A short course in Python for astronomers

Neal Jackson

1. General introduction

The Python language has major advantages - it is

- concise - I can usually write Python scripts in about a quarter of the length of a C program
- intuitive - it is very easy to guess what you need to type to do something you want
- scripted - there is no compilation
- *free!!* - you don't need to spend hundreds of pounds on a licence
- high-level - many basic, and lots of not-so-basic, functions have already been written, leaving you to concentrate on the problem you want to solve
- adopted by many users - if you have a problem, googling "python" followed by your problem usually gets you something useful.

It has one disadvantage; it's slower than compiled languages such as Fortran or C. This disadvantage can be considerably mitigated in the way you write code.

Start python by typing

```
python
```

It is an interactive language - you can test things, and then put them into a script to run non-interactively. To run non-interactively, type the commands into a file (conventionally ending in .py) and run the script with

```
python filename.py
```

Comment lines in scripts are introduced with # (anything after a hash on a line is ignored by the processor). Python is fussy about indentation: each line must be unindented in a simple script[1]. If you need to continue on to the next line, you need to write a backslash before continuing with the next line. To exit, use `CTRL-D`; to get help on something, type `help(something)`, and if you want to find out what you can do to something, type `dir(something)`.

Using existing routines usually requires the import command. For example, to use mathematical functions you need to type

```
import math
```

which is American for "maths", or write it in a script. You can then access a function called `name` by using `math.name(arguments)`. If there are imports that you do a lot, then you may wish to make a file called `pythonstart.py` which contains all the things you would like Python to do on each startup. In order for Python to see this file, it must be in a directory with the logical name `PYTHONSTARTUP`, which you can arrange in Linux by

[1]As we will see later, code inside loops is indented, but all such code must be indented consistently

```
setenv PYTHONSTARTUP [directory-name]/pythonstart.py
```

Finally, you can use Python as a calculator:

```
2+3
```

Exercises

1. [If you are working on a Linux system, either remotely via VNC or on your own machine.] Make yourself a pythonstart file and put the `setenv` declaration in your `/home/[username]/.cshrc` file. You will then only need to edit this file in future, in order to load programs by default.

2. Try to get python to generate an error message, by typing something which you think it will not understand, or trying to divide by zero. Try to get at least three different error messages.

3. Use Python to find the sin of 40 degrees. (Note that like most calculators, Python uses radians, and that the value of pi is `math.pi`).

---

2. Data types

Python has five main datatypes: `integer, float, string, complex, boolean`.

Integers and floats

Note the result of

```
2.0/3.0
2./3.
2/3.
2/3
2%3
19%4
15%6
```

Hence the rule that any float in an expression implies the rest are float not integers. As usual with programming languages, division of two integers yields an integer, any remainder being ignored. What do you think the `%` operator does?

String data types

Type the following:

```
a='test'
b='this is a '
c=2
d=2.0
```

Examine values just by typing the variable name e.g.

```
a
```

or using the print statement (arguments separated by commas)

```
print 'The value of a is',a
```

What would you predict for the following? Try (and record) them; they illustrate a number of very important functions.

```
b+a
c*a
d*a
a-b
b[0]
b[1]
b[2]
c==2         # note the use of == as a test
c==3
'his' in a
'his' in b   # note the use of 'in' - it is very useful!
len(b)
len(a)+len(b)
int(a)       # NB the int function turns things into integers
str(d)       # ... and str into strings, if possible
str(len(a)+len(b))*2
```

Moral: Python has a habit of doing what you would expect - or giving an error if something "obviously" doesn't make sense.

Operations on string data types

Here's one thing you can't do: try `b[2]='q'` and note the complaint. Moral: you cannot replace elements of strings.

Now try something that does work:

```
b.replace('s','z')
```

Note this does not change the value of b! You need to assign to another variable to do this e.g. `e=b.replace('s','z')`

You can get a complete list of things you can do to strings: `dir(b)`

Try a few. Useful ones include `b.swapcase()`, `b.rfind('a')`, `b.split(' ')`, `b.split('s')`. Work out what each of these does (the last one returns a list; see the next section).

Complex numbers

These work as you might expect:

```
a=2.0+1.0j
b=1.0+2.0j
```

Try a few operations e.g. `a+b,a-b,a/b,a.real,b.imag`.

Boolean

Start with two variables

```
a=True
b=False
```

Try the following: `not(a)`, `a&b`, `a&a`, `b&b` (and the same three with `|` instead of `&`. Deduce what these operators mean.

Exercises

1. Write a script whose first line is

`a='wombat'`

and which prints the first two and last two letters of the string `a`. When this works, experiment with `a='do'` and `a='z'` as the first line.

2. If `a='wombat'`, find an expression which does not involve the letter `'b'` or `'B'` that will give a) `w*mbat`, b) `wombatWQMBAT`, c) `WombatWombat`, d) `['wo','bat']` (two strings). e) `'00000000000000wombat'` (without typing any '0's). You will need to look at `dir(a)` for at least one of these.

3. Write a script which begins by defining two strings `a` and `b`, and outputs a Spoonerised version of the two strings separated by a hyphen, i.e.

```
a='wombat'
b='kangaroo'
[ your code here ]
```

returns `kombat-wangaroo`.

4. If `a=2.473+3.476j`, write a script to write `a` in the form `[float]e**[float]i`.

5. Hence or otherwise write a script to rotate a complex number by a given number of degrees in the complex plane - e.g.

```
a=1.0+1.0j
b=90.0
[your-code-here]
```

returns `-1.0+1.0j`.

---

3. Control flow

You need to know four basic control loops; `if`, `for`, `while` and `try`. Note carefully (i) the colons (ii) the use of `==` for a test of equality, and (iii) the indentation structure. Not indenting things is an error. Guess what each will do, and then try each one to check that you know what you will get in each case.

```
a=3

if a==2:
   b=3
elif a==3:
   b=5
else:
   b=7

for i in range (3,10):
   if i>5:
       print i
   else:
       print i,'is not bigger than 5'

i=0
while i<20:
    i+=1
    print 'Still in the loop and i is',i


a=3
try:
    b=a/0  # NB! normally stops with an error if you do this
except:
    b=4
    print b
```

In some cases you can get away with "implicit" loops; for example, `a=2 if b==3 else 0` will work.

A very useful application of the `for` loop is the ability to iterate simply over arrays[2] making use of `in`:

```
a = ['gnome','goblin','hobgoblin','troll']
for i in a:
    if 'b' in i:
        print i
```

Loops can be exited or repeated using the `break` and `continue` statements. (If you are familiar with C, they work the same way.) `break` causes an immediate exit from the loop at that point; `continue` abandons execution of the current loop iteration and starts again at the next one. Note that you can write an unending loop using a statement that is always true, such as `while 1==1:`, or more simply, `while True:`. Such loops must have a `break` statement in them somewhere (and you must be reasonably confident that it will be triggered).

[2]This is jumping ahead a little; see section 5 for more on arrays

Important comment: It is highly recommended that you *do not* use loops for any serious processing. For example, if you have an image a with 1000×1000 pixels and try something like

```
for iy in range(1000):
    for ix in range(1000):
        do-something(a[iy,ix])
```

this will typically take an order of magnitude longer than it needs to. The correct way to do this is described in the later section on `numpy`.

Exercises

1. Given a string `a`, write a program to write its letters vertically accompanied by their position in the string; so that `a='wombat'` would yield

```
1 w
2 o
3 m
4 b
5 a
6 t
```

Amend the program so that it does not print any line containing the letter `m`, and stops if the line contains the letter `a`.

2. Write a program to find the sum of all numbers up to 50 (without using the $n(n + 1)/2$ formula).

3. Write a program to generate and print a Fibonacci sequence (1,1,2,3,5,8....) stopping when the number gets to 1000.

4. Write a program to find all prime numbers up to 1000.

---

4. Simple input and output

a) Input from the terminal can be read into a Python string by `raw_input`:

```
a=raw_input('Enter something: ')
[type something]
print a
```

The information read can be typecast in the standard way, e.g. `float(a)`.

b) Input from a file can be read using the standard routines `open`, `close` and `readline`:

```
f = open ('myfile.txt')
s = f.readline()
f.close()
```

The first line will be read in the above example. The whole file can be read, line by line, using an always-true `while` loop and knowing that an empty string `''` is returned at the end of the file. Alternatively you can use `f.readlines()` which returns all space-separated entities in the whole file in a single list of lines. (You need further processing with `split` to give a complete array of words in the file).

c) If you have a file consisting of columns of floating-point data, you can read it with the routine `loadtxt`, which is provided by the `numpy` package. See later.

d) Output to the terminal is done using the `print` statement, which we have already met:

```
print s,angle,flux
```

will print the information from three variables, separated by a space, and terminating in a newline. You can format these in any way you like, using a syntax similar to C. For example,

```
print 'The answers are %3.1f  %10s  %7d\n\n' % (s,angle,flux)
```

prints `s` as a floating point number in a field of length 3, with 1 character after the decimal point, `angle` as a string of length 10, and `flux` as an integer of length 7. Note that the variables must be declared inside brackets (a form known as a tuple: see section 5), otherwise an error message will be generated. Note also that

```
'\n'
```

is a special character which forces a new line.

e) Output to a file is done using the `write` statement, for example:

```
f.write ('The answers are %3.1f  %10s  %7d\n\n' % (s,angle,flux))
```

The file must have been opened with an additional argument to `open` in order to access the file for writing: `f=open ('myfile','w')`. Don't forget to use `f.close()` at the end, or the file is not guaranteed to contain what you have written to it.

f) There are two other types of file writing you should know about: a compact and fast way of saving data known as a pickle; and reading and writing of FITS files. These will be covered later.

Exercises

1. Write a program which will prompt for the user to enter a string followed by an integer, loop until the user enters a legitimate string followed by a legitimate integer (and nothing else), and print the integer followed by the string.

2. Write a program which will read a whole file full of text, and then output the number of times the word 'wombat' (in any case) is used, and the total of all integers present in the file. So, for example, a file containing

```
Wombat said: I think that 100
flowers bloom; no, said the other wombat -
it is more like 200 flowers.
```

would return `2 300`. Amend the program to make sure that any word containing `wombat`, such as `wombats'`, will be counted.

3. Write a program which will read in lines from a file and select the first three fields (separated by blank space). The program should write a file called `error` which contains lines which look like `Line 362 did not have 3 fields`, to identify those which did not have 3 fields. The program should then write, into a file called `output`, the first three fields in the form of a string of width 10, followed by two floats of width 10 each with 3 places after the decimal point. In case this is not possible, the fields should be written as strings of width 10.

---

5. Arrays

Python provides several types of array:

Lists. These are structures containing several individual pieces of data; you can mix data types within a list. They are declared using square brackets and indexed starting at 0. They can be modified item-by-item. Again, guess what the following does, and then try it:

```
a = [3.0,5,'hello',3+4j]

a[0]+1.5
a[2]+',world'
a[2]+1.5

a[0]+=1.5; a[0]  # note that you can put two commands on one line with a ;
```

Lists behave like strings under multiplication. (Try `a*3`, for example.) You can tell the number of elements in a list using the `len` function: `len(a)` will return `4` here.

Tuples are like lists but are immutable (cannot be modified item by item).

```
a = (3.0,4.0)
a[0]+1         # this is legal
a[0]+=1        # try this - will give an error
```

Numpy arrays are the closest things to arrays from most computer languages. The script must explicitly import the `numpy` package with the statement

```
import numpy as np
```

(the "as np" is optional but will save typing later). You can then, for example, declare a one-dimensional array with the `np.array` command:

```
a = np.array([2.,3.,4.,5.])
```

or for a 2×2 two-dimensional array:

`a = np.array([[2.,3.],[4.,5.]])`. You can also declare an empty array with `a=np.array([])` and append to it later. You can also load an array from a text file (for instance, one called `file.txt`) using `a=np.loadtxt('file.txt')`. This assumes that the text file is rectangular, has fields separated by spaces, and consists of only fields interpretable as floating point numbers. If the latter assumption is not true, you can instead read in an array of strings using the `dtype` argument: `a=np.loadtxt('file.txt',dtype='S')`. If the delimiters are not spaces (for example commas, as in CSV files) you would use the `delimiter` argument: `a=np.loadtxt('file.txt',delimiter=',')`

Note that unlike lists, numpy arrays behave the way that you would expect under arithmetic operations. Try, for example `a*2.0` or `a/2.0`. Like lists, however, and like all Python arrays, they are indexed starting at zero. Multi-dimensional arrays always have the most-rapidly varying index last. This sounds complicated, but you will understand it if you try the following loop (note the comma syntax for arrays of >1 dimension):

```
for i in range(2):
    for j in range(2):
        print a[i,j]
```

One very useful feature of these arrays is the powerful syntax for array indexing. First, you can take subarrays using colons. For example, if `a` is an image, then

```
b = a[2:5,3:6]
```

takes a sub-image from the fourth to the sixth x-pixel, inclusive, and the third to the fifth y-pixel. Note that the x-axis of an image, being the fastest varying, is the *second* argument; that both indices of the array begin at zero; and that the number after the colon represents the first index which is *not* part of the output array. All of these three points are frequent sources of error.

Two other things to note: if you leave off an index, Python does the sensible thing (i.e. goes from the beginning or to the end), and if you use negative indices, Python counts down from the last pixel instead of upwards from the bottom. So if you wanted to shave off the top row and right-hand column of an image, you want `b=a[:-1,:-1]`.

A second very powerful feature is implicit array indexing. Suppose that array `a` is a two dimensional array:

```
10 17 20
13 41 50
18 19 34
```

where the horizontal axis is the fastest varying. You would declare this array using:

```
a=np.array([[10,17,20],[13,41,50],[18,19,34]]) .
```

(Try also the command `a.sum()`). Then the command

```
b=a[a[:,1]>20]
```

will select all rows of the array whose second member is greater than 20. This will leave `b` containing just the middle line:

```
b=[[13,41,50]]
```

Note that the output is *still a 2-D array*, because the input is a 2-D array. To get a one-dimensional array you would need

```
b=a[a[:,1]>20][0].
```

If you want more than one condition, you need to use brackets and `&` and `|` for "and" and "or" respectively, for example:

```
b=a[(a[:,1]>18)&(a[:,1]<20)][0]
```

will create a 1-D array containing just the last row of `a`.

Finally, there is a way to do implicit array looping in condition statements. Suppose you have two arrays, `a` and `b`, and you want something to happen if all members of `a` are members of `b`. You can then use

```
if all (x in b for x in a):
```

as the test.

If you have a simple array, you can use `np.append` to add to the array. For example,

```
a=np.array([2.,3.,4.])
b=np.append(a, np.array([5.,6.]))
```

produces an array of length 5 in `b` (note that `a` remains unchanged). You can also append floats one by one to `a` (e.g. `np.append (a, 5.0)`). You can even append a string, in which case all of the array will be changed to string type. (Try it).

Exercises

1. Write a script to return the standard deviation of a portion of a 2-D array, specified by the user, from a text file of floating point numbers, also specified by the user. Make your own text file of numbers with your favourite text editor, and verify that it works.

A simple version of the FIRST catalogue (Becker et al. 1995, a catalogue of radio sources at 20cm made with the VLA) can be found on `www.jb.man.ac.uk/~ njj/first_simple.txt` (also available on a memory stick during the session). Its columns for each source are: right ascension, declination, peak flux, total flux, major axis of fitted component (in arcseconds), minor axis (if the component is resolved, otherwise this field is 0.0), and component position angle.

2. From the FIRST catalogue, print out a list of the right ascension and declination (both in degrees) of all sources which have peak flux density of 100mJy or more. Append a 'C' to those sources which are not resolved. Also find the total flux density of all sources south of the equator.

3. Write a script to read in a file and return an array which contains the index of all fields which could contain a date between 1900 and 2020, e.g.

```
Although solar eclipses (Alpha et al. 1980) might be granular
(Bethe & Gamow 2000), it is thought...
```

would return an array `[6,13]`.

---

6. Array operations

Here is a list of operations on numpy arrays. They are drawn from a trawl through my Python scripts. This list is not exhaustive; remember that you can get a complete list from `dir(np)` and help on any function with e.g. `help(np.loadtxt)`. Note that numpy also provides all of the mathematical functions you are ever likely to need, with reasonably obvious names, for example `np.sqrt`, `np.cosh`, `np.exp`... The general principle is that you should always try to use these "canned" routines for array operations, rather than writing your own iterations with a `for` loop, for speed reasons. Try out each of them until you are happy that you know what they do.

(a) Some that do exactly what they say on the tin:

`a=np.copy(b)`; `a=np.median(b)`; `a=np.gradient(b)`; `a=np.transpose(b)`; `a=np.sort(b)`. If in doubt, try it out with a sample array `b`.

(b) Some simple array operations provided by Python:

`a.max()`, `a.min()`	Maximum/minimum value of array
`a.T`	Transpose of an array
`a.clip(x,y)`	Clip values in array below `x` or above `y`
`a.ndim`	Number of dimensions of array
`a.shape`	Returns a tuple containing dimensions of array
`a.reshape(i,j)`	Reshapes an array to dimensions `i,j`
`a.ravel()`	Flattens an array to one dimension
`a.argmax()`	Index of maximum value (array is flattened first)
`a.astype('type')`	Changes type of all elements of array to `type` (returns error if this is not possible for any element)

(c) Range and array setup functions

`np.arange (start,stop,incr)`	Returns an array of numbers, beginning at `start`, separated by `incr`, and continuing until just below `stop`. With just one argument, returns an array of numbers beginning at 0 and ending just below that argument.
`np.linspace (start,stop,nelm)`	As `np.arange`, but the third argument is the number of elements.
`np.logspace (start,stop,nelm)`	As `np.linspace`, but separated in logarithmic intervals.
`np.random.random(nelm)`	Array of `nelm` random floats from 0-1. (You can turn this into a 2-D image by postfixing `.reshape(nx,ny)`).
`np.random.randn(nelm)`	Array of `nelm` Gaussian deviates with mean 0.0 and σ^2=1.0
`np.meshgrid (x,y)`	Evaluate functions on a grid.

The last function is not particularly obvious, but is very useful. To understand what it does, try:

```
x = y = np.arange (-5,5,1.0)
xx,yy = np.meshgrid (x,y)
```

Try looking at `xx`, `yy`. Also try `np.hypot(xx,yy)` to see what is going on; you will get an image returned in which each pixel contains its own distance from the point you specified with `np.arange`. This means that, for example, you could then write a simple function to generate a Gaussian of width w with

```
z=np.hypot(xx,yy)
gauss_image = np.exp(-z*z/(w*w))
```

This is the most general way of avoiding writing explicit loops (and thus incurring large time overheads) when processing images.

(d) Stacking and splitting functions

There are three main routines for stacking arrays: `np.dstack`, `np.hstack` and `np.vstack`. Each takes a single argument, namely a tuple of arrays to be stacked next to or on top of each other. They perform perpendicular, horizontal and vertical stacking respectively.

Investigate them by trying the following:

```
a=np.array([[2.,3.],[4.,5.]])
np.dstack ((a,a))
np.hstack ((a,a))
np.vstack ((a,a))
```

Also try `np.tile(a,(2,3))`.

(e) Element-by-element placement and modification

These routines should always be used instead of writing loops over indices. The one you are likely to use most is `np.putmask (array, condition, value)`, which replaces elements of the array which fulfil `condition` with `value`, e.g.

```
a=np.array([[np.nan,3.],[4.,5.]])
np.putmask (a,a>4.5,1.0)
a
```

This replaces just one element of the array. Note the use of `np.nan`, which is the "not-a-number" value used by Python to represent undetermined array elements (for example, bad pixels in an image). If you wanted to replace NaN elements with zero, you would use `np.putmask(a,np.isnan(a),0.0)`. Note also that this routine modifies the value of `a`.

If you want to know where in an array something is, you are likely to need `np.argwhere`. For example,

```
a=np.array([1.,2.,3.,4.,5.,4.,3.,2])
np.argwhere(a==3.0)
```

returns a 2×1 array, `[[2],[6]]`. You can do this for >1D arrays too; for example, try the following and be sure you understand the output:

```
a=np.array([[1.,2.,3.,4.],[5.,4.,3.,2]])
np.argwhere(a==3.0)
```

You can delete or insert rows or columns of an array with the two routines `np.delete(a,index,axis)` or `np.insert(a,index,axis)`. Try, for example,

```
a=np.array([[0,1,2,3],[4,5,6,7],[8,9,10,11],[12,13,14,15]])
b=np.delete(a,[0,2],0)       # delete 1st and 3rd rows
b
b=np.delete(a,[0,2],1)       # delete 1st and 3rd columns
b
```

Also try `np.insert`.

A further, slightly more complicated, function is `np.take (array,index_array,[axis,out,mode])` whose function is to select subarrays in a wide variety of ways, based on indices given in the second argument. Again, this is best illustrated by example:

```
a=np.array([[1,2,3],[4,5,6]])
np.take(a,[1],axis=1)      -> [[2],[5]]
np.take(a,[1],axis=0)      -> [4,5,6]
np.take(a,[0,1],axis=0)    -> [[1,2,3],[4,5,6]]
np.take(a,[0,1],axis=1)    -> [[1,2],[4,5]]
```

If there is something you need to do to an array which is not covered, again try `dir(np)` and the `help` function. You may wish to check `np.place`, `np.put`, `np.roll`.

A function which is often useful is `np.unique`. This returns the sorted, unique members of an array, flattening it to one dimension if necessary. For example, `np.unique([4,5,3,4,2,3,5])` gives `[2,3,4,5]`.

Exercises

Do not use `for` loops in the following exercises.

1. Make a 2-D image full of empty sky, whose size and counts per pixel can be specified by the user. The noise in each pixel should be the square root of the number of counts.

2. Make a 2-D image whose values at each pixel are the distance from a pixel which is specified by the user. Then make a 2-D image which contains a Gaussian with a user-specified width, centre (in pixels) and peak flux.

3. Make a $n\times 2$ array of RA and declination (in degrees) of all FIRST sources with total flux density >100mJy. Find the two >100mJy sources with the smallest separation in declination.

7. Functions, subroutines and system interaction

So far you have written simple scripts without subroutines or function calls. Like C or Fortran, Python provides a way to do this using the `def` statement. Here is an example; suppose we have a file `myprog.py` which consists of the following code

```
import numpy as np

def func1 (a,multiplier=2.0,index=3):
    try:
        b = func2 (a,multiplier,index)
        return b
    except:
        print 'Error'
        return np.nan


def func2 (a,m,i):
    return m * a[i]
```

The code takes an input array `a`, multiplies its `index`th element by `multiplier`, and returns this. If an error is found (for example if the array is shorter than length `index`), a Not-A-Number value is returned. Note the following features:

- `numpy` must be declared at the beginning using the `import` statement.
- Each subroutine is introduced by the `def` declaration, which must end with a colon. All subsequent lines are indented, until the end of the function.
- Arguments are given in brackets. Any argument which does not contain an equals sign must be given when the program is called. Any argument which contains an equals sign need not be specified; if not, the default value is given after the equals sign. Required arguments must precede defaulted arguments.
- Note the use of the `try: except:` syntax. Because `func2` is inside one of these loops, any error encountered in `func2` (such as `a[i]` not existing because the array is shorter than `index` elements) will not cause the program to stop, but will trigger the `except` statement.
- The `return` statement exits the current subroutine and returns a value. You do not need to return a value. You can return scalar values or arrays, or return more than one quantity (e.g. `return b,b*4`).

This script can be run from the python command line by importing the file, and running `func1`:

```
import myprog
myprog.func1(np.array([1,2,3,4]),2,3)
```

There are various alternatives. You can import using `from myprog import *`; in this case you can invoke `func1` instead of `myprog.func1`, although you then do risk clashes if a subroutine called `func1` is declared in another program. You can give the arguments in a different order by specifying the name of the argument: for example `myprog.func1([1,2,3,4],index=2,multiplier=3)`.

Interaction with the operating system is carried out by declaring `import os` at the beginning of a script. This module contains a large number of routines for performing system functions from within a Python program, of which by far the most useful is `os.system(command)`. This sends `command` to be executed by the operating system.

A further useful piece of system interaction is provided by `import glob`. `glob` contains one useful program, `glob`, which provides an unsorted list of files in the current working directory which match the argument. This provides a way to run Python code on particular files within a directory. Thus, the following is a fairly standard piece of code:

```
import glob
import numpy as np
for i in np.sort (glob.glob('*.fits')):
    do\_something(i)
```

will run the function `do_something` on every file in the current directory which ends in the suffix `.fits`.

---

8. Further input and output: pickles, text and FITS

There are three further important things you need to know about input and output.

a) In theory, you can use `for` loops to save any Python structure to a disk file in text format. This is generally slow and inefficient. Python/numpy provides a useful way to save a structure and retrieve it later:

```
a=np.array([1,2,3,4,5,6])
np.save('thing',a)
b=np.load('thing.npy')
```

This code copies `a` into `b`, via a disk file called `thing.npy` (note that the `.npy` extension is added automatically by `np.save`. This file is not readable with an editor, but only with `np.load`; retrieval is typically a factor 5-10 faster than from a text file.

b) `numpy` provides a way of reading text files, each of whose lines contain the same number of columns. This is the `np.loadtxt` routine. All rows and columns are assumed to be interpretable as floating-point numbers. If this is not the case, you can read the file as an array of strings using

```
a = np.loadtxt(filename, dtype='S'))
```

and then use other `numpy` routines to deal with the array. For example, if you know that all of the second column consists of floats, then you can specify `col2 = np.asarray(a,dtype='float')`.

A more flexible veration of `np.loadtxt` is `np.genfromtxt` which can handle missing values in a variety of ways. Read the help file for full details.

c) Another important type of file is the FITS file. This is an extremely common way of storing astronomical data, including both images and radio interferometer visibility data. A FITS file consists of a number of 80-character headers, containing data about the file (for example, the number of axes and size of each axis of an image, together with the coordinates of the centre of the image. The `pyfits` module handles interaction of Python with FITS files. It is very flexible and powerful, and has its own separate documentation on the `pyfits` web page[3]. To first order, all you need to know is

```
import pyfits

a=getdata('filename.fits')
h=getheader('filename.fits')
pyfits.writeto('newfile.fits',a,h)
```

In this example, data from `filename.fits` is read into a Python array `a`, and the header information into the list structure `h`. These are then written out into `newfile.fits`; you can treat `a` exactly as any other `numpy` array and modify it before writing it out. You can also modify fields of the header. For example, `h['CRVAL1']=130.0` changes the first (right-ascension) coordinate of the image reference pixel to 130 degrees. You can get a complete list of what is in the header using `print h`. Again, the `pyfits` documentation goes into more detail.

Exercises

1. Write a script to write a 100×100 FITS file containing a Gaussian with a centre position, central flux and width specified by the user.

2. For each file ending in `.fits` in the current directory, find the brightest pixel and its position in right ascension and declination; write this information in columns in a file.

3. Write a script to forcibly convert a rectangular array to floating point, replacing any values that cannot be converted with Not-A-Number.

4. Write a script to take the resulting array and write out another file consisting of the second and third column, provided that either of the two is not NaN, for all rows in which the first column is greater than 0. The program should exit gracefully if an error occurs (e.g. the file has fewer than three columns).

---

9. Scientific functions: numpy and scipy

`numpy` provides some higher-level functions, notably polynomial fitting; further routines are provided by the Scientific Python package, `scipy.` This section gives an overview of the most important, which occur over and over again in data analysis and simulation: polynomial fitting, Fourier methods, array interpolation, and optimization/function minimization.

a) Polynomial fitting

[3]You may or may not be able to install pyfits directly on your system. If not, it is provided by `astropy` in the package `astropy.io.fits`.

This is done using the `numpy.polyfit` routine. It has three compulsory arguments: x, the sample point array, y, the array of measurements, and *deg*, the desired degree of the polynomial to fit.

b) Fourier transformation

There are several Fourier packages available for Python. Probably the best is the scipy package `fftpack`, which you can import using, for example,

```
import scipy as sp
from scipy import fftpack as ff
```

This package provides both 1-D and 2-D Fourier transforms. For 2-D FFTs, the forward transform is given by `ff.fft2 (a)` and the backward transform `ff.ifft2 (a)`. Note that in general the arrays produced are complex; if you want the real parts only you need to take `a.real` or `a.imag`.

c) Interpolation

A useful and often needed 1-D interpolation routine can be found in numpy:

```
a = np.interp (x, xp, fp, left=None, right=None)
```

Given a set of measured values `fp` at points `xp`, the routine returns an interpolated array `a` of values evaluated at a different set of points, `x`.

There is also an `interp2d` routine in the package `scipy.interpolate`. Here's the example from the `scipy` documentation. You create an "interpolator" using the function defined on the existing grid (`xx,yy` here) and then interpolate on to a new, finer grid (`x,y`).

```
from scipy import interpolate
x = np.arange(-5.01, 5.01, 0.25)
y = np.arange(-5.01, 5.01, 0.25)
xx, yy = np.meshgrid(x, y)
z = np.sin(xx**2+yy**2)
f = interpolate.interp2d(x, y, z, kind='cubic')
xnew = np.arange(-5.01, 5.01, 1e-2)
ynew = np.arange(-5.01, 5.01, 1e-2)
znew = f(xnew, ynew)
```

d) Optimization

This routine is accessed by importing the `scipy.optimize` package, followed by `from scipy.optimize import fmin`. The `fmin` routine uses a downhill simplex algorithm to find the minimum of a function (and the `optimize` package also provides other ways of doing this).

Let's suppose that you have a subroutine called `myfunc` which calculates the value of a function, given an array of input parameters, which are to be adjusted in order to minimise the function value, `x0` and an array of other information, `x`. Your function is required to have these two arguments:

```
def myfunc (x0, *x):
    [ some code here ]
    return value
```

The subroutine which performs the minimisation must first set the values of `x0` to some reasonable initial values, and the values of `*x` in order to pass any supplementary information which is needed by `myfunc` to calculate the current function value, but which is not to be adjusted. You then call the `fmin` routine from within the main program using

```
xopt,fopt = fmin(myfunc, x0, x)
```

and on exit, `xopt` will contain the optimized parameter array, in the same order as `x0`, and `fopt` will contain the final minimised value for the function.

e) Integration of functions

Here's how to integrate a function (in this case a Bessel function, $J_{2.5}(x)$, from 0.0 to 4.5):

```
import numpy as np
import scipy as sp
from scipy import integrate
from scipy import special
result = integrate.quad(lambda x: special.jv(2.5,x), 0, 4.5)
print result
```

Note: A lambda function is a small anonymous function that is restricted to a single expression here x is the argument and the function calculates the Bessel function for value x.

Now compare with the analytic solution:

```
I = sqrt(2/pi)*(18.0/27*sqrt(2)*cos(4.5)-4.0/27*sqrt(2)*sin(4.5)+ \
    sqrt(2*pi)*special.fresnel(3/sqrt(pi))[0])
print I
```

Exercise. 1. Find the coefficients of the second-order polynomial $ax^2 + bx + c$ which give the best fit to the function $y = e^{-x}$ between $x = 0$ and $x = 1$, using discrete values of the function at intervals of 0.1 in x. Do this in two ways: (i) by a polynomial fit and (ii, harder) by optimization.

---

10. Plotting with matplotlib

All plotting functions are imported with the following incantation:

```
import matplotlib
from matplotlib import pyplot as plt
```

Depending on the machine you are running on, you may also need to specify `matplotlib.use('Qt4Agg')` (it's a long story).

a) Plotting 2-D images

If you have a 2-dimensional array `a`, this can be plotted with `plt.imshow(a)`, followed by `plt.show()` to put the plot on the screen. Try this with an arbitrary array `a`.

The default is to interpolate between pixels and to plot the y-axis the "wrong" way round - i.e. with $y = 0$ at the top. You can change this using

```
plt.rcParams['image.interpolation']='nearest'
plt.rcParams['image.origin']='lower'
```

If you like this way better, then you may wish to include this in your `pythonstartup.py`.

You can also add colour wedges and make multiple plots. For your favourite array `a`, try

```
plt.subplot(221);plt.imshow(a);plt.colorbar()
plt.subplot(224);plt.imshow(a*2.0);plt.colorbar()
plt.show()
```

You can get labels for the plot with `plt.xlabel`, `plt.ylabel`, and `plt.title`. You can change the colour lookup using the argument `cmap=matplotlib.cm.XX`, where `XX` is one of a list which you can get using `dir(matplotlib.cm)`. You can also change the x and y-limits using the two-argument functions `plt.xlim` and `plt.ylim` (which you need to invoke *after* the `plt.imshow`).

If you want different black and white levels, you can get them using the arguments `vmin` and `vmax` to `plt.imshow`.

Finally, to save the figure instead of viewing it on the screen, use `plt.savefig(filename)`. If `filename` ends in `.ps`, you will get a postscript file, otherwise PNG is the default.

b) Scatter and line plots

Points specified by an abscissa array x and ordinate array y can be plotted using `plt.plot(x,y,'ct')`. `'c'` here should be replaced by a letter representing the colour you want; all are reasonably obvious except that black is `'k'`. `'t'` should be replaced by the type of point you want: `'-'` for lines, `'--'` for dotted lines, `'o'` for circles, `'s'` for squares. See `help(plt.plot)` for a full list, and help on more complicated options.

You can plot several different arrays on one plot; all you need is to do several `plt.plot` calls before the `plt.show` or `plt.savefig`.

If you are particularly fussy about the tick labels on a plot you can make custom tick arrays. For example, `plt.xticks([0.0,100.0,200.0])` will plot just three ticks and tick labels on the x-axis, at 0,100 and 200.

You can write text on plots; `plt.text(x,y,s)` will do this, where `x` and `y` are floating point numbers and `s` is a string.

Quite often you will need to make plots with logarithmic axes. Matplotlib provides three functions for this: `plt.loglog`, `plt.semilogx` and `plt.semilogy`. Apart from the scaling of the axes, these routines function in the same way as `plt.plot`.

c) Histograms

These are easier; given an array `a`, you need only `plt.hist(a)`. Optional arguments include `bins`, the number of bins and `range`, a Python list of two numbers for the x-range for the plot.

---

11. astropy and aplpy: astronomy packages

`astropy` is a general-purpose astronomical package for performing functions such as coordinate transformation, cosmological calculations, common statistical calculations, FITS and VO file handling, and astrometry. It is complemented by `aplpy`, which is a general-purpose plotting package for astronomy. In particular, it handles astronomical coordinates, which `matplotlib` does not.

Full documentation is available on `docs.astropy.org`. Here is a code snippet which does something relatively common, i.e. calculate the angular distance between two points on the sky.

```
import astropy
from astropy.coordinates import SkyCoord

def astropy_sep (tra1,tdec1,tra2,tdec2):
    sc1 = SkyCoord(tra1, tdec1, frame = 'fk5', unit='degree')
    sc2 = SkyCoord(tra2, tdec2, frame = 'fk5', unit = 'degree')
    return((sc1.separation(sc2)).degree)
```

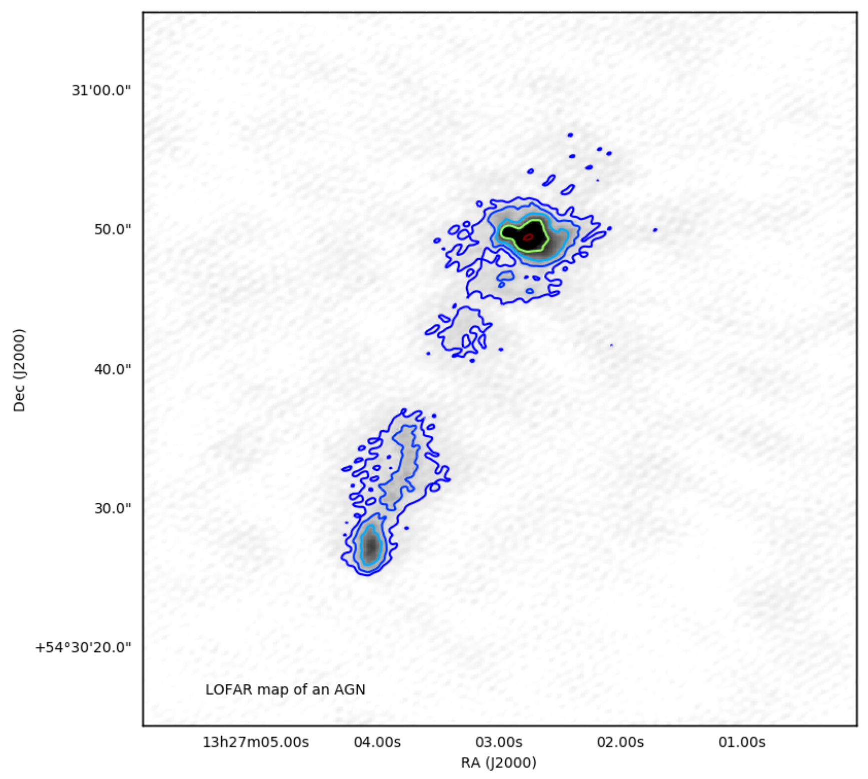
You can also convert coordinates from floating-point to hexadecimal strings:

```
>>> import astropy; from astropy import coordinates
>>> s = coordinates.SkyCoord(123.57475,42.58942,unit='degree')
>>> ss = scoord.to_string(style='hmsdms')
>>> print ss
08h14m17.94s +42d35m21.912s
>>> print ss.replace('d',':').replace('m',':').\
         replace('h',':').replace('s','')
08:14:17.94 +42:35:21.912
```

`aplpy` can do a wide range of astronomical plots, using the tasks `aplpyFITSFigure` to define a data object, which can then be plotted using `show_colorscale` and `show_contour`. Here are a few examples. First, a simple greyscale+contour plot of a FITS file:

```
>>> import aplpy
>>> gc=aplpy.FITSFigure('1327+5430.fits')
INFO: Setting slices=[0, 0] [aplpy.core]
WARNING: FITSFixedWarning: 'spcfix' made the change 'Changed SPECSYS to 'TOPOCENT''. [astropy
>>> gc.show_colorscale(cmap=matplotlib.cm.gray_r,vmin=0.0,vmax=0.005)
>>> gc.show_contour(levels=[-0.0005,0.0005,0.001,0.002,0.004,0.008])
>>> gc.add_label(0.2,0.05,'LOFAR map of an AGN',relative='axes')
>>> gc.save('1327.png')
```

produces



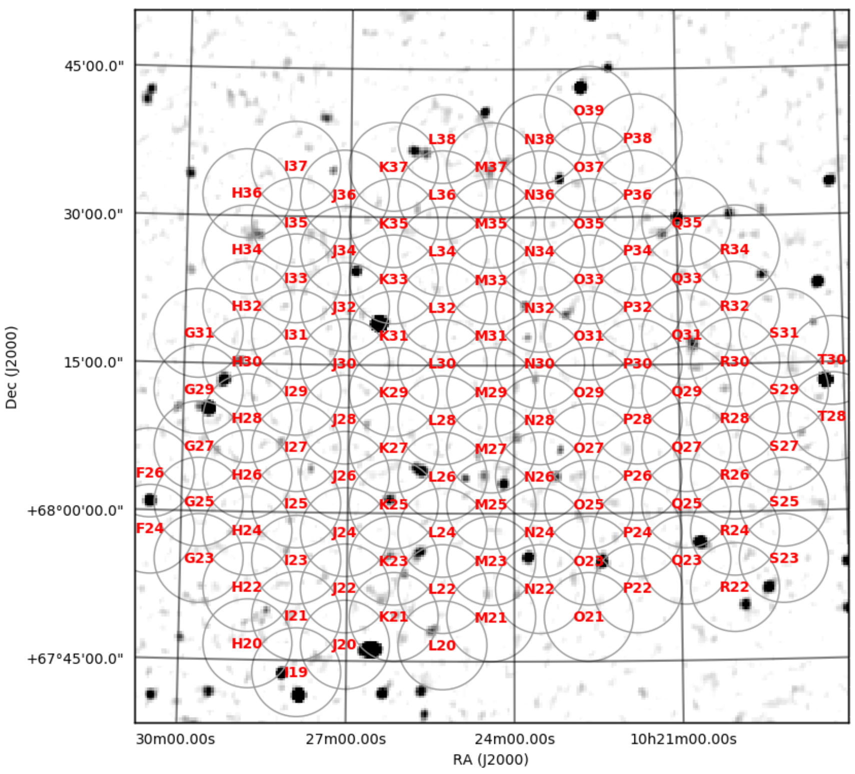
Here is a more extended example, which covers most things you are likely to need to do. A FITS file is read in, which consists of a VLA observation, and plotted in greyscale. Annotations are then written on the field from a list of pointings (in RA, dec), of which the first few lines is:

```
G29 157.432854 68.205000
G31 157.438392 68.300000
H32 157.218204 68.347500
I35 157.000854 68.490000
```

Here's the script:

```
import aplpy
pointings_str = np.loadtxt('superclass_pointings_112',dtype='S')
pointings = np.asarray(pointings_str[:,1:],dtype='f')
vla_radius = np.ones_like(pointings[:,0])*0.075
gc = aplpy.FITSFigure('nvss_superclass.fits')
gc.show_colorscale(cmap='gray_r',vmin=0.5e-3,vmax=5.e-3)
gc.add_grid()
gc.grid.set_color('black')
gc.show_circles(pointings[:,0],pointings[:,1],vla_radius,color='#999999')
lpointings = pointings_str[:,0]
for i in range(len(pointings)):
    s = lpointings[i]
    p = pointings[i]
    gc.add_label(p[0],p[1],s,size=10,color='red',weight='extra bold')
gc.savefig('output_fig.png')
```

and the result:



Finally, here is an example of how to create a grid of arbitrary size, using more advanced `pyfits` routines to set a user-defined size in RA and declination, and a user-defined image size and pixel size. A set of pointings of the same format as in the last example is read and plotted as circles. Finally, a set of text files are read in, each of whose rows represent vertices of a polygon which defines an area within the grid.

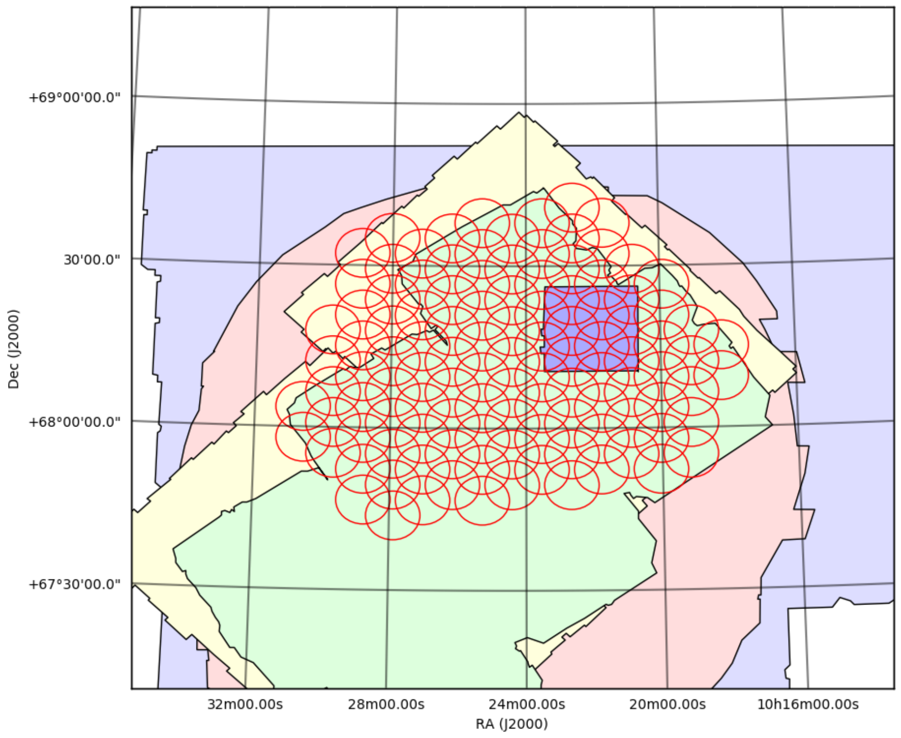
The script looks like:

```
import numpy as np, os
import astropy,aplpy
import astropy.io.fits as pyfits
plt.rcParams['image.origin'] = 'lower'

def mkfits (rasiz,decsiz,imsiz,pixsiz):
    hdu=pyfits.PrimaryHDU(np.zeros((imsiz,imsiz)))
    hdu.header.update({'CTYPE1':'RA---SIN'})
    hdu.header.update({'CRVAL1':rasiz})
    hdu.header.update({'CRPIX1':imsiz/2.})
    hdu.header.update({'CDELT1':-pixsiz})
    hdu.header.update({'CTYPE2':'DEC--SIN'})
    hdu.header.update({'CRVAL2':decsiz})
    hdu.header.update({'CRPIX2':imsiz/2.})
    hdu.header.update({'CDELT2':pixsiz})
    hdu.header.update({'EQUINOX':2000.0})
    hdu.header.update({'EPOCH':2000.0})
    os.system('rm temp.fits')
    hdu.writeto('temp.fits')


pointings_str = np.loadtxt('superclass_pointings_112',dtype='S')
pointings = np.asarray(pointings_str[:,1:],dtype='f')
lpointings = pointings_str[:,0]
lovell_radius = np.ones_like(pointings[:,0])*0.075
mkfits (156.10333,68.25,3600,2.1/3600)
gc = aplpy.FITSFigure('temp.fits')
gc.add_grid()
gc.grid.set_color('black')
gc.show_polygons([np.loadtxt('cov1.txt')],facecolor='#ddddff',edgecolor='black')
gc.show_polygons([np.loadtxt('cov2.txt')],facecolor='#ffdddd',edgecolor='black')
gc.show_polygons([np.loadtxt('cov3.txt')],facecolor='#ffffdd',edgecolor='black')
gc.show_polygons([np.loadtxt('cov4.txt')],facecolor='#ffffdd',edgecolor='black')
gc.show_polygons([np.loadtxt('cov5.txt')],facecolor='#ddffdd',edgecolor='black')
gc.show_polygons([np.loadtxt('cov6.txt')],facecolor='#aaaaff',edgecolor='black')
gc.show_circles(pointings[:,0],pointings[:,1],lovell_radius,color='red')
gc.savefig('figure_1.png')
```

and the final figure looks like


+69°00'00.0"
30'00.0"
+68°00'00.0"
+67°30'00.0"
Dec (J2000)
32m00.00s
28m00.00s
24m00.00s
20m00.00s
10h16m00.00s
RA (J2000)