Worming your way around Python

Introduction

The aim of this problem set is to exercise your python know-how and also to get some experience with topcat just because it is good to know (you might already know all this - in which case you can skip all this).

Getting started

Getting some data

The first task is to get hold of some data to plot - the focus here will be on tabular data for now and we will use the Exoplanet encyclopaedia at <u>exoplanet.eu</u>. Go to that site and download the catalogue in VOTable format. You can use any table file you want, for instance from Vizier, but if you want to do the problems below for your enjoyment, the exoplanet file is needed.

Getting topcat

The topcat program is a super-useful utility program that everyone working with astronomical data should be aware of. For machine-learning it is convenient for exploring and quickly converting catalogues - but we will look at how to do this in Python as well below.

The first task is to start-up Topcat. It might be installed already but if you need it for your own computer you can find it at:

```
http://www.star.bris.ac.uk/~mbt/topcat/
```

Go there and download the topcat-full.jar file and start this. If this is presenting problems, try the WebStart version a bit further down on the page. If it is installed, just open a terminal window and type topcat. All ok? You should see a display like this, depending on the computer you use
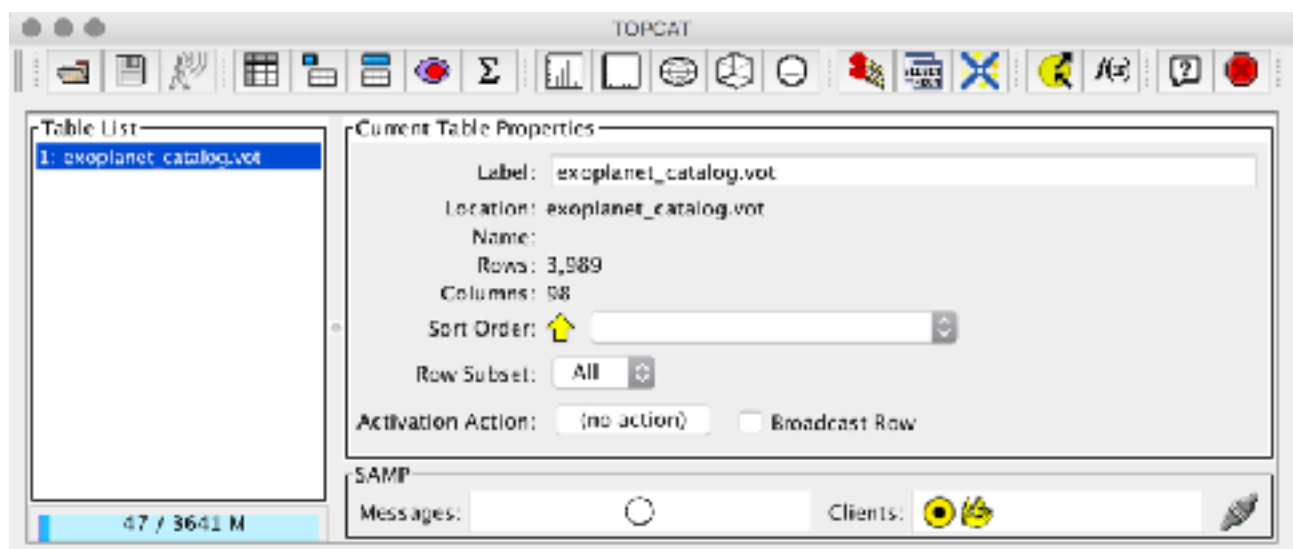
topcat can load most types of tables that astronomers use and it has a useful table viewer and convenient plotting functions. These will be described a bit in a lecture but it is useful to know how to do the basics:

These buttons are used to show the data in table format (left-most), see generic information about the table (second button), see an overview of columns (third button) and to define subsets of the data (right-most).
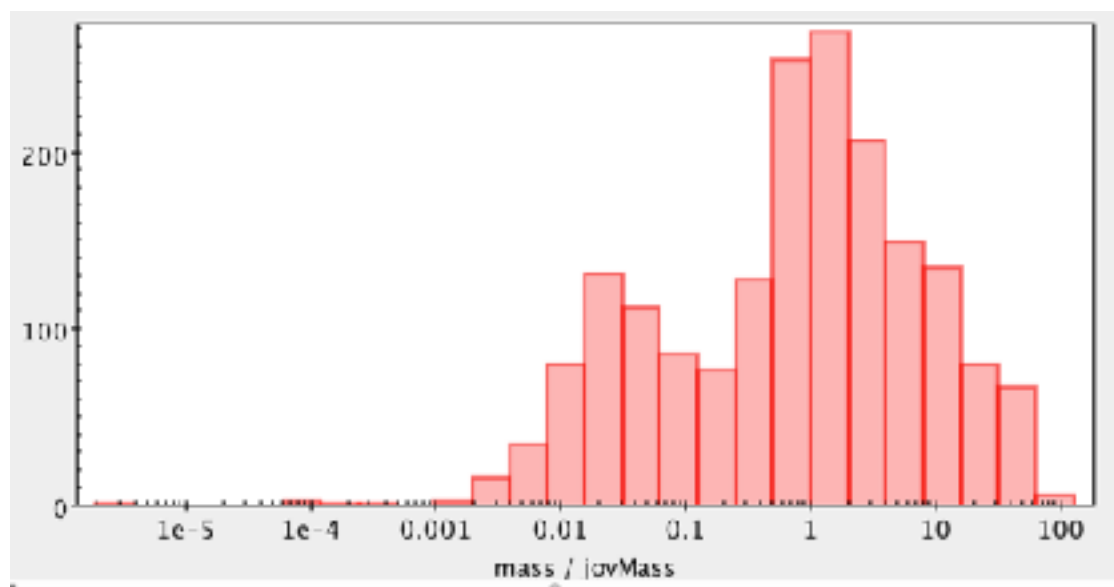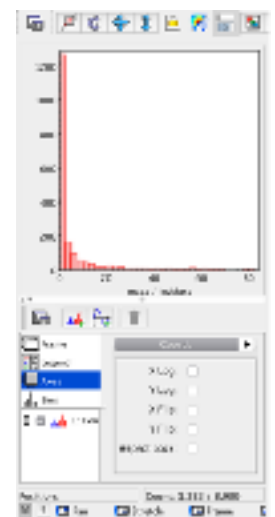

These buttons are used to plot the data. The first creates histograms and the second plots scatter plots (x versus y).
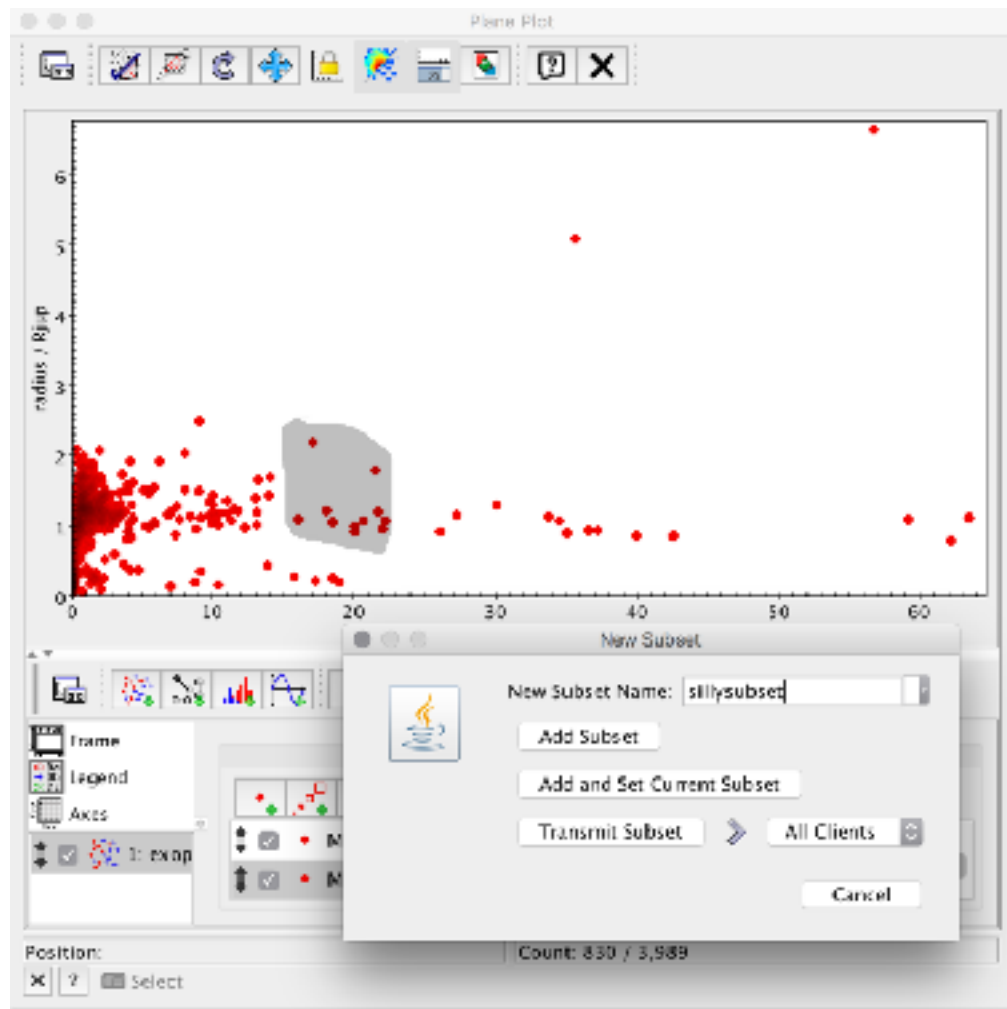
Click on the table button to bring up a display that looks like this:

you can use this interface to browse the table and when you have plotted columns in this table you can click on a table row to highlight that object in the plot, and vice versa. But for now, click the histogram button to create a histogram of the mass of the exoplanets:

That actually looks a bit sub-optimal. You can see that there are a lot of low-mass exo-planets but they all end up in a single bin on the left. In cases like this it is often useful to use a logarithmic x-axis, so click on the 'Axes' option in the left panel and tick the option for X log in the panel. This should give you a result looking like this:

You can manipulate this histogram in various ways. When you click on the dataset you bring up a menu with three buttons: Position, Subsets and Form. Position allows you to select the variable you want to show (and an optimal weight) and the table to plot. Subsets allows you to decide which subset to show (see below) and Form allows you to modify the appearance plus adding various density estimators - a topic we will return to later in the course. If you click on the KDE button indicated below you get a kernel density estimate which attempts to reconstruct the density distribution in a smooth way. Play around with this - it will be useful later for developing intuition for parameters in density estimators. It might be a good idea to first go to the 'Bins' entry and 'General' and choose 'Normalise=area'

Besides simple plots you might also find topcat to be useful for looking at complex subsets and

identifying outliers. As an example, let us create a scatter plot of mass versus radius. At the same time also open the table browser side by side. When you now click on individual symbols the table will move to the appropriate row - very convenient at times. For instance it might look like:

Finally if you click on the irregular symbol you can start defining a complex region. Clicking on the button a second time will save it. You can see an example result on the following page.

While topcat is great for many things and is definitely a tool you should have in your toolbox, it has some clear limitations: some types of interactions are not easily reproducible - and if it isn't reproducible it is not suitable for publication - but might be useful in an early exploratory stage of course.

Topcat is also not particularly suitable when you have many tables or you need to make very complex transformations of columns (but you can do quite a lot). For this you can use the command-line interface called stil, but I suspect most people will prefer to use Python so I will turn to this next.

Moving to python

Topcat is very versatile and useful but in this course we need to use python. This is not a good place to learn the basics of python - a good place is http://learnpythonthehardway.org/book/
Make sure you learn about dictionaries and lists and be prepare to forget some of that!

Now - gone through that? Good! Now, let us touch a bit on NumPy. This is the framework we will use for most of our calculations in our course so you need to gain some familiarity with this. We (and most everyone else) normally import this using:

```
In [1]: import numpy as np
```

(note that I use ipython (see below) and therefore each input line (what you type) will be prefixed by In [<number>] and the output will be prefixed Out [<number>] when I bother to write it.

NumPy provides access to powerful array features and allows you to operate on quantities without looping over them element by element (a very slow operation in Python).

Creating arrays

```
In [2]: a = np.array([3.1, 4, 5, 2])     # From list to array

In [3]: a*2
Out[3]: array([  6.2,   8. ,  10. ,   4. ])

In [4]: np.arange(4)              # Similar to range()
Out[4]: array([0, 1, 2, 3])

In [5]: np.linspace(2, 3, num=3) # three numbers evenly distributed between 2 and 3
Out[5]: array([ 2. ,  2.5,  3. ])

In [6]: nothing = np.zeros(5)

In [7]: ones = np.ones(5)

In [8]: x = np.random.random(size=10)  # Random array

In [9]: xd = np.arange(5, dtype=np.int)   # Create an array of integer type

In [10]: xd.dtype
Out[10]: dtype('int64')    # On my 64-bit computer - you might get a different result
```

Note in particular how I created an array of a particular type - in some situations it is crucial that you have the right type, and in others you can save space by working with the smallest type necessary. The default is a float array.

Operating on arrays
When you have the arrays you can access individual elements and then do operations on all elements in one go - what often is called vectorised access, or in python lingo these are ufuncs - universal functions.

```
In [2]: x = np.random.random(size=10)
```

```
In [3]: x.mean()                 # or np.mean(x)
Out[3]: 0.56451589378025646      # Your number will differ

In [4]: np.sin(x[1:3])           # Note how I access two elements
Out[4]: array([ 0.33083056,  0.50987037])
```

And for multi-dimensional array you can thread automatically over a dimension:

```
In [5]: x = np.ones((2, 3))

In [6]: y = np.arange(-1, 2)

In [7]: y
Out[7]: array([-1,  0,  1])

In [8]: x
Out[8]:
array([[ 1.,  1.,  1.],
       [ 1.,  1.,  1.]])

In [9]: x + y
Out[9]:
array([[ 0.,  1.,  2.],
       [ 0.,  1.,  2.]])
```

This can be very powerful when used right and very confusing when used wrongly! If you make use of this in a complex function, make sure to document clearly what you are doing - or you will be confused in the future, and you will cause untold headache to people that will use your code.

There are lots of Numpy functions - a few that are useful are: `np.min/np.max` for calculation of minimum and maximum, `np.argmin/np.argmax` for giving the index of the minimum or maximum element, `np.mean/np.median/np.std` for means, medians and standard deviations of a vector, `np.isfinite` to check which elements in an array are finite

Array access
Numpy also has a powerful indexing capability for arrays.

```
In [10]: x = np.array([1,1,2,3,5])

In [11]: x[1]
Out[11]: 1

In [12]: x[3]
Out[12]: 3

In [13]: x[0:3]
Out[13]: array([1, 1, 2])

In [14]: x = np.array([[1,2], [2,3], [4, 5]]) # Create a 3x2 array

In [15]: x
Out[15]:
array([[1, 2],
       [2, 3],
       [4, 5]])

In [16]: x[: ,1]      # Get column #2 (counting starts at 0)
Out[16]: array([2, 3, 5])
```

```
In [17]: x.shape          # Get the dimensions of this array
Out[17]: (3, 2)
```

You can also index and give steps
```
In [19]: x = np.linspace(-3, 3, num=7)

In [20]: x
Out[20]: array([-3., -2., -1.,  0.,  1.,  2.,  3.])

In [21]: x[::2]     # Get only every second element
Out[21]: array([-3., -1.,  1.,  3.])

In [22]: x[::2] = 0     # Set every second element to zero

In [23]: x
Out[23]: array([ 0., -2.,  0.,  0.,  0.,  2.,  0.])
```

for even more complex selections, the Numpy `where` function is very useful:

```
In [24]: x = np.array([1, 3, 5, -2, 4, 9])

In [25]: inds = np.where(x > 3)

In [26]: inds
Out[26]: (array([2, 4, 5]),)

In [27]: x[inds]
Out[27]: array([5, 4, 9])
```

see also the `np.select/np.take/np.choose` functions - all powerful and potentially confusing but worth learning.

Simple plotting

The main plotting package for python is matplotlib - this is a powerful package and of course with power comes some complexity - so you need to invest some time into learning this package well. If you work in ipython it is convenient to use the %pylab magic to work

```
In [1]: %pylab
Using matplotlib backend: MacOSX
Populating the interactive namespace from numpy and matplotlib

In [2]: import numpy as np

In [3]: import matplotlib.pyplot as plt

In [4]: x = np.linspace(-3, 3, num=100)

In [5]: y = np.sin(x*x/np.pi)+x

In [6]: plt.plot(x, y)
Out[6]: [<matplotlib.lines.Line2D at 0x10d0a6690>]

In [7]: plt.xlim(-1, 1)
Out[7]: (-1, 1)
```

and you can save your figure to various formats using `plt.savefig`. There are many other ways of using matplotlib and you need to experiment a bit.

I will now assume you are ok with python - so now we go a good step further.

Intermezzo 1: Working with ipython and friends

An efficient working environment is always important - most of us do not prepare dinner in our bathroom, and so it is with python as well. There are many ways to interact with python and integrated coding environments abound. It is not the aim of this course to guide you through these - you need to explore on your own, but it is probably a good idea to look at ipython.

This is a replacement for the normal python command line - if you type python on the command line you get a display more or less like this:

```
leukermeer [33] > python
Python 2.7.8 (default, Nov 10 2014, 08:19:18)
[GCC 4.9.2 20141101 (Red Hat 4.9.2-1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

you can then proceed to type in commands, and it will work fine. However it is quite limited in what it provides and ipython (ipython.org) is a powerful alternative, although it takes a bit longer to start up. If you start ipython you get:

```
leukermeer [34] > ipython
Python 2.7.8 (default, Nov 10 2014, 08:19:18)
Type "copyright", "credits" or "license" for more information.

IPython 1.1.0 -- 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]:
```

which already tells you there are is more help to be had. One particularly nice aspect is that it helps you find out what an object can do. As an example let us see what the functions are for a normal numpy variable:

```
In [1]: import numpy as np

In [2]: a = np.zeros(5)

In [3]: a.<PRESS TAB HERE!>
a.T             a.clip       a.dot        a.item        a.prod       a.setasflat    a.swapaxes
a.all           a.compress   a.dtype      a.itemset     a.ptp        a.setfield     a.take
a.any           a.conj       a.dump       a.itemsize    a.put        a.setflags     a.tofile
a.argmax        a.conjugate  a.dumps      a.max         a.ravel      a.shape        a.tolist
a.argmin        a.copy       a.fill       a.mean        a.real       a.size         a.tostring
a.argsort       a.ctypes     a.flags      a.min         a.repeat     a.sort         a.trace
a.astype        a.cumprod    a.flat       a.nbytes      a.reshape    a.squeeze      a.transpose
a.base          a.cumsum     a.flatten    a.ndim        a.resize     a.std          a.var
a.byteswap      a.data       a.getfield   a.newbyteorder a.round     a.strides      a.view
a.choose        a.diagonal   a.imag       a.nonzero     a.searchsorted a.sum
```

so as you can see - if you press <TAB> you get a list of what functions you have. So you can now find out what the sum of this variable is:

```
In [3]: a.sum()
Out[3]: 0.0
```

You can also write your code in some text editor and then paste it into ipython - to do that you select the text and then type %paste in ipython to glue it in. This can be a very efficient way to try out new pieces of code.

Notebooks

In this course we will make use of notebooks - and with that I do not mean the traditional paper ones (but they are of course very useful still!) - but rather the Jupyter notebook facility. This allows you to edit code and make plots interspersed with text in a very powerful structure. This is very widely used these days and works for various languages (although it grew out of python/ipython).

To try this out you should type:

```
> jupyter notebook
```

on the command line. Depending on the settings on your computer this will either open up your web browser with a notebook overview page, or print a URL that you need to copy into your browser to access the notebook overview.

Notebooks are very interactive creatures and you quickly get a handle on this. Thus I have added an example notebook alongside this document which gives some experience with this.

Notebooks are very convenient, but they are not always the right tool, at least through the whole project. Consider for instance a situation where you are developing a large piece of software, say you are creating a code to create simulated spectroscopic observations for a future space mission. It is then very useful to use a notebook in at least two ways:

1. To test out subsystems of the full code to ensure that things work well. In this case it is definitely a good idea to write well-documented and well-structured code in the notebook so that you can easily export what you have written to enter into your larger piece of software.
2. To test final code. In a proper code base you should have tests integrated in the package but the reality is that in most slightly complex Python code you need to do other kinds of tests and checking of the code. For this a notebook can be very handy - not just for running the tests, but also for documenting what you have been doing.

FILL IN!! In the future the Jupyter notebook will be replaced by the Jupyter Lab. You can find more information on this here and here, but this is not urgent and certainly not needed for this course.

Reading tables into Python

To continue this very simple introduction we need to read in a table to python. To do that you first need to think about how tables are stored in astronomy: basically they are stored in four different broad categories: plain text tables, which can have a huge range of formats, structured text tables for instance implemented using a version of XML, binary tables such as FITS, and finally in databases, which we will look at later.

Until relatively recently this plethora could be a headache - and a lot of time can easily be wasted in converting a table from one format to another. This is considerably easier now with the powerful Table interface in the `astropy` package and this is a good choice for reading any type of tables into Python. It does not solve all cases but it does a pretty good job in general and it reads all commonly encountered table formats.

I also in general find that it is convenient to convert my tables to a format that is readable with all the various types of software I use. Very often a FITS table is a good lower common denominator - but note that the XML based VOTable is a much more flexible and powerful format albeit often slower and readers are less wide-spread than FITS readers.

So let us assume that you saved the exoplanet table into `exoplanet_catalog.vot`, then reading this in as a table into Python is as easy as

```
In [1]: from astropy.table import Table

In [2]: t = Table().read('exoplanet_catalog.vot')

<usually some warnings>
```

And `t` is now a table object which can easily be printed:

```
In [3]: print(t)
     name      planet_status   mass  ... star_detected_disc star_magnetic_field        star_alternate_names
                               jovMass ...
-------------- ------------- ------- ... ------------------ ------------------- --------------------------------------------
     11 Com b     Confirmed    19.4 ...               None                None                                         None
     11 Oph b     Confirmed    21.0 ...               None                None                        Oph 11A, Oph 1622-2405
     11 UMi b     Confirmed    10.5 ...               None                None                                         None
     14 And b     Confirmed    5.33 ...               None                None                                         None
```

You can get the list of column names by printing t.colnames and you get a single column by subsetting with these names:

```
In [3]: age_star = t['star_age']

In [9]: np.mean(age_star)
Out[9]: 5.0354198878343395
```

(your numbers might differ from mine, depending on the version of the code). Note that not all exoplanet entries have a known stellar age but the table columns are implemented as Masked Columns and when you call `np.mean` the missing data are ignored.

If you have FITS tables you can choose not to use astropy, and do something like
```
from astropy.io import fits
fname = 'exoplanets.fits'
fits.getdata(fname, 1)
```

Ok, I assume you have done all this - so now you can try things out:

Task 1: How many of the planets in your list were found by Kepler?

Assume that any exoplanet with a name that starts with 'Kepler' was detected by Kepler and use Python to count the occurrences.

I found a total of 937 - by the time you do this there might be more.

Task 2: What is the mean mass of the planets in the exoplanet.eu catalogue?

Do this in two ways:
1. Find out what the appropriate numpy function is called and use that.
2. Write a function in python to do the same calculation.

Intermezzo: modularity of code

It is almost always best to split your code into smaller units - at the very least using functions. As an example, here is a **bad** way to calculate the sum of the N first numbers from 0 to 5:

```
x = np.arange(1, 1+1)
print "Up to {0} the sum is {1}".format(1, x.sum())

x = np.arange(1, 2+1)
print "Up to {0} the sum is {1}".format(2, x.sum())

x = np.arange(1, 3+1)
print "Up to {0} the sum is {1}".format(3, x.sum())

x = np.arange(1, 4+1)
print "Up to {0} the sum is {1}".format(4, x.sum())
```

It works, but it is easy to make mistakes and later changes to the code are harder and for anything but the simplest examples this is hard to maintain and understand.

For that reason it is best to split off tasks like this into functions. The task above can be written:

```
def sumN (N):
    x = np.arange(1, N+1)
    print "Up to {0} the sum is {1}".format(N, x.sum())

for i in range(1, 5):
    sumN(i)
```

This is now much easier to generalise and modifications are easy as well because everything is in one particular place.

The bottom line: You almost always want to use functions! And cut-and-paste of code is a bad habit to get into!

Task: Exoplanets in the Hertzsprung-Russel diagram

The Hertzsprung-Russel (HR) diagram is a convenient way to visualise stellar populations. A typical plot of this shows the effective temperature of a star on the x-axis and the luminosity of the star in some band on the y-axis - both typically on logarithmic scales.

a) Plot a HR diagram for the host stars of the exoplanets. It is perfectly fine if the luminosity axis is in arbitrary units - indeed it is perhaps easiest to convert mag_v to absolute magnitude.

There are two points off at around 30,000K - what kind of stars are those?

Check: Did you add axis labels?

> **Reversing axes**
> The convention in astronomy is that the x axis has high temperatures on the left and low on the right - the opposite of the default.
>
> It is easy to do this matplotlib:
> ```
> import matplotlib.pyplot as plt
> plt.plot(x, y)
> plt.gca().invert_yaxis()
> ```

b) *More challenging:* That is a fairly low information plot - now let us add information to the plot by colouring each point according to the logarithm of the mass of the exo-planet.

c) *More challenging:* Next create the plot but now scale the symbol size according to the radius of the star. Does your diagram look anything like one of the HR diagrams you find on the net?

The two last points are a bit harder because you need to search a bit around and also making the plot look good is more work. They are not intrinsically hard though.

d) Find all stars with >2 planets identified and create a plot illustrating all these extra-solar systems and compare to our own solar system. [for your sanity you might want to just show a few rather than all].

Comments & documenting code

It is very important to add comments and instructions to your code. Try to make it a habit that you always add a comment that says what a function does when you write one. Add comments everywhere you write an equation that is anything but extremely trivial.

Comments are also very useful for yourself later, for others using your code and for anyone who you ask to help you with your code!

In python you add comments by placing # before and you add help text to a function using a structure of this kind:

```
def my_func(x):
    """This function will carry out a fancy function.
x — The argument to this function
    """

    return x*x
```

In essence it should be a string literal as the first argument after the declaration of the function. This then becomes the help text so if you later do help(my_func) you will get:
```
In [3]: help(my_func)
Help on function my_func in module __main__:
```

```
my_func(x)
    This function will carry out a fancy function
    x -- the argument to this function
```

And this is very useful, particularly for other people and for complex functions. If you want to learn more about this see https://www.python.org/dev/peps/pep-0257/.

It is also useful to know that some of these particular tasks are easier to do in topcat than in Python - but now you at least have a record of what you have done and can modify it easily in the future.

Random numbers and error propagation

While science is all about reproducibility, it is often the case that we need random numbers. Indeed random numbers are crucial for much of science and you do need to know how to create them. Luckily, it is quite easy with Python. The package you need is the random package in numpy:

```
In [1]: import numpy.random as r
```

The random number methods are now available as r.<something>, where 'something' is the name of the type of random numbers. We usually think about random numbers as being drawn from some distribution - the simplest one being the uniform distribution. This is defined over a finite range and any value is equally likely. In the numpy random library we can draw numbers from this using the uniform function:

```
In [2]: x = r.uniform(low=1, high=2.5, size=500)
```

will draw 500 random numbers between 1 and 2.5 with a uniform distribution.

There are many other distributions and uses for this - to give a more practical introduction there is a Jupyter notebook on the GitHub site (see the ProblemSets/Set0 directory).