>> i = np.zeros_like(f)  # array of zeros with same shape/type as f
array([[ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.],
       [ 0.,  0.,  0.,  0.]]))
```

You can change the dimensions of existing arrays:

```
>>> w = np.arange(12)
>>> w.shape = [3, 4]  # does not modify the total number of elements
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]]),

>>> x = np.arange(5)
>>> x
array([0, 1, 2, 3, 4]),

>>> y = x.reshape(5, 1)
>>> y = x.reshape(-1, 1)  # Same thing but Numpy figures out correct length
>>> y
array([[0],
       [1],
       [2],
       [3],
       [4]]))
```

It is possible to operate with arrays of different dimensions as long as they fit well (broadcasting):

```
>>> x + y * 10
array([[ 0,  1,  2,  3,  4],
       [10, 11, 12, 13, 14],
       [20, 21, 22, 23, 24],
       [30, 31, 32, 33, 34],
       [40, 41, 42, 43, 44]])
```

Array access and slicing

Numpy provides powerful methods for accessing array elements or particular subsets of an array, e.g. the 4th column or every other row. This is called slicing. The outputs below illustrate basic slicing, but you don't need to type these examples. The ">>>" indicates the input to Python:

```
>>> a = np.arange(20).reshape(4,5)
>>> a
array([[ 0,  1,  2,  3,  4],
      [ 5,  6,  7,  8,  9],
      [10, 11, 12, 13, 14],
      [15, 16, 17, 18, 19]])

>>> a[2, 3]  # select element in row 2, col 3 (counting from 0)
13

>>> a[2, :]  # select every element in row 2
array([10, 11, 12, 13, 14])

>>> a[:, 0]  # select every element in col 0
array([ 0,  5, 10, 15])

>>> a[0:3, 1:3]
array([[ 1,  2],
       [ 6,  7],
       [11, 12]])
```

Numpy arrays vs Python lists

The main differences between Numpy arrays and Python lists are the following:

- Python lists can contain anything, including other lists, objects, or complex data structures.
- When you slice a Python list it returns a copy, whereas slicing a Numpy array returns a reference
- Vector math does not work on lists. For example, multiplying a list by an int `n` gives `n` copies of the list, adding another list concatentates, and multiplying by a float gives an error.

2.2.5 About Scipy

The Scipy package is a package that provides common scientific tools for Scientists, and consists mainly of a collection of functions.

If you are interested in finding out about Scipy, click through to the main SciPy Reference Manual and skim the tutorial. Keep this repository of functionality in mind whenever you need some numerical functionality that isn't in Numpy: there is a good chance it is in Scipy:

- Basic functions in Numpy (and top-level scipy)
- Special functions (scipy.special)
- Integration (scipy.integrate)
- Optimization (optimize)
- Interpolation (scipy.interpolate)
- Fourier Transforms (scipy.fftpack)
- Signal Processing (signal)
- Linear Algebra
- Statistics
- Multi-dimensional image processing (ndimage)
- File IO (scipy.io)
- Weave

2.2.6 Matplotlib

Matplotlib is a python 2-d plotting library which produces publication quality figures in a variety of formats and interactive environments across platforms. Matplotlib can be used in Python scripts, the Python and IPython shell, web application servers, and six graphical user interface toolkits.

Documentation

The matplotlib documentation is extensive and covers all the functionality in detail. The documentation is littered with hundreds of examples showing a plot and the exact source code making the plot:

- Matplotlib home page: key pylab plotting commands in a table.
- Matplotlib manual
- Thumbnail gallery: hundreds of thumbnails linking to the source code used to make them (find a plot like the one you want to make).
- Code examples: extensive examples showing how to use matplotlib commands.

Hints on getting from here (an idea) to there (a plot)

- Start with Screenshots section of the manual for a broad idea of the plotting capabilities
- Most of the high-level plotting functions are in the `pyplot` module and you can find them quickly by searching for `pyplot.<function>`, e.g. `pyplot.errorbar`.

Pylab and Pyplot and NumPy

Let's demystify what's happening when you use `ipython --pylab` and clarify the relationship between **pylab** and **pyplot**.

matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB. This is just a package module that you can import:

```
import matplotlib.pyplot
print sorted(dir(matplotlib.pyplot))
```

Likewise pylab is also a module provided by matplotlib that you can import:

```
import pylab
```

This module is a thin wrapper around `matplotlib.pylab` which pulls in:

- Everything in `matplotlib.pyplot`
- All top-level functions `numpy`,``numpy.fft``, `numpy.random`,
- `numpy.linalg`
- A selection of other useful functions and modules from matplotlib

There is no magic, and to see for yourself do

```
import matplotlib.pylab
matplotlib.pylab??       # prints the source code!!
```

When you do `ipython --pylab` it (essentially) just does:

```
from pylab import *
```

In a lot of documentation examples you will see code like:

```
import matplotlib.pyplot as plt  # set plt as alias for matplotlib.pyplot
plt.plot([1,2], [3,4])
```

This is good practice for writing scripts and programs, since it is clear that the plot command is from the `matplotlib.pyplot` package. We'll adopt this alias for the examples in our course. But once you're familiar with the commands, you can omit the `plt.` when plotting in the interactive ipython shell to reduce the amount of typing needed.

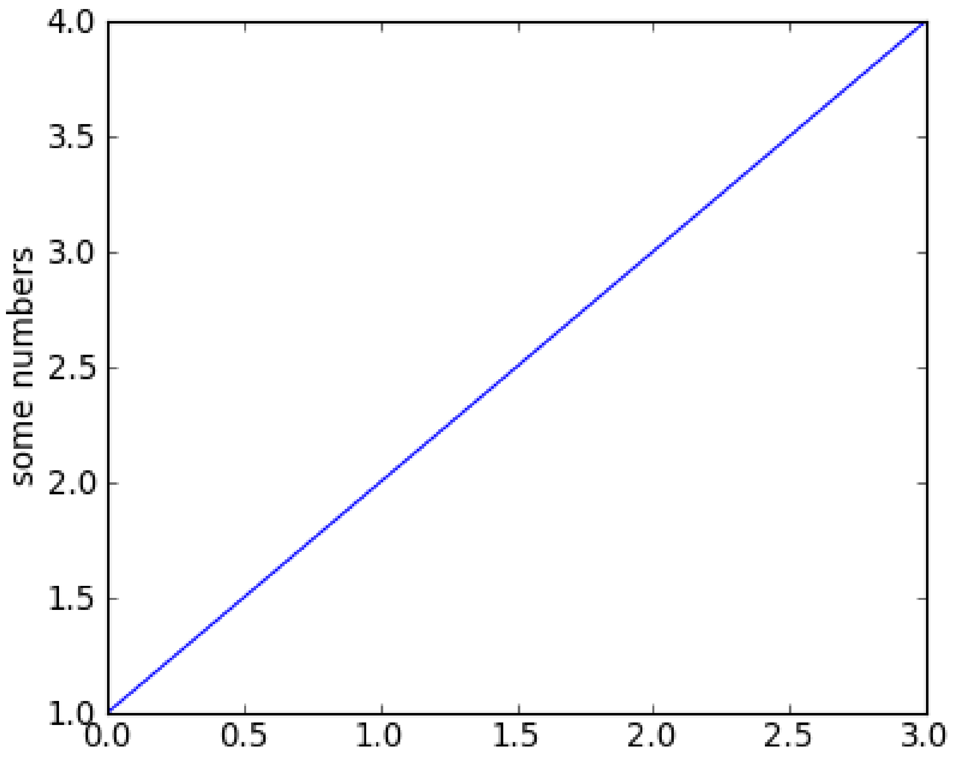
See Matplotlib, pylab, and pyplot: how are they related? for a more discussion on the topic.

Plotting 1-d data

The matplotlib tutorial on Pyplot (Copyright (c) 2002-2009 John D. Hunter; All Rights Reserved and license) is an excellent introduction to basic 1-d plotting. **The content below has been adapted from the pyplot tutorial source with some changes and the addition of exercises.**

matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB. Each `pyplot` function makes some change to a figure: eg, create a figure, create a plotting area in a figure, plot some lines in a plotting area, decorate the plot with labels, etc.... matplotlib.pyplot is stateful, in that it keeps track of the current figure and plotting area, and the plotting functions are directed to the current axes:

```
plt.figure()          # Make a new figure window
plt.plot([1,2,3,4])
plt.ylabel('some numbers')
```



You may be wondering why the x-axis ranges from 0-3 and the y-axis from 1-4. If you provide a single list or array to the plot() command, matplotlib assumes it is a sequence of y values, and automatically generates the x values for you. Since python ranges start with 0, the default x vector has the same length as y but starts with 0. Hence the x data are `[0, 1, 2, 3]`.

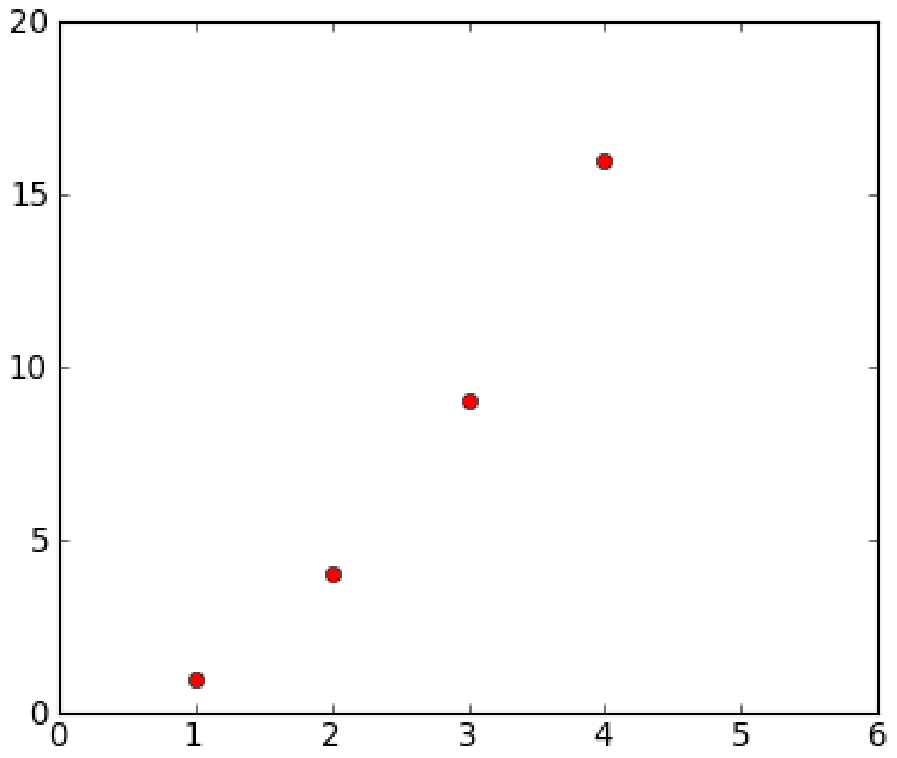
plot() is a versatile command, and will take an arbitrary number of arguments. For example, to plot x versus y, you can issue the command:

```
plt.clf()    # this clears the existing figure
plt.plot([1,2,3,4], [1,4,9,16])
```

plot() is just the tip of the iceberg for plotting commands and you should study the page of matplotlib screenshots to get a better picture.

For every x, y pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB, and you concatenate a color string with a line style string. The default format string is ‘b-‘, which is a solid blue line. For example, to plot the above with red circles, you would issue:

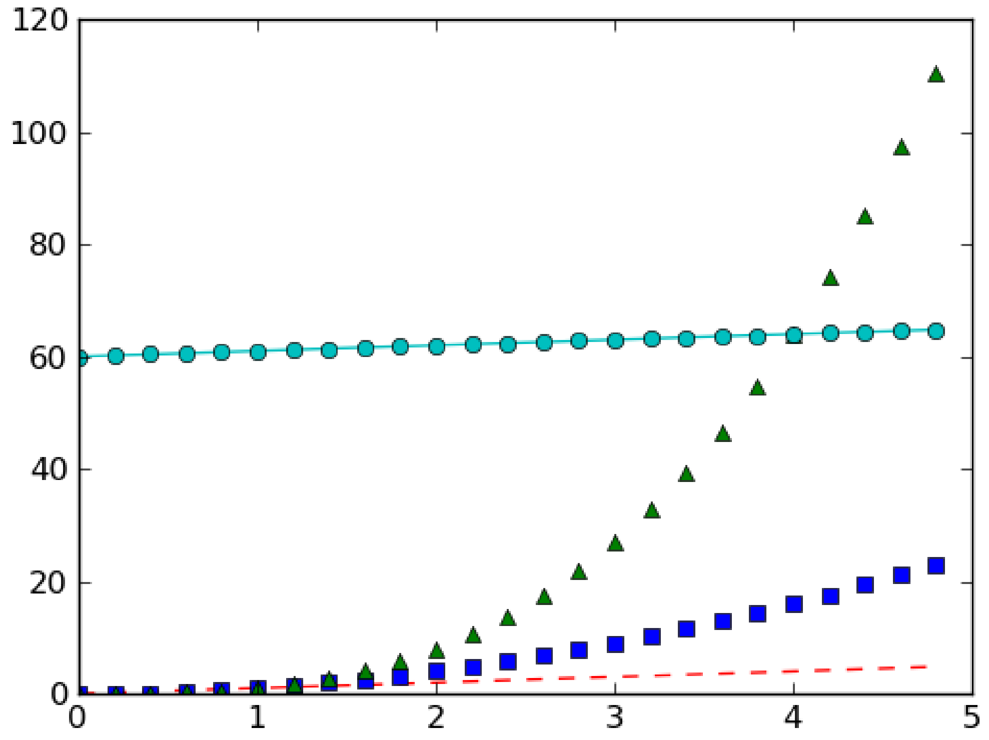
```
plt.clf()
plt.plot([1,2,3,4], [1,4,9,16], 'ro')
plt.axis([0, 6, 0, 20])
```



See the plot() documentation for a complete list of line styles and format strings. The axis() command in the example above takes a list of `[xmin, xmax, ymin, ymax]` and specifies the viewport of the axes.

If matplotlib were limited to working with lists, it would be of limited use for numeric processing. Generally, you will use NumPy arrays. In fact, all sequences are converted to numpy arrays internally. The example below illustrates plotting several lines with different format styles in one command using arrays:

```
# evenly sampled time at 200ms intervals
t = np.arange(0., 5., 0.2)

# red dashes, blue squares and green triangles
# then filled circle with connecting line
plt.clf()
plt.plot(t, t, 'r--')
plt.plot(t, t**2, 'bs')
plt.plot(t, t**3, 'g^')
plt.plot(t, t+60, 'co-')
```



Exercise: Plot a sine curve

Make a plot of a sin curve between 0 and 4*pi shown by a green curve. Remember that the value of *pi* and the sine function are in the numpy namespace (`np.pi` and `np.sin`).

```
plt.clf()
x = np.linspace(0, 4*np.pi, 100)
plt.plot(x, np.sin(x), 'g')
plt.xlim(0, 4*np.pi)
plt.ylim(-1.1, 1.1)
```

Controlling line and marker properties

What are lines and markers?

A matplotlib "line" is an object containing a set of points and various attributes describing how to draw those points. The points are optionally drawn with "markers" and the connections between points can be drawn with various styles of line (including no connecting line at all).

Lines have many attributes that you can set: linewidth, dash style, antialiased, etc; see Line2D. There are two commonly-used ways to set line properties:

- Use keyword args:

```
x = np.arange(0, 10, 0.25)
y = np.sin(x)
plt.clf()
plt.plot(x, y, linewidth=4.0)
```

- Use the setter methods of the `Line2D` instance. `plot` returns a list of lines; eg `line1, line2 = plot(x1,y1,x2,x2)`. Below I have only one line so it is a list of length 1. I use tuple unpacking in the `line, = plot(x, y, ’o’)` to get the first element of the list:

  ```
  plt.clf()
  line, = plt.plot(x, y, ’-’)
  line.set_<TAB>
  ```

  Now change the line color, noting that in this case you need to explicitly redraw:

  ```
  line.set_color(’m’) # change color
  plt.draw()
  ```

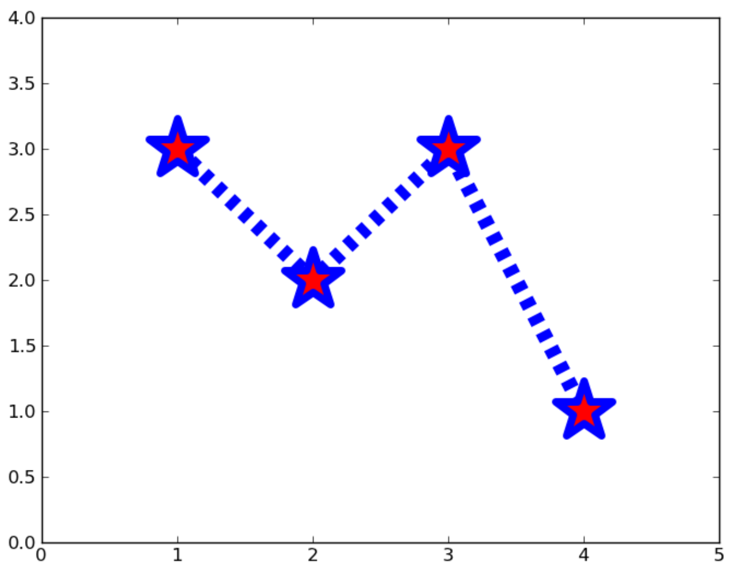
Wait, plot() makes a plot and returns a value?

Take note that plot() both generates the plot you asked for, and returns a list of line objects. This happens for most matplotlib plotting commands. They will generate the image, contours, histogram, or whatever you want, and also return an object that you can then use to later adjust the properties of the plotted object.

Here are some of the Line2D properties.

Property	Value Type
alpha	a float that controls the opacity
antialiased or aa	[True | False]
color or c	any matplotlib color
dash_capstyle	[’butt’ | ‘round’ | ‘projecting’]
dash_joinstyle	[’miter’ | ‘round’ | ‘bevel’]
dashes	sequence of on/off ink in points
data	(array xdata, array ydata)
figure	The figure in which the plot was made
label	a string used for the legend (see below)
linestyle or ls	[‘-‘ | ‘–‘ | ‘-.’ | ‘:’ | ‘steps’ | ...]
linewidth or lw	float value in points
lod	[True | False]
marker	[‘+’ | ‘,’ | ‘.’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ...]
markeredgecolor or mec	any matplotlib color
markeredgewidth or mew	float value in points
markerfacecolor or mfc	any matplotlib color
markersize or ms	float
markevery	None | integer | (startind, stride)
solid_capstyle	[’butt’ | ‘round’ | ‘projecting’]
solid_joinstyle	[’miter’ | ‘round’ | ‘bevel’]
visible	[True | False]
xdata	array
ydata	array
zorder	a number determining the plot order (controls overlaps)

Exercise: Make this plot



Make a plot that looks similar to the one above. You should be able to do this just by using the format string and keyword arguments to adjust the line and marker properties.

```
x = [1, 2, 3, 4]
y = [3, 2, 3, 1]
plt.clf()
plt.plot(x, y, '--', linewidth=10)
plt.plot(x, y, '*r', markeredgecolor='b', markeredgewidth=5, markersize=40)
plt.xlim(0, 5)
plt.ylim(0, 4)
```

2.3 Python in Astronomy

Before you proceed

Download this file and then expand it and go into this directory:

```
tar xvzf python4cardiff.tar.gz
cd python4cardiff
```

Then start up IPython with the `--pylab` option to enable easy plotting:

```
ipython --pylab
```

2.3.1 Astropy

Astropy is a community-developed core Python package for Astronomy (with the term used in the broad sense, from Solar System work to Cosmology). The first public release (v0.2) took place on February 20th 2013, bug-fixes are released a regular intervals.

Installation instructions for the most current version and the full documentation can be found on www.astropy.org.

After this workshop you will be familiar with some parts of Astropy, a second workshop will cover the remaining functionality.

Handling FITS files

Note: If you are already familiar with PyFITS, *astropy.io.fits* is in fact the same code as the latest version of PyFITS, and you can adapt old scripts that use PyFITS to use Astropy by simply doing:

```
from astropy.io import fits as pyfits
```

However, for new scripts, we recommend the following import:

```
from astropy.io import fits
```

Documentation

For more information about the features presented below, you can read the astropy.io.fits docs.

Data

The data used in this page (`gll_iem_v02_P6_V11_DIFFUSE.fit`) is an old version of the LAT Background Model (Pass 6 V11 Diffuse front+back) which was chosen so as not to have to download the larger more recent file.

Reading FITS files and accessing data

Opening a FITS file is relatively straightforward. We can open the LAT Background Model included in the tutorial files:

```
>>> from astropy.io import fits
>>> hdulist = fits.open('gll_iem_v02_P6_V11_DIFFUSE.fit')
```

The returned object, `hdulist`, behaves like a Python list, and each element maps to a Header-Data Unit (HDU) in the FITS file. You can view more information about the FITS file with:

```
>>> hdulist.info()
Filename: gll_iem_v02_P6_V11_DIFFUSE.fit
No.    Name         Type      Cards   Dimensions   Format
0    PRIMARY     PrimaryHDU      34   (720, 360, 30)   float32
1    ENERGIES    BinTableHDU     19   30R x 1C      [D]
```

As we can see, this file contains two HDUs. To access the primary HDU, which contains the main data, you can then do:

```
>>> hdu = hdulist[0]
```

The `hdu` object then has two important attributes: `data`, which behaves like a Numpy array, can be used to access the data, and `header`, which behaves like a dictionary, can be used to access the header information. First, we can take a look at the data:

```
>>> hdu.data.shape
(30, 360, 720)
```

This tells us that it is a 3-d cube. We can now take a peak at the header

```
>>> hdu.header
SIMPLE  =                    T / Written by IDL:  Thu Jan 20 07:19:05 2011
BITPIX  =                  -32 /
NAXIS   =                    3 / number of data axes
NAXIS1  =                  720 / length of data axis 1
NAXIS2  =                  360 / length of data axis 2
NAXIS3  =                   30 / length of data axis 3
EXTEND  =                    T / FITS dataset may contain extensions
COMMENT   FITS (Flexible Image Transport System) format is defined in 'Astronomy
COMMENT   and Astrophysics', volume 376, page 359; bibcode: 2001A&A...376..359H
FLUX    =        8.42259635886 /
CRVAL1  =                   0. / Value of longitude in pixel CRPIX1
CDELT1  =                  0.5 / Step size in longitude
CRPIX1  =                360.5 / Pixel that has value CRVAL1
CTYPE1  = 'GLON-CAR'           / The type of parameter 1 (Galactic longitude in
CUNIT1  = 'deg     '           / The unit of parameter 1
CRVAL2  =                   0. / Value of latitude in pixel CRPIX2
CDELT2  =                  0.5 / Step size in latitude
CRPIX2  =                180.5 / Pixel that has value CRVAL2
CTYPE2  = 'GLAT-CAR'           / The type of parameter 2 (Galactic latitude in C
CUNIT2  = 'deg     '           / The unit of parameter 2
CRVAL3  =                  50. / Energy of pixel CRPIX3
CDELT3  =    0.113828620540137 / log10 of step size in energy (if it is logarith
CRPIX3  =                   1. / Pixel that has value CRVAL3
CTYPE3  = 'photon energy'      / Axis 3 is the spectra
CUNIT3  = 'MeV     '           / The unit of axis 3
CHECKSUM= '3fdO3caL3caL3caL'   / HDU checksum updated 2009-07-07T22:31:18
DATASUM = '2184619035'         / data unit checksum updated 2009-07-07T22:31:18
DATE    = '2009-07-07'         /
FILENAME= '$TEMPDIR/diffuse/gll_iem_v02.fit' /File name with version number
TELESCOP= 'GLAST   '           /
INSTRUME= 'LAT     '           /
ORIGIN  = 'LISOC   '           /LAT team product delivered from the LISOC
OBSERVER= 'MICHELSON'          /Instrument PI
HISTORY Scaled version of gll_iem_v02.fit for use with P6_V11_DIFFUSE
```

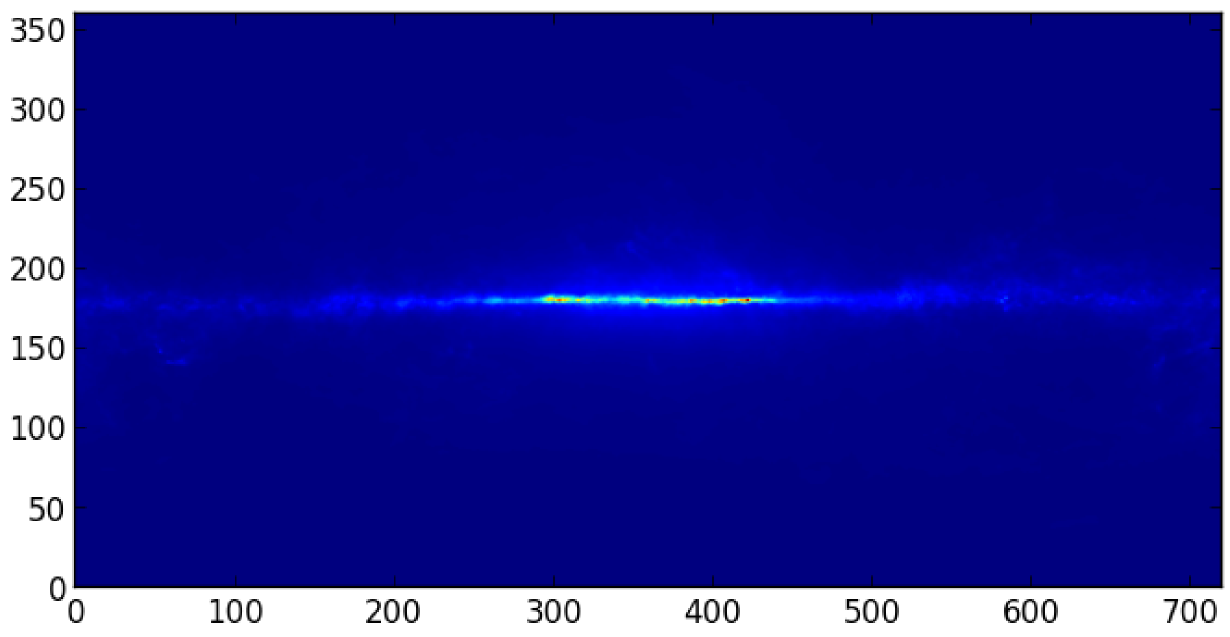
which shows that this is a Plate Carrée (-CAR) projection in Galactic Coordinates, and the third axis is photon energy. We can access individual header keywords using standard item notation:

```
>>> hdu.header['TELESCOP']
'GLAST'

>>> hdu.header['INSTRUME']
'LAT'
```

Provided that we started up `ipython` with the `--pylab` flag, we can plot one of the slices in photon energy:

```
>>> plt.imshow(hdu.data[0,:,:], origin='lower')
```

which gives:



Note that this is just a plot of an array, so the coordinates are just pixel coordinates at this stage. The data is stored with longitude increasing to the right (the opposite of the normal convention), but the Level 3 problem at the bottom of this page shows how to correctly flip the image.

Modifying data or header information in a FITS file object is easy. We can update existing header keywords:

```
>>> hdu.header['TELESCOP'] = "Fermi Gamma-ray Space Telescope"
```

or add new ones:

```
>>> hdu.header['MODIFIED'] = '26 Feb 2013'  # adds a new keyword
```

and we can also change the data, for example extracting only the first slice in photon energy:

```
>>> hdu.data = hdu.data[0,:,:]
```

Note that this does not change the original FITS file, simply the FITS file object in memory. Note that since the data is now 2-dimensional, we can remove the WCS keywords for the third dimension:

```
hdu.header.remove('CRPIX3')
hdu.header.remove('CRVAL3')
hdu.header.remove('CDELT3')
hdu.header.remove('CUNIT3')
hdu.header.remove('CTYPE3')
```

You can write the FITS file object to a file with:

```
>>> hdu.writeto('lat_background_model_slice.fits')
```

if you want to simply write out this HDU to a file, or:

```
>>> hdulist.writeto('lat_background_model_slice_allhdus.fits')
```

if you want to write out all of the original HDUs, including the modified one, to a file.

Creating a FITS file from scratch

If you want to create a FITS file from scratch, you need to start off by creating an HDU object:

```
>>> hdu = fits.PrimaryHDU()
```

and you can then populate the data and header attributes with whatever information you like:

```
>>> import numpy as np
>>> hdu.data = np.random.random((128,128))
```

Note that setting the data automatically populates the header with basic information:

```
>>> hdu.header
SIMPLE  =                    T / conforms to FITS standard
BITPIX  =                  -64 / array data type
NAXIS   =                    2 / number of array dimensions
NAXIS1  =                  128
NAXIS2  =                  128
EXTEND  =                    T
```

and you should never have to set header keywords such as `NAXIS`, `NAXIS1`, and so on manually. We can then set additional header keywords:

```
>>> hdu.header['telescop'] = 'Python Observatory'
```

and we can then write out the FITS file to disk:

```
>>> hdu.writeto('random_array.fits')
```

If the file already exists, you can overwrite it with:

```
>>> hdu.writeto('random_array.fits', clobber=True)
```

Convenience functions

In cases where you just want to access the data or header in a specific HDU, you can use the following convenience functions:

```
>>> data = fits.getdata('gll_iem_v02_P6_V11_DIFFUSE.fit')
>>> header = fits.getheader('gll_iem_v02_P6_V11_DIFFUSE.fit')
```

To get the data or header for an HDU other than the first, you can specify the extension name or index. The second HDU is called `energies`, so we can do:

```
>>> data = fits.getdata('gll_iem_v02_P6_V11_DIFFUSE.fit', extname='energies')
```

or:

```
>>> data = fits.getdata('gll_iem_v02_P6_V11_DIFFUSE.fit', ext=1)
```

and similarly for `getheader`.

Accessing Tabular Data

Tabular data behaves very similarly to image data such as that shown above, but the data array is a structured Numpy array which requires column access via the item notation:

```
>>> from astropy.io import fits
>>> hdulist = fits.open('gll_psc_v08.fit')

>>> hdulist[1].name
'LAT_Point_Source_Catalog'

>>> hdulist[1].data['RAJ2000']
array([  2.33711034e-01,   4.38849270e-01,   6.79812014e-01, ...,
         3.59759430e+02,   3.59859894e+02,   3.59906921e+02], dtype=float32)

>>> hdulist[1].data['DEJ2000']
array([ -7.81549788, -41.99647903,  62.33962631, ..., -30.62516785,
        67.86333466,  65.73053741], dtype=float32)
```

Practical Exercises

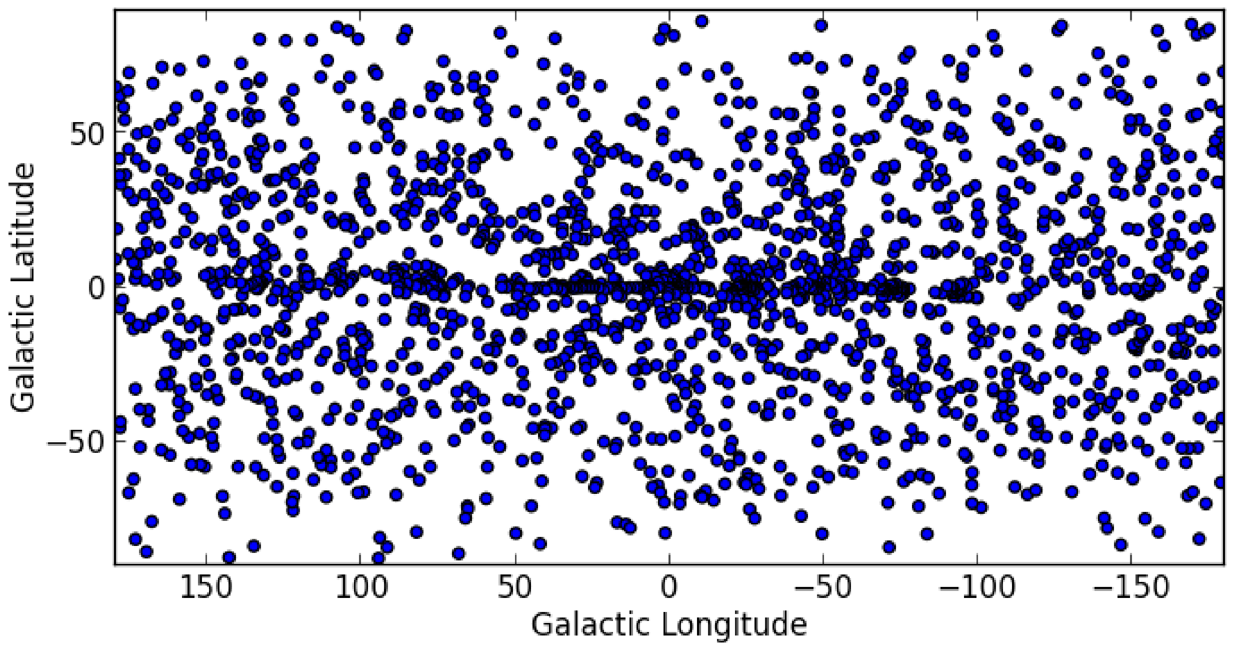
Level 1

Try and read in one of your own FITS files using `astropy.io.fits`, and see if you can also plot the array values in Matplotlib. Also, examine the header, and try and extract individual values. You can even try and modify the data/header and write the data back out - but take care not to write over the original file!

Level 2

Read in the LAT Point Source Catalog and make a scatter plot of the Galactic Coordinates of the sources (complete with axis labels). Bonus points if you can make the plot go between -180 and 180 instead of 0 and 360 degrees. Note that the Point Source Catalog contains the Galactic Coordinates, so no need to convert them.

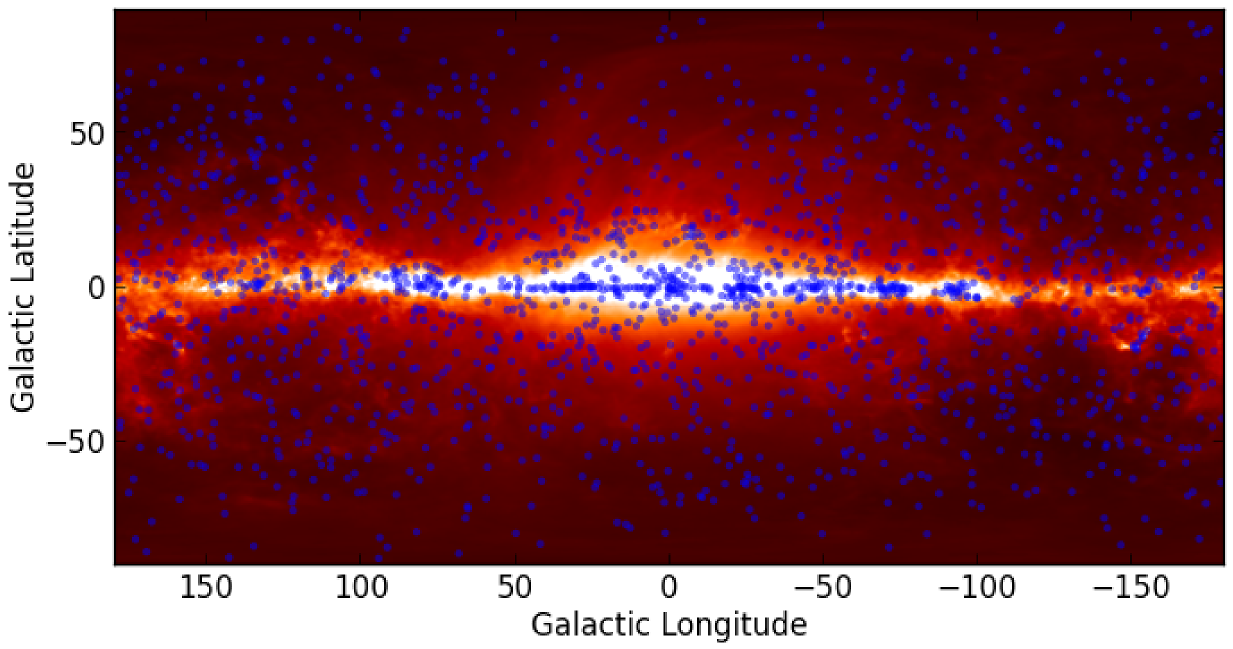
```
from astropy.io import fits
from matplotlib import pyplot as plt

# Read in Point Source Catalog
hdulist = fits.open('gll_psc_v08.fit')
psc = hdulist[1].data

# Extract Galactic Coordinates
l = hdulist[1].data['GLON']
b = hdulist[1].data['GLAT']

# Coordinates from 0 to 360, wrap to -180 to 180 to match image
l[l > 180.] -= 360.

# Plot the image
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1, aspect='equal')
ax.scatter(l, b)
ax.set_xlim(180., -180.)
ax.set_ylim(-90., 90.)
ax.set_xlabel('Galactic Longitude')
ax.set_ylabel('Galactic Latitude')
fig.savefig('fits_level2.png', bbox_inches='tight')
```



Level 3

Using Matplotlib, make an all-sky plot of the LAT Background Model in the Plate Carée projection showing the LAT Point Source Catalog overlaid with markers, and with the correct coordinates on the axes. You should do this using only `astropy.io.fits`, Numpy, and Matplotlib (no WCS or coordinate conversion library). Hint: the -CAR projection is such that the x pixel position is proportional to longitude, and the y pixel position to latitude. Bonus points for a pretty colormap.

```python
from astropy.io import fits
from matplotlib import pyplot as plt

# Read in Background Model
hdulist = fits.open('gll_iem_v02_P6_V11_DIFFUSE.fit')
bg = hdulist[0].data[0, :, :]

# Read in Point Source Catalog
hdulist = fits.open('gll_psc_v08.fit')
psc = hdulist[1].data

# Extract Galactic Coordinates
l = hdulist[1].data['GLON']
b = hdulist[1].data['GLAT']

# Coordinates from 0 to 360, wrap to -180 to 180 to match image
l[l > 180.] -= 360.

# Plot the image
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.imshow(bg ** 0.5, extent=[-180., 180., -90., 90.], cmap=plt.cm.gist_heat,
          origin='lower', vmin=0, vmax=2e-3)
ax.scatter(l, b, s=10, edgecolor='none', facecolor='blue', alpha=0.5)
ax.set_xlim(180., -180.)
ax.set_ylim(-90., 90.)
ax.set_xlabel('Galactic Longitude')
ax.set_ylabel('Galactic Latitude')
fig.savefig('fits_level3.png', bbox_inches='tight')
```



WCS Transformations

Note: If you are already familiar with PyWCS, *astropy.wcs* is in fact the same code as the latest version of PyWCS, and you can adapt old scripts that use PyWCS to use Astropy by simply doing:

```
from astropy import wcs as pywcs
```

However, for new scripts, we recommend the following import:

```
from astropy.wcs import WCS
```

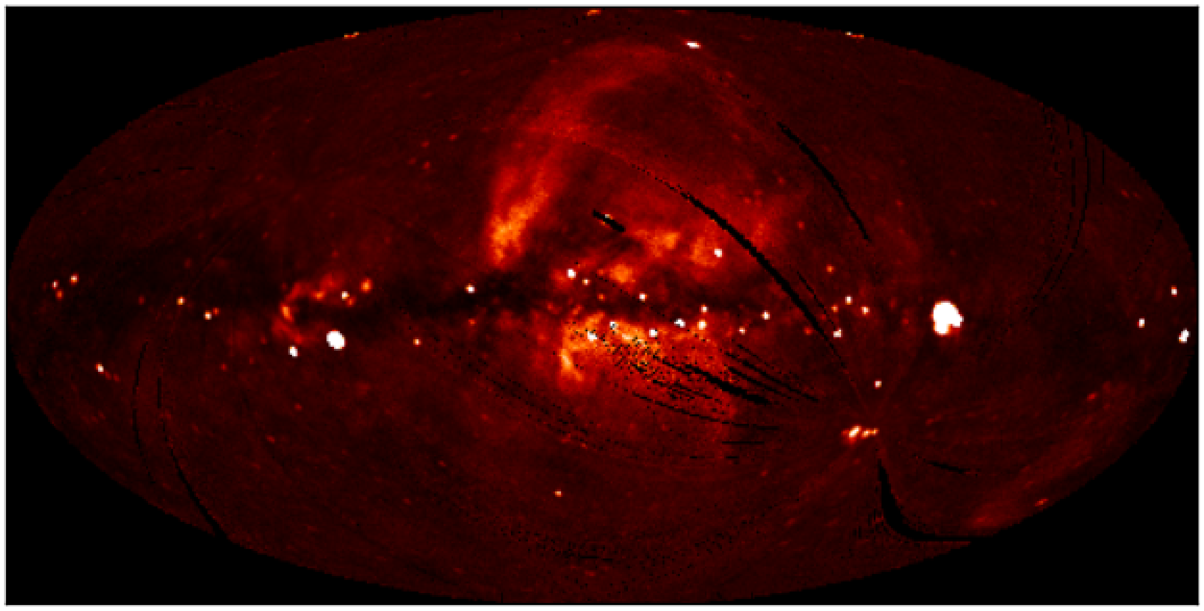
since most of the user-level functionality is contained within the *WCS* class.

Documentation

For more information about the features presented below, you can read the astropy.wcs docs.

Data

The data used in this page (`ROSAT.fits`) is a map of the Soft X-ray Diffuse Background from the ROSAT XRT/PSPC in the 3/4 keV band, in an Aitoff projection:



Representing WCS transformations

The World Coordinate System standard is often used in FITS files in order to describe the conversion from pixel to world (e.g. equatorial, galactic, etc.) coordinates. Given a FITS file with WCS information, such as `ROSAT.fits`, we can create an object to represent the WCS transformation either by directly supplying the filename:

```
>>> from astropy.wcs import WCS
>>> w = WCS('ROSAT.fits')
```

or the header of the FITS file:

```
>>> from astropy.io import fits
>>> from astropy.wcs import WCS
>>> header = fits.getheader('ROSAT.fits')
>>> w = WCS(header)
```

Pixel to World and World to Pixel transformations

Once the WCS object has been created, you can use the following methods to convert pixel to world coordinates:

```
>>> wx, wy = w.wcs_pix2world(250., 100., 1)
>>> print wx, wy
352.67460912268814 -15.413728717834152
```

This converts the pixel coordinates (250, 100) to the native world coordinate system of the transformation. Note the third argument, set to `1`, which indicates whether the pixel coordinates should be treated as starting from (1, 1) (as FITS files do) or from (0, 0). Converting from world to pixel coordinates is similar:

```
>>> px, py = w.wcs_world2pix(0., 0., 1)
>>> print px, py
240.5 120.5
```

Working with arrays

If many coordinates need to be transformed, then it is possible to use Numpy arrays:

```
>>> import numpy as np
>>> px = np.linspace(200., 300., 10)
>>> py = np.repeat(100., 10)
>>> wx, wy = w.wcs_pix2world(px, py, 1)
>>> print wx
[  31.31117136   22.6911179    14.09965438    5.52581152  356.9588445
  348.38809541  339.80285857  331.19224432  322.54503641  313.84953796]
>>> print wy
[-15.27956026 -15.34691039 -15.39269292 -15.4170814  -15.42016742
 -15.40196251 -15.36239844 -15.30132572 -15.21851046 -15.11362923]
```

Practical Exercises

Level 1

Try converting more values from pixel to world coordinates, and try converting these back to pixel coordinates. Do the results agree with the original pixel coordinates? Also, what are the world coordinates of the pixel at (1, 1), and why?

The final pixel coordinates should always agree with the starting ones, since each pixel covers a unique world coordinate position. The world coordinates of the pixel at (1, 1) are not defined:

```
w.wcs_pix2world(1, 1, 1)
[array(nan), array(nan)]
```

because the pixel lies outside the coordinate grid. Thus, not all pixels in an image have a valid position on the sky.

Level 2

Extract and print out the values in the ROSAT map at the position of the LAT Point Sources (from the FITS tutorial)

```
import numpy as np
from astropy.io import fits
from astropy.wcs import WCS

# Read in LAT Point Source Catalog
hdulist_cat = fits.open('gll_psc_v08.fit')
psc = hdulist_cat[1].data

# Extract Galactic Coordinates
l = hdulist_cat[1].data['GLON']
b = hdulist_cat[1].data['GLAT']

# Read in ROSAT map
hdulist_im = fits.open('ROSAT.fits')

# Extract image and header
image = hdulist_im[0].data
header = hdulist_im[0].header

# Instantiate WCS object
```

```
w = WCS(header)

# Find pixel positions of LAT sources. Note we use ''0'' here for the last
# argument, since we want zero based indices (for Numpy), not the FITS
# pixel positions.
px, py = w.wcs_world2pix(l, b, 0)

# Find the nearest integer pixel
px = np.round(px).astype(int)
py = np.round(py).astype(int)

# Find the ROSAT values (note the reversed index order)
values = image[py, px]

# Print out the values
print values
```

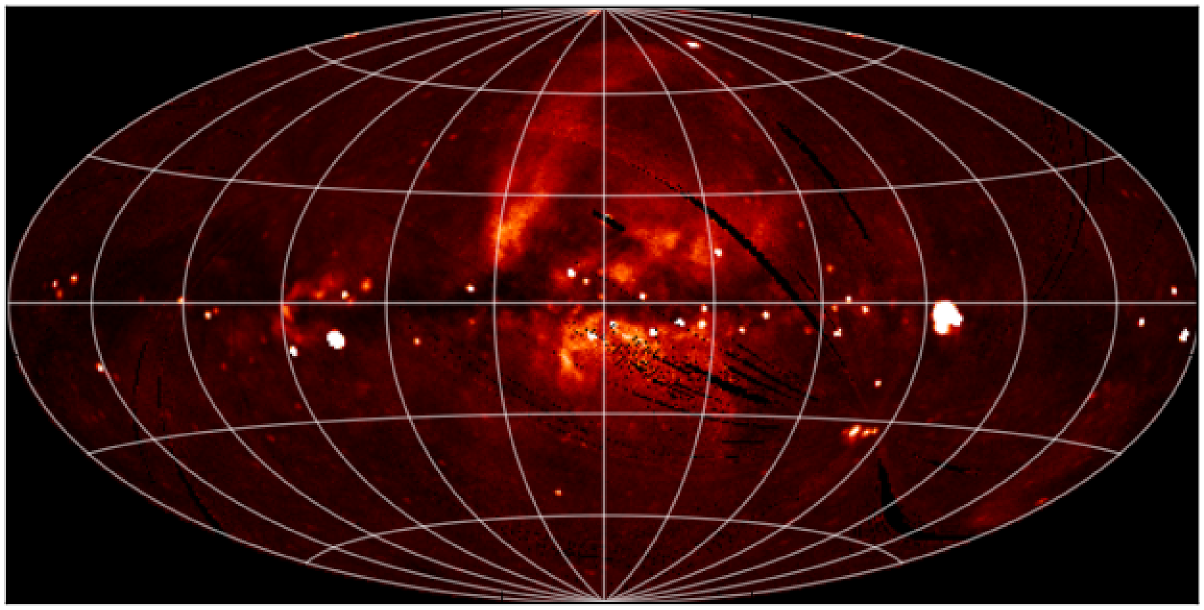
which gives:

```
[ 123.7635498   163.27642822  221.76609802 ...,  255.07995605  100.35219574
   87.62506104]
```

Level 3

Make a Matplotlib plot of the image showing gridlines for longitude and latitude overlaid (e.g. every 30 degrees).

```
import numpy as np
from matplotlib import pyplot as plt
from astropy.io import fits
from astropy.wcs import WCS

# Read in file
hdulist = fits.open('ROSAT.fits')

# Extract image and header
image = hdulist[0].data
header = hdulist[0].header

# Instantiate WCS object
w = WCS(header)

# Plot the image
fig = plt.figure()
ax = fig.add_subplot(1, 1, 1)
ax.imshow(image, cmap=plt.cm.gist_heat,
          origin='lower', vmin=0, vmax=1000.)

# Loop over lines of longitude
for lon in np.linspace(-180., 180., 13):
    grid_lon = np.repeat(lon, 100)
    grid_lat = np.linspace(-90., 90., 100)
    px, py = w.wcs_world2pix(grid_lon, grid_lat, 1)
    ax.plot(px, py, color='white', alpha=0.5)

# Loop over lines of latitude
for lat in np.linspace(-60., 60., 5):
    grid_lon = np.linspace(-180., 180., 100)
    grid_lat = np.repeat(lat, 100)
    px, py = w.wcs_world2pix(grid_lon, grid_lat, 1)
    ax.plot(px, py, color='white', alpha=0.5)

ax.set_xlim(0, image.shape[1])
ax.set_ylim(0, image.shape[0])
ax.set_xticklabels('')
ax.set_yticklabels('')
fig.savefig('wcs_extra.png', bbox_inches='tight')
```



Tabular data

Astropy includes a class for representing arbitrary tabular data in `astropy.table`, called `Table`. This class can be imported with:

```
from astropy.table import Table
```

You may need to also import the `Column` class, depending on how you are definining your table (see below):

```
from astropy.table import Table, Column
```

Documentation

For more information about the features presented below, you can read the astropy.table docs.

Constructing and Manipulating tables

There are a number of ways of constructing tables. One simple way is to start from existing lists or arrays:

```
>>> from astropy.table import Table
>>> a = [1, 4, 5]
>>> b = [2.0, 5.0, 8.2]
>>> c = ['x', 'y', 'z']
>>> t = Table([a, b, c], names=('a', 'b', 'c'))
```

There are a few ways to examine the table. You can get detailed information about the table values and column definitions as follows:

```
>>> t
<Table rows=3 names=('a','b','c')>
array([(1, 2.0, 'x'), (4, 5.0, 'y'), (5, 8.2, 'z')],
      dtype=[('a', '<i8'), ('b', '<f8'), ('c', '|S1')])
```

If you print the table (either from the noteboook or in a text console session) then a formatted version appears:

```
>>> print t
  a   b   c
--- --- ---
  1 2.0   x
  4 5.0   y
  5 8.2   z
```

Now examine some high-level information about the table:

```
>>> t.colnames
['a', 'b', 'c']

>>> len(t)
3
```

Access the data by column or row using the same syntax as for Numpy structured arrays:

```
>>> t['a']       # Column 'a'
<Column name='a' units=None format=None description=None>
array([1, 4, 5])

>>> t['a'][1]    # Row 1 of column 'a'
4

>>> t[1]         # Row obj for with row 1 values
<Row 1 of table
  values=(4, 5.0, 'y')
  dtype=[('a', '<i8'), ('b', '<f8'), ('c', '|S1')]>

>>> t[1]['a']    # Column 'a' of row 1
4
```

One can retrieve a subset of a table by rows (using a slice) or columns (using column names), where the subset is returned as a new table:

```
>>> print t[0:2]      # Table object with rows 0 and 1
 a   b   c
--- --- ---
  1 2.0   x
  4 5.0   y

>>> t['a', 'c']  # Table with cols 'a', 'c'
 a   c
--- ---
  1   x
  4   y
  5   z
```

Modifying table values in place is flexible and works as one would expect:

```
>>> t['a'] = [-1, -2, -3]           # Set all column values
>>> t['a'][2] = 30                  # Set row 2 of column 'a'
>>> t[1] = (8, 9.0, "W")            # Set all row values
>>> t[1]['b'] = -9                  # Set column 'b' of row 1
>>> t[0:2]['b'] = 100.0             # Set column 'c' of rows 0 and 1
>>> print t
 a    b    c
--- ----- ---
 -1 100.0   x
  8 100.0   W
 30   8.2   z
```

Add, remove, and rename columns with the following:

```
>>> t.add_column(Column(data=[1, 2, 3], name='d')))
>>> t.remove_column('c')
>>> t.rename_column('a', 'A')
>>> t.colnames
['A', 'b', 'd']
```

Adding a new row of data to the table is as follows:

```
>>> t.add_row([-8, -9, 10])
>>> len(t)
4
```

Lastly, one can create a table with support for missing values, for example by setting `masked=True`:

```
>>> t = Table([a, b, c], names=('a', 'b', 'c'), masked=True)
>>> t['a'].mask = [True, True, False]
>>> t
<Table rows=3 names=('a','b','c')>
masked_array(data = [(--, 2.0, 'x') (--, 5.0, 'y') (5, 8.2, 'z')],
             mask = [(True, False, False) (True, False, False) (False, False, False)],
       fill_value = (999999, 1e+20, 'N'),
            dtype = [('a', '<i8'), ('b', '<f8'), ('c', '|S1')])

>>> print t
 a   b   c
--- --- ---
 -- 2.0   x
 -- 5.0   y
  5 8.2   z
```

Finally, every table can have meta-data attached to it via the `meta` attribute, which can be used like a Python dictionary:

```
>>> t.meta['creator'] = 'me'
```

Reading and writing tables

`Table` objects include `read` and `write` methods that can be used to easily read and write the tables to different formats. The tutorial directory contains a file named rosat.vot which is the ROSAT All-Sky Bright Source Catalogue (1RXS) (Voges+ 1999) in the VO Table format.

You can read this in as a `Table` object by simply doing:

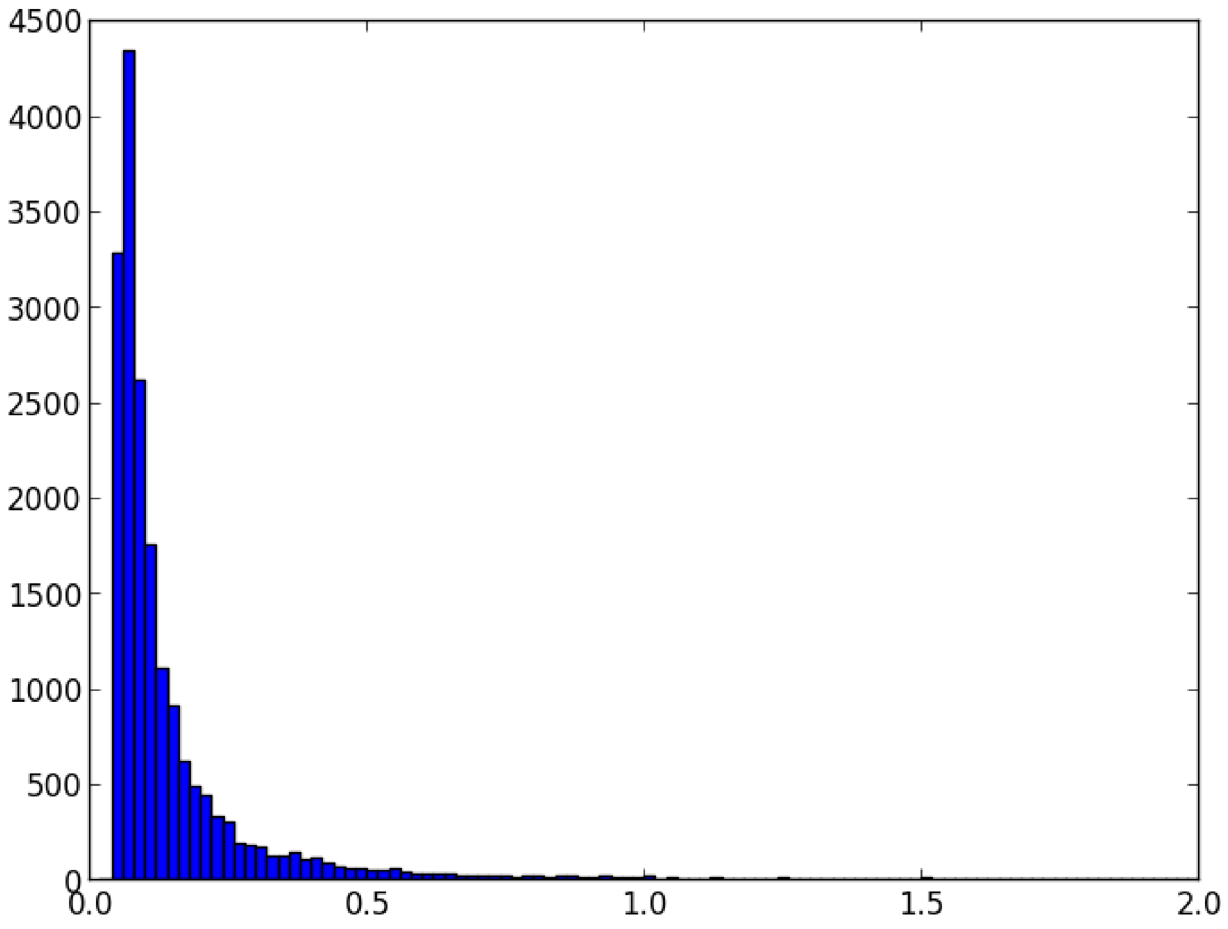
```
>>> t = Table.read('rosat.vot', format='votable')
```

(just ignore the warnings, which are due to Vizier not complying with the VO standard). We can see a quick overview of the table with:

```
>>> print t
     _1RXS         RAJ2000   DEJ2000  PosErr NewFlag   Count    e_Count   HR1  e_HR1  HR2  e_HR2 Exter
---------------- --------- --------- ------ ------- --------- --------- ----- ----- ----- ----- -----
J000000.0-392902   0.00000 -39.48403     19    __..      0.13     0.035  0.69  0.25  0.28  0.24
J000007.0+081653   0.02917   8.28153     10    TT..      0.19     0.021  0.89  0.10  0.24  0.13
J000010.0-633543   0.04167 -63.59528     11    __..      0.19     0.031 -0.36  0.13 -0.35  0.23     1
J000011.9+052318   0.04958   5.38833      7    __..      0.26     0.026  0.24  0.10  0.00  0.13
J000012.6+014621   0.05250   1.77250     11    __..     0.081     0.016  0.05  0.20  0.00  0.26     1
J000013.5+575628   0.05625  57.94125      8    __..      0.12     0.017  0.57  0.12  0.32  0.14
J000019.5-261032   0.08125 -26.17556     12    __..      0.12     0.022 -0.26  0.17  0.19  0.29
             ...       ...       ...    ...     ...       ...       ...   ...   ...   ...   ...    ..
J235929.2-255851 359.87164 -25.98083     10    _T..      0.23     0.028 -0.43  0.11 -0.30  0.26     1
J235929.3+334329 359.87207  33.72472     11    __..      0.16     0.024 -0.62  0.12 -0.56  0.66     1
J235930.9-401541 359.87875 -40.26139     18    __..      0.13     0.037 -0.73  0.18  0.02  0.82
J235940.9-314342 359.92041 -31.72847     19    __..     0.058     0.017  0.17  0.30  0.33  0.34
J235941.2+830719 359.92166  83.12195     10    __..     0.066     0.011  0.72  0.14  0.19  0.17
J235944.7+220014 359.93625  22.00389     17    __..     0.052     0.015 -0.01  0.27  0.37  0.35
J235959.1+083355 359.99625   8.56528     10    __..      0.12     0.018  0.54  0.13  0.10  0.17
```

Since we are using IPython with the `--pylab` option, we can easily make a histogram of the count rates:

```
>>> plt.hist(t['Count'], range=[0., 2], bins=100)
```



It is easy to select a subset of the table matching a given criterion:

```
>>> t_bright = t[t['Count'] > 0.2]
>>> len(t_bright)
3627
```

Criteria can be combined:

```
>>> t_sub = t[(t['RAJ2000'] > 230.) & (t['RAJ2000'] < 260.) &
              (t['DEJ2000'] > -60.) & (t['DEJ2000'] < -20)]

>>> len(t_sub)
642
```

Note about FITS tables

In Astropy 0.2, FITS tables cannot be read/written directly from the `Table` class. To create a `Table` object from a FITS table, you can use `astropy.io.fits` to read in the table to a Numpy array, then initialize the table with it:

```
>>> from astropy.io import fits
>>> from astropy.table import Table
>>> data = fits.getdata('catalog.fits', 1)
>>> t = Table(data)
```

and to write out, you can use `astropy.io.fits`, converting the table to a Numpy array:

```
>>> fits.writeto('new_catalog.fits', np.array(t))
```

The main drawback of the current approach is that table metadata like UCDs and other FITS header keywords are lost. Future versions of Astropy will support reading/writing FITS tables directly from the `Table` class.

Practical Exercises

Level 1

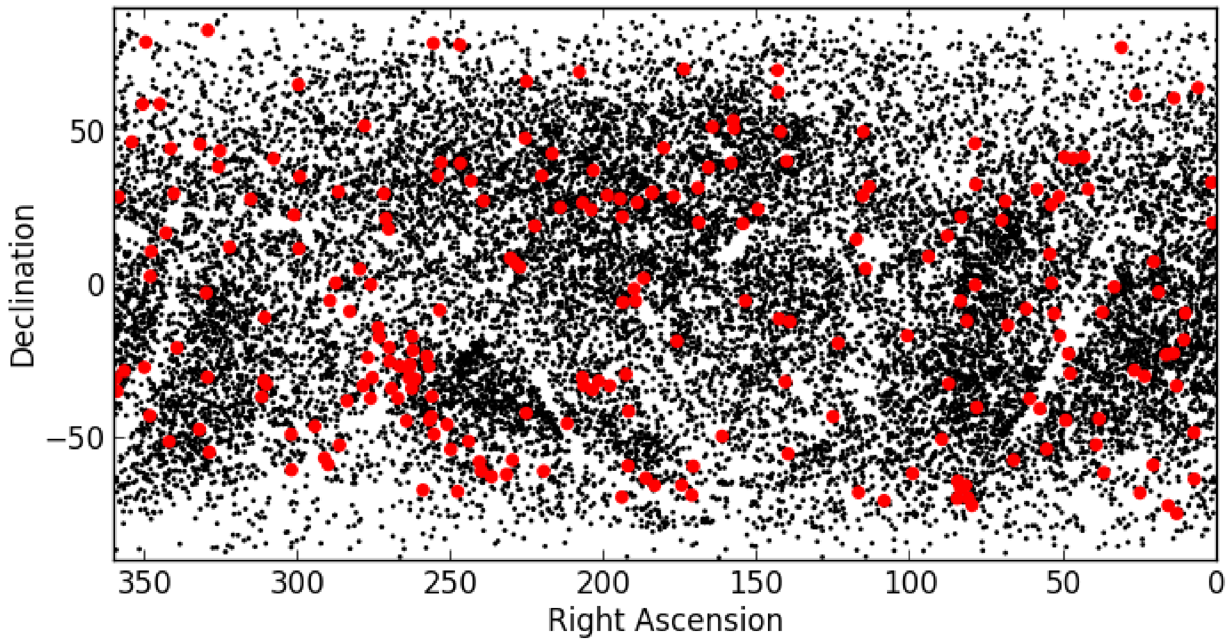
Try and find a way to make a table of the ROSAT point source catalog that contains only the RA, Dec, and count rate. Hint: you can see what methods are available on an object by typing e.g. `t.` and then pressing `<TAB>`. You can also find help on a method by typing e.g. `t.add_column?`.

```
>>> t.keep_columns(['RAJ2000', 'DEJ2000', 'Count'])
>>> print t
 RAJ2000   DEJ2000    Count
--------- --------- ---------
  0.00000 -39.48403      0.13
  0.02917   8.28153      0.19
  0.04167 -63.59528      0.19
  0.04958   5.38833      0.26
  0.05250   1.77250     0.081
  0.05625  57.94125      0.12
  0.08125 -26.17556      0.12
      ...       ...       ...
359.87207  33.72472      0.16
359.87875 -40.26139      0.13
359.92041 -31.72847     0.058
359.92166  83.12195     0.066
359.93625  22.00389     0.052
359.99625   8.56528      0.12

Note that you can also do this with::

>>> t_new = t['RAJ2000', 'DEJ2000', 'Count']
>>> print t_new
 RAJ2000   DEJ2000    Count
--------- --------- ---------
  0.00000 -39.48403      0.13
  0.02917   8.28153      0.19
  0.04167 -63.59528      0.19
  0.04958   5.38833      0.26
  0.05250   1.77250     0.081
  0.05625  57.94125      0.12
  0.08125 -26.17556      0.12
      ...       ...       ...
359.87207  33.72472      0.16
359.87875 -40.26139      0.13
359.92041 -31.72847     0.058
359.92166  83.12195     0.066
359.93625  22.00389     0.052
359.99625   8.56528      0.12
```

Level 2

Make an all-sky equatorial plot of the ROSAT sources, with all sources shown in black, and only the sources with a count rate larger than 2. shown in red.

```
from astropy.table import Table
from matplotlib import pyplot as plt

t = Table.read('rosat.vot', format='votable')
t_bright = t[t['Count'] > 2.]

fig = plt.figure()
ax = fig.add_subplot(1,1,1, aspect='equal')
ax.scatter(t['RAJ2000'], t['DEJ2000'], s=1, color='black')
ax.scatter(t_bright['RAJ2000'], t_bright['DEJ2000'], color='red')
ax.set_xlim(360., 0.)
ax.set_ylim(-90., 90.)
ax.set_xlabel("Right Ascension")
ax.set_ylabel("Declination")

fig.savefig('tables_level2.png', bbox_inches='tight')
```



Level 3

Try and write out the ROSAT catalog into a format that you can read into another software package (see here for more details). For example, try and write out the catalog into CSV format, then read it into a spreadsheet software package (e.g. Excel, Google Docs, Numbers, OpenOffice).

To write out the file:

```
>>> t.write('rosat2.csv', format='ascii', delimiter=',')
```

Then you should be able to load it into another software package.

Unit manipulation

Documentation

For more information about the features presented below, you can read the astropy.units docs.

Representing units and quantities

Astropy includes a powerful framework for units that allows users to attach units to scalars and arrays, and manipulate/combine these, keeping track of the units.

Since we may want to use a number of units in expressions, it is easiest and most concise to import the units module with:

```
from astropy import units as u
```

though note that this will conflict with any variable called `u`.

Units can then be accessed with:

```
>>> u.m
Unit("m")

>>> u.pc
Unit("pc")

>>> u.s
Unit("s")

>>> u.kg
Unit("kg")
```

We can create composite units:

```
>>> u.m / u.kg / u.s**2
Unit("m / (kg s2)")
```

The most useful feature about the units is the ability to attach them to scalars or arrays, creating `Quantity` objects:

```
>>> 3. * u.m
<Quantity 3.0 m>

>>> import numpy as np

>>> np.array([1.2, 2.2, 1.7]) * u.pc / u.year
<Quantity [ 1.2  2.2  1.7] pc / (yr)>
```

Combining and converting units

Quantities can then be combined:

```
>>> q1 = 3. * u.m

>>> q2 = 5. * u.cm / u.s / u.g**2

>>> q1 * q2
<Quantity 15.0 cm m / (g2 s)>
```

and converted to different units:

```
>>> (q1 * q2).to(u.m**2 / u.kg**2 / u.s)
<Quantity 150000.0 m2 / (kg2 s)>
```

The units and value of a quantity can be accessed separately via the `value` and `unit` attributes:

```
>>> q = 5. * u.pc

>>> q.value
5.0

>>> q.unit
Unit("pc")
```

Advanced features

The units of a quantity can be decomposed into a set of base units using the `decompose()` method. By default, units will be decomposed to S.I.:

```
>>> (3. * u.cm * u.pc / u.g / u.year**2).decompose()
<Quantity 929.5706802193289 m2 / (kg s2)>
```

To decompose into c.g.s. units, one can do:

```
>>> (3. * u.cm * u.pc / u.g / u.year**2).decompose(u.cgs.bases)
<Quantity 9295.70680219329 cm2 / (g s2)>
```

Using physical constants

The astropy.constants module contains physical constants relevant for Astronomy, and these are defined with units attached to them using the `astropy.units` framework. If we want to compute the Gravitational force felt by a 100. * u.kg space probe by the Sun, at a distance of 3.2au, we can do:

```
>>> from astropy.constants import G

>>> F = (G * 1. * u.M_sun * 100. * u.kg) / (3.2 * u.au)**2

>>> F
<Quantity 6.517421874999999e-10 m3 solMass / (AU2 s2)>

>>> F.to(u.N)
<Quantity 0.05792707869188191 N>
```

The full list of available physical constants is shown here (and additions are welcome!).

Practical Exercises

Level 1

What is 1 barn megaparsecs in teaspoons?

```
>>> (1. * u.barn * u.Mpc).to(u.tsp)
<Quantity 0.626035029893 tsp>
```

Level 2

What is 3 nm^2 Mpc / m^3 in dimensionless units?

```
>>> (3. * u.nm**2 * u.Mpc / u.m**3).decompose()
<Quantity 92570.327444 >
```

or to just get the numerical value:

```
>>> (3. * u.nm**2 * u.Mpc / u.m**3).decompose().value
92570.327444015755
```

Level 3

Try and convert 250 microns to Ghz using the units framework. You will need to look through the documentation for astropy.units to see how this can be made to work.

```
>>> (250 * u.micron).to(u.GHz)
...
UnitsException: 'micron' (length) and 'GHz' (frequency) are not convertible

>>> (250. * u.micron).to(u.GHz, equivalencies=u.spectral())
Out[5]: <Quantity 1199.169832 GHz>
```

Celestial Coordinates

Documentation

For more information about the features presented below, you can read the astropy.coordinates docs.

Representing and converting coordinates

Astropy includes a framework to represent celestial coordinates and transform between them. Astropy 0.2 only includes a few common coordinate systems (ICRS, FK4, FK5, and Galactic), but future versions will include more built-in coordinate systems, and users can already define their own systems. In addition, while the current version only supports transformation of individual scalar coordinates, arrays will be supported in future.

Coordinate objects are instantiated with a flexible and natural approach that supports both numeric angle values and (limited) string parsing:

```
>>> from astropy import coordinates as coord
>>> from astropy import units as u
>>> coord.ICRSCoordinates(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg>
>>> coord.ICRSCoordinates('00h42m44.3s +41d16m9s')
<ICRSCoordinates RA=10.68458 deg, Dec=41.26917 deg>
```

The individual components of a coordinate are `Angle` objects, and their values are accessed using special attributes:

```
>>> c = coord.ICRSCoordinates(ra=10.68458, dec=41.26917, unit=(u.degree, u.degree))
>>> c.ra
<RA 10.68458 deg>
>>> c.ra.hours
```

```
0.7123053333333333
>>> c.ra.hms
(0.0, 42, 44.2992000000001)
>>> c.dec
<Dec 41.26917 deg>
>>> c.dec.radians
0.7202828960652683
```

To convert to some other coordinate system, the easiest method is to use attribute-style access with short names for the built-in systems:

```
>>> c.galactic
<GalacticCoordinates l=121.17422 deg, b=-21.57283 deg>
```

but explicit transformations via the `transform_to` method are also available:

```
>>> c.transform_to(coord.GalacticCoordinates)
<GalacticCoordinates l=121.17422 deg, b=-21.57283 deg>
```

The `astropy.coordinates` subpackage also provides a quick way to get coordinates for named objects (with an internet connection). All coordinate classes have a special class method, *from_name()*, that accepts a string and queries Sesame to retrieve coordinates for that object:

```
>>> c_eq = coord.ICRSCoordinates.from_name("M16")
>>> c_eq
<ICRSCoordinates RA=274.70000 deg, Dec=-13.80670 deg>
```

This works for any coordinate class:

```
>>> c_gal = coord.GalacticCoordinates.from_name("M16")
>>> c_gal
<GalacticCoordinates l=16.95408 deg, b=0.79335 deg>
```

Note: This is intended to be a convenience, and is very simple. If you need precise coordinates for an object you should find the appropriate reference for that measurement and input the coordinates manually.

Practical Exercises

Level 1

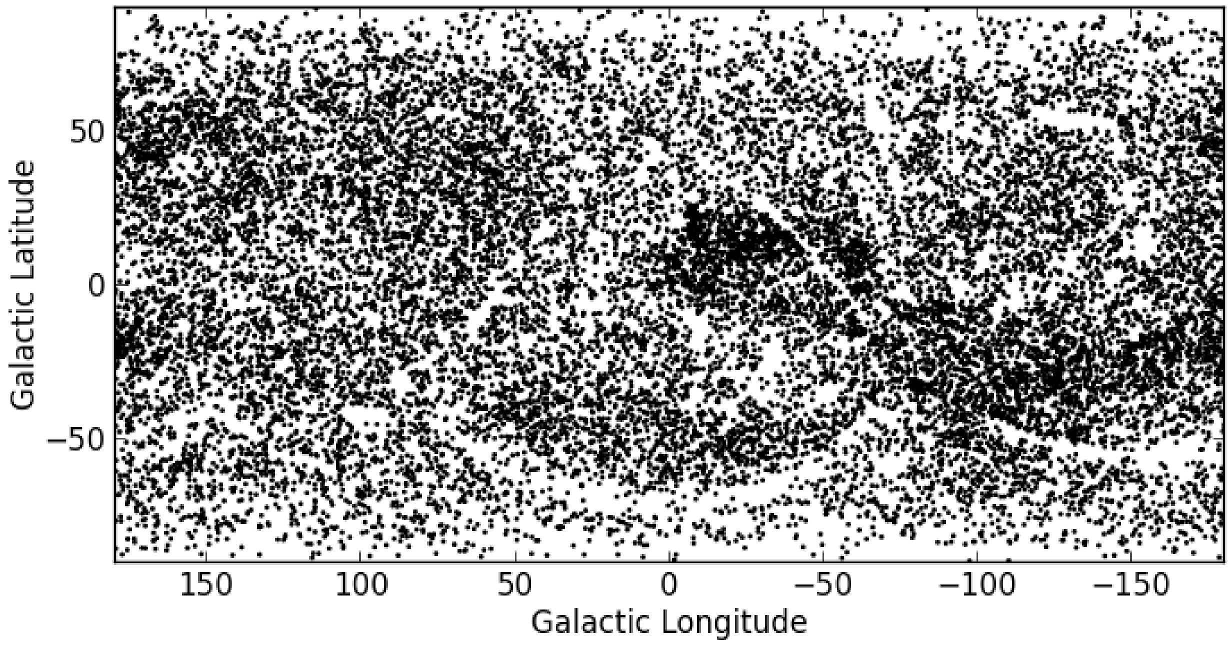
Find the coordinates of the Crab Nebula in ICRS coordinates, and convert them to Galactic Coordinates

```
>>> from astropy import coordinates as coord
>>> crab = coord.ICRSCoordinates.from_name('M1')
>>> print crab
<ICRSCoordinates RA=83.63308 deg, Dec=22.01450 deg>
>>> crab_gal = crab.transform_to(coord.GalacticCoordinates)
>>> print crab_gal
<GalacticCoordinates l=-175.44248 deg, b=-5.78434 deg>
```

Level 2

Using the ROSAT Point source catalog (from *Tabular data*), convert all the equatorial coordinates to Galactic coordinates, and make a new plot (but don't worry about the bright sources).

```
from astropy import units as u
from astropy import coordinates as coord
from astropy.table import Table
from matplotlib import pyplot as plt

t = Table.read('rosat.vot', format='votable')

l = []
b = []
for row in t:
    eq = coord.FK5Coordinates(row['RAJ2000'], row['DEJ2000'], unit=(u.degree, u.degree))
    gal = eq.transform_to(coord.GalacticCoordinates)
    l.append(gal.l.degrees)
    b.append(gal.b.degrees)

fig = plt.figure()
ax = fig.add_subplot(1,1,1, aspect='equal')
ax.scatter(l, b, s=1, color='black')
ax.set_xlim(180., -180.)
ax.set_ylim(-90., 90.)
ax.set_xlabel("Galactic Longitude")
ax.set_ylabel("Galactic Latitude")

fig.savefig('coord_level2.png', bbox_inches='tight')
```



Level 3

Try and make a plot similar to this!

2.3.2 APLpy

APLpy (the Astronomical Plotting Library in Python) is a Python module aimed at producing publication-quality plots of astronomical imaging data in FITS format. It effectively provides a layer on top of Matplotlib to enable plotting of

Astronomical images, and allows users to:

- Make plots interactively or using scripts
- Show grayscale, colorscale, and 3-color RGB images of FITS files
- Generate co-aligned FITS cubes to make 3-color RGB images
- Overlay any number of contour sets
- Overlay markers with fully customizable symbols
- Plot customizable shapes like circles, ellipses, and rectangles
- Overlay ds9 region files
- Overlay coordinate grids
- Show colorbars, scalebars, and beams
- Easily customize the appearance of labels and ticks
- Hide, show, and remove different contour and marker layers
- Pan, zoom, and save any view as a full publication-quality plot
- Save plots as EPS, PDF, PS, PNG, and SVG

Documentation

The APLpy Documentation contains all the information needed to run APLpy successfully. The most important page is the Quick Reference Guide which provides concise instructions for all of the APLpy functions. In particular, note that the interface to APLpy is slightly different to Matplotlib, so not all Matplotlib commands will work in APLpy. Be sure to check out the quick reference guide!

Getting started

Import the `aplpy` module (note the lowercase module name):

```
import aplpy
```

And create a new figure to plot the FITS file with:

```
f = aplpy.FITSFigure('ngc2264_cutout.fits')
```

This should open up a matplotlib window and will show an empty set of axes with coordinates. From now on, you will interact with the figure by calling methods associated with `f`. For example, show the image as a grayscale:

```
f.show_grayscale()
```

The automatic settings for the stretch should be decent, but there are of course options to allow custom min/max levels. You can now try panning around and zooming in like you would do in Matplotlib, and you will notice the coordinates updating. Press the Home button to reset the view.

Next, let's overlay a set of contours from a different image:

```
f.show_contour('ngc2264_cutout_mips.fits', levels=10)
```

We can also add a coordinate grid:

```
f.add_grid()
```

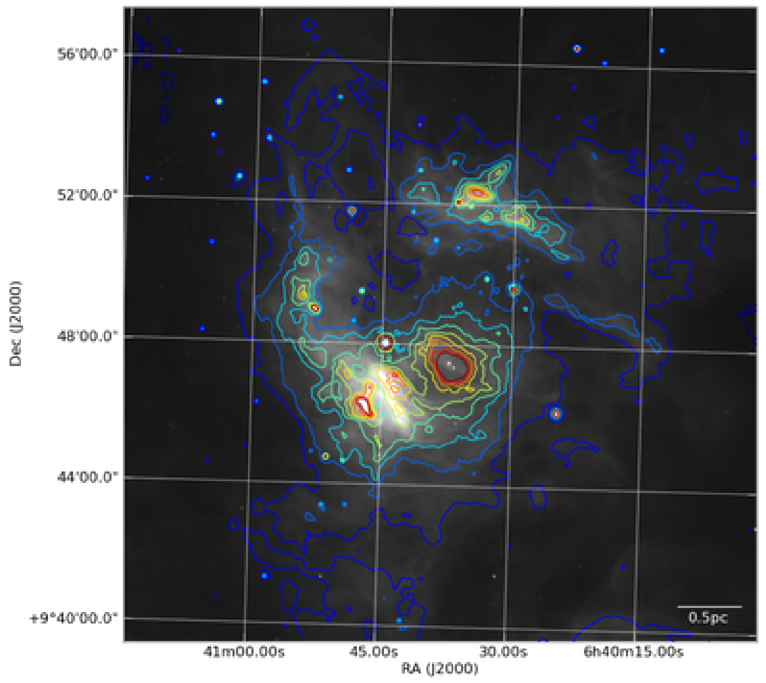
And finally let's add a scalebar to make the plot look sciency:

```
f.add_scalebar(0.03, '0.5pc', color='white')
```

We can now save our masterpiece either by clicking on the Save icon in the matplotlib window, or doing:

```
f.save('my_first_plot.eps')
```

The latter is recommended because it will automatically figure out the best resolution with which to output your plot. Your plot should look something like this:



Exercise 1

Use the `help` or `?` functionality in `ipython` to figure out how to set the min/max levels on the grayscale manually, and to change the stretch function to a square-root stretch. Also use the Use the Quick Reference Guide to figure out how to change the grayscale to a colorscale.

To manually set the levels:

```
f.show_grayscale(vmin=0., vmax=200.)
```

To additionally use a square-root stretch:

```
f.show_grayscale(vmin=0.,vmax=200., stretch='sqrt')
```

To change to a colorscale:

```
f.show_colorscale()
```

Note that the colormap can be set using for example:

```
f.show_colorscale(cmap='gist_heat')
```

where the value of the cmap argument can be any of the names listed on this page.

Exercise 2

Use the Quick Reference Guide to manually set the tick spacing on both axes. In the default view for the example FITS file above, the arcseconds in the declination are not useful (they are always zero). Try and change the format of the y-axis labels so that they only include degrees and arcminutes.

To set the tick spacing:

```
f.ticks.set_xspacing(0.05)
f.ticks.set_yspacing(0.05)
```

To show the y-axis labels in dd:mm format:

```
f.tick_labels.set_yformat('dd:mm')
```

Exercise 3

Use APLpy to plot one of your own FITS images! If you don't have any FITS files at hand, you can play with the M82 files provided in the data directory for the workshop (`m82_wise`).

If you have trouble downloading the file, then start up IPython (`ipython --pylab`) and enter:

```
import urllib2, tarfile
url = 'http://python4astronomers.github.com/_downloads/m82_wise.tar'
tarfile.open(fileobj=urllib2.urlopen(url), mode='r|').extractall()
cd m82_wise
ls
```

2.4 Where to learn more/get help

- IPython Tab-completion
- Google
- StackOverflow
- astropy mailing list
- Colleagues!

CHAPTER

THREE

ACKNOWLEDGMENTS

This workshop makes use of the template developed for the Practical Python for Astronomers workshop at http://python4astronomers.github.com, which was developed by Tom Aldcroft, Tom Robitaille, Brian Refsdal, Gus Muench and the Smithsonian Astrophysical Observatory creation.

Authors Thomas Robitaille