Bio 723

Scientific Computing for Biologists

Paul M. Magwene

Fall 2014

Contents

1 Getting your feet wet with Python

1.0.1 Starting the Python interpretter

The Python interpretter can be started in a number of ways. The simplest way is to open a shell (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, OS X) 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 an interface called IPython. Recent versions of IPython provides terminal and GUI-based shells as well as a web-browser interface called the IPython Notebook.

1.0.2 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:

```
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.
```

1.0.3 Using Python as a Calculator

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

Some things to remember about mathematical operations in Python:

- 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) # 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
```

1.0.4 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
>>>
```

Comparison Operators in Python

The standard comparison operators for numerical values are available in Python. Comparison operators return Boolean values:

```
>>> 4 < 5 # less than
True
>>> 4 > 3.99 # greater than
True
>>> 4 <= 4.0 # less than or equal to
True
>>> 4 >= 5 # greater than or equal to
False
>>> 4 == 4.0 # test equality
True
```

Strings

The string data type is used to represent ordered sets of characters, such as text:

```
>>> s1 = 'It was the best of times'
>>> type(s1)
<type 'str'>
>>> s2 = 'it was the worst of times'
>>> s1 + s2  # string concatenation
'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.

1.0.5 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?
[]
# slice from last to first, stepping backwards by 2
>>> y[-1:0:-2]
[9, 7, 5, 3, 1]
```

1.0.6 Using NumPy arrays

The Python user community has developed a module called NumPy for efficient numerical computing in Python. The basic data structure in the NumPy library is an array. Arrays can be used to reprsent both vectors and matrices, common mathematical structures we'll use throughout the course. Below are some examples illustrating the use of NumPy arrays:

```
>>> 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 error above illustrates that basic arithmetic operations involving pairs of NumPy arrays in Python require that the arrays be of equal length.

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

```
>>> x[x < 5]  # get the elements of x where that are less than 5
array([2, 4])
```

```
>>> x[x >= 6]  # elements of x that are greater than or equal to 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’. There are also a variety of more complicated indexing functions available for NumPy; see the Indexing Routines in the Numpy docs.

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])
```

2 Bivariate Data

2.1 Plotting Bivariate Data in R

Let's use a dataset called `iris` (included in the standard R distribution) to explore bivariate relationships between variables. This data set was made famous by R. A. Fisher who used it to illustrate many of the fundamental statistical methods he developed. The data set consists of four morphometric measurements on specimens of three different iris species. Use the R help to read about the iris data set (`?iris`). We'll be using this data set repeatedly in future weeks so familiarize yourself with it.

```
> ?iris
> names(iris)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
[5] "Species"
> unique(iris$Species)
[1] setosa     versicolor virginica
Levels: setosa versicolor virginica
> dim(iris)
[1] 150   5
```

2.1.1 Bivariate scatter plots

We'll start with the conventional 'variable space' representation of bivariate relationships – the scatter plot.

```
> plot(iris$Sepal.Length, iris$Sepal.Width)
```

This plots Sepal Length on the x-axis and Petal Length on the y-axis. Here's an alternate way to generate the same plot:

```
> plot(Petal.Length ~ Sepal.Length, data = iris)
```

Did you notice what is different between the two versions above? In the second version, you can think of the tilde ('~') as short-hand for 'function of'. So the plotting call above can be translated roughly as "Plot Petal.Length as a function of Sepal.Length, where these variables can be found in the iris data set".

From these plot it is immediately obvious that these two variables are positively associated (i.e. when one increases the other tends to increase). You will also notice there seem to be distinct clusters of points in the plot. Recall that the iris data set consists of three different species. Let's regenerate the plot, this time coloring the points according to the species names. First, let's note that the Species column is a categorical variable, which in R we refer to as a 'factor'.

```
> iris$Species
  [1] setosa     setosa     setosa     setosa ...
 [51] versicolor versicolor versicolor versicolor ...
[101] virginica  virginica  virginica  virginica  ...
  ....
Levels: setosa versicolor virginica
> is.factor(iris$Species)
[1] TRUE
> levels(iris$Species)
[1] "setosa"     "versicolor" "virginica"
> nlevels(iris$Species)
[1] 3
> typeof(iris$Species)
[1] "integer"
```

The `is.factor()` function tests whether a vector is a factor, the `levels()` function returns the categorical labels associated with the factor, and `nlevels()` gives the total number of levels. Factor levels are represented internally as integers, as the `typeof()` function call illustrates. You can use the function `unclass()` to show the corresponding integer representations for a vector of factors:

```
> unclass(iris$Species)
  [1] 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 ...
 [59] 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 ...
[117] 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 ...
attr(,"levels")
[1] "setosa"     "versicolor" "virginica"
```

As you can see, the 'setosa' specimens have the value 1, 'versicolor' have the value 2, and 'virginica' the value 3.

Because of the mapping between factor levels and integers, we can use a variable of factors as indices into another vector, effectively creating a mapping between the factor levels, and the elements of the vector that is being indexed. This is shown below:

```
> clrs <- c('red','green','blue')
> clrs[iris$Species]
  [1] "red"   "red"   "red"   "red"   "red"   "red"   "red" ...
 [57] "green" "green" "green" "green" "green" "green" "green" ...
 [99] "green" "green" "blue"  "blue"  "blue"  "blue"  "blue" ...
```

With that mapping in mind, let's reconstruct our scatter plot:

```
> plot(Petal.Length ~ Sepal.Length, data = iris, col = clrs[iris$Species],
    main="Petal Length vs. Sepal Length")
> legend( "topleft", pch = 1, col = clrs, legend = levels(iris$Species ))
```

In addition to plotting and coloring the bivariate scatter, we added a title to the plot using the `main` argument and created a legend, using the `legend()` function. Your output should look like Figure 2.1.

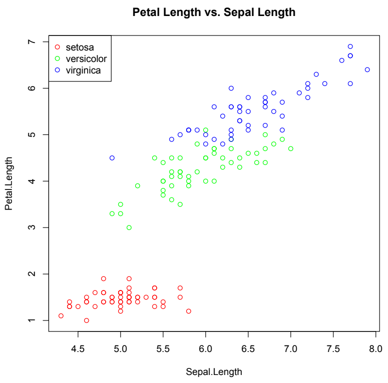

Figure 2.1: Scatter plot created from the iris data set using the `plot` function.

2.2 Introducing ggplot2

Pretty much any statistical plot can be thought of as a mapping between data and one or more visual representations. For example, in a bivariate scatter plot we map two ordered sets of numbers (the variables of interest) to points in the Cartesian plane (x,y-coordinates). In our example above, we further embellished our plot with another mapping in which we mapped the Species labels to different colors.

This notion of representing plots in terms of their mappings is a powerful idea which is central to an approach for plotting that is represented in the R package `ggplot2`.

2.2.1 Installing ggplot2

Like all R packages, `ggplot2` can be installed either from the command line or via the GUI. Here's a reminder of how to do so from the command line:

```
> install.packages("ggplot2", dependencies=T)
```

2.2.2 Aesthetic and Geometric mappings in ggplot2

ggplot2 considers two types of mappings from data to visual representations: 1) 'aesthetic mappings', which determine the way that data are represented in a plot (e.g. symbols, colors) and 2) 'geometry' or 'geom' mappings which determine the type of geometric representation that a plot uses.

The primary plotting function in `ggplot2` is `ggplot()`. The first argument to `ggplot` is always a data frame. The data frame is the one that ggplot will use to look for all the mappings that you define in the subsequent pieces of the plot. The nice thing about this is that there is no need to use the dollar sign notation. As you've seen, you can get similar behavior in base plots by specifying the 'data' argument.

The second argument to `ggplot()` is always a function called `aes()`. `aes()` takes named arguments. Each argument name is the 'aesthetic' that you want mapped to a particular variable (column) in the data.

The final piece of information that we need to draw our plot is the 'geom'. All geoms are encoded as R functions. The syntax used to add them to a plot is simply a '+' sign. There are many different ggplot geoms for different plot types. We'll explore a few of the built-in geoms in this chapter; additional geoms will come up in later weeks.

2.2.3 Scatter plots using ggplot2

Let's recreate our iris scatter plot using the function `ggplot` from the ggplot2 library:

```
> library(ggplot2)
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                   col = Species)) + geom_point()
```

Following the requirement outline above, `iris` is our data frame, the call to `aes` set's up our aesthetic mapping, and we're specifying the use of the point geom (`geom_point()`) to map the x- and y-values in the aesthetic mapping to points in the Cartesian plane. In the function call above, we told `ggplot` that we wanted the sepal length on the x axis, the petal length on the y axis, and the colors to be encoded by the species. However, we could choose any number of other aesthetic mappings. For example, could use shape instead of color to represent the Species labels:

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                   shape = Species)) + geom_point()
```

or alternately, size:

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                   size = Species)) + geom_point()
```

We can even combine multiple aesthetics in a single plot:

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length,
                   col = Species, shape = Species)) + geom_point()
```

The resulting plot is shown in Figure 2.2.

There's a number of advantages to using `ggplot` rather than trying to replicate this plot with base graphics functions in R:

1. The legend is automatically drawn for you.

2. The code is very easy to change. Rather than having to figure out how to manually map a point size onto a variable using some difficult R code, it's just as simple as saying to set the 'size' equal to a 'variable'.

3. It's easy to swap around variables from one aesthetic mapping to another.

Having a good understanding of both the base plotting functions and a powerful package like `ggplot2` allows you maximum flexibility in terms of the statistical graphics you are able to produce.

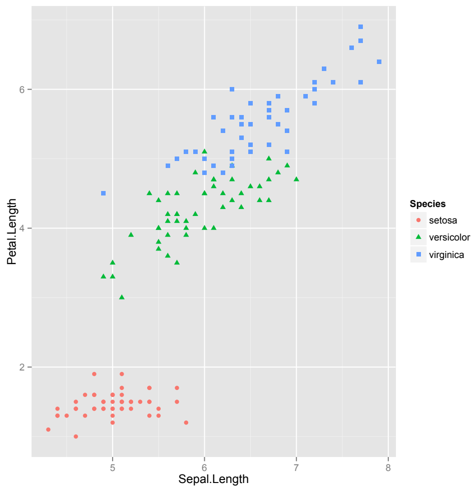

Figure 2.2: Scatter plot created from the iris data set using the `ggplot` function.

2.2.4 Some additional ggplot geoms

So far we've only looked at a single geom (`geom_point()`). Let's revisiting some of the univariate plots from last week using ggplot.

Boxplots `geom_boxplot()` constructs boxplots in ggplot.

```
> ggplot(iris, aes(x = Species, y = Sepal.Length, col=Species)) +
      geom_boxplot()
```

Histograms `geom_histogram()` is used to construct histogram plots in ggplot.

```
> ggplot(iris, aes(x = Sepal.Length)) + geom_histogram()
```

Here we let `ggplot` pick the default bin widths. Below we show how to change the bin width:

```
> ggplot(iris, aes(x = Sepal.Length)) + geom_histogram(binwidth=0.25)
```

If we want to color histogram by species identity you need to set the `position = 'identity'` in the call to `geom_histogram`:

```
> ggplot(iris, aes(x = Sepal.Length, fill=Species)) +
        geom_histogram(binwidth=0.25, position='identity',alpha=0.65)
```

The above code also set the transparency of the bar fills using the `alpha` argument. As an alternative to overlaying the histogram bins for each species, you can show the bins side-by-side using the argument `position = 'dodge'`.

```
> ggplot(iris, aes(x = Sepal.Length, fill=Species)) +
        geom_histogram(binwidth=0.25, position='dodge')
```

Density plots geom_density() creates density plots in ggplot.

```
> ggplot(iris, aes(x = Sepal.Length, fill=Species)) +
        geom_density(alpha=0.65)
```

There's also a 2D version of the density plot, created using geom_density2d(). This can be usefully combined with geom_points() to create a bivariate scatter plot with density contours.

```
> ggplot(iris, aes(x = Sepal.Length, y = Petal.Length, col = Species)) +
    geom_point() + geom_density2d(alpha=0.25)
```

Scatter plots with marginal density plots The file scatterWithMargins.R from the course wiki contains a function that uses multiple calls to ggplot() to combine two marginal density plots with a scatter plot. To use this function you'll need to install a package called "gridExtra":

```
> install.packages("gridExtra", dependencies=T)
```

Then import the new function from scatterWithMargins.R and use it as so:

```
> source('scatterWithMargins.R')
> scatterWithMargins(iris, "Sepal.Length", "Petal.Length", "Species")
```

This produces the plot shown in Figure 2.3.

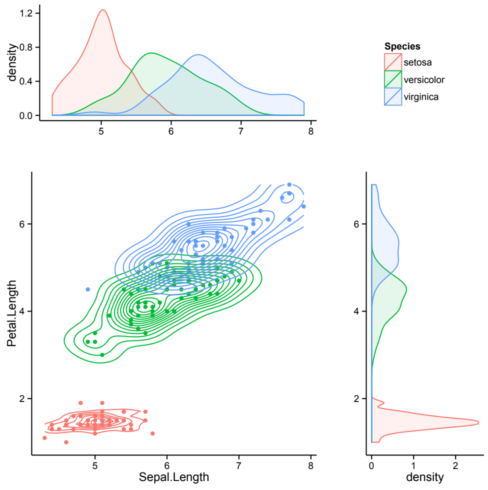

Figure 2.3: Figure produced by the scatterWithMargins function from the course wiki.

2.3 Vector Mathematics in R

As you saw last week R vectors support basic arithmetic operations that correspond to the same operations on geometric vectors. For example:

```
> x <- 1:15
> y <- 10:24
> x
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
> y
 [1] 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24

> x + y             # vector addition
 [1] 11 13 15 17 19 21 23 25 27 29 31 33 35 37 39
> x - y             # vector subtraction
 [1] -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9 -9
> x * 3             # multiplication by a scalar
 [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45
```

R also has an operator for the dot product, denoted `%*%`. This operator also designates matrix multiplication, which we will discuss next week. By default this operator returns an object of the R matrix class. If you want a scalar (or the R equivalent of a scalar, i.e. a vector of length 1) you need to use the `drop()` function.

```
> z <- x %*% x
> class(z)      # note use of class() function
[1] "matrix"
> z
     [,1]
[1,] 1240
> drop(z)
[1] 1240
```

In lecture we saw that many useful geometric properties of vectors could be expressed in the form of dot products. Let's start with some two-dimensional vectors where the geometry is easy to visualize:

```
> a <- c(1, 0) # the point (1,0)
> b <- c(0, 1) # the point (0,1)
```

Now let's draw our vectors:

```
# create empty plot w/specified x- and y- limits
# the 'asp=1' argument maintains the scaling of the x- and y-axes
# so that units are equivalent for both axes (i.e. squares remain squares)
> plot(c(-2,2),c(-1,2),type='n', asp=1)

# draw an arrow from origin (0,0) to x,y coordinates of vector "a"
# the length argument changes the size of the arrowhead
# use the R help to read more about the arrows function
> arrows(0, 0, a[1], a[2], length=0.1)
```

```
# and now for the vector "b"
> arrows(0, 0, b[1], b[2], length=0.1)
```

You should now have a figure that looks like the one below: Let's see what the dot

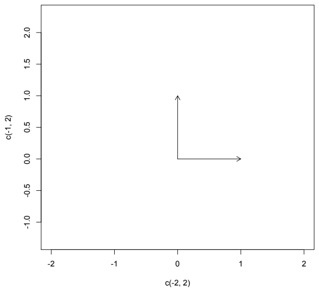

Figure 2.4: A simple vector figure.

product can tell us about these vectors. First recall that we can calculate the length of a vector as the square-root of the dot product of the vector with itself ($|\vec{a}|^2 = \vec{a} \cdot \vec{a}$)

```
> len.a <- drop(sqrt(a %*% a))
> len.a
[1] 1
> len.b <- drop(sqrt(b %*% b))
```

How about the angle between a and b?

```
> dot.ab <- a %*% b
> dot.ab
     [,1]
[1,]    0
> cos.ab <- (a %*% b)/(len.a * len.b)
> cos.ab
     [,1]
[1,]    0
```

A key point to remember dot product of two vectors is zero if, and only if, they are orthogonal to each other (regardless of their dimension).

2.4 Writing Functions in R

So far we've been mostly using R's built in functions. However the power of a true programming language is the ability to write your own functions.

The general form of an R function is as follows:

```
funcname <- function(arg1, arg2) {
 # one or more expressions
 # last expression is the object returned
 # or you can explicitly return an object
}
```

To make this concrete, here's an example where we define a function in the interpreter and then put it to use:

```
> my.dot <- function(x,y){
+ # don't type the '+' symbols, these show continuation lines
+   return(sum(x*y))
+ }

> a <- 1:5
> b <- 6:10
> a
[1] 1 2 3 4 5
> b
[1]  6  7  8  9 10
> my.dot(a,b)
[1] 130
> my.dot
function(x,y){
  return(sum(x*y))
}
```

If you type a function name without parentheses R shows you the function's definition. This works for built-in functions as well (thought sometimes these functions are defined in C code in which case R will tell you that the function is a '.Primitive').

2.4.1 Putting R functions in Scripts

When you define a function at the interactive prompt and then close the interpreter your function definition will be lost. The simple way around this is to define your R functions in a script that you can than access at any time.

I R Studio choose `File > New > R Script`. This will bring up a blank editor window. Enter your function into the editor and save the source file in your R working directory with a name like `vecgeom.R`.

```
# functions defined in vecgeom.R

veclength <- function(x) {
  # Given a numeric vector, returns length of that vector
  sqrt(drop(x %*% x))
}

unitvector <- function(x) {
  # Return a unit vector in the same direction as x
  x/veclength(x)
}
```

There are two functions defined above, one of which calls the other. Both take single vector arguments. These functions have no error checking to insure that the arguments passed to the functions are reasonable but R's built in error handling will do just fine for most cases.

Once your functions are in a script file you can make them accesible by using the `source()` function (See also the `Source` tab button in the R Studio GUI):

```
> source("vecgeom.R")
> x <- c(1,0.4)
> veclength(x)
[1] 1.077033
> ux <- unitvector(x)
> ux
[1] 0.9284767 0.3713907
> veclength(ux)
[1] 1
> a
[1] 1 2 3 4 5
> veclength(a)
[1] 7.416198
> ua <- unitvector(a)
> ua
[1] 0.1348400 0.2696799 0.4045199 0.5393599 0.6741999
> veclength(ua)
[1] 1
```

Note that our functions work with vectors of arbitrary dimension.

Assignment 2.1

Write a function that uses the dot product and the `acos()` function to calculate the angle (in radians) between two vectors of arbitrary dimension. By default, your function should return the angle in radians. Also include a logical (Boolean) argument that will return the answer in degrees. Test your function with the following two vectors: `x = [-3, -3, -1, -1, 0, 0, 1, 2, 2, 3]` and `y = [-8, -5, -3, 0, -1, 0, 5, 1, 6, 5]`. The expected angle for these test vectors is 0.441 radians (25.3 degrees).

Let's also add the following function to `vecgeom.R` to aid in visualizaing 2D vectors:

```
draw.vectors <- function(a, b, colors=c('red', 'blue'), clear.plot=TRUE){

    # figure out the limits such that the origin and the vector
    # end points are all included in the plot
    xhi <- max(0, a[1], b[1])
    xlo <- min(0, a[1], b[1])
    yhi <- max(0, a[2], b[2])
    ylo <- min(0, a[2], b[2])

    xlims <- c(xlo, xhi)*1.10 # give a little breathing space around
        vectors
    ylims <- c(ylo, yhi)*1.10
    if (clear.plot){
        plot(xlims, ylims, type='n', asp=1, xlab="x-coord", ylab="y-coord")
    }
    arrows(0, 0, a[1], a[2], length=0.1, col=colors[1])
    arrows(0, 0, b[1], b[2], length=0.1, col=colors[2])
}
```

You can use this new function as follows:

```
# you need to source the file everytime you change it
> source("/Users/pmagwene/Downloads/vecgeom.R")
> x <- c(1,0.4)
> y <- c(0.2, 0.8)
> draw.vectors(x,y)  # draw the original vectors
```

The resulting figure should resemble the one below.

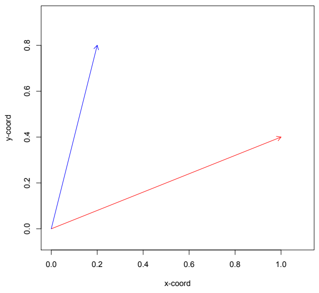

Figure 2.5: Another vector figure.

Notice that we included a `clear.plot` argument in our `draw.vectors` function. I included this so we could add additional vectors to our plot, without overwriting the old vectors, as demonstrated below:

```
# draw the unit vectors that point in the same directors as the original
    vectors
> ux <- unitvector(x)
> uy <- unitvector(y)
> draw.vectors(ux, uy, colors=c('black', 'green'), clear.plot=F)
```

Unlike the other functions we wrote, `draw.vectors` only works properly with 2D vectors. Since any pair of vectors defines a plane, it is possible to generalize this function to work with arbitrary pairs of vectors.

Assignment 2.2

Write a function, `vproj()`, that takes two vectors, $\vec{x}$ and $\vec{y}$, and returns a list containing the projection of $\vec{y}$ on $\vec{x}$ and the component of $\vec{y}$ in $\vec{x}$:

$$P_{\vec{x}}(\vec{y}) = \left(\frac{\vec{x} \cdot \vec{y}}{|\vec{x}|}\right) \frac{\vec{x}}{|\vec{x}|}$$

and

$$C_{\vec{x}}(\vec{y}) = \frac{\vec{x} \cdot \vec{y}}{|\vec{x}|}$$

Use the test vectors from Assignment 2.1 to test your function. The list returned by your function for these test vectors should resemble that shown below:

```
> vproj(x, y)

$proj
 [1] -6 -6 -2 -2  0  0  2  4  4  6

$comp
[1] 12.32883
```

2.5 Vector Geometry of Correlation and Regression

Let's return to our use of the dot product to explore the relationship between variables. First let's add a function to our module, `vecgeom.R`, to calculate the cosine of the angle between to vectors.

```
# add to vecgeom.R

vec.cos <- function(x,y) {
  # Calculate the cos of the angle between vectors x and y
  len.x <- veclength(x)
  len.y <- veclength(y)
  return( (x %*% y)/(len.x * len.y) )
}
```

We can then use this function to examine the relationships between the variables in the iris dataset. For now let's just work with the *I. setosa* specimens. Read the help file for `subset()`.

```
> setosa <- subset(iris, Species == 'setosa', select = -Species)
> dim(setosa)
[1] 50  4
> names(setosa)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
```

Often times it's useful to look at many bivariate relationships simultaneously. The `pairs()` function allows you to do this:

```
> pairs(setosa)
```

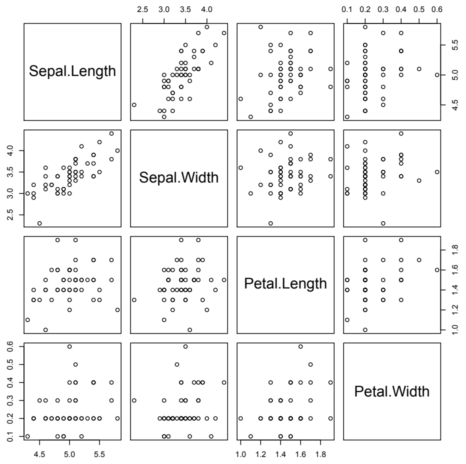

Figure 2.6: Output of the `pairs()` function for the *I. setosa* specimens in the `iris` dataset.

First we'll center the setosa dataset using the `scale()` function. `scale()` has two logical arguments `center` and `scale`. By default both are `TRUE` which will center *and* scale the variables. But for now we just want to center the data. `scale()` returns a matrix object so we use the `data.frame` function to cast the object back to a data frame.

```
> source("/Users/pmagwene/Downloads/vecgeom.R")
> ctrd <- scale(setosa,center=T,scale=F)
> class(ctrd)
[1] "matrix"
> names(ctrd)
NULL
> ctrd <- data.frame(scale(setosa,center=T,scale=F))
> class(ctrd)
[1] "data.frame"
> names(ctrd)
[1] "Sepal.Length" "Sepal.Width"  "Petal.Length" "Petal.Width"
> vec.cos(ctrd$Sepal.Length, ctrd$Sepal.Width)
          [,1]
[1,] 0.7425467
> vec.cos(ctrd$Sepal.Length, ctrd$Petal.Length)
          [,1]
[1,] 0.2671758
> vec.cos(ctrd$Sepal.Length, ctrd$Petal.Width)
          [,1]
[1,] 0.2780984
```

Consider the values above in the context of the scatter plots you generated with the `pairs()` function; and then recall that for mean-centered variables, $\text{cor}(X,Y) = r_{XY} = cos\,\theta = \frac{\vec{x}\cdot\vec{y}}{|\vec{x}||\vec{y}|}$. So our `vec.cos()` function, when applied to centered data, is equivalent to calculating the correlation between x and y. Let's confirm this using the built in `cor()` function in R:

```
> cor(setosa$Sepal.Length, setosa$Sepal.Width)
[1] 0.7425467
> cor(setosa)  # called like this will calculate all pairwise correlations
             Sepal.Length Sepal.Width Petal.Length Petal.Width
Sepal.Length    1.0000000   0.7425467    0.2671758   0.2780984
Sepal.Width     0.7425467   1.0000000    0.1777000   0.2327520
Petal.Length    0.2671758   0.1777000    1.0000000   0.3316300
Petal.Width     0.2780984   0.2327520    0.3316300   1.0000000
```

2.5.1 Bivariate Regression in R

R has a flexible built in function, `lm()` for fitting linear models. Bivariate regression is the simplest case of a linear model.

```
> setosa.lm <- lm(Sepal.Width ~ Sepal.Length, data=setosa)
> class(setosa.lm)
[1] "lm"
> names(setosa.lm)
 [1] "coefficients"  "residuals"     "effects"       "rank"
 [5] "fitted.values" "assign"        "qr"            "df.residual"
 [9] "xlevels"       "call"          "terms"         "model"
> coef(setosa.lm)
 (Intercept) Sepal.Length
  -0.5694327    0.7985283
```

The function `coef()` will return the intercept and slope of the line representing the bivarariate regression. For a more complete summary of the linear model you've fit use the `summary()` function:

```
> summary(setosa.lm)

Call:
lm(formula = Sepal.Width ~ Sepal.Length, data = setosa)

Residuals:
     Min       1Q   Median       3Q      Max
-0.72394 -0.18273 -0.00306  0.15738  0.51709

Coefficients:
             Estimate Std. Error t value Pr(>|t|)
(Intercept)   -0.5694     0.5217  -1.091    0.281
Sepal.Length   0.7985     0.1040   7.681 6.71e-10 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘’. 0.1 ‘’  1
```

```
Residual standard error: 0.2565 on 48 degrees of freedom
Multiple R-squared: 0.5514, Adjusted R-squared: 0.542
F-statistic: 58.99 on 1 and 48 DF,  p-value: 6.71e-10
```

As demonstrated above, the `summary()` function spits out key diagnostic information about the model we fit. Now let's create a plot illustrating the fit of the model.

```
> plot(Sepal.Width ~ Sepal.Length, data=setosa, xlab="Sepal Length (cm)",
    ylab="Sepal Width (cm)", main="Iris setosa")
> abline(setosa.lm, col='red', lwd=2, lty=2)  # see ?par for info about lwd
     and lty
```

Your output should resemble the figure below. Note the use of the function `abline()` to plot the regression line. Calling `plot()` with an object of class `lm` shows a series of diagnostic plots. Try this yourself.

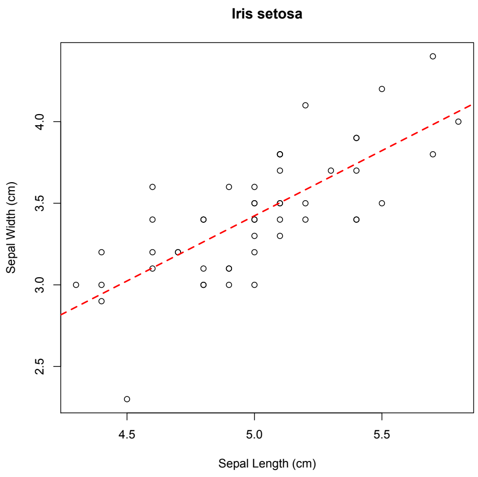

Figure 2.7: Linear regression of Sepal Width on Sepal Length for *I. setosa*.

Assignment 2.3

Write your own regression function (i.e. your code shouldn't refer to the built in regression functions) for mean centered vectors in R. The function will take as it's input two vectors, $\vec{x}$ and $\vec{y}$. The function should return:

1. a list containing the mean-centered versions of these vectors
2. the regression coefficient b in the mean centered regression equation $\hat{\vec{y}} = b\vec{x}$
3. the coefficient of determination, R^2

Demonstrate your regression function by using it to carry out regressions of Sepal.Length on Sepal.Width separately for the 'versicolor' and 'virginica' specimens from the iris data set. Include `ggplot` created plots in which you use the `geom_point()` and `geom_abline()` functions to illustrate your calculated regression line. To test your function, compare your regression coefficients and coefficient of determination to the same values returned by the built in `lm()` function.