Getting Acquainted with R

Starting the default R GUI

Starting R is simple. If you're using Windows simply navigate to the R subfolder from the Start Menu. On a Unix/Linux system invoke the program by typing `R`. On OS X start R by clicking the R icon from the Dock or in the Applications folder.

The OS X and Windows version of R provide a simple GUI interface that simplifies certain tasks. When you start up the R GUI you'll be presented with a single window, the R console. The rest of this document will assume you're using R under Windows or OS X.

R Studio

RStudio is a new IDE for R being developed under an open source model (see the RStudio GitHub Site). It provides a nicely designed graphical interface that is consistent across platforms. It can even run as a server, allowing you to access R via a web interface!

Check out the RStudio Docs for detailed info on configuring Rstudio. We'll go over some of RStudio's nice features in class.

Accessing the Help System on R

R comes with fairly extensive documentation and a simple help system. You can access HTML versions of R documentation under the Help menu. The HTML documentation also includes information on any packages you've installed. Take a few minutes to browse through the R HTML documentation.

The help system can be invoked from the console itself using the `help` function or the `?` operator.

```
> help(length)
> ?length
> ?log
```

What if you don't know the name of the function you want? You can use the `help.search()` function.

```
> help.search("log")
```

In this case `help.search(log)` returns all the functions with the string 'log' in them. For more on help.search type `?help.search`. Other useful help related functions include `apropos()` and `example()`.

Navigating Directories in R

When you start the R environment your 'working directory' (i.e. the directory on your computer's file system that R currently 'sees') defaults to a specific directory. On Windows this is usually the same directory that R is installed in, on OS X it is typically your home directory. Here are examples showing how you can get information about your working directory and change your working directory.

```
> getwd()
[1] "/Users/pmagwene"
> setwd("/Users")
> getwd()
[1] "/Users"
```

Note that on Windows you can change your working directory by using the `Change dir...` item under the `File` menu.

To get a list of the file in your current working directory use the `list.files()` function.

```
> list.files()
[1] "Shared" "pmagwene"
```

Using R as a Calculator

The simplest way to use R is as a fancy calculator.

```
> 3.14 * 2.5^2
[1] 19.625
> pi * 2.5^2 # R knows about some mathematical constants such as Pi
[1] 19.63495
> cos(pi/3)
[1] 0.5
> sin(pi/3)
[1] 0.8660254
> log(10)
[1] 2.302585
> log10(10) # log base 10
[1] 1
> log2(10) # log base 2
[1] 3.321928
> (10 + 2)/(4-5)
[1] -12
> (10 + 2)/4-5 # compare the answer to the above
[1] -2
```

Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing statements.

```
> sqrt(pi)
[1] 1.772454
> sqrt(-1)
[1] NaN
Warning message:
NaNs produced in: sqrt(-1)
> sqrt(-1+0i)
[1] 0+1i
```

What happened when you tried to calculate `sqrt(-1)`? –1 is treated as a real number and since square roots are undefined for the negative reals, R produced a warning message and returned a special value called `NaN` (Not a Number). Note that square roots of negative complex numbers are well defined so `sqrt(-1+0i)` works fine.

```
> 1/0
[1] Inf
```

Division by zero produces an object that represents infinite numbers.

Comparison Operators

You've already been introduced to the most commonly used arithmetic operators. Also useful are the comparison operators:

```
> 10 < 9  # less than
[1] FALSE
> 10 > 9  # greater than
[1] TRUE
> 10 <= (5 * 2) # less than or equal to
```

```
[1] TRUE
> 10 >= pi # greater than or equal to
[1] TRUE
> 10 == 10 # equals (note that '=' is an alternative assignment operator)
[1] TRUE
> 10 != 10 # does not equal
[1] FALSE
> 10 == (sqrt(10)^2) # Are you surprised by the result? See below..
[1] FALSE
> 4 == (sqrt(4)^2) # Even more confused?
[1] TRUE
```

Comparisons return boolean values. Be careful to distinguish between `==` (tests equality) and `=` (the alternative assignment operator equivalent to `<-`).

How about the last two statement comparing two values to the square of their square roots? Mathematically we know that both $(\sqrt{10})^2 = 10$ and $(\sqrt{4})^2 = 4$ are true statements. Why does R tell us the first statement is false? What we're running into here are the limits of computer precision. A computer can't represent $\sqrt{10}$ exactly, whereas $\sqrt{4}$ can be exactly represented. Precision in numerical computing is a complex subject and beyond the scope of this course. Later in the course we'll discuss some ways of implementing sanity checks to avoid situations like that illustrated above.

Working with Vectors in R

Vector Arithmetic and Comparison

Remember that in R arithmetic operations work on vectors as well as on single numbers (in fact single numbers *are* vectors).

```
> x <- c(2, 4, 6, 8, 10)
> x * 2
[1]  4  8 12 16 20
> x * pi
[1]  6.283185 12.566371 18.849556 25.132741 31.415927
> y <- c(0, 1, 3, 5, 9)
> x + y
[1]  2  5  9 13 19
> x * y
[1]  0  4 18 40 90
> x/y
[1]      Inf 4.000000 2.000000 1.600000 1.111111
> z <- c(1, 4, 7, 11)
> x + z
[1]  3  8 13 19 11
Warning message:
longer object length
        is not a multiple of shorter object length in: x + z
```

When vectors are not of the same length R 'recycles' the elements of the shorter vector to make the lengths conform. In the example above `z` was treated as if it was the vector `(1, 4, 7, 11, 1)`.

The comparison operators also work on vectors as shown below. Comparisons involving vectors return vectors of booleans.

```
> x > 5
[1] FALSE FALSE  TRUE  TRUE  TRUE
> x != 4
[1]  TRUE FALSE  TRUE  TRUE  TRUE
```

Indexing Vectors

```
> length(x)
[1] 5
> x[1]
[1] 2
> x[4]
[1] 8
> x[6]
[1] NA
> x[-1]
[1]  4  6  8 10
> x[c(3,5)]
[1]  6 10
```

For a vector of length n, we can access the elements by the indices $1 \ldots n$. Trying to access an element beyond these limits returns a special constant called `NA` (Not Available) that indicates missing or non-existent values.

Negative indices are used to exclude particular elements. `x[-1]` returns all elements of `x` except the first.

You can get multiple elements of a vector by indexing by another vector. In the example above `x[c(3,5)]` returns the third and fifth element of `x`.

Combining Indexing and Comparison

A very powerful feature of R is the ability to combine the comparison operators with indexing. This facilitates data filtering and subsetting. Some examples:

```
> x <- c(2, 4, 6, 8, 10)
> x[x > 5]
[1]  6  8 10
> x[x < 4 | x > 6]
[1]  2  8 10
```

In the first example we retrieved all the elements of `x` that are larger than 5 (read 'x where x is greater than 5').

In the second example we retrieved those elements of `x` that were smaller than four *or* greater than six. The symbol `|` is the 'logical or' operator. Other logical operators include `&` ('logical and' or 'intersection') and `!` (negation).

Combining indexing and comparison is a powerful concept and one you'll probably find useful for analyzing your own data.

Generating Regular Sequences

Creating sequences of numbers that are separated by a specified value or that follow a particular patterns turns out to be a common task in programming. R has some built-in operators and functions to simplify this task.

```
> s <- 1:10
> s
 [1]  1  2  3  4  5  6  7  8  9 10
> s <- 10:1
> s
 [1] 10  9  8  7  6  5  4  3  2  1
> s <- seq(0.5,1.5,by=0.1)
> s
 [1] 0.5 0.6 0.7 0.8 0.9 1.0 1.1 1.2 1.3 1.4 1.5
> s <- seq(0.5, 1.5, 0.33) # 'by' is the 3rd argument
                           # so you don't have to specify it
> s
[1] 0.50 0.83 1.16 1.49
```

`rep()` is another way to generate patterned data.

```
> rep(c("Male","Female"),3)
[1] "Male"   "Female" "Male"   "Female" "Male"   "Female"
> rep(c(T,T, F),2)
[1]  TRUE  TRUE FALSE  TRUE  TRUE FALSE
```

Some Useful Functions

You've already seem a number of functions (e.g. `sin()`, `log`, `length()`, etc). Functions are called by invoking the function name followed by parentheses containing zero or more *arguments* to the function. Arguments can include the data the function operates on as well as settings for function parameter values. We'll discuss function arguments in greater detail below.

Creating vectors

An important function that you've used extensively but we've glossed over is the `c()` function. This is short for 'concatenate' or 'combine' and as you've seen it combines it's arguments to form a vector.

For vectors of more than 10 or so elements it gets tiresome and error prone to create vectors using `c()`. For medium length vectors the `scan()` function is very useful.

```
> test.scores <- scan()
1: 98 92 78 65 52 59 75 77 84 31 83 72 59 69 71 66
17:
Read 16 items
> test.scores
 [1] 98 92 78 65 52 59 75 77 84 31 83 72 59 69 71 66
```

When you invoke `scan()` without any arguments the function will read in a list of values separated by white space (usually spaces or tabs). Values are read until `scan()` encounters a blank line or the end of file (EOF) signal (platform dependent).

Note that we created a variable with the name '`test.scores`'. If you have previous programming experience you might be surpised that this works. Unlike most languages, R (and S) allow you to use periods in variable names. Descriptive variable names generally improve readability but they can also become cumbersome (e.g. `my.long.and.obnoxious.variable.name`). As a general rule of thumb use short variable names when working at the interpreter and more descriptive variable names in functions.

Useful Numerical Functions

Let's introduce some additional numerical functions that are useful for operating on vectors.

```
> sum(test.scores)
[1] 1131
> min(test.scores)
[1] 31
> max(test.scores)
[1] 98
> range(test.scores) # min and max returned as a vector of length 2
[1] 31 98
> sorted.scores <- sort(test.scores)
> sorted.scores
 [1] 31 52 59 59 65 66 69 71 72 75 77 78 83 84 92 98
> w <- c(-1, 2, -3, 3)
> abs(w) # absolute value function
```

Function Arguments in R

Function arguments can specify the data that a function operates on or parameters that the function uses. Some arguments are required, while others are optional and are assigned default values if not specified.

Take for example the `log()` function. If you examine the help file for the `log()` function you'll see that it takes two arguments, refered to as '`x`' and '`base`'. The argument `x` represents the numeric vector you pass to the function and is a required argument (see what happens when you type `log()` without giving an argument). The argument `base` is optional. By default the value of `base` is $e = 2.71828\ldots$. Therefore by default the `log()` function returns natural logarithms. If you want logarithms to a different base you can change the `base` argument as in the following examples:

```
> log(2) # log of 2, base e
[1] 0.6931472
> log(2,2) # log of 2, base 2
[1] 1
> log(2, 4) # log of 2, base 4
[1] 0.5
```

Simple Input in R

The `c()` and `scan()` functions are fine for creating small to medium vectors at the interpreter, but eventually you'll want to start manipulating larger collections of data. There are a variety of functions in R for retrieving data from files.

The most convenient file format to work with are tab delimited text files. Text files have the advantage that they are human readable and are easily shared across different platforms. If you get in the habit of archiving data as text files you'll never find yourself in a situation where you're unable to retrieve important data because the binary data format has changed between versions of a program.

Using `scan()` to input data

`scan()` itself can be used to read data out of a file. Download the file `algae.txt` from the class website and try the following (after changing your working directory):

```
> algae <- scan('algae.txt')
Read 12 items
> algae
 [1] 0.530 0.183 0.603 0.994 0.708 0.006 0.867 0.059 0.349 0.699 0.983 0.100
```

One of the things to be aware of when using `scan()` is that if the data type contained in the file can not be coerced to doubles than you must specify the data type using the `what` argument. The `what` argument is also used to enable the use of `scan()` with columnar data. Download `algae2.txt` and try the following:

```
> algae.table <- scan('algae2.txt', what=list('',double(0)))
                        # note use of list argument to what
> algae.table
[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"

[[2]]
 [1] 0.530 0.183 0.603 0.994 0.708 0.006 0.867 0.059 0.349 0.699 0.983 0.100

> algae.table[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov" "Dec"
> algae.table[[2]]
 [1] 0.530 0.183 0.603 0.994 0.708 0.006 0.867 0.059 0.349 0.699 0.983 0.100
```

Use help to learn more about `scan()`.

Using `read.table()` to input data

`read.table()` (and it's derivates - see the help file) provides a more convenient interface for reading tabular data. Using the file `turtles.txt`:

```
> turtles <- read.table('turtles.txt', header=T)
> turtles
   sex length width height
1    f     98    81     38
2    f    103    84     38
3    f    103    86     42
  # output truncated
> names(turtles)
[1] "sex"    "length" "width"  "height"
> length(turtles)
[1] 4
> length(turtles$sex)
[1] 48
```

What kind of data structure is `turtles`? What happens when you call the `read.table()` function without specifying the argument `header=T`?

You'll be using the `read.table()`}function frequently. Spend some time reading the documentation and playing around with different argument values (for example, try and figure out how to specify different column names on input).

Note: `read.table()` is more convenient but `scan()` is more efficient for large files. See the R documentation for more info.

Basic Statistical Functions in R

There are a wealth of statistical functions built into R. Let's start to put these to use.

If you wanted to know the mean carapace width of turtles in your sample you could calculate this simply as follows:

```
> sum(turtles$width)/length(turtles$width)
[1] 95.4375
```

Of course R has a built in `mean()` function.

```
mean(turtles$width) [1] 95.4375
```

One of the advantages of the built in `mean()` function is that it knows how to operate on lists as well as vectors:

```
> mean(turtles)
      sex     length      width    height
       NA 124.68750  95.43750  46.33333
Warning message:
argument is not numeric or logical: returning NA in: mean.default(X[[1]], ...)
```

Can you figure out why the above produced a warning message? Let's take a look at some more standard statistical functions:

```
> min(turtles$width)
[1] 74
> max(turtles$width)
[1] 132
> range(turtles$width)
[1]  74 132
> median(turtles$width)
[1] 93
> summary(turtles$width)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
  74.00   86.00   93.00   95.44  102.00  132.00
> var(turtles$width) # variance
[1] 160.6769
> sd(turtles$width)  # standard deviation
[1] 12.67584
```

Simple Plots in R

One of the advantages of R is it's ability to produce a variety of plots and statistical graphics. Try out the following:

```
> hist(turtles$width)  # histogram plot
> hist(turtles$width,10) # produces a histogram with 10 bins
> hist(turtles$width,breaks=10, xlab="Carapace Width", probability=T)
>
> boxplot(turtles$width) # simple box plot
> boxplot(list(turtles$length, turtles$width, turtles$height),
+        names=c("Carapace\nLength","Carapace\nWidth","Carapace\nHeight"),
+        ylab="millimeters") # a fancy box plot showing multiple variables
> title("Turtle Shell Variables")
>
> plot(turtles$length, turtles$width)
> plot(turtles$length ~ turtles$width) # how does this differ from the plot above?
> plot(turtles$length, turtles$width, xlab="Carapace Length(mm)",
+      ylab="Carapace Width(mm)")
> title("Relationship Between\nLength and Width")
```

To get a sense of some of the graphical power of R try the `demo()` function:

```
> demo(graphics)
```

Getting Acquainted with Python

Starting the Python interpretter

The Python interpretter can be started in a number of ways. The simplest way is to open a terminal and type `python` . Go ahead and do this to make sure you have a working version of the default Python interpretter available on your system. From within the default interpretter you can type `Ctrl-d` (Unix,Mac) or `Ctrl-z` (Windows) to stop the interpretter and return to the command line.

For interactive use, the default interpretter isn't very feature rich, so the Python community has developed a number of GUIs or shell interfaces that provide more functionality. For this class we will be using a shell interface called IPython.

Recent versions of IPython (v0.11) provides both terminal and GUI-based shells. The EPD installer will place a number of shortcuts on your Start Menu or in Launchpad on OS X 10.7, including ones that read `PyLab` and

QtConsole. These are a terminal based and GUI based versions of IPython respectively, both of which automatically load key numerical and plotting libraries. Click on both of these icons to compare their interfaces.

To get the functionality of PyLab from the terminal, run the following command:

```
    $ ipython --pylab
```

To get the equivalent of QtConsole you can run ipython with the following arguments:

```
    $ ipython qtconsole --pylab
```

If you'd prefer a dark background, call QtConsole as so:

```
    $ ipython qtconsole --pylab --colors=linux
```

QtConsole is a recent addition to IPython and there may still be bugs to be sorted out, but it provides some very nice features like 'tooltips' (shows you useful information about functions as you type) and the ability to embed figures and plots directly into the console, and the ability to save a console session as a web page (with figures embedded!).

Quick IPython tips

IPython has a wealth of features, many of which are detailed in its documentation. There are also a number of videos available on the IPython page which demonstrate some of it's power. Here are a few key features to get you started and save you time:

- *Don't retype that long command!* — You can scroll back and forth through your previous inputs using the up and down arrow keys (or `Ctrl-p` and `Ctrl-n`); once you find what you were looking forward you can edit or change it. For even faster searching, start to type the beginning of the input and then hit the up arrow.
- *Navigate using standard Unix commands* — IPython lets you use standard Unix commands like `ls` and `cd` and `pwd` to navigate around your file system (even on Windows!)
- *Use <Tab> for command completion* — when your navigating paths or typing function names in you can hit the `<Tab>` key and IPython will show you matching functions or filenames (depending on context). For example, type `cd ./<Tab>` and IPython will show you all the files and subdirectories of your current working directory. Type a few of the letters of the names of one of the subdirectories and hit `<Tab>` again and IPython will complete the name if it finds a unique match. Tab completeion allows you to very quickly navigate around the file system or enter function names so get the hang of using it.

Accessing the Documentation in Python

Python comes with extensive HTML documentation and the Python interpreter has a help function that works similar to R's `help()`.

```
>>> help(sum)
Help on built-in function sum in module __builtin__:

sum(...)
    sum(sequence, start=0) -> value

    Returns the sum of a sequence of numbers (NOT strings) plus the value
    of parameter 'start'.  When the sequence is empty, returns start.
```

IPython also lets you use proceed the function name with a question mark, like in R:

```
In [1]: ?sum
Type:       builtin_function_or_method
Base Class: <type 'builtin_function_or_method'>
String Form:<built-in function sum>
Namespace:  Python builtin
Docstring:
sum(sequence[, start]) -> value

Returns the sum of a sequence of numbers (NOT strings) plus the value
of parameter 'start' (which defaults to 0).  When the sequence is
empty, returns start.
```

Using Python as a Calculator

As with R, the simplest way to use Python is as a fancy calculator. Let's explore some simple arithmetic operations:

```
>>> 2 + 10   # this is a comment
12
>>> 2 + 10.3
 12.300000000000001  # 0.3 can't be represented exactly in floating point precision
>>> 2 - 10
-8
>>> 1/2  # integer division
0
>>> 1/2.0  # floating point division
0.5
>>> 2 * 10.0
20.0
>>> 10**2  # raised to the power 2
100
>>> 10**0.5  # raised to a fractional power
3.1622776601683795
>>> (10+2)/(4-5)
-12
>>> (10+2)/4-5  # compare this answer to the one above
-2
```

In addition to integers and reals (represented as floating points numbers), Python knows about complex numbers:

```
>>> 1+2j    # Engineers often use 'j' to represent imaginary numbers
(1+2j)
>>> (1 + 2j) + (0 + 3j)
(1+5j)
```

Some things to remember:

- Integer and floating point division are not the same in Python. Generally you'll want to use floating point numbers.
- The exponentiation operator in Python is `**`
- Be aware that certain operators have precedence over others. For example multiplication and division have higher precedence than addition and subtraction. Use parentheses to disambiguate potentially confusing statements.
- The standard math functions like `cos()` and `log()` are not available to the Python interpeter by default. To use these functions you'll need to `import` the math library as shown below.

For example:

```
>>> 1/2
0
>>> 1/2.0
0.5
>>> cos(0.5)

Traceback (most recent call last):
  File "<pyshell#2>", line 1, in -toplevel-
    cos(0.5)
NameError: name 'cos' is not defined
>>> import math     # make the math module available
>>> math.cos(0.5) # access the cos() function in the math module
0.87758256189037276
>>> pi      # pi isn't defined in the default namespace

Traceback (most recent call last):
  File "<pyshell#5>", line 1, in -toplevel-
    pi
NameError: name 'pi' is not defined
>>> math.pi # however pi is defined in math
3.1415926535897931
>>> from math import * # bring everything in the math module into the current namespace
>>> pi
3.1415926535897931
>>> cos(pi)
-1.0
```

Comparison Operators in Python

The comparison operators in Python work the same was as they do in R (except they don't work on lists default). Repeat the comparison excercises given above.

More Data Types in Python

You've already seen the three basic numeric data types in Python - integers, floating point numbers, and complex numbers. There are two other basic data types - Booleans and strings.

Here's some examples of using the Boolean data type:

```
>>> x = True
>>> type(x)
<type 'bool'>
>>> y = False
>>> x == y
False
>>> if x is True:
...     print 'Oh yeah!'
...
Oh yeah!
>>> if y is True:
...     print 'You betcha!'
... else:
...     print 'Sorry, Charlie'
...
Sorry, Charlie
>>>
```

And some examples of using the string data type:

```
>>> s1 = 'It was the best of times'
>>> type(s1)
<type 'str'>
>>> s2 = 'it was the worst of times'
>>> s1 + s2
'It was the best of timesit was the worst of times'
>>> s1 + ', ' + s2
'It was the best of times, it was the worst of times'
>>> 'times' in s1
True
>>> s3 = "You can nest 'single quotes' in double quotes"
>>> s4 = 'or "double quotes" in single quotes'
>>> s5 = "but you can't nest "double quotes" in double quotes"
  File "<stdin>", line 1
    s5 = "but you can't nest "double quotes" in double quotes"
                                    ^
SyntaxError: invalid syntax
```

Note that you can use either single or double quotes to specify strings.

Simple data structures in Python: Lists

Lists are the simplest 'built-in' data structure in Python. List represent ordered collections of arbitrary objects.

```
>>> l = [2, 4, 6, 8, 'fred']
>>> l
[2, 4, 6, 8, 'fred']
>>> len(l)
5
```

Python lists are zero-indexed. This means you can access lists elements `0` to `len(x)-1`.

```
>>> l[0]
2
>>> l[3]
8
>>> l[5]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
```

You can use negative indexing to get elements from the end of the list:

```
>>> l[-1] # the last element
'fred'
>>> l[-2] # the 2nd to last element
8
>>> l[-3] # ... etc ...
6
```

Python lists support the notion of 'slices' - a continuous sublist of a larger list. The following code illustrates this concept:

```
>>> y = range(10)  # our first use of a function!
>>> y
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> y[2:8]
```

```
[2, 3, 4, 5, 6, 7]
>>> y[2:-1] # the slice
[2, 3, 4, 5, 6, 7, 8]
>>> y[-1:0] # how come this didn't work? see the next example...
[]
>>> y[-1:0:-2]  # slice from last to first, stepping backwards by 2
[9, 7, 5, 3, 1]
```

Using NumPy arrays

As mentioned during lecture, Python does not have a built-in data structure that behaves in quite the same way as do vectors in R. However, we can get very similar behavior using a library called `NumPy`.

NumPy does not come with the standard Python distribution, but it does come as an included package if you use the Enthought Python distribution. Alternately you can download NumPy from the SciPy project page at: http://numpy.scipy.org.

The NumPy package comes with documentation and a tutorial. You can access the documentation at `http://docs.scipy.org/doc/`

```
>>> from numpy import array # a third form of import
>>> x = array([2,4,6,8,10])
>>> -x
array([ -2,  -4,  -6,  -8, -10])
>>> x ** 2
array([  4,  16,  36,  64, 100])
>>> pi * x # assumes pi is in the current namespace
array([  6.28318531,  12.56637061,  18.84955592,  25.13274123,  31.41592654])
>>> y = array([0, 1, 3, 5, 9])
>>> x + y
array([ 2,  5,  9, 13, 19])
>>> x * y
array([ 0,  4, 18, 40, 90])
>>> z = array([1, 4, 7, 11])
>>> x+z
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: shape mismatch: objects cannot be broadcast to a single shape
```

The last example above shows that, unlike R, NumPy arrays in Python are not 'recycled' if lengths do not match.

Remember that lists and arrays in Python are zero-indexed rather than one-indexed.

```
>>> x
array([ 2,  4,  6,  8, 10])
>>> len(x)
5
>>> x[0]
2
>>> x[1]
4
>>> x[4]
10
>>> x[5]

Traceback (most recent call last):
  File "<pyshell#52>", line 1, in -toplevel-
    x[5]
IndexError: index out of bounds
```

Numpy arrays support the comparison operators and return arrays of booleans.

```
    >>> x < 5
    array([ True, True, False, False, False], dtype=bool)
    >>> x >= 6
    array([0, 0, 1, 1, 1])
```

NumPy also supports the combination of comparison and indexing that R vectors can do. There are also a variety of more complicated indexing functions available for NumPy; see the Indexing Routines in the Numpy docs.

```
>>> x[x < 5]
array([2, 4])
>>> x[x >= 6]
array([ 6,  8, 10])
>>> x[(x<4)+(x>6)]  # 'or'
array([ 2,  8, 10])
```

Note that Boolean addition is equivalent to 'or' and Boolean multiplication is equivalent to 'and'.

Most of the standard mathematical functions can be applied to NumPy arrays however you must use the functions defined in the `numpy` module.

```
>>> x
array([ 2,  4,  6,  8, 10])
>>> import math
>>> math.cos(x)

Traceback (most recent call last):
  File "<pyshell#67>", line 1, in -toplevel-
    math.cos(x)
TypeError: only length-1 arrays can be converted to Python scalars.
>>> import numpy
>>> numpy.cos(x)
array([-0.41614684, -0.65364362,  0.96017029, -0.14550003, -0.83907153])
```

Simple Plots in Python

The Matplotlib package is the de facto standard for producing publication quality scientific graphics in Python. Matplotlib is included with the EPD and was automatically pulled into the interpretter namespace if you're using the IPython PyLab or QtConsole configurations.

Here are some simple plotting examples using matplotlib:

```
    # only necessary if not using pylab
    >>> from pylab import *

    >>> import numpy as np # use a shorter alias

    # load the turtle data using the numpy.loadtxt function (see ?np.loadtxt)
    # we skipped the first row (header) and the first column (info about sex)
    # we'll see how to deal with more sophisticated data import next week

    >>> turt = np.loadtxt('turtles.txt', skiprows=1, usecols=(1,2,3))
    >>> turt.shape
    (48, 3)

    # draw bivariate scatter plot
    >>> scatter(turt[:,0], turt[:,1])
    # give the axes some labels and a title for the plot
    >>> xlabel('Length')
    >>> ylabel('Width')
    >>> title('Turtle morphometry')
```

Introduction to Literate Programming

Sweave for R

Sweave documents weave together documentation/discussion and code into a single document. The pieces of code and documentation are referred to as 'chunks'. R comes with a set of tools that allow you to extract just the code, or to turn the entire document into a nicely formatted report.

Here's a simple Sweave document to get you started:

```
\documentclass{article}
\begin{document}

This is a very simple Sweave file. It includes only a single
code chunk.

<<>>=
z <- rnorm(30, mean=0, sd=1)
summary(z)
@

That code chunk generated a random sample of 30 observations
drawn from a normal distribution with mean zero and standard
deviation one.

\end{document}
```

Type those lines into a text editor and save the document with the name `sweave1.Rnw`.

Let's break down the various pieces of the document. The first two lines and the last line represent LaTeX commands.

```
\documentclass{article}
\begin{document}
    ....
\end{document}
```

For simple Sweave documents that's all the LaTeX you need to learn. However, learning a little bit more about the document preparation system gives you the option of producing very nicely formatted output as we'll see in a little bit.

The R code is preceeded by the text `<<>>=`. This tells Sweave that you're starting a code chunk. The `@` symbol after the code chunk tells Sweave that you're going back to writing documentation chunks.

If you already have a working installation of LaTeX on your computer you can now compile this into a nicely formatted document from the R interpretter. Change the R working directory so that it's in the same directory where you saved the `sweave1.Rnw` file. The execute the following commands:

```
> library(tools)  # makes the texi2dvi function available
> Sweave('sweave1.Rnw') # compiles our Sweave document into 'sweave1.tex'
> texi2dvi("sweave1.tex", pdf = TRUE)
```

These commands will produce two new documents – `sweave1.tex` and a PDF document, `sweave1.pdf`. You can open the `sweave1.tex` file in any text editor and you'll see that it's just a slightly modified version of the `sweave1.Rnw` file you created. The PDF file contains the nicely formatted report.

If you got an error message the most likely reason is that R can't find the path to your LaTeX executable. To check this try:

```
> texi2dvi('sweave1.tex', pdf=TRUE, quiet=FALSE)
```

If that's the case, you can fix that by typing the following into the R command line (on OS X):

```
> Sys.setenv("PATH" = paste(Sys.getenv("PATH"),"/usr/texbin",sep=":"))
```

Then try executing the `texi2dvi` command again. If that solved your problem you can make this permanent by adding that line to your .Rprofile (located in `/Users/yourname` on OS X; if the file doesn't already exist go ahead and create it). If that doesn't work please see me for troubleshooting help.

RStudio makes Sweaving easy!

RStudio hides some of the complexity of Sweaving documents. Simply create or open your Sweave document (use the .Rnw extension) in RStudio and then hit the `Compile PDF` button. If your LaTeX setup is working, and the document and code are valid, Rstudio will compile everything behind the scenes and pop up a nice PDF.

A fancier Sweave document

Let's get a little bit fancier and show how we can create graphics and use some LaTeX formatting features to produce a nicer document.

```
\documentclass[letterpaper]{article}
\usepackage[margin=0.75in]{geometry}

\title{My Second Sweave Report}
\author{John Q. Public}

\begin{document}
\maketitle
This is a still a simple Sweave file. However,
now it includes several code chunks and several
\LaTeX\ specific commands.

\section{Sampling from the random normal distribution}

<<>>=
z <- rnorm(30, mean=0, sd=1)
summary(z)
@

That code chunk generated a random sample of 30
observations drawn from a normal distribution with mean
zero ($\mu = 0$) and standard deviation one ($\sigma = 1$).


\section{Generating figures}

We can also automatically imbed graphics in our
report. For example, the following will generate
a histogram.
```

```
% this tells Sweave to set the graphics
% to be half the width of the text
\setkeys{Gin}{width=0.5\textwidth}
<<fig=TRUE>>=
hist(z)
@

\end{document}
```

Notice how we put an argument, `fig=TRUE` within the second code chunk delimiter. This will tell Sweave to automatically imbed a figure with the histogram graphic we created into our report. Save this as `sweave2.Rnw` and repeat the above steps to compile it into a PDF report.

For a full overview of Sweave's capabilities see the documentation for Sweave availabe at `http://www.stat.uni-muenchen.de/~leisch/Sweave/`.

Pweave for literate programming in Python

Pweave uses almost exactly the same syntax as Sweave to delimit code and document chunks. Here's a simple Pweave document.

```
\documentclass[letterpaper]{article}
\usepackage[margin=0.75in]{geometry}
\usepackage{graphicx} % unlike Sweave, Pweave doesn't
                      % pull this in automatically

\title{My First Pweave Report}
\author{John Q. Public}

\begin{document}
\maketitle
This is a still a simple Pweave file. As in our Sweave example,
there are several code chunks and we've included a figure.

\section{Sampling from the random normal distribution}

<<>>=
from numpy import random
z = random.normal(loc=0, scale=1, size=30)
@

That code chunk generated a random sample of 30
observations drawn from a normal distribution with mean
zero ($\mu = 0$) and standard deviation one ($\sigma = 1$).


\section{Generating figures}

We can also automatically imbed graphics in our
report. For example, the following will generate
a histogram.

% this tells Sweave to set the graphics
% to be half the width of the text
\setkeys{Gin}{width=0.5\textwidth}
<<fig=True>>=
import pylab
pylab.hist(z)
```

```
@

\end{document}
```

Note that the second code chunk, we wrote `fig=True` in the Pweave document, whereas we wrote `fig=TRUE` for the Sweave document. This minor difference reflects the different syntax for boolean values in R and Python.

Save that code in a text file called `pweave1.Pnw` and from the bash shell (*not* in the Python interpretter) type the following command:

```
Pweave -f "tex" pweave1.Pnw
```

The option `-f "tex"` tells Pweave to output a file (Note: from the Windows command prompt you must use double quotes around "tex", on Unix-based systems either single our double quotes work fine). Assuming you got no error messages, you can then compile this to PDF using the following command:

```
pdflatex pweave1.tex
```