Introduction to R Walkthrough

Demitri Muna
SciCoder Workshop

Data Types

Simple assignment
```
a = 12
```

Create a vector (c is for 'combine').
```
b = c(100, 110, 120)
c = c(10, 20, 30)
d = b + c
```

Access elements by square brackets. R indices are 1-based. Can grow vectors dynamically.
```
e = c(1,2,3)
e[4] = 4
```

Adding different lengths of vectors also works, producing a warning *only* if the shorter length is not a multiple of the longer length.
```
b = c(100, 110, 120)
c = c(50, 50)
d = b + c
```

Concatenate vectors
```
c(b,c)
```

Vector multiplication
```
c(10, 25, 40) * 10^2
c(1, 2, 3) * c(2, 10, 20) # vector dot product
```

Create a vector from a sequence. Starts from 1 unless otherwise specified (with the *from* keyword). Note the named arguments as in python. The order of the *named* parameters doesn't matter, but there is an order if they are not. Unlike (non-numpy) python, non-integer steps are ok.
```
seq(10)
seq(from=100, to=106, by=2)
seq(100, 106, 2)
seq(to=106, by=2, from=100)
```

Any function can operate on a vector.
```
sqrt(seq(1,100,by=10))
```

Create an empty vector of some size (good for predefining vectors of known length).
```
v = vector(length=10)
v[1] = 12
```

Boolean values
```
t = TRUE
```

```
f = FALSE
```

Return the length of a vector.

```
length(v)
```

Control Structures

Ref: http://cran.r-project.org/doc/manuals/R-lang.html#Control-structures

for loop

```
for (i in seq(len(e)) {
     v[i] = someFunction(i)
}
```

But this can also be written as:

```
v = someFunction(v)
```

Generate distributions

```
n = 10000 # number of points to generate
uniform = runif(n) # uniform distribution
hist(uniform)

normal = rnorm(n, mean=5, sd=2) # normal distribution
hist(normal)
```

Simple Plots

```
plot(uniform)
```

Note that if only one vector is given, it assumes:

```
plot(seq(length(v)), v)
```

Operations can be made in the plot statement:

```
plot(sqrt(uniform)) # square root
plot(uniform ^(1/3)) # cube root
```

Add labels:

```
plot(uniform, xlab="X label", ylab="Y label", main="Random Values On
(0,1)")
```

Statistical functions

min, max, length, mean, sd

Reading data from a file

Assuming the data is in a delimited format (the delimiter can be white space or tabs). Lines beginning with "#" are automatically ignored (see `comment.char`). Can skip a given number of lines (see `skip`). Can directly read files that are compressed with gzip, bzip2, xz or lzma.

```
d = read.table("filename.dat")
```

If there is a header line, also space/tab separated (much preferred format):

```
d = read.table("filename.dat", header=TRUE)
```

For a different separator, e.g. ("|"):

```
d = read.table("filename.dat", header=TRUE, sep="|")
```

Make the directory that contains the sample file the current working directory (`setwd("dir")` or cmd-D on the Mac), then read the sample file. Note that tab completion works.

```
d = read.table("Sample R data.txt", header=TRUE)
```

By default, columns are read as text, but even if you forget this the data can be plotted without worrying about it. However, any numeric operations (e.g. col1 - col2) will fail unless the data is converted to be numeric. If all columns are numeric, it's easiest to specify this at the initial read (see the documentation to set the type on a column by column basis):

```
d = read.table("Sample R data.txt", header=TRUE, colClasses =
"numeric")
```

The variable "d" is a *data frame*. Think of this as you would a spreadsheet (or IDL structure). Each column can be accessed by the name specified in the header. To see the fields it contains:

```
names(d)
```

Each vector (column) is accessed as variable$*name*. Let's plot the fields:

```
plot(d$area, d$height)
```

If the file had no header, R provides default header names for each column: V1, V2, V3, etc. For such a file, the first column can be referred to as `d$V1`. If the file does not have a header, you can supply column names when you read the file:

```
d = read.table("Sample R data.txt", col.names=c('a', 'b', 'c')
```

Subsets of data frames can be made like this:

```
d_sub = [100:200,] # rows 100-200, all columns
```

Subset with conditionals:

```
d_sub = subset(d, a1 > 1 & a3 < 3) # where a1,a3 are fields in d
```

New columns can be added:

```
d$new\_column = d$a + d$b
```

Plot has a *long* list of possible parameters (see ?plot and ?par). Let's scale the size of the points drawn (cex) and change the color:

```
plot(d$area, d$height, cex=0.5, col="steelblue")
```

Change the x range of the plot (note the vector form as the parameter), subselect the first 200 points, and add a grid:

```
plot(d$area[1:200], d$height [1:200], cex=0.5, col="steelblue",
xlim=c(0,2), ylim=c(0,60))
grid()
```

Draw a horizontal line at height=40 and a vertical line at area=1:

```
abline(h=40, col='blue')
abline(v=1, col='green')
```

Fit a line to the data using lm (linear model):

```
fit = lm(d$height ~ d$area) # note the order is y ~ x
```

Draw the fit - note that abline can actually just take the object returned by lm.

```
abline(fit, col='blue')
```

Of course, it could have been done in one step if you didn't need the values:

```
abline(lm(d$height ~ d$area), col='blue')
```

Let's draw that plot again, but this time use a subset of data:

```
plot(d$area[d$height < 40], d$height[d$height < 40], cex=0.5,
col="steelblue", xlim=c(0,2), ylim=c(0,60))
```

Can specify multiple criteria:

```
d$area[d$height < 40 & d$height > 10]
```

Plots can be one line, but more complicated plots can be constructed by a series of commands. You can create a 'script' to generate a plot.

Histograms

Plot a histogram of d$area.

```
hist(d$area)
```

R is very good at guessing what the bin size should be based on the data (and you can even select one of a few algorithms to use). You can also supply a vector of bins to use or the number of bins. If you specify a single number, it is only a suggestion. See the difference between these:

```
hist(d$area, br=10)
hist(d$area, br=20)
hist(d$area, br=50)
hist(d$area, br=75)
hist(d$area, br=100, col='steelblue')
```

Note that br=10 and 20 are the same, as are br=50 and 75. Also, many of the plot keywords are available in other places.

Aside: if you look in the documentation, the parameter keyword is actually "breaks", not "br". You only need to type as much of the keyword that makes it unique.

Let's define our own bins:

```
bins = seq(0,10,2.5/100)
hist(d$area, br=bins, xlim=c(0,2.5))
```

The data on the far left is due to noise, so let's place a cut on that to remove some of it.

```
hist(d$area[d$height > 8], br=bins, xlim=c(0,2.5))
```

Let's fit the curve in the center to a Gaussian to find the mean and the standard deviation. To do this, we need x and y values for the peak. Let's select a range of x between 0.75 and 1.75 and then plot those points alone. To do that, we need the x values of the subset. To get this (and other information), we can get a histogram object back from the 'hist' function. We'll go back to using the computed bins.

```
h = hist(d$area, br=200, xlim=c(0,2.5))
```

With this object, you can plot the data as points:

```
plot(h$mids[h$mids > 0.75 & h$mids < 1.75], h$counts[h$mids > 0.75 & 
h$mids < 1.75])
```

Let's shorten for convenience:

```
x = h$mids[h$mids > 0.75 & h$mids < 1.75]
y = h$counts[h$mids > 0.75 & h$mids < 1.75]
```

We'll perform a non-linear minimization on these points. This can be done for any function you define, so let's define a Gaussian function:

```
gaus = function(x, const, mean, sigma) {
     const * exp(-0.5 * ((x-mean)/sigma)^2)
}
```

Note that you can type the bare name of the function (as you would a variable) to display the definition.

Perform the fit, providing initial guesses in a list (think Python dictionary). The trace shows the fitting progress. Best advice: plot the data first to make a visual initial value guess!

```
fit = nls(y ~ gaus(x, const, mean, sigma), start=list(const=500,
mean=1.1, sigma=0.2), trace=TRUE)
```

See the result of the by entering `fit` or `summary(fit)`.

Extract the values programmatically:

```
const = summary(fit)$parameters[1]
mean = summary(fit)$parameters[2]
sigma = summary(fit)$parameters[3]
```

Overplot the result. Rather than plot a series of points (and worrying about generating *x* and *y* values from the function), we can actually plot a smooth function. Note the *add* keyword that will overlay the line on an existing plot, without it a new plot is made. Also note the range goes past the plot window – this is fine as it will be truncated to the existing window.

```
curve(gaus(x, const, mean, sigma), from=0, to=2, add=TRUE, col='red',
lwd=2)
```

Let's look at the original data again with this fit, raising the height threshold to remove the noise altogether. Note that you can plot and return the 'hist' object at one time with the *plot=TRUE*.

```
h = hist(d$area[d$height > 10], br=200, xlim=c(0,2.5), plot=TRUE)
curve(gaus(x, const, mean, sigma), from=0, to=2, add=TRUE, col='red')
```

Clearly, this a single Gaussian does not fit the data. What model might be a better fit? Let's try two Gaussians.

```
model = function(x, const1, mean1, sigma1, const2, mean2, sigma2) {
     g1 = gaus(x, const1, mean1, sigma1)
     g2 = gaus(x, const2, mean2, sigma2)
     g1 + g2
}
```

Perform the fit using all data points (note we now have six free parameters):

```
x = h$mids
y = h$counts
fit = nls(y ~ model(x, const1, mean1, sigma1, const2, mean2, sigma2),
start=list(const1=2500, mean1=1.5, sigma1=0.2, const2=500, mean2=0.5,
sigma2=0.1), trace=TRUE)
```

Extract the values from the fit.

```
const1 = summary(fit)$parameters[1]
mean1 = summary(fit)$parameters[2]
sigma1 = summary(fit)$parameters[3]
const2 = summary(fit)$parameters[4]
mean2 = summary(fit)$parameters[5]
sigma2 = summary(fit)$parameters[6]
```

Overplot the curves, including our model as a dashed line.

```
curve(gaus(x, const1, mean1, sigma1), col='seagreen', add=TRUE,
lwd=2)
curve(gaus(x, const2, mean2, sigma2), col='steelblue', add=TRUE,
lwd=2)
curve(model(x, const1, mean1, sigma1, const2, mean2, sigma2),
col='grey30', add=TRUE, lwd=3, lty='dashed')
```

Plotting Tips

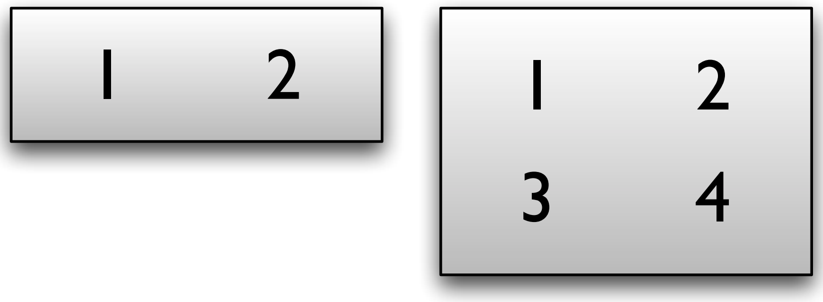
Look at `?par` for most plotting parameters.

Multiple plots in a window

Create a vector describing the layout of the window in rows and columns and pass it to `split.screen()`. For example, one row and two columns would be:

```
split.screen(c(1,2))
```

The screens are implicitly numbered from left to right, top to bottom.



Set the focus to a screen with `screen(n)`, where *n* is the number of the screen, then execute the plot command. The `quartz()` command opens an empty window on the Mac, and `x11()` does the same on Linux.

```
quartz()
split.screen(c(1,2))
screen(1)
hist(rnorm(1000, mean=5, sd=2), xlab="normal distribution")
screen(2)
hist(runif(1000), xlab="uniform distribution")
```

If you execute `split.screen()` on a sub-window, it will be divided accordingly. You can use this to create complex layouts. Note that calling `screen(n)` on a previously drawn plot will erase it by default.

To draw items within a plot:

```
text(x, y, 'string')
segments(x1, y1, x2, y2, col="red") # draw a line segment
rect(x1, y1, x2, y2, cex=1.2, col="blue") # draws a rectangle between
                                            the coordinates
```

```
legend("topright", c("series 1", "series 2"), lwd=c(2, 2),
col=c("red", "blue")) # draw a plot legend
```

Useful parameters:
`lty` - line type (one of "blank", "solid", "dashed", "dotted", "dotdash", "longdash", or "twodash")
`lwd` - line width (default = 1, higher values are thicker)
`cex` - size of text and symbols plotted (as a multiplier)
`col` - color
`pch` - plot symbol, can be a string or number mapping to a symbol (see `?points`)

Color arrays can be generated with built-in functions (`rgb()`, `hsv()`, `hcl()`, `gray()` and `rainbow()`).

```
hist(rnorm(1000, mean=5, sd=2), xlab="normal distribution",
col=rainbow(10))
```

Libraries

There are a large number of external libraries available in R. Most are available in one of these two repositories:
http://cran.r-project.org/web/packages/
http://www.bioconductor.org/install/

Once installed, the library can be loaded with the `library` command. This is analogous to Python's 'import'. It only needs to be called once per session. Once the library command is called, the package's documentation is added to the online help.

Here is a simple example that uses the scatterplot3d library to plot random points generated on a sphere. Note the generated plot can be rotated.

```
library(scatterplot3d)
u = runif(1000, 0, 1)
v = runif(1000, 0, 1)

theta = runif(1000, 0, 2*pi)
phi = asin(runif(1000, 0, 2)-1)

x = cos(phi) * cos(theta)
y = sin(theta) * cos(phi)
z = sin(phi)

plot3d(x,y,z)
```

Error Bars

The built in plot/par functionality doesn't support error bars, but these are easily added to an existing plot using the Hmisc package:

```
errbar(x, y, yplus=y+err, yminus=y-err, add=TRUE)
```

Math Symbols in Text (including plot labels)

Math expressions are created using the `expression()` function, e.g.

```
plot(1,1, xlab=expression(x~~lambda~~60*degree))
```

Here '~' indicates a space. Note the lack of quotes. The alternate form is to concat strings and expressions with '*':

```
plot(1,1, xlab=expression("time (" * mu * "s)"))
```

To use variables within the expression, use `substitute()`:

```
x = 8
y = 1200
plot(1,1, main=substitute(phi~'='~2^c1~c2, list(c1=x, c2=y)))
```

Or use the `bquote()` function:

```
plot(1,1, main=bquote(phi~'='~2^.(x)~.(y)))
```

Plot titles are normally drawn in bold, but using expression turns this off:

```
plot(1,1, main=expression(x^2))
```

You can turn it back on like this:

```
plot(1,1, main=expression(bold(x^2)))
```

To use expression strings in a plot legend:

```
legend("bottomright",expression("Linear fit", "±1.5"*sigma*"  from
fit"))
```

See `demo(plotmath)` and `?plotmath` for more details.

Random Notes

Use paste() to concatenate strings. Be default, strings are separated with a space; use `sep=''` to have no separator (or a different separator).

```
paste('This', 'is', 'a', 'sentence.')
```

Mimic Python's split() function:

```
unlist(strsplit("a=b=c", "="))
```

Note that the last parameter is a regular expression:

```
unlist(strsplit("a.b.c", "\\."))
```

R Graphics Library
http://addictedtor.free.fr/graphiques/thumbs.php

Selecting points in a polygon:
https://stat.ethz.ch/pipermail/r-sig-geo/2006-December/001581.html
http://www.nceas.ucsb.edu/scicomp/GISSeminar/UseCases/PointInPolygonAnalysis/point_in_poly.html

The union of two polygons (even when overlapping) can be done with:

```
p = p1 | p2
```

Unix environment variables

```
Sys.getenv()
```

Can access individual variables as:

```
currdir = getwd()
setwd(Sys.getenv("PROJECT"))
source("xxx.R")
setwd(currdir)
```

Calling command line programs

The 'intern' parameter says to return the output as a string.

```
paste(system("echo $HOME", intern=T),"sourcefile.r",sep="/")
```

To source something in your home directory regardless of the working directory:

```
source(paste(system("echo $HOME", intern=T),"sourcefile.r",sep="/"))
```

Plotting a density line over a plot
This is similar to a histogram, useful in showing the density distribution where there are too many points piled on top of one another. Where 'x' is the range data and 20 is a vertical scaling factor:

```
dens = density(x)
lines(dens$x, dens$y*20/max(dens$y), col='grey60')
```

Pointers for 2D Histograms
See hexbin package.
http://addictedtor.free.fr/graphiques/RGraphGallery.php?graph=139
(smoothscatter)
http://biomserv.univ-lyon1.fr/library/ade4/html/s.kde2d.html
http://www.statmethods.net/graphs/scatterplot.html
help(sunflowerplot)

Writing a plot directly to a file (Mac or Linux)

Postscript:

```
postscript(file="my_plot.ps")
```

<plotting commands>

```
dev.off()
```

EPS:

```
postscript("my_plot.eps", width = 8.0, height = 10.0,
horizontal=FALSE, onefile=FALSE, paper="special",
encoding="TeXtext.enc")
```

<plotting commands>

```
dev.off()
```

PDF:

```
pdf("my_plot.pdf", width = 8.0, height = 10.0, onefile=FALSE,
paper="special")
```

<plotting commands>

```
dev.off()
```

Plotting Date Values

Dates should be plotted as such – don't make a user do the math in their head to figure out the plot. The standard date format is YYYY-MM-DD HH:MM:SS (ISO 8601). If you specify this and cast the data as a date, then you can specify the format to display the dates in the plot command.

```
date_strings = c('2005-07-19 09:10:10', '2005-07-19 12:10:10',
'2005-07-19 19:10:10', '2005-07-20 09:10:10')
plot(as.POSIXct(date_strings), seq(1,4))
```

It's probably better to specify the axis yourself in these cases:

```
plot(as.POSIXct(date_strings), seq(length(date_strings)), xaxt='n')
axis.POSIXct(1, at=seq(as.POSIXct(date_strings[1]),
as.POSIXct(date_strings[NROW(date_strings)]), by="hours"), format="%b
%d %d:%H")
```

Note that "by='hours'" can even be something like "by='6 hours'" – any unit of time:

```
plot(as.POSIXct(date_strings), seq(length(date_strings)), xaxt='n')
axis.POSIXct(1, at=seq(as.POSIXct(date_strings[1]),
as.POSIXct(date_strings[NROW(date_strings)]), by="4 hours"),
format="%b %d %d:%H")
```

See `?strptime` for format options.