Bio 723

Scientific Computing for Biologists

Paul M. Magwene and Colin S. Maxwell

Fall 2013

Contents

1. Getting your feet wet with R

1.1. Getting Acquainted with R

1.1.1. Installing R

The R website is at http://www.r-project.org/. I recommend that you spend a few minutes checking out the resources, documentation, and links on this page. Download the appropriate R installer for your computer from the Comprehensive R Archive Network (CRAN). A direct link can be found at: http://cran.stat.ucla.edu/. As of mid August 2012 the latest R release is verison 2.15.1.

The R installer will install appropriate icons under the `Start Menu` (Windows) or `Applications` Folder (OS X). On OS X it will install two icons – "R" and "R64", corresponding to 32-bit and 64-bit versions of the executable. The 64 bit version, which allows access to much larger amounts of your comptuer's RAM, is suitable for dealing with very large data sets.

1.1.2. Starting and R Interactive Session

The OS X and Windows version of R provide a simple GUI interface for using R in interactive mode. When you start up the R GUI you'll be presented with a single window, the R console. See the your textbook, The Art of R Programming (AoRP) for a discussion of the difference between R's interactive and batch modes.

1.1.3. 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 in the GUI. 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()`.

1.1.4. 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, while the corresponding item is found under the `Misc` menu on OS X.

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

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

1.1.5. Using R as a Calculator

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

```
> 10 + 2 # addition
[1] 12
> 10 - 2 # subtraction
[1] 8
> 10 * 2 # multiplication
[1] 20
> 10 / 2 # division
[1] 5
> 10 ^ 2 # exponentiation
[1] 100
> 10 ** 2 # alternate exponentiation
[1] 100
> sqrt(10) # square root
[1] 3.162278
> 10 ^ 0.5 # same as square root
[1] 3.162278
> exp(1) # exponential function
[1] 2.718282
```

```
> 3.14 * 2.5^2
[1] 19.625
> pi * 2.5^2 # R knows about some 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.

1.1.6. 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
[1] TRUE
> 10 != 10 # does not equal
[1] FALSE
> 10 == (sqrt(10)^2) # 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 do 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.

1.1.7. Working with Vectors in R

Vectors are the core data structure in R. Vectors store an ordered list of items all of the same type. Learning to compute effectively with vectors and one of the keys to efficient R programming. Vectors in R always have a length (accessed with the `length()` function) and a type (accessed with the `typeof()` function).

The simplest way to create a vector at the interactive prompt is to use the `c()` function, which is short hand for 'combine' or 'concatenate'.

```
> x <- c(2,4,6,8)
[1] "double"
> length(x)
[1] 4
> y <- c('joe','bob','fred')
> typeof(y)
[1] "character"
> length(y)
[1] 3
> z <- c() # empty vector
> length(z)
[1] 0
> typeof(z)
[1] "NULL"
```

You can also use `c()` to concatenate two or more vectors together.

```
> v <- c(1,3,5,7)
> w <- c(-1, -2, -3)
> vwx <- c(v,w,x)
> vwx
 [1]  1  3  5  7 -1 -2 -3  2  4  6  8
```

Vector Arithmetic and Comparison

The basic 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
```

If you try and apply arithmetic operations to non-numeric vectors, R will warn you of the error of your ways:

```
> w <- c('foo', 'bar', 'baz', 'qux')
> w**2
Error in w^2 : non-numeric argument to binary operator
```

Note, however that the comparison operators can work with non-numeric vectors. The results you get will depend on the type of the elements in the vector.

```
>  w == 'bar'
[1] FALSE  TRUE FALSE FALSE
> w < 'cat'
[1] FALSE  TRUE  TRUE FALSE
```

Indexing Vectors

For a vector of length n, we can access the elements by the indices $1 \ldots n$. We say that R vectors (and other data structures like lists) are 'one-indexed'. Many other programming languages, such as Python, C, and Java, use zero-indexing where the elements of a data structure are accessed by the indices $0 \ldots n-1$. Indexing errors are a common source of bugs. When moving back and forth between different programming languages keep the appropriate indexing straight!

Trying to access an element beyond these limits returns a special constant called `NA` (Not Available) that indicates missing or non-existent values.

```
> x <- c(2, 4, 6, 8, 10)
> 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
```

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 as '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
# 'by' is the 3rd argument so you don't have to specify it
> s <- seq(0.5, 1.5, 0.33)
> 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
```

1.1.8. Some Useful Functions

You've already seem a number of functions (`c()`, `length()`, `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 longer vectors

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). We'll see how to read in data from files below.

Note that we created a variable with the name `test.scores`. If you have previous programming experience you might be surprised that this works. Unlike most languages, R allows 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,max returned as a vec of len 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
```

1.1.9. 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 (type `?log` now) 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
```

Because base 2 and base 10 logarithms are fairly commonly used, there are convenient aliases for calling log with these bases.

```
> log2(8)
[1] 3
> log10(100)
[1] 2
```

1.1.10. Lists in R

R lists are like vectors, but unlike a vector where all the elements are of the same type, the elements of a list can have arbitrary types (even other lists).

```
> l <- list('Bob', pi, 10, c(2,4,6,8))
```

Indexing of lists is different than indexing of vectors. Double brackets (`x[[i]]`) return the element at index i, single bracket return a list containing the element at index i.

```
> l[1] # single brackets
[[1]]
[1] "Bob"

> l[[1]] # double brackets
[1] "Bob"
> typeof(l[1])
[1] "list"
> typeof(l[[1]])
[1] "character"
```

The elements of a list can be given names, and those names objects can be accessed using the `$` operator. You can retrieve the names associated with a list using the `names()` function.

```
> l <- list(name='Bob', age=27, years.in.school=10)
> l
$name
[1] "Bob"

$age
[1] 27

$years.in.school
[1] 10

> l$years.in.school
[1] 10
> l$name
```

```
[1] "Bob"
> names(l)
[1] "name"            "age"             "years.in.school"
```

1.1.11. 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.

1.1.12. 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

> algae.table
[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
[12] "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
[12] 0.100

> algae.table[[1]]
 [1] "Jan" "Feb" "Mar" "Apr" "May" "Jun" "Jul" "Aug" "Sep" "Oct" "Nov"
[12] "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
[12] 0.100
```

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

1.1.13. 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. Download the `turtles.txt` data set from the class wiki. The data in `turtles.txt` are a set of linear measurements representing dimensions of the carapace (upper shell) of painted turtles (*Chrysemys picta*), as reported in Jolicoeur and Mosimmann, 1960; Growth 24: 339-354.

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.

1.1.14. 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
```

1.2. Exploring Univariate Distributions in R

1.2.1. Histograms

One of the most common ways to examine a the distribution of observations for a single variable is to use a histogram. The `hist()` function creates simple histograms in R.

```
> hist(turtles$length) # create histogram with fxn defaults
> ?hist # check out the documentation on hist
```

Note that by default the `hist()` function plots the frequencies in each bin. If you want the probability densities instead set the argument `freq=FALSE`.

```
> hist(turtles$length,freq=F) # y-axis gives probability density
```

Here's some other ways to fine tune a histogram in R.

```
> hist(turtles$length, breaks=12) # use 12 bins
> mybreaks = seq(85,185,8)
> hist(turtles$length, breaks=mybreaks) # specify bin boundaries
> hist(turtles$length, breaks=mybreaks, col='red') # fill the bins with red
```

1.2.2. Density Plots

One of the problems with histograms is that they can be very sensitive to the size of the bins and the break points used. You probably noticed that in the example above as we changes the number of bins and the breakpoints to generate the histograms for the `turtles$length` variable. This is due to the discretization inherent in a histogram. A 'density plot' or 'density trace' is a continuous estimate of a probability distribution from a set of observations. Because it is continuous it doesn't suffer from the same sensitivity to bin sizes and break points. One way to think about a density plot is as the histogram you'd get if you averaged many individual histograms each with slightly different breakpoints.

```
> d <- density(turtles$length)
> plot(d)
```

A density plot isn't entirely parameter free – the parameter you should be most aware of is the 'smoothing bandwidth'.

```
> d <- density(turtles$length) # let R pick the bandwidth
> plot(d,ylim=c(0,0.020)) # gives ourselves some extra headroom on y-axis
> d2 <- density(turtles$length, bw=5) # specify bandwidth
> lines(d2, col='red') # use lines to draw over previous plot
```

The bandwidth determines the standard deviation of the 'kernel' that is used to calculate the density plot. There are a number of different types of kernels you can use; a Gaussian kernel is the R default and is the most common choice. In the example above, R picked a bandwidth of 8.5 (the black line in our plot). When we specified a smaller bandwith of 5, the resulting density plot (red) is less smooth. There exists a statistical literature on picking 'optimum' kernel sizes. In general, larger data sets support the use of smaller kernels. See the R documentation for more info on the `density()` function and references to the literature on density estimators.

The `lattice` package is an R library that makes it easier to create graphics that show conditional distributions. Here's how to create a simple density plot using the `lattice` package.

```
> library(lattice)
> densityplot(turtles$length) # densityplot defined in lattice
```

Notice how by default the `lattice` package also drew points representing the observations along the x-axis. These points have been 'jittered' meaning they've been randomly shifted by a small amount so that overlapping points don't completely hide each other. We could have produced a similar plot, without the lattice package, as so:

```
> d <- density(turtles$length)
> plot(d)
> nobs <- length(turtles$length)
> points(jitter(turtles$length), rep(0,nobs))
```

Notice that in our version we only jittered the points along the x-axis. You can also combine a histogram and density trace, like so:

```
> hist(turtles$length, 10, xlab='Carapace Length (mm)',freq=F)
```

```
> d <- density(turtles$length)
> lines(d, col='red', lwd=2) # red lines, with pixel width 2
```

Notice the use of the `freq=F` argument to scale the histogram bars in terms of probability density.

Finally, let's some of the features of `lattice` to produce density plots for the 'length' variable of the turtle data set, conditional on sex of the specimen.

```
> densityplot(~length | sex, data = turtles)
```

There are a number of new concepts here. The first is that we used what is called a 'formula' to specify what to plot. In this case the formula can be read as 'length conditional on sex'. We'll be using formulas in several other contexts and we discuss them at greater length below. The `data` argument allows us to specify a data frame or list so that we don't always have to write arguments like `turtles$length` or `turtles$sex` which can get a bit tedious.

1.2.3. Box Plots

Another common tool for depicting a univariate distribution is a 'box plot' (sometimes called a box-and-whisker plot). A standard box plot depicts five useful features of a set of observations: the median (center most line), the upper and lower quartiles (top and bottom of the box), and the minimum and maximum observations (ends of the whiskers).

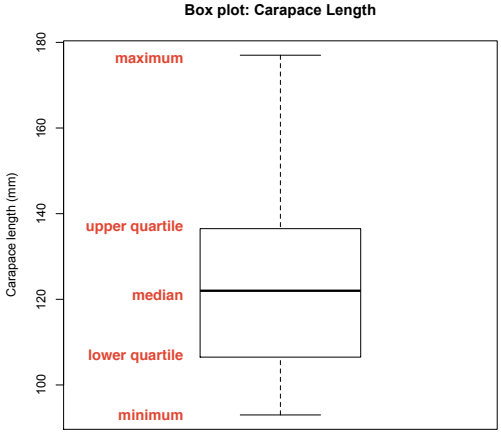

Figure 1.1.: A box plot represents a five number summary of a set of observations.

There are many variants on box plots, particularly with respect to the 'whiskers'. It's always a good idea to be explicit about what a box plot you've created depicts.

Here's how to create box plots using the standard R functions as well as the lattice package:

```
> boxplot(turtles$length)
```

```
> boxplot(turtles$length, col='darkred', horizontal=T) # horizontal version
> title(main = 'Box plot: Carapace Length', ylab = 'Carapace length (mm)')
> bwplot(~length,data=turtles) # using the bwplot function from lattice
```

Note how we used the `title()` function to change the axis labels and add a plot title.

Historical note – The box plot is one of many inventions of the statistician John W. Tukey. Tukey made many contributions to the field of statistics and computer science, particularly in the areas of graphical representations of data and exploratory data analysis.

1.2.4. Bean Plots

My personal favorite way to depict univariate distributions is called a 'beanplot'. Beanplots combine features of density plots and boxplots and provide information rich graphical summaries of single variables. The standard features in a beanplot include the individual observations (depicted as lines), the density trace estimated from the observations, the mean of the observations, and in the case of multiple beanplots an overall mean.

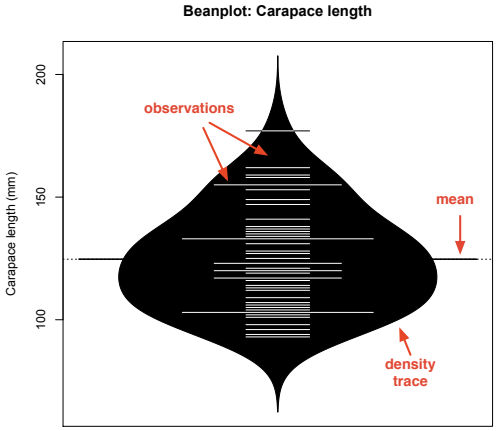

Figure 1.2.: Beanplots combine features of density and box plots.

The `beanplot` package is not installed by default. To download it and install it use the R package installer under the `Packages & Data` menu. If this is the first time you use the package installer you'll have to choose a CRAN repository from which to download package info (I recommend you pick one in the US). Once you've done so you can search for 'beanplot' from the Package Installer window. You should also check the 'install dependencies' check box.

Once the beanplot package has been installed check out the examples to see some of the capabilities:

```
> library(beanplot)
```

Note the use of the `library()` function to make the functions in the `beanplot` library available for use. Here's some examples of using the `beanplot` function with the turtle data set:

```
> beanplot(turtles$length) # note the message about log='y'
> beanplot(turtles$length, log='') # DON'T do the automatic log transform
> beanplot(turtles$length, log='', col=c('white','blue','blue','red'))
```

In the final version we specified colors for the parts of the beanplot. See the explanation of the `col` argument int he beanplot function for details.

We can also compare the carapace length variable for male and female turtles.

```
> beanplot(length ~ sex, data = turtles, col=list(c('red'),c('black')),
names = c('females','males'),xlab='Sex', ylab='Caparace length (mm)')
```

Note the use of the formula notation to compare the carapace length variable for males and females. Note the use of the list argument to `col`, and the use of vectors within the list to specify the colors for female and male beanplots.

There is also a asymmetrical version of the beanplot which can be used to more directly compare distributions between two groups. This can be specified by using the argument `side='both'` to the beanplot function.

```
> beanplot(length~sex, data=turtles, col=list(c('red'),c('black')),names=c(
    'females','males'),xlab='Sex', ylab='Carapace length (mm)',side='both')
```

Plots like this one are very convenient for comparing distributions between samples grouped by treatment, sex, species, etc.

We can also create a beanplot with multiple variables in the same plot if the variables are measured on the same scale.

```
> beanplot(turtles$length, turtles$width, turtles$height, log='',
names=c('length','width','height'), ylab='carapace dimensions (mm)')
```

1.2.5. Demo Plots in R

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

```
> demo(graphics)
```

A. The Unix Command Line

A.1. Platform specific issues

OS X

If your computer has Linux or Mac OS X installed you already have a native Unix environment. All of the command line tools we'll use in this tutorial are already available to you.

Unix on Windows

If your computer runs Windows you can have access to a Unix-like environment by installing a program called Cygwin (http://www.cygwin.com). Cygwin is free, open source, and provides a convenient installer for common Unix programs. Download the installer program (`setup.exe`) and place it in a `c:\cygwin` directory (I recommend that you use this directory as the installation directory as well).

During the installation you will have the choice of installing additional tools. I recommend you install the following packages (use the search box to find them):

- curl
- mintty

curl is a command line tool for transferring data over networks. It makes it easy to download packages and source code with a single text command. mintty is a terminal window program for Cygwin that provides a nicer interface than the standard Windows terminal.

The first time you run Cygwin I recommend you start it with the default terminal emulator (the Cygwin shortcut on your desktop will link to the default). After that you can interface with the Cygwin tools via mintty. During the Cygwin installation a mintty shortcut was put in your Start Menu (under the Cygwin folder). Since you'll be using the terminal frequently I recommend you pin the mintty shortcut to your taskbar (available via right clicking the mintty icon on Windows 7; copy the shortcut and drag the copy to your taskbasr on older version of Windows).

Directory structure under Cygwin

NOTE: all the commands that follow should be executed from within the cygwin bash shell (i.e. in the terminal window you get when you click the Cygwin or mintty shortcuts).

Cygwin creates a set of subdirectories that mirrors the standard Unix file system (`/bin`, `/usr`, `/var`, `/home`, etc). When you start the bash shell you will be in your home directory (`/home/<username>`).

Cygwin will treat the directory where you installed it (`c:\cygwin` if you followed my instructions above) as if that was the 'root' directory of the Unix file system. So when you type `cd /home` from the shell in Cygwin, you're really in `c:\cygwin\home`. To access the standard Windows drive names, Cygwin provides a mapping through a directory called `/cygdrive`. For example, to list the contents of `c:\Python27` from Cygwin you would type:

```
$ ls /cygdrive/c/Python27
```

Setting up symbolic links and aliases in bash under Cygwin

The bash shell (the default shell under Cygwin) can be customized and configured to suit your needs. Let's start by creating a convenient link between your cygwin home directory and your Windows home directory.

```
$ cd ~  # makes sure your in your home directory
$ ln -s /cygdrive/c/<username> ~/winhome
```

`ln -s` is a command that create a "symbolic" or "soft" link between files and directories. This makes it convenient to quickly navigate from your cygwin home directory as so:

```
$ cd ~/winhome
$ ls
... list of files and directories in your home directory...
```

A.2. The Unix philosophy

Doug McIlroy who invented the concept of the Unix 'pipe' (discussed below) summarized the Unix philosphy as follows:

> "This is the Unix philosophy: Write programs that do one thing and do it well. Write programs to work together. Write programs to handle text streams, because that is a universal interface."

This is not some rigid set of specifications, but rather an approach to writing simple programs that can be tied together in useful ways to accomplish larger, more complex tasks. Most of the standard Unix commands are written with this philosophy in mind. The scripts you will develop over the next three class session will follow the same philosophy (and take advantage of other software tools that also use the Unix approach).

A.3. The Unix command line

A Unix/Linux command line environment is the de-facto standard for building bioinformatics pipelines. While the command line may not be particularly user friendly, various aspects of how Unix is designed make it very powerful for constructing analysis pipelines. We'll review some of those design aspects here.

A.3.1. Unix tools

You'll need the following tools. Under Cygwin these are all easily installed using the GUI installation interface (or installed by default). On Linux or OS X many of these are already installed by default.

- less - a 'pager' (convenient for viewing files)
- curl - a package for retrieving files using a variety ofInternet protocols; found under 'Net' in the cygwin installer.
- gzip - a file compression utility
- tar - a file archiving utility (usually used in conjunction with gzip)
- awk - a text processing programming language.

Basic Unix commands

You should familiarize yourself with the basic unix commands covered in the UNIX Tutorial for Beginners (see link on class website). Here are some of the more common ones you'll need to navigate around your file system:

- `ls` – list the content of a directory e.g. `ls /home/`
- `cd` – change directory. e.g. `cd /home/pmagwene/tmp`
- `pwd` – display the name of the present working directory.
- `mv` – move a file. e.g. `mv myfile newfile`
- `rm` – remove (delete) a file. Be careful with this one! e.g. `rm tmpfile`
- `find` – find files that match a given patern. e.g. `find . -name "*.txt"` (matches all files in the current directory that end with '.txt').
- `man` – show the manual pages for a command. e.g. `man ls`
- `less` – show the contents of a file, displaying one page at a time. e.g. `less somefile.txt` (use the space bar to advance, `b` to go back, `q` to quit)

The bash shell

The bash shell is the default shell on most Linux systems, Cygwin, and recent versions of OS X. The shell itself provides a useful framework for interacting with the operating system. Shell scripts can be written to make the shell environment even more powerful. We'll explore how to do this in today's exercises.

Here's a few efficiency tips to keep in mind when working in the bash shell:

- Scroll backwards and forwards through your command history using the up and down arrow keys
- Use the tab key to invoke command and file-name completion to keep your typing to a minimum
- Use `<ctrl-r>` and then start typing a word or phrase to search on; this invokes the history search mode to do a reverse incremental search of previous commands
  - Once you've found the command you were searching for hit `<Enter>` to execute it or `<ctrl-j>` to retrieve the command for further editing.
  - Use `<ctrl-g>` or `<ctrl-c>` to cancel the history search mode

Everything is a file or a process

One of the aspects of Unix that makes it easy to tie programs together is that the operating systems treats pretty much everything as either a *file* or a *process*. There are three categories of files in Unix: plain files (e.g. text files, image files, word documents, video files, the code for a program, etc.), directories (e.g. your home directory, the root directory), and devices (e.g. the keyboard, a printer, a display screen, etc). The same basic set of commands can be applied to all three types of files.

A *process* is an instance of a running program. Everytime you start a program the operating system creates a process ID (PID) that is associated with that process. Processes typically operate on data in the form of files (of any of the three types) and return data that is sent to a file (again, any of the three file types). Any given process can start multiple subprocesses (also called child processes), however a process can only have one parent. For example, when you logon to a Unix system you are typically working in a shell process (common shells include bash, tcsh, csh, etc.). When you type a command like `ls` this creates a child process. The parent process is temporarily suspended until the child process returns its output. The `ls` process takes a file as input (the current directory by default), and returns it's output to the display associated with the shell (represented by a device file).

Redirection and Pipes

Because Unix treats everything as a file or process, it's easy to change the source of input and the destination for output. There are several special operators that allow one to change the source/destination of input and output from a process. These are:

- > (redirect output operator)
- >> (append output operator)
- < (redirect input operator)
- | (pipe)

We'll give a few example of redirection and pipes using the commands `ls` (list directory contents), and `grep` (find lines matching a pattern):

```
$ ls -l
total 4184
drwx------+   43 pmagwene  staff     1462 Nov 13 18:14 Desktop
drwx------+   20 pmagwene  staff      680 Sep 23 12:32 Documents
... output truncated ...
$ ls -l > ex1.out # redirect output to file
```

[bash]

In the example above we redirected the output of the `ls -l` command to a file named `ex1.out`. Open the file to confirm this. Now we'll 'pipe' the output of `ls` to the `grep` command.

```
$ ls -l | grep 'Nov'
drwx------+   43 pmagwene  staff     1462 Nov 13 18:14 Desktop
drwx------+ 1285 pmagwene  staff    43690 Nov 13 18:48 Downloads
drwx------+   14 pmagwene  staff      476 Nov 10 11:51 Pictures
-rw-r--r--     1 pmagwene  staff     1427 Nov 14 14:13 ex1.out
-rw-r--r--     1 pmagwene  staff   604976 Nov  1 12:21 rolland-etal-2000-cAMP.pdf
```

In this example we used grep to show all the lines of the `ls` output that have the string 'Nov' in them.

Now we'll combine those two commands and redirect the output to another file.

```
$ ls -l | grep 'Nov' > ex2.out
```

Finally, let's use the append output operator to append to our file lines with 'Oct' as well.

```
$ ls -l | grep 'Oct' >> ex2.out
```

If we had used the redirection operator rather than append than it would have overwritten the previous contents of the file rather than adding the output to what was already there. Open `ex2.out` to confirm that your commands worked as expected.

Review chapter 3 of the UNIX Tutorial for Beginners (see link on class website) for more examples illustrating the use of redirection and pipes. We'll be using pipes and redirection throughout these hands-on exercises.

A.3.2. Using curl to retrieve files from the net

`curl` is a command line tool for transfering data to or from a server using a variety of different protocols including FTP, HTTP, SCP, etc. `curl` is available by default in recent versions of OS X. If you're running Cygwin on Windows you may have to install it from the "Net" subdirectory in the Cygwin installer.

Using `curl` is relatively straightforward. Here, we'll use it to download a file that we're going to use for today's exercises:

```
$ curl -O http://downloads.yeastgenome.org/curation/chromosomal_feature/
  saccharomyces_cerevisiae.gff
```

Type `man curl` or check out an online version of the manual for more information on using curl: http://curl.haxx.se/docs/manual.html.

A.3.3. The GFF3 File Format

GFF3 (GENERIC FEATURE FORMAT VERSION 3) is a text-based format for representing genomic features. It is widely used by the genomics research community for representing sequence features associated with genome projects. All of the major genome databases provide data in GFF3 format and most of the software tools used by the reserach community can parse GFF3 formatted files.

You can read the details of the GFF3 format here: http://www.sequenceontology.org/gff3.shtml. Notice that a GFF3 file consists 9 columns, separated by tabs. Read the above web page to understand what each of these 9 columns represents.

A.4. Tools for manipulating text

Many types of data, including GFF3 files, are structured text files. Because of this it's useful to have a handle on some of the major tools that Unix provides for manipulating such files.

A.4.1. head and tail

`head` and `tail` respectively show the first n and last n lines of a file (default $n = 10$). These can be useful for quickly checking out what's in a file. `tail` is especially useful for looking at log files to see the last few entries entered in a log.

```
$ head saccharomyces_cerevisiae.gff
... < output truncated > ...
$ tail saccharomyces_cerevisiae.gff
... < output truncated > ...
```

Use the -n argument to specify the number of lines you'd like to see:

```
$ head -n 3 saccharomyces_cerevisiae.gff
##gff-version 3
#date Mon Nov 15 19:50:13 2010
#
```

A.4.2. less

`less` is 'pager' program that allows you to scroll through a file (or standard input) page by page.

```
$ less saccharomyces_cerevisiae.gff
```

From within `less` you can scroll forward by hitting the space bar, or the 'f' key, backward by typing 'b'. To search for a particular word or pattern in the file type '/' followed by the word of interest and then hit return. All the instances of that word / pattern will be highlighted. For example, from within `less` type `/gene<RET>`, where `<RET>` means hit the enter or return key, to find all instances of the word 'gene' in the file. Type `q` to quit `less`.

A.4.3. echo

`echo` simply writes it's string argument to standard output (i.e. it echos what you type).

```
$ echo "hello, world"
hello, world
$ echo "These are the times that try men's souls"
These are the times that try men's souls
```

A.4.4. cat

`cat`, short for 'concatenate', is a utility for concatenating and printing text. Here are some examples of it's use:

```
$ echo "some text here" > file1.txt
$ echo "some more text" > file2.txt
$ cat file1.txt file2.txt > file1plus2.txt
$ cat file1plus2.txt
some text here
some more text
```

A.4.5. wc

`wc` is a program that counts the number of words, lines, and characters in a file. You can also specify you only want one of those counts using options like `-l` (count only words).

```
$ wc saccharomyces_cerevisiae.gff
  168490  299825 18871650 saccharomyces_cerevisiae.gff
$ wc -l saccharomyces_cerevisiae.gff
  168490 saccharomyces_cerevisiae.gff
```

A.4.6. cut

`cut` is a utility for subsetting words, bytes or columns of a text file. For example:

```
$ cut -f1-3 saccharomyces_cerevisiae.gff | less
```

In the above we use cut to show the first three fields of the file, and then we pipe it to `less` to examine one page of text at a time . The default field delimiter in `cut` is a tab (`\t`), but you can specify other delimiters with the `-d` option. You don't have to use adjacent columns with `cut`. For example,

```
$ cut -f1,3-5,7 saccharomyces_cerevisiae.gff | less
```

This allows us to look at the first column, and columns 3-5 and 7, corresponding to the seqid (=chromosome), the feature type, the feature start and stop coordinates (1-based), and the strand on which the feature is defined.

Notice how in addition to the fields, `cut` also gave use the header information at the beginning of the file. We can use the `-s` option to suppress lines that don't have the field delimiter character:

```
$ cut -s -f1,3-5,7 saccharomyces_cerevisiae.gff > out.txt
```

Notice this time we redirected the output of the command to a file, `out.txt`.

A.4.7. sort

The `sort` utility sorts lines of text. By default `sort` interprets an entire line of text as the key for sorting and sorts in dictionary order. For example, to see the default sorting:

```
$ sort out.txt | less
```

We can use the `-k` option to specify the field to sort on. For example, this is how we can sort on the second column of `out.txt`:

```
$ sort -k2 out.txt | less
```

Another useful option to sort is `-u` which tells sort to output only the first instance of a set of identical keys. Try and figure out what the following command does before running it:

```
$ cut -s -f3 saccharomyces_cerevisiae.gff | sort -u
```

A.4.8. grep

`grep` is a tool for doing regular expression matching on lines of a file. Regular expressions are a way to specificy search patterns in strings. The simplest type of regular expression is to just search for a specific word, as illustrated here:

```
$ grep "gene" out.txt | less
```

The above command simply returns all the lines in `out.txt` that have the word "gene" in them. Let's use this in a slightly different way to count instances of different features in the file:

```
$ grep "gene" out.txt | wc -l
    6720
$ grep "pseudogene" out.txt | wc -l
```

```
      21
$ grep "telomere" out.txt | wc -l
      32
```

NOTE: the numbers of matches may change somewhat between releases of the curated yeast genome. If the numbers you get above or below are *slightly* different from what is shown here don't worry.

We can get a little fancier if we use the "extended" `grep` syntax (specified using the `-E` option). Here's how we can search for lines that match on any of a set of terms (the vertical bar | indicates an "OR" operator):

```
$ grep -E "tRNA|rRNA|snRNA|snoRNA" out.txt | wc -l
     409
```

Note that we have to be careful about what `grep` matches, for example:

```
$ grep "chr01" out.txt | grep "gene" | wc -l
    121
```

Note how we piped two `grep` commands together to get the equivalent of AND ("chr01" AND "gene"). However, there's a very subtle problem with this command as constructed. We search on the word "gene" but "gene" is also a substring of "psuedogene" and hence "pseudogene" features also generate matches. What we really want is whole word matches. We can do that as follows:

```
$ grep "\<chr01\>" out.txt | grep "\<gene\>" | wc -l
     117
```

This uses what are called "POSIX character classes" to match possible sets of characters. A list of the POSIX character classes is linked to on the course wiki. Here's the equivalent call for counting genes on chromosome IV:

```
$ grep "\<chr04\>" out.txt | grep "\<gene\>" | wc -l
     836
```

We've only just scratched the surface of regular expressions. Regular expressions are a very powerful tool and there are whole books on the topic. I'll post a number of links on the course wiki to online tutorials on `grep` and regular expressions.

A.4.9. tr

`tr` is a utility for translating characters within a text stream. `tr` can be useful for converting delimiters from one file type to another. For example, let's say we wanted to analyze the file `out.txt` in a program that expected comma separated values (csv) instead of tab-delimited fields. `tr` makes that conversion easy:

```
$ cat out.txt | tr "\t" "," > out.csv
```

Note that `tr` only reads from standard input so we used the `cat` program to feed the lines of text to `tr`.

A.4.10. awk

awk is a programming language designed for processing structured text files. You can use it to write short one liners or to write full blown programs. It turns out that some form of text file manipulation is often a necessary first step in most bioinformatics analyses, so awk often comes in very handy. We'll use awk to illustrate how you might transitions from simple command line usage into slightly more complicated scripts.

One simple thing we can do with awk is to use it to re-order fields in a structured data file:

```
awk '{print $2, $1, $4, $5}' out.txt | less
```

In the command above the the dollar signs followed by numbers refer to the fields of the file. With it's default setting awk operates line by line, so you can interpret the above statement as saying: "for each line, print the fields 2, 1, 4 and 5".

The basic syntax of awk is often depicted in the form `pattern {action}`. The above command only specified an action, so it was applied to every line. By contrast, in the example below we specify a pattern. The pattern can be read as – "if the 3rd field is 'chromosome'". For all lines that match that pattern the correspoding action is applied; in this case "print fields 1 and 5" (the chromosome name and its length):

```
awk '$3=="chromosome" {print $1, $5}' saccharomyces_cerevisiae.gff
```

Here's another `pattern {action}` pair that shows how we could find all gene features with length less than 300:

```
$ awk '$3 == "gene" && ($5 - $4) < 300 {print $0 }' \
  saccharomyces_cerevisiae.gff | wc -l
     446
```

&& is the AND operator. Read this as "if the 3rd fields is 'gene' AND the the 5th field minus the 4th field is less than 300."

In this last example we added one more condition – we looked for the word 'Dubious' in the 9th field. The results indicate that a significant proportion of these small genes are classified as 'Dubious'.

```
$ awk '$3 == "gene" && ($5 - $4) < 300 && match($9, "Dubious") {print $0
  }' saccharomyces_cerevisiae.gff | wc -l
     163
```