159.302 Artificial Intelligence
Lab 1: NumPy and Matplotlib

Stephen Marsland

1 Introduction

The purpose of this lab is simply to get you started using Numerical Python (Numpy) and Matplotlib. There are two sets of tasks for you to do. The first set aims to get you started with using NumPy to do very simple computations, while the second will get you started loading, playing with, and plotting data. These two things will enable you to do the labs in the machine learning sections of the course much more easily.

There is a brief introduction to Python and NumPy at the end of these notes. It's actually the text of Chapter 16 of my book on Machine Learning (CRC Press, 2009) and I'm probably not meant to give it out, so please keep it private.

2 Your Tasks

2.1 Using NumPy and Matplotlib

- Make an array `a` of size 6×4 where every element is a 2.
- Make an array `b` of size 6×4 that has 3 on the leading diagonal and 1 everywhere else. (You can do this without loops.)
- Can you multiply these two matrices together? Why does `a * b` work, but not `dot(a,b)`?
- Compute `dot(a.transpose(),b)` and `dot(a,b.transpose())`. Why are the results different shapes?
- Write a function that prints some output on the screen and make sure you can run it in Eclipse or whichever environment you are using.
- Now write one that makes some random arrays and prints out their sums, the mean value, etc.
- Write a function that consists of a set of loops that run through an array and count the number of ones in it. Do the same thing using the `where()` function (use `info(where)` to find out how to use it).

2.2 Loading and Visualising Data

In general, the datasets that we use in this course will have been nicely prepared, so that they consist of a set of rows of data, of size n rows by m columns, often with ',' or a similar character between the values. This will mean that we have m datapoints, and that they are each of size n. Several of these datasets will come from the UCI Machine Learning Repository at `http://archive.ics.uci.edu/ml/`, and it is one of those that you will play with today. Go on to that website and look for the `iris` dataset, and download it. This is a very famous 'toy' dataset for testing learning algorithms out. It consists of a set of measurements from some iris flowers. The dataset was originally used by one of the fathers of statistical machine learning, Ronald Fisher.

This is a dataset for supervised classification learning. There are four datapoints in each row, followed by the name of the type of iris, and there are 150 examples of the three types of iris, 50 of each. Unfortunately, the name of the type of flower is actually given, and the algorithms that we use will generally require that everything is numeric, so the first thing to do is write some code that reads in the file, replaces the names of the flowers by class labels ('0', '1' and '2') will do very well, and then saves it as a new file. You can do this yourself, hopefully, but if not, here is a function that does the job:

```
def preprocessIris(infile,outfile):

    stext1 = 'Iris-setosa'
    stext2 = 'Iris-versicolor'
    stext3 = 'Iris-virginica'
    rtext1 = '0'
    rtext2 = '1'
    rtext3 = '2'

    fid = open(infile,"r")
    oid = open(outfile,"w")

    for s in fid:
        if s.find(stext1)>-1:
            oid.write(s.replace(stext1, rtext1))
        elif s.find(stext2)>-1:
            oid.write(s.replace(stext2, rtext2))
        elif s.find(stext3)>-1:
            oid.write(s.replace(stext3, rtext3))
    fid.close()
    oid.close()
```

The command to read in a matrix of numbers in NumPy is:

```
iris = np.loadtxt('iris_proc.data',delimiter=',')
```

assuming that you have imported NumPy as `np`. You will also need to have imported Matplotlib, for example as `pl`, to do be able to do the following tasks.

- Plot the first and second columns of the iris dataset. Can you see the three classes of data at all?
- Use the `where()` function in NumPy to plot the three different classes of iris as different shapes and colours.
- Repeat this for different pairs of columns and see which seem most useful.
- Subtract the mean of values in each column from the column entries, and divide by the standard deviation. Does this make things any clearer?

3 A Very Brief Introduction to Python and Numerical Python

The examples in this book are all written in Python, and the various graphs and results were also created in that language, using the code available via the book website. The purpose of this chapter is to give a brief introduction to using Python, and particularly NumPy, the numerical library for Python.

3.1 Installing Python and Other Packages

The Python language is very compact, but there are huge numbers of extensions and libraries available to make it more suited to a wide variety of tasks. Almost all of the examples in the book use NumPy, a set of numerical libraries, and the figures are produced using Matplotlib. Both of these packages have syntax that is similar to MATLAB. There are a few places where examples also use SciPy, the scientific programming libraries.

An Internet search will turn up working distributions as self-extracting zip files for the major operating systems, which will include the Python interpreter and all of the packages that are used in the book, amongst others. If you download individual packages, then they generally come with a setup script (`setup.py`) that can be run from a shell. Package webpages generally give instructions.

3.2 Getting Started

There are two ways that Python is commonly used. The first is as an interactive command environment, such as *iPython* or *IDLE*, which are commonly bundled with the Python interpreter. Starting Python with one of these (using Start/IPython in Windows, or by typing `python` at a command prompt in other operating systems) results in a script window with a command prompt (which will be shown as `>>>`). Unlike with C or Java, you can type commands at this prompt and the interpreter will run the commands and display the results, if any, on the screen. You can write functions in a text editor and run them from the command prompt by calling them by name. We'll see more about functions in Section 3.3.

As well as iPython there are several other Python IDEs and code editors available for various operating systems. The one that I use is the Java-based IDE Eclipse, which is freely available on the Internet. There is an extension of Eclipse for Python called PyDev that works very well. It includes all of the usual syntax highlighting and development help, you can run programs directly, and you can also set up an interactive Python environment so that you can test small pieces of code to see how they work.

The best way to get used to any language is, of course, to write programs in it. There is lots of code in the book and practical programming assignments along the way, but if you haven't used Python before then it will help if you get used to the language prior to working on the code examples in the book. Section 3.3 describes how to get started writing Python programs, but here we will begin by using the command line to see how things work. This can be in iPython or IDLE, by typing `python` at the command prompt, or within the console in the PyDev Eclipse extension.

Creating a variable in Python is easy: you give it a name and assign a value. While Python is strongly typed (so that variables that contain integers don't suddenly change to holding strings or floats without being told to) it performs all the declaration and creation of variables for you, unlike lower level languages like C. So typing `>>> a = 3` at the command prompt (note that the `>>>` is the command prompt, so you only actually type `a = 3`) defines `a` as an integer variable and gives it value 3. To see the effect of the integer typing, type `>>> a/2`, and you will see that the answer is `1`. What Python actually does is compute the answer in the most accurate of the types that are included in the calculation, but since `a` is an integer, and so is `2`, it returns the answer as an integer. You can see this using the `type()` function; `type(3/2)` returns `<type 'int'>`. So `>>> a/2.0` will work perfectly well, since the type of `2.0` is a float (`type(3/2.0) = <type 'float'>`. When writing floats you can abbreviate them to `2.` without the zero if you really want to save typing one character. To see the value of a variable you can just type its name at the command prompt, or use `>>> print a`, or whatever the name of the variable is.

You can perform all of the usual arithmetic operators on numbers, adding them up, etc. Raising numbers to a power is performed by `a**2` or `pow(a,2)`. In fact, you can use Python as a perfectly good calculator at the command line.

Just like in many other languages, comparison is performed using the double equals (`==`). It returns Boolean values `True` (1) and `False` (0) to tests like `>>> 3 < 4` and `>>> 3 == 4`. The other arithmetic comparisons are also available: `<, <=, >, >=` and these can be chained (so `3<x<6` performs the two tests and only returns `True` if both are true). The not-equal-to test is `!=` or `<>`, and there is another useful comparison: `is` checks if two variables point to the same object. This might not seem important, but Python works by reference, which means that the command `>>> a = b` does not put a copy of the value of `b` into `a`, but rather assigns to `a` a reference to the variable `b`. This can be a trap for the unwary, as will be discussed more shortly. The normal logical operators are slightly unusual in Python, with the normal logical operators using the words `and, or` and `not`; the symbols `&, |` perform bit-wise and/or. These bit-wise operators are actually quite useful, as we'll see later.

In addition to integer and floating point representations of numbers, Python also deals with strings, which are described by using single or double quotes (' or ") to surround them: `>>> b = 'hello'`. For strings, the `+` operator is overloaded (given a new meaning), which is concatenation: merging the strings. So `>>> 'a' + 'd'` returns the new string `'ad'`.

Having made the basic data types, Python then allows you to combine them into three different basic data structures:

Lists A list is a combination of basic data types, surrounded by square brackets. So `>>> mylist = [0, 3, 2, 'hi']` is a perfectly good list that contains integers and a string. This ability to store different types inside a list gives you a hint that Python handles lists differently to the way other languages handle arrays. This comes about because Python is inherently object-oriented, so that every variable that you make is simply an object, and so a list is just a collection of objects. This is why the type of the object does not matter. It also means that you can have lists of lists without a problem: `>>> newlist = [3, 2, [5, 4, 3], [2, 3, 2]]`.

Accessing particular elements of a list simply requires giving it an index. Like C, but unlike MATLAB, Python indices start at 0, so `>>> newlist[0]` returns the first element (3). You can also index from the end using a minus sign, so `>>> newlist[-1]` returns the last element, `>>> newlist[-2]` the last-but-one, etc. The length of a list is given by `len`, so `>>> len(newlist)` returns 4. Note that `>>> newlist[3]` returns the list in the 4th location of `newlist` (i.e., `[2, 3, 2]`). To access an element of that list you need an extra index: `>>> newlist[3][1]` returns 3.

A useful feature of Python is the slice operator. This is written as a colon (`:`) and enables you to access sections of a list easily, such as `>>> newlist[2:4]` which returns the elements of `newlist` in positions 2 and 3 (the arguments you use in a slice are inclusive at the start and exclusive at the end, so the second parameter is the first index that is excluded). In fact, the slice can take three operators, which are [start:stop:step], the third element saying what stepsize to use. So `>>> newlist[0:4:2]` returns the elements in locations 0 and 2, and you can use this to reverse the order of a list: `>>> newlist[::-1]`. This last example shows a couple of other refinements of the slice operator: if you don't put a value in for the first number (so it looks like `[:3]`) then the value is taken as 0, and if you don't put a value for the second operator (`[1:]`) then it is taken as running to the end of the list. These can be very useful, especially the second one, since it avoids having to calculate the length of the list every time you want to run through it. `>>> newlist[:]` returns the whole string.

This last use of the slice operator, returning the whole string, might seem useless. However, because Python is object-oriented, all variable names are simply references to objects. This means that copying a variable of type `list` isn't as obvious as it could be. Consider the following command: `>>> alist = mylist`. You might expect that this has made a copy of `mylist`, but it hasn't. To see this, use the following command `>>> alist[3] = 100` and then have a look at the contents of `mylist`. You will see that the 3rd element is now 100. So if you want to copy things you need to be careful. The slice operator lets you make actual copies using: `>>> alist = mylist[:]`. Unfortunately, there is an extra wrinkle in this if you have lists of lists. Remember that lists work as references to objects. We've just used the slice operator to return the values of the objects, but this only works for one level. In location 2 of `newlist` is another list, and the slice operator just copied the reference to that embedded list. To see this, perform `>>> blist = newlist[:]` and then `>>> blist[2][2] = 100` and have a look at `newlist` again. What we've done is called a shallow copy, to copy everything (known as a deep copy) requires a bit more effort. There is a `deepcopy` command, but to get to it we need to `import` the `copy` module using `>>> import copy` (we will see more about importing in Section 3.3.1). Now we can call `>>> clist = copy.deepcopy(newlist)` and we finally have a copy of a complete list.

There are a variety of functions that can be applied to lists, but there is another interesting feature of the fact that they are objects. The functions (methods) that can be used are part of the object class, so they modify the list itself and do not return a new list (this is known as working in place). To see this, make a new list `>>> list = [3, 2, 4, 1]` and suppose that you want to print out a list of the numbers sorted into order. There is a function `sort()` for this, but the obvious `>>> print list.sort()` produces the output `None`, meaning that no value was returned. However, the two commands `>>> list.sort()` followed by `>>> print list` do exactly what is required. So functions on lists modify the list, and any future operations will be applied to this modified list.

Some other functions that are available to operate on lists are:

`append(x)` adds `x` to the end of the list

`count(x)` counts how many times `x` appears in the list

`extend(L)` adds the elements in list `L` to the end of the original list

`index(x)` returns the index of the first element of the list to match `x`

`insert(i, x)` inserts element `x` at location `i` in the list, moving everything else along

`pop(i)` removes the item at index `i`

`remove(x)` deletes the first element that matches `x`

`reverse()` reverses the order of the list `sort()` we've already seen

You can compare lists using `>>> a==b`, which works elementwise through the list, comparing each element against the matching one in the second list, returning True if the test is true for each pair (and the two lists are the same length), and False otherwise.

Tuples A tuple is an immutable list, meaning that it is read-only and doesn't change. Tuples are defined using round brackets, e.g., `>>> mytuple = (0, 3, 2, 'h')`. It might seem odd to have them in the language, but they are useful if you want to create lists that cannot be modified, especially by mistake.

Dictionaries In the list that we saw above we indexed elements by their position within the list. In a dictionary you assign a key to each entry that you can use to access it. So suppose you want to make a list of the number of days in each month. You could use a dictionary (shown by the curly braces): `>>> months = {'Jan': 31, 'Feb': 28, 'Mar': 31}` and then you access elements of the dictionary using their key, so `>>> months['Jan']` returns 31. Giving an incorrect key results in an exception error.

The function `months.keys()` returns a list of all the keys in the dictionary, which is useful for looping over all elements in a dictionary. The `months.values()` function returns a list of values instead, while `months.items()` gives a list of tuples containing everything. There are lots of other things you can do with dictionaries.

There is one more data type that is built directly into Python, and that is the `file`. This makes reading from and writing to files very simple in Python: files are opened using `>>> input = open('filename')`, closed using `>>> input.close()` and reading and writing are performed using `readlines()` (and `read()`, and `writelines()` and `write()`). There are also `readline()` and `writeline()` functions, that read and write one line at a time.

3.2.1 Python for MATLAB and R users

With the NumPy package that we are using there are a great many similarities between MATLAB or R and Python. There are useful comparison websites for both MATLAB and R, but the main thing that you need to be aware of is that indexing starts at 0 instead of 1 and elements of arrays are accessed with square brackets instead of round ones. After that, while there are differences, the similarity between the three languages is striking.

3.3 Code Basics

Python has a fairly small set of commands and is designed to be fairly small and simple to use. In this section we'll go over the basic commands and other programming details. There are lots of good resources available for getting started with Python; a few books are listed at the end of the chapter, and an Internet search will provide plenty of other resources.

3.3.1 Writing and Importing Code

Python is a scripting language, meaning that everything can be run interactively from the command line. However, when writing any reasonable sized piece of code it is better to write it in a text editor or IDE and then run it. Eclipse or Idle and other GUIs provide their own code writing editors, but you can also use any text editor available on your machine. It is a good idea to use one that is consistent in its tabbing, since the white space indentation is how Python blocks code together.

The file can contain a script, which is simply a series of commands, or a set of functions and classes. In either case it should be saved with a `.py` extension, which Python will compile into a `.pyc` file when you first load it. Any set of commands or functions is known as a module in Python, and to load it you use the `import` command. The most basic form of the command is `import packagename`, but it is almost always better to use `import packagename as name`. If you import a script file then Python will run it immediately, but if it is a set of functions then it will not run anything.

To run a function you use `>>> name.functionname()`, where `name` is the name of the module and `functionname` the relevant function. Arguments can be passed as required in the brackets, but even if no arguments are passed then the brackets are still needed.

When developing code at a command line there is one slightly irritating feature of Python, which is that `import` only works once for a module. Once a module has been imported, if you change the code and want Python to work on the new version then you need to use `>>> reload(name)`. Using `import` will not give any error messages, but it will not work, either. This is not a problem when using the PyDev extensions for Eclipse.

Many modules contain several subsets, so when importing you may need to be more specific. You can import particular parts of a module in this way using `from x import y`, or to import everything use `from x import *`. Finally, you can specify the name that you want to import the module as, by using `from x import y as z`. You now have access to the functions within that module, which can be called by `z.name()`.

Program code also needs to import any modules that it uses, and these are usually declared at the top of the file (although they don't need to be, but can be added anywhere). There is one other thing that might be confusing, which is that Python uses the `pythonpath` variable to tell it where to look for code. Eclipse doesn't include other packages in your current project on the path, and so if you want it to find those packages, you have to add them to the path using the Properties menu item. If you are not using Eclipse, then you will need to add modules to the path. This can be done using something like:

```
import sys
sys.path.append('mypath')
```

3.3.2 Control Flow

The most obviously strange thing about Python for those who are used to other programming languages is that the indentation means something: white space is the way that blocks of code are shown. So if you have a loop or other construct then the equivalent of `begin ... end` or the braces { } in other languages is a colon (:) after the keyword and indented commands following on. This looks quite strange at first, but is actually quite nice once you get used to it. The other thing that is unusual is that you can have an (optional) `else` clause on loops. This clause runs when the loop terminates normally. If you break out of a loop using the `break` command then the `else` clause does not run.

The control structures that are available are `if`, `for`, and `while`. The `if` statement syntax is:

```
if statement:
    commands
elif:
    commands
else:
    commands
```

The most common loop is the `for` loop, which differs slightly from other languages in that it iterates over a list of values:

```
for var in set:
    commands
else:
    commands
```

There is one very useful command that goes with this `for` loop, which is the `range` command, which produces a list output. Its most basic form is simply `>>> range(4)`, which produces the list `[0, 1, 2, 3]`. However, it can also take 2 or 3 arguments, and works in the same way as in the slice command, but with commas between them instead of colons: `>>> range(start,stop,step)`. This can include going down instead of up a list, so `>>> range(5,-3,-2)` produces `[5, 3, 1, -1]` as output.

Finally, there is a `while` loop:

```
while condition:
    commands
else:
    commands
```

3.4 Functions

Functions are defined by:

```
def name(args):
    commands
    return value
```

The `return value` line is optional, but enables you to return values from the function (otherwise it returns `None`). You can list several things to return in the line with commas between them, and they will all be returned. Once you have defined a function you can call it from the command line and from within other functions. Python is case sensitive, so with both function names and variable names, `Name` is different to `name`.

As an example, here is a function that computes the hypotenuse of a triangle given the other two distances (`x` and `y`). Note the use of '#' to denote a comment:

```
def pythagorus(x,y):
    """ Computes the hypotenuse of two arguments"""
    h = pow(x**2+y**2,0.5)
    # pow(x,0.5) is the square root
    return h
```

Now calling `pythagorus(3,4)` gets the expected answer of `5.0`. You can also call the function with the parameters in any order provided that you specify which is which, so `pythagorus(y=4,x=3)` is perfectly valid. When you make functions you can allow for default values, which means that if fewer arguments are presented the default values are given. To do this, modify the function definition line: `def pythagorus(x=3,y=4):`

3.4.1 The `doc` String

The help facilities within Python are accessed by using `help()`. For help on a particular module, use `help('modulename')`. (So using `help(pythagorus)` in the previous example would return the description of the function that is given there). A useful resource for most code is the `doc` string, which is the first thing defined within the function, and is a text string enclosed in three sets of double quotes (`"""`). It is intended to act as the documentation for the function or class. It can be accessed using `>>> print functionname.__doc__`. The Python documentation generator `pydoc` uses these strings to automatically generate documentation for functions, in the same way that `javadoc` does.

3.4.2 `map` and `lambda`

Python has a special way of performing repeated function calls. If you want to apply the same function to every element of a list you don't need to loop over the elements of the list, but can instead use the `map` command, which looks like `map(function,list)`. This applies the function to every element of the list. There is one extra tweak, which is the fact that the function can be `anonymous` (created just for this job without needing a name) by using the `lambda` command, which looks like `lambda args :  command`. A `lambda` function can only execute one command, but it enables you to write very short code to do relatively complicated things. As an example, the following instruction takes a list and cubes each number in it and adds 7:

```
map(lambda x:pow(x,3)+7,list)
```

Another way that `lambda` can be used is in conjunction with the `filter` command. This returns elements of a list that evaluate to `True`, so:

```
filter(lambda x:x>=2,list)
```

returns those elements of the list that are greater than or equal to 2. NumPy provides simpler ways to do these things for arrays of numbers, as we shall see.

3.4.3 Exceptions

Like other modern languages, Python allows for the trapping of exceptions. This is done through the `try ... except ... else` and `try... finally` constructions. This example shows the use of the most common version. For more details, including the types of exceptions that are defined, see a Python programming book.

```
try:
    x/y
except ZeroDivisonError:
    print "Divisor must not be 0"
except TypeError:
    print "They must be numbers"
except:
    print "Something unspecified went wrong"
else:
    print "Everything worked"
```

3.4.4 Classes

For those that wish to use it in this way, Python is fully object-oriented, and classes are defined (with their constructor) by:

```
class myclass(superclass):

    def __init__(self,args):

    def functionname(self,args):
```

If a superclass is not specified then the class does not inherit from elsewhere. The `__init__(self,args)` function is the constructor for the class. There can also be a destructor `__del__(self)`, although they are rarely used. Accessing methods from the class uses the `classname.functionname()` syntax. The `self` argument can be ignored in all function calls, since Python fills it in for you, but it does need to be specified in the function definition. Many of the examples in the book are based on classes provided on the book website. You need to be aware that you have to create an instance of the class before you can run it. So to import and run the class you need to use:

```
import myclass
var = myclass.myclass()
var.function()
```

3.5 Using NumPy and Matplotlib

Most of the commands that are used in this book actually come from the NumPy and Matplotlib packages, rather than the basic Python language. More specialised commands are described thoughout the book in the places where they become relevant. There are lots of examples of performing tasks using the various functions within NumPy on its website. Getting information about functions within NumPy is easy, because there is a special command: `info()`; for example, to find out about the sum command, use `info(sum)`.

NumPy has a base collection of functions and then additional packages that have to be imported as well if you want to use them. To import the NumPy base library and get started you use:

```
>>> from numpy import *
```

3.5.1 Arrays

The basic data structure that is used for numerical work, and by far the most important one for the programming in this book, is the array. This is exactly like multi-dimensional arrays (or matrices) in any other language; it consists of one or more dimensions of numbers or chars. Unlike Python lists, the elements of the array all have the same type, which can be Boolean, integer, real, or complex numbers.

Arrays are made using a function call, and the values are passed in as a list, or set of lists for higher dimensions. Here are one-dimensional and two-dimensional arrays (which are effectively arrays of arrays) being made. Arrays can have as many dimensions as you like up to a language limit of 40 dimensions, which is more than enough for this book.

```
>>> myarray = array([4,3,2])
>>> mybigarray = array([[3, 2, 4], [3, 3, 2], [4, 5, 2]])
>>> print myarray
[4 3 2]
>>> print mybigarray
[[3 2 4]
 [3 3 2]
 [4 5 2]]
```

Making arrays like this is fine for small arrays where the numbers aren't regular, but there are several cases where this is not true. There are nice ways to make a set of the more interesting arrays, such as those shown next.

Array Creation Functions

`arange()` Produces an array containing the specified values, acting as an array version of `range()`. For example, `arange(5) = array([0, 1, 2, 3, 4])` and `arange(3,7,2) = array([3, 5])`.

`ones()` Produces an array containing all ones. For both `ones()` and `zeros()` you need two sets of brackets when making arrays of more than one dimension. `ones(3) = array([ 1., 1., 1.])` and `ones((3,4)) =`

```
array([[ 1.,  1.,  1., 1,]
[ 1.,  1.,  1., 1.]
[ 1.,  1.,  1., 1.]])
```

You can specify the type of arrays using `a = ones((3,4),dtype=float)`. This can be useful to ensure that you don't run into problems with integer casting, although NumPy is fairly good at casting things as floats.

`zeros()` Similar to `ones()`, except that all elements of the matrix are zero.

`eye()` Produces the identity matrix, i.e., the 2D matrix that is zero everywhere except down the leading diagonal, where it is one. Given one argument it produces the square identity: `eye(3) =`

```
[[ 1.  0.  0.]
 [ 0.  1.  0.]
 [ 0.  0.  1.]]
```

while with two arguments it fills spare rows or columns with zeros: `eye(3,4) =`

```
[[ 1.  0.  0.  0.]
 [ 0.  1.  0.  0.]
 [ 0.  0.  1.  0.]]
```

`linspace(start,stop,spacing)` Produces a matrix with linearly spaced elements. The nice thing is that you specify the number of elements, not the spacing. `linspace(3,7,3) = array([ 3., 5., 7.])`

`r_[]` **and** `c_[]` Perform row and column concatenation, including the use of the slice operator: `r_[1:4,0,4] = array([1, 2, 3, 0, 4])`. There is also a variation on `linspace()` using a `j` in the last entry: `r_[2,1:7:3j] = array([ 2.  , 1.  , 4.  , 7.])`. This is another nice feature of NumPy that can be used with `arange()` and `mgrid()` as well. The `j` on the last value specifies that you want 3 equally spaced points starting at 0 and running up to (and including) 7, and the function works out the locations of these points for you. The column version is similar.

The array `a` used in the next set of examples was made using `>>> a = arange(6).reshape(3,2)`, which produces:

```
array([[0, 1],
       [2, 3],
       [4, 5]])
```

Indexing elements of an array is performed using square brackets '[' and ']', remembering that indices start from 0. So `a[2,1]` returns `5` and `a[:,1]` returns `array([1, 3, 5])`. We can also get various pieces of information about an array and change it in a variety of different ways, as follows.

Getting information about arrays, changing their shape, copying them

`ndim(a)` Returns the number of dimensions (here `2`).

`size(a)` Returns the number of elements (here `6`).

`shape(a)` Returns the size of the array in each dimension (here `(3, 2)`). You can access the first element of the result using `shape(a)[0]`.

`reshape(a,(2,3))` Reshapes the array as specified. Note that the new dimensions are in brackets. One nice thing about `reshape()` is that you can use '-1' for 1 dimension within the reshape command to mean 'as many as is required.' This saves you doing the multiplication yourself. For this example, you could use `reshape(a,(2,-1))` or `reshape(a,(-1,2))`.

`ravel(a)` Makes the array one-dimensional (here `array([0, 1, 2, 3, 4, 5])`).

`transpose(a)` Compute the matrix transpose. For the example:

```
[[0 2 4]
 [1 3 5]]
```

`a[::-1]` Reverse the elements of each dimension.

`a.min(), a.max(a), a.sum(a)` Returns the smallest or largest element of the matrix, or the sum of the elements. Often used to sum the rows or columns using the `axis` option: `a.sum(axis=0)` for columns and `a.sum(axis=1)` for rows.

Note that `min`, `max`, and `sum` did not take the array as a parameter, but specified it first. This avoids confusion, since they are overloaded functions: `min(1,2)` returns `1`, the smallest of 1 and 2. All of the commands in this list can be written using the '.' notation if you wish, e.g., `a.reshape(2,-1)`.

Just like the rest of Python, NumPy generally deals with references to objects, rather than the objects themselves. So to make a copy of an array you need to use `c = a.copy()`.

There are two other very useful ways to make arrays in NumPy. They are `mgrid` and `ogrid`, the first of which is an equivalent of MATLAB's `meshgrid` command. Using `m = mgrid[0:3,0:2]` should give you a flavour of the command—it produces a set of all the (x, y) coordinates of points in one three-dimensional matrix, with the x-values in `m(0,:,:)` and the y-values in `m(1,:,:)`. You can also use the `j` indicator in the third parameter to specify the number of entries, and have NumPy compute the spacing of the elements for you. The `ogrid` command has identical syntax, but returns two one-dimensional arrays showing the values in x and y.

Once you have defined matrices, you need to be able to add and multiply them in different ways. As well as the array `a` used above, for the following set of examples two other arrays `b` and `c` are needed. They have to have sizes relating to array `a`. Array `b` is the same size as `a` and is made by `>>> b = arange(3,9).reshape(3,2)`, while `c` needs to have the same inner dimension; that is, if the size of `a` is `(x, 2)` then the size of `c` needs to be `(2, y)` where the values of `x` and `y` don't matter. For the examples `>>> c = transpose(b)`. Here are some of the operations you can perform on arrays and matrices:

Operations on arrays

`a+b` Matrix addition. Output for the example is:

```
array([[ 3,  5],
       [ 7,  9],
       [11, 13]])
```

`a*b` Element-wise multiplication. Output:

```
array([[ 0,  4],
       [10, 18],
       [28, 40]])
```

`dot(a,c)` Matrix multiplication. Output:

```
array([[ 4,  6,  8],
       [18, 28, 38],
       [32, 50, 68]])
```

`pow(a,2)` Compute exponentials of elements of matrix. Output:

```
array([[ 0,  1],
       [ 4,  9],
       [16, 25]])
```

`pow(2,a)` Compute number raised to matrix elements. Output:

```
array([[ 1,  2],
       [ 4,  8],
       [16, 32]])
```

Matrix subtraction and element-wise division are also defined, but the same trap that we saw earlier can occur with division, namely that `a/3` returns an integer not a float if `a` is an array of integers.

There is one more very useful command on arrays, which is the `where()` command. This has two forms: `x = where(a>2)` returns the indices where the logical expression is true in the variable `x`, while `x = where(a>2,0,1)` returns a matrix the same size as `a` that contains 0 in those places where the expression was true in `a` and 1 everywhere else. To chain these conditions together you have to use the bitwise logical operations, so that `indices = where((a[:,0]>3) | (a[:,1]<3))` returns a list of the indices where either of these statements is true.

3.5.2 Random Numbers

There are some good random number features within NumPy, which you need to import using `>>> from numpy.random import *`. To find out about the functions use `help(random)` once NumPy has been imported, but the more useful functions are:

`random.rand(matsize)` produces uniformly distributed random numbers between 0 and 1 in an array of size `matsize`

`random.randn(matsize)` produces zero mean, unit variance Gaussian random numbers

`random.normal(mean,stdev,matsize)` produces Gaussian random numbers with specifed mean and standard deviation

`random.uniform(low,high,matsize)` produces uniform random numbers between low and high

`random.randint(low,high,matsize)` produces random integer values between low and high

3.5.3 Linear Algebra

NumPy has a reasonable linear algebra package that performs standard linear algebra functions. The functions are available as `linalg.inv(a)`, etc., where `a` is an array and possible functions are (if you don't know what they all are, don't worry: they will be defined where they are used in the book):

`linalg.inv(a)` Compute the inverse of (square) array `a`

`linalg.pinv(a)` Compute the pseudo-inverse, which is defined even if `a` is not square

`linalg.det(a)` Compute the determinant of `a`

`linalg.eig(a)` Compute the eigenvalues and eigenvectors of `a`

3.5.4 Plotting

The plotting functions that we will be using are in the Matplotlib package. These are designed to look exactly like the MATLAB plotting functions. The entire set of functions, with examples, are given on the Matplotlib webpage, but the two most important ones that we will need are `plot` and `hist`. In order to use Matplotlib, you have to import it. For some reason it is called `pylab`, so the relevant command is `>>> import pylab as pl`, which gives you access to the plotting commands. Matplotlib also provides some MATLAB functionality, and sometimes this can overwrite NumPy functions. The best way to avoid this is to ensure that you always import PyLab before NumPy. When producing plots they sometimes do not appear. This is usually because you need to specify the command `>>> ion()` which turns interactive plotting on. If you are using Matplotlib within Eclipse it has a nasty habit of closing all of the display windows when the program finishes. To get around this, issue a `show()` command at the end of your function.

The basic plotting commands of Matplotlib are demonstrated here, for more advanced plotting facilities see the package webpage.

The following code (best typed into a file and executed as a script) computes a Gaussian function for values -2 to 2.5 in steps of 0.01 and plots it, then labels the axes and gives the figure a title:

```
from pylab import *
from numpy import *

gaussian = lambda x: exp(-(0.5-x)**2/1.5)
x = arange(-2,2.5,0.01)
y = gaussian(x)
plot(x,y)
xlabel('x values')
ylabel('exp(-(0.5-x)**2/1.5')
title('Gaussian Function')
show()
```

Further Reading

Python has become incredibly popular for both general computing and scientific computing. Because writing extension packages for Python is simple (it does not require any special programming commands: any Python module can be imported as a package, as can packages written in C), many people have done so, and made their code available on the Internet. Any search engine will find many of these, but a good place to start is the Python Cookbook website.

If you are looking for more complete introductions to Python, some of the following may be useful:

- M.L. Hetland. *Beginning Python: From Novice to Professional.* Apress Inc., Berkeley, CA, USA, 2nd edition, 2008.
- G. van Rossum and F.L. Drake Jr., editors. *An Introduction to Python.* Network Theory Ltd, Bristol, UK, 2006.
- W.J. Chun. *Core Python Programming.* Prentice-Hall, New Jersey, USA, 2006.
- B. Eckel. *Thinking in Python.* Mindview, La Mesa, CA, USA, 2001.
- T. Oliphant. Guide to NumPy, e-book, 2006. The official guide to NumPy by its creator.