Basic Introduction to R

Ontario HPC Summer School Central 2019

Marcelo Ponce

ADVANCED RESEARCH COMPUTING at the UNIVERSITY OF TORONTO

June 24th, 2019

Material for this session

All the material for the HPC Summer School can be found here:

scinet.courses/438

The slides for this class can be found here:

scinet.courses/447

and the Summer School website:
https://support.scinet.utoronto.ca/education/go.php/438/index.php

An introduction to R

Our adventure in R will cover the following:

- Getting R started.
- Primitive types
- Lists
- Vectors
- Matrices, Arrays
- Data frames
- Statistics
- Functions
- *pply: lapply, apply, tapply
- Scripting

If you have questions, please ask.

R history

R has been around for a while, and is well-developed:

- Introduced in 1996 as an evolution of the S language.
- R is designed for exploring and analysing data.
- Home page: http://www.r-project.org.
- Community packages are stored at CRAN - Comprehensive R Archive Network.
- A new full version of R is released each year. We're currently on release 3.6.0

About R

Some important things to know about R:

- R is a scripting language, meaning an interpreter executes commands one line at a time (not a compiled language).
- R can be used interactively, with or without IDE (RStudio).
- R has a large repository of community packages.
- R is all about data analysis: it is not a general purpose language.
  - Several important features (numerics, visualization) are baked into the language, not add-ons.
  - Not as useful outside of number crunching.
- R is designed with interactive data exploration in mind.
  - Lots of surprising things "just work" interactively.
  - But this design can make it a little difficult to debug large non-interactive programs.
- R is closer to functional in approach.

Starting R

How you start the R interpreter depends upon your system:

- Windows: several graphical R interfaces exist (RCommander, Rgui, RStudio). Launch whichever one you have installed.
- Mac: similar to Windows, but running from the command line is also an option.
- Linux: open a terminal. Type "R". Use "q()" to quit.
- web-interface: JuPyteR notebook.

Open up your R interface now. Raise your hand if you don't think it's working. Please follow along by entering the commands on the slides, and playing with the output.

Starting R on the TEACH cluster

Alternatively, you can log into Graham/Cedar/Niagara and run R there.

```
myUSER@mycomp ~>
myUSER@mycomp ~> ssh USERNAME@teach.scinet.utoronto.ca -X
[USERNAME@gra-loginX ~]$
myUSER@mycomp ~> ssh scinetguestXXX@teach.scinet.utoronto.ca -X
[USERNAME@gra-loginX ~]$
[USERNAME@gra-loginX ~]$
[USERNAME@gra-loginX ~]$ module spider r
[USERNAME@gra-loginX ~]$ module load gcc/7.3.0 r/3.5.1
[USERNAME@gra-loginX ~]$
[USERNAME@gra-loginX ~]$ R
>
```

We won't be using parallel R capabilities, so you should be able to just work either on your own laptop, or on a login/development node for this session.

R data types

Once you start your session you will get an interactive prompt:

- Enter commands at the prompt, the interpreter interprets them.
- The "!" is the NOT operator.
- "paste" converts the input to strings, and then concatenates them.
- The "class" function returns names of the classes from which the object inherits.
- Like Python, R uses the # symbol to start comments.

```
>
> a <- 1
> b <- 1.73
> d <- "hello"
> e <- FALSE
> f <- "world"
>
> a + b
[1] 2.73
> !e
[1] TRUE
> paste(d,f)
[1] "hello world"
> class(b)
[1] "numeric"
>
> # a comment
>
```

R data types, continued

R and Python have similar primitive types:

- integer
- "numeric": floating types (as with Python, double precision)
- logicals
- character strings

But there are some differences:

- R: idiomatic assignment operator is "<-".
- logical literals are shoutier (TRUE/FALSE, or T/F).
- variables can have periods in their names.

The strength of R lies in its data structures. Note that, like Python, R is case sensitive ("A" is not the same as "a").

R lists

Lists are the most basic "container" data type in R:

- The "list" function will generate a list from the inputs.
- "str" stands for "structure". It gives a description of the argument.
- In R, the "start:finish" notation returns a sequence running from start to finish, inclusive.

```
> l <- list(a, b, d, e, f, pi)
> str(l)
List of 6
$ : num 1
$ : num 1.73
$ : chr "hello"
$ : logi FALSE
$ : chr "world"
$ : num 3.14
>
> l[[6]]
[1] 3.141593
> l[1:2]
[[1]]
[1] 1

[[2]]
[1] 1.73
>
```

R lists, continued

As with Python lists, the values can be of various types - including lists.
Note:

- Indexing individual items in a list is done with [[ ]].
- Indexing starts at 1, as with most scientific computing languages. (Indices, not offsets.)
- Slicing is done with [start:finish], and the last item is included (unlike Python).
- Note that [[-1]] will return an error message.
- What does slicing with a negative number do - eg, l[-1]? This is very different behaviour than Python.

R named lists

- Named lists allow you to access elements by name, rather than by index.
- If you don't finish your line in R, but hit enter, it will display the "+" symbol, indicating that it's waiting for more input.
- You can access pieces of a named list with the "$".
- The "names" function returns the names of a named list, data frame, etc.

```
>
> named.list <- list(value = 5,
+ word = "text", number = 7.3)
> str(named.list)
List of 3
 $ value : num 5
 $ word  : chr "text"
 $ number: num 7.3
>
> named.list$value
[1] 5
> named.list[["number"]]
[1] 7.3
> names(named.list)
[1] "value"   "word"   "number"
>
```

R named lists, pop quiz!

Pop quiz: add a new entry to your ”named.list” list:

- call the new entry ”mybool”
- give the entry a value of FALSE

R named lists, pop quiz!

Pop quiz: add a new entry to your ”named.list” list:

- call the new entry ”mybool”
- give the entry a value of FALSE

```
>
> names(named.list)
[1] "value"   "word"   "number"
>
> named.list$mybool = F
>
> str(named.list)
List of 4
 $ value :  num 5
 $ word  :  chr "text"
 $ number:  num 7.3
 $ number:  logi FALSE
>
```

```
>
> names(named.list)
[1] "value"   "word"   "number"
>
> named.list['mybool'] = F
>
> str(named.list)
List of 4
 $ value :  num 5
 $ word  :  chr "text"
 $ number:  num 7.3
 $ number:  logi FALSE
>
```

R vectors

Unlike Python, vectors are built into the language:

- Homogeneous (same type)
- Compact
- Not nested
- Like numpy vectors

```
> a <- c(1,2,3)
> b <- c("Hello", "World", "From", "A", "Vector")
> str(b)
chr [1:5] "hello" "World" "From" "A" "Vector"
> d <- 1:17
> str(d)
int [1:17] 1 2 3 4 5 6 7 8 9 10 ...
>
```

The "c" command combines values in to a vector or list.

R vectors, continued

There are many ways to create vectors in R:

```
>
> 1:10
[1] 1 2 3 4 5 6 7 8 9 10
> seq(2, 20, 4)
[1] 2 6 10 14 18
> paste("A", 1:5, sep = "")
[1] "A1" "A2" "A3" "A4" "A5"
> rep(letters[1:5], 3)
[1] "a" "b" "c" "d" "a" "b" "c" "d" "a" "b" "c" "d"
> is.vector(1:10)
[1] TRUE
>
```

And others.

Using sample to create vectors

The "sample" function samples from a vector:

- By default, sample removes the previously sampled elements from the set.
- As such, you can't sample more than the set size.
- To keep sampled elements in the set, use the "replace = TRUE" argument.
- If you want consistent samples (for testing and debugging purposes), set the seed to the same value before sampling.

```
>
> sample(1:10, 4)
[1] 10 5 4 7
> sample(1:10, 4)
[1] 2 6 10 3
> sample(1:10, 4, replace = TRUE)
[1] 7 10 3 7
>
> set.seed(2)
> sample(1:10, 4, replace = TRUE)
[1] 2 8 6 2
> set.seed(2)
> sample(1:10, 4, replace = TRUE)
[1] 2 8 6 2
>
```

Diagnostic functions

R is loaded with functions for figuring out what things are:

- The "typeof" function returns the type or storage mode of the object.
- To check to see if something is a vector, use "is.vector".
- The "summary" function returns a summary of the object's properties. It is often used in the context of summarizing the results of model-fitting functions.

```
> typeof(l)
[1] "list"
> is.vector(l)
[1] TRUE
> summary(d)
 Min. 1st Qu.  Median  Mean 3rd Qu.  Max.
    1       5       9     9      13    17
>
```

Note that lists are considered vectors in R.

Appending to vectors

Unlike numpy arrays in Python, you can add elements to the end of existing vectors:

- Use sparingly! It's better to fill the whole length you need first, using seq() or rep(), rather than set elements as needed.
- Increasing length of vector/list one at a time is:
  - slow
  - at risk of causing memory problems

```
>
> # probably bad, certainly slow
> a <- c(1,2,3)
> a <- c(a,4)
> a <- c(a,5)
> a
[1] 1 2 3 4 5
>
> # probably bad,
> # certainly funny-looking
> a[length(a) + 1] <- 6
> a
[1] 1 2 3 4 5 6
>
```

Appending to vectors, continued

It's much better to allocate your vector once, and then set the elements as you go.

```
>
> # good
> a <- rep(0,5)
> a[4] <- 4
> a[5] <- 5
> a
[1] 0 0 0 4 5
>
```

If you extend your vector one element at a time, the contents of the vector must be copied each time the vector is extended. This is slow.

R vectors behave intuitively

As with numpy vectors, most operations happen automatically on vectors:

```
> a <- 1:5 + 1
> a
[1] 2 3 4 5 6
> b <- rep(2.,5)
> a * b
[1] 4 6 8 10 12
> sin(a)
[1] 0.9092974 0.1411200 -0.7568025 -0.9589243 -0.2794155
>
> d <- sample(c(TRUE,FALSE), 5, replace = TRUE)
> d
[1] TRUE FALSE FALSE TRUE FALSE
> !d
[1] FALSE TRUE TRUE FALSE TRUE
> a[d]
[1] 2 5
>
```

More slicing

You can slice with:

- vectors of integers
- ranges (which are really just vectors of integers)
- vectors of booleans (which pull out the values corresponding to TRUE, an in the last example on the last slide)

```
>
> a[2:4]
[1] 3 4 5
> a[c(1,2,4)]
[1] 2 3 5
> a[-c(1,2,3)]
[1] 5 6
> a[seq(1,5,2)]
[1] 2 4 6
>
```

Boolean operators

- The statement "a < 4" returns where this statement is TRUE.
- The "==" is the equivalence test ("is this equal to this?"). "!=" does the opposite.
- The & symbol is the "AND" operator.
- The | symbol is the "OR" operator.
- The "which" command will give the indices of the TRUE entries.

```
> a < 4
[1] TRUE TRUE FALSE FALSE FALSE
> a[a < 4]
[1] 2 3
>
> b <- seq(1, 60, 13)
> b
[1] 1 14 27 40 53
>
> b == 38
[1] FALSE FALSE FALSE FALSE FALSE
> (b > 5) & (b < 50)
[1] FALSE TRUE TRUE TRUE FALSE
> (b < 10) | (b > 30)
[1] TRUE FALSE FALSE TRUE TRUE
>
> which((b < 10) | (b > 30))
[1] 1 4 5
```

Not Available (NA)

Let's try extending the "a" vector by another 3 items, and only set the last one:

```
> a
[1] 2 3 4 5 6
>
> length(a)
[1] 5
>
> a[length(a) + 3] <- 9
>
> a
[1] 2 3 4 5 6 NA NA 9
>
```

NA (Not Available) is used to represent missing or invalid data. The right thing to do with NAs will depend on the application, but we will often need to deal with NAs specifically.

NA, continued

We can use the "is.na" function to pick out NAs:

```
>
> is.na(a)
[1] FALSE FALSE FALSE FALSE FALSE TRUE TRUE FALSE
>
> a[!is.na(a)]
[1] 2 3 4 5 6 9
>
> a[is.na(a)] <- -1
>
> a
[1] 2 3 4 5 6 -1 -1 9
>
```

Recall that the "!" symbol is the "NOT" operator.

NA, continued more

Most math operations on NAs will return NA; but most generally have built-in optional ways of dealing with them.

```
>
> a[a == -1] <- NA
>
> a
[1] 2 3 4 5 6 NA NA 9
>
> sum(a)
[1] NA
>
> sum(a, na.rm = TRUE)
[1] 29
>
```

Help!

But what if you don't know how to use the function, or don't know the optional arguments? You can use the help function, which is also accessed using '?':

```
>
> help(sum)
.
.
.
>
> ? sum
.
.
.
>
```

Press 'q' to exit the help page (on a Linux system).

Help, continued

Some authors of functions have even been kind enough to give you examples of how to use their functions:

```
>
---------------------------------------------------------------------------
> example(sum)

sum> ## Pass a vector to sum, and it will add the elements together.
sum> sum(1:5)
[1] 15

sum> ## Pass several numbers to sum, and it also adds the elements.
sum> sum(1, 2, 3, 4, 5)
[1] 15
.
.
.
sum> ## ...  unless we exclude missing values explicitly:
sum> sum(1:5, NA, na.rm = TRUE)
[1] 15
---------------------------------------------------------------------------
>
```

Pop quiz!

Create a vector of 100 elements whose elements are randomly drawn from a normal distribution with a mean of 200 and a standard deviation of 20. The function that you are looking for is called 'rnorm'. Use 'help' to figure out how to use it.

Pop quiz!

Create a vector of 100 elements whose elements are randomly drawn from a normal distribution with a mean of 200 and a standard deviation of 20. The function that you are looking for is called 'rnorm'. Use 'help' to figure out how to use it.

```
>
> mynorm <- rnorm(100, 200, 20)
>
> length(mynorm)
[1] 100
>
> mean(mynorm)
[1] 199.5294
>
> sd(mynorm)
[1] 19.12821
>
```

Matrices, arrays

Vectors are generalized into matrix and array types - matrices are 2D and are created by specifying at least one of the two dimensions:

- The "`matrix`" function returns a 1D vector by default.
- To make it 2D, specify either "`nrow`" or "`ncol`", but make sure it divides the number of elements evenly.
- The "`rnorm`" function draws from the normal distribution.

```
> A <- matrix(rnorm(9), nrow = 3)
>
> class(A)
[1] "matrix"
> A
            [,1]       [,2]       [,3]
[1,] -0.02574689  0.1676856 -1.2806814
[2,]  0.17226639 -0.7561610 -1.0730775
[3,] -0.35919061  0.2674792 -0.6362443
>
```

Matrices, continued

As you might expect, you can use matrices to do matrix math:

- The "`solve`" function solves the equation $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a matrix and $\mathbf{x}$ and $\mathbf{b}$ are vectors.
- If available, "`solve`" will use a LAPACK routine on your machine to solve this system. That means it's fast.
- What's with the %*% symbol?

```
>
> b <- 1:3
>
> A %*% b
          [,1]
 [1,] -3.532420
 [2,] -4.559288
 [3,] -1.732965
>
> solve(A,b)
[1] -8.943326 -3.229387 -1.023876
>
```

R special operators

R contains a number of "special operators":

- %*% is matrix multiply.
- %o% is outer product.
- %x% is Kronecker product.
- %% is the modulus.
- %/% is integer division.
- And there are others as well.

```
> A
             [,1]       [,2]       [,3]
[1,] -0.02574689  0.1676856 -1.2806814
[2,]  0.17226639 -0.7561610 -1.0730775
[3,] -0.35919061  0.2674792 -0.6362443
```

```
> a <- 1:3;   b <- 3:5
```

```
> a * b
[1] 3 8 15
```

```
> a %*% b
     [,1]
[1,]   26
```

```
> a %o% b
     [,1] [,2] [,3]
[1,]    3    4    5
[2,]    6    8   10
[3,]    9   12   15
```

```
> a %x% b
[1] 3 4 5 6 8 10 9 12 15
```

```
> 9 %% 4
[1] 1
```

```
> 9 / 4
[1] 2.25
```

```
> 9 %/% 4
[1] 2
```

More matrix operations

A few more miscellaneous matrix operations:

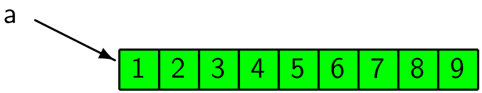
- The "`dim`" command can be used to change the dimensions of a matrix.
- Note that R is *column major*. Do you see how this manifests itself in the way the elements get arranged?
- The "`t`" command gives the transpose of a matrix.

```
> a = sample(c(T,F), 6, replace = T)
> a
[1] FALSE TRUE TRUE TRUE FALSE FALSE
> dim(a) <- c(3,2)
> a
      [,1]  [,2]
 [1,] FALSE  TRUE
 [2,]  TRUE FALSE
 [3,]  TRUE FALSE
> dim(a) <- c(2,3)
> a
      [,1] [,2]  [,3]
 [1,] FALSE TRUE FALSE
 [2,]  TRUE TRUE FALSE
> t(a)
      [,1]  [,2]
 [1,] FALSE  TRUE
 [2,]  TRUE  TRUE
 [3,] FALSE FALSE
```

R is column major

A 1D array is linear in memory, but so is a 2D array.

```
> a <- 1:9
> dim(a) <- c(3,3)
> a
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> a[2,3]
[1] 8
>
```



This is how the numbers are stored in memory. If you loop over the 2D array, which index should you loop over (first or second)?

Arrays

Arrays can have any rank (not necessarily just 2D).

Arrays are just high-dimension versions of matrices.

```
> B <- array(1:12, c(2, 3, 2))
> class(B)
[1] "array"
> B
, , 1
      [,1] [,2] [,3]
 [1,]    1    3    5
 [2,]    2    4    6


, , 2
      [,1] [,2] [,3]
 [1,]    7    9   11
 [2,]    8   10   12
> C <- array(1:12, c(2, 6))
> class(C)
[1] "matrix"
>
```

Built-in datasets

R contains built-in datasets that can be used for practicing.

```
> data()
Data sets in package datasets:

 AirPassengers            Monthly Airline Passenger Numbers 1949-1960
 BJsales                  Sales Data with Leading Indicator
 BJsales.lead (BJsales)   Sales Data with Leading Indicator
 BOD                      Biochemical Oxygen Demand
 CO2                      Carbon Dioxide Uptake in Grass Plants
 .
 .
 .
```

```
>
```

```
> str(faithful)
data.frame':   272 obs.  of 2 variables:
$ eruptions: num 3.6 1.8 3.33 2.28 4.53 ...
$ waiting  : num 79 54 74 62 85 55 88 85 51 85 ...
```

```
>
```

Type 'q' to get out of the 'data' menu.

Data frames

Data frames are a building block for data analysis in R; in Python, pandas data frames are based on them.

A data frame is a list of vectors. Each vector (a column of the frame) has the same length, but different columns may have different types.

Thus every row of the frame is a list.

```
> data <- trees
> class(data)
[1] "data.frame"
> str(data)
'data.frame':  31 obs.  of 3 variables:
 $ Girth : num  8.3 8.6 8.8 10.5 10.7 10.8 11 11 11.1 11.2 ...
 $ Height: num  70 65 63 72 81 83 66 75 80 75 ...
 $ Volume: num  10.3 10.3 10.2 16.4 18.8 19.7 15.6 18.2 22.6 19.9 ...
>
```

Slicing data frames

Accessing parts of the data frame makes a lot more sense when you remember it's just a list of vectors.

```
>
> names(data)
[1] "Girth" "Height" "Volume"
>
> data[1:3,"Girth"]
[1] 8.3 8.6 8.8
>
> data[2,]
  Girth Height Volume
2   8.6     65   10.3
>
> data$Height[4:5]
[1] 72 81
>
```

Updating data frames

Performance tip: While you can update individual items in a data frame via slicing:

```
> data[1, "Girth"] <- 8.7
```

It turns out this is extremely slow and memory intensive, for boring reasons. If you have do a number of such updates, try to minimize the number of updates to the data frame.

It's better to extract a column, update it, and then update the whole column at once:

```
> Girth <- data$Girth
> Girth <- 2. * Girth + 1
> data$Girth <- Girth
```

Getting external data

Getting data from online is as simple as putting in the URL:

```
> data = read.csv("https://support.scinet.utoronto.ca/~mponce/courses/
datasets/Dental-2011-2012.csv")
```

```
> str(data)
 $ Quarter: Factor w/ 3 levels "Q1","Q2","Q3":  1 1 1 1 1 1 1 1 ...
                  ⋮
 $ Total  : num 9317 14948 23136 18546 40536 ...
```

```
> colnames(data)
 [1]  "Quarter"                    "Year"
 [3]  "Data"                       "CCG_Code"
 [5]  "AT_CODE"                    "Region_Code"
 [7]  "Patient_type"               "Band_1"
 [9]  "Band_2"                     "Band_3"
 [11]  "Urgent_Occasional"         "Free___Arrest_of_Bleeding"
 [13]  "Free___Bridge_Repairs"     "Free___Denture_Repair"
 [15]  "Free___Prescription_Issue" "Free___Removal_of_sutures"
 [17]  "Total"
```

The original URL for this data is

`http://www.hscic.gov.uk/catalogue/PUB07163/nhs-dent-stat-udas-eng-2011-2012-anx5.csv`

Getting external data, continued

We can do some initial exploration of the data by plotting it.

- boxplot, shockingly, creates a boxplot.
- The tilde symbol indicates that we are creating a "formula".
- The boxplot uses this the determine what to plot against what.

```
>
> boxplot(data$Total ~  data$Patient_Type)
>
> boxplot(data$Urgent_Occasional ~  data$Patient_Type)
>
```

Excel files

You can also read Excel files using R, though not out of box.

There are many packages out there that will do this, but you'll need to download them separately.

- gdata
- XLConnect
- xlsx
- readXL

```
>
> install.packages("xlsx")
>
> library(xlsx)
>
> data = read.xlsx('datalist.xls',
+     sheetName = 'Sheet1')
>
> names(data)
[1] "Soc_Sec_Num" "Name"
"First.name"
[4] "Gender" "Title" "Salary"
"Category"
>
```

R and statistics

Without a doubt, one of R's strongest features is its statistical libraries.

- Every distribution is there.
- Everything is baked in.
- If it's not baked in, someone's already written your function or distribution for you. Go find it.
- Many of the standard functions are written in R itself, so you can go read the source code and determine exactly what has been done.
- (The name of the project is "The R Project for Statistical Computing".)

You could make a worse choice for statistical programming language.

R and statistics, the normal distribution

We start with our old friend, the normal distribution:

- 'dnorm(x)' returns the value of the normal distribution at x.
- 'pnorm(x)' returns the cumulative distribution function.
- 'qnorm(x)' returns the quantile function.

```
>
> x <- seq(-4, 4, 0.01)
>
> plot(x, dnorm(x), type = "l")
>
> plot(x, pnorm(x))
>
> qnorm(0.25)
[1] -0.6744898
>
> qnorm(c(0.025, 0.975))
[1] -1.959964 1.959964
>
```

R and statistics, a t-test example

Let's perform a Student's t test:

- Consider the two samples, given by 'A' and 'B'.
- Question: what is the probability that these two samples are drawn from the same distribution?
- The answer: about 8%.

```
> A <- array(c(97, 98, 78, 80, 81, 84,
+ 85, 85, 86, 94, 100, 102, 103, 104))
> B <- array(c(76, 87, 89, 90, 94, 96,
+ 98, 99, 100, 106, 109, 112, 113, 105))
>
> t.test(A, B, var.equal = T)

Two Sample t-test

data:  A and B
t = -1.8423, df = 26, p-value = 0.07686
alternative hypothesis:  true difference
in means is not equal to 0
95 percent confidence interval:
-14.6589062 0.8017634
sample estimates:
mean of x mean of y
91.21429 98.14286
>
```

Basic building blocks of Programming

All programming languages have some basic “building” blocks

- looping constructs
- conditionals to handle decision making
- the ability to group commands into functions or modules

Basic building blocks of Programming

All programming languages have some basic “building” blocks

- looping constructs
- conditionals to handle decision making
- the ability to group commands into functions or modules

➡ R has all these features available

Basic building blocks of Programming

All programming languages have some basic "building" blocks

- looping constructs
- conditionals to handle decision making
- the ability to group commands into functions or modules

➡ R has all these features available

Code Blocks

- Normally, R treats each line as a statement and executes it immediately
- Using { and } can wrap multiple lines into a single statement
- R will run a code block at the entry of a new line after the closing }

Functions in R

R tends towards functional programming; functions aren't normally called for side effects.

Functions return whatever values are needed.

Typical upsides/downsides to functional programming:

- Easy to read, understand, debug.
- Makes parallelism somewhat easier.
- Requires lots of temporary memory (copies being made).

R functions

What do the following do?

```
> a <- 1:5
```

```
> doubleVector <- function(x) {x <- x * 2}
```

```
> doubleVector2 <- function(x) {x * 2}
```

```
> a
[1] 1 2 3 4 5
```

```
> doubleVector(a)
```

```
> a
```

R functions

What do the following do?

```
> a <- 1:5
> doubleVector <- function(x) {x <- x * 2}
> doubleVector2 <- function(x) {x * 2}
> a
[1] 1 2 3 4 5
> doubleVector(a)
> a
[1] 1 2 3 4 5
>
```

R functions

What do the following do?

```
> a <- 1:5
> doubleVector <- function(x) {x <- x * 2}
> doubleVector2 <- function(x) {x * 2}
> a
[1] 1 2 3 4 5
> doubleVector(a)
> a
[1] 1 2 3 4 5
> a <- doubleVector2(a)
> a
```

R functions

What do the following do?

```
> a <- 1:5
> doubleVector <- function(x) {x <- x * 2}
> doubleVector2 <- function(x) {x * 2}
> a
[1] 1 2 3 4 5
> doubleVector(a)
> a
[1] 1 2 3 4 5
> a <- doubleVector2(a)
> a
[1] 2 4 6 8 10
>
```

R passes by value, not by reference, unless the argument is not being modified.

R return statement

Did you notice the lack of "return" statement in the function definition?

- R tends toward functional programming.
- The "return" statement tends to be unnecessary in this context, as state changes are discouraged.
- However, you may include it for clarity.

```
>
> doubleVector2 <- function(x)
+ {x * 2}
>
> doubleVector3 <- function(x) {
+ return(x * 2)
+ }
>
> doubleVector2(a)
[1] 2 4 6 8 10
> doubleVector3(a)
[1] 2 4 6 8 10
>
```

R iterators

Iterators are a fundamental part of Python, but are a recent addition to R.

- An iterator allows a programmer to traverse a list/array/container without loading the entire container into memory first.
- If I need to loop over one million elements, I don't want to load all of the elements into memory unless I really need to.
- Use the 'iterators' package.

```
>
> library(iterators)
>
> names <- c("Bob", "Mary",
+ "Jack", "Jane")
>
> str(names)
chr [1:4] "Bob" "Mary" "Jack" "Jane"
>
> inames <- iter(names)
> str(inames)
List of 4
$ state    :<environment: 0x1a5496b8>
$ length   :  int 4
$ checkFunc:function (...)
$ recycle  :  logi FALSE
- attr(*, "class")= chr [1:2]
"containeriter" "iter"
>
```

R iterators

Once you have your iterator you can advance to the next element using "nextElem".

In what can only be described as a bizarre design decision, iterators do not work with for loops. Nor is there an easy way to create your own iterating functions.

You can, however, use iterators with "foreach". We'll discuss that this afternoon.

```
>
> nextElem(inames)
[1] "Bob"
>
> nextElem(inames)
[1] "Mary"
>
> people <- data.frame(names = names,
+ ages = c(17, 23, 41, 19))
>
> ipeople <- iter(people, by = "row")
>
> nextElem(ipeople)
[1] "Bob" 17
>
```

R loops

R has three types of loops:

- The "for" loop. Note that the list or vector being looped over can be of any type.
- The "while" loop.
- The "repeat" loop (not illustrated here). The repeat loop requires the use of a "break" statement; you may as well use a while loop.

```
>
> for (i in list('cow', 1, F)) {
+ print(i) }
[1] "cow"
[1] 1
[1] FALSE
>
> i <- 1
> while(i < 4) {
+ print(i)
+ i <- i + 1 }
[1] 1
[1] 2
[1] 3
>
```

Conditionals

R has the usual types of conditionals:

- The "if" conditional.
- The "switch" conditional. This is analogous to the "case" conditional seen in other languages.

```
>
> for (i in 1:3) {
+ if (i < 2) {
+ print(i) }
}
[1] 1
>
> for (animal in c('cow', 'pig', 'sheep')) {
+ switch(animal,
+ cow = { print('moo') },
+ sheep = { print('baaa') },
+ pig = { print('oink') },
+ stop("Wrong type of animal.")
+ )
+ }
[1] "moo"
[1] "oink"
[1] "baaa"
>
```

The 'apply' family of functions

The apply family of functions make it very easy, and fast, to repeatedly apply a function to a lot of individual elements.

Many parallel routines are parallel versions of these higher-level functions.

- `lapply`: apply a function to each element of a list/vector.
- `sapply`: simpify the `lapply` return list to a vector or array if possible.
- `apply`: apply a function to rows, columns, or elements of an array.
- `tapply`: apply a function to subsets of a list/vector.
- `mapply`: apply a function to the "transpose" of a list. Pass two lists of length three; apply function to first items of lists, then second, then third.

lapply

The function "lapply" repeatedly applies a function to each element of a list or vector. Let's say we wanted to show that as N grows larger, the mean of N normally distributed random numbers tended to zero.

```
> mean.n.rnorm <- function(n) return(mean(rnorm(n)))
> ns <- c(1, 10, 100, 1000)
> lapply(ns, mean.n.rnorm)
[[1]]
[1] -0.2720619

[[2]]
[1] 0.1122716

[[3]]
[1] 0.1562597

[[4]]
[1] -0.0007317103
>
```

lapply, continued

We could even do this in two steps, applying rnorm to the list of ns, and then mean to the list of vectors:

```
> ns <- c(1, 10, 100, 1000)
```

```
> random.nums <- lapply(ns, rnorm)
```

```
> means <- lapply(random.nums, mean)
```

```
> means
[[1]]
[1] -1.143101

[[2]]
[1] 0.1152926

[[3]]
[1] -0.02110213

[[4]]
[1] 0.004479629
```

sapply

And we can get that final result as a sometimes-more-convenient vector rather than list with "`sapply`":

```
>
> ns <- c(1, 10, 100, 1000)
> random.nums <- lapply(ns, rnorm)
> means <- sapply(random.nums, mean)
> means
[1] -1.143100822 0.115292582 -0.021102134 0.004479629
> means * means
[1] 1.306679e+00 1.329238e-02 4.453001e-04 2.006707e-05
>
```

sapply/vapply

Performance tip: if you know the size and type that `sapply` will return, create such a vector/matrix and use vapply, passing it the example object as the third parameter (everything else stays the same). This can be substantially faster, and more memory-efficient for large outputs.

apply

The 'apply' function is used on matrices and arrays. In this case, it's just easier to demonstrate:

```
>
> A <- matrix(1:9, nrow = 3, ncol = 3)
> A
      [,1] [,2] [,3]
 [1,]    1    4    7
 [2,]    2    5    8
 [3,]    3    6    9
> apply(A, MARGIN = 1, max)   # max of each row
[1] 7 8 9
> apply(A, MARGIN = 2, max)   # max of each col
[1] 3 6 9
>
```

apply, continued

`apply` applies a function to the rows (`MARGIN = 1`) or columns (`MARGIN = 2`) of an array (also relevant: `rowSums`, `colSums`).

You can also apply it to each element by using `MARGIN = 1:2`.

Here we square each element of the array:

```
> A <- matrix(1:9, nrow = 3, ncol = 3)
> A
     [,1] [,2] [,3]
[1,]    1    4    7
[2,]    2    5    8
[3,]    3    6    9
> apply(A, MARGIN = 1:2, function(x) {x**2})   # square of each item
     [,1] [,2] [,3]
[1,]    1   16   49
[2,]    4   25   64
[3,]    9   36   81
>
```

tapply

Finally, `tapply` is also a lot easier to just demonstrate than explain:

```
> data <- morley
>
> str(data)
'data.frame':  100 obs.  of 3 variables:
 $ Expt : int  1 1 1 1 1 1 1 1 1 1 ...
 $ Run  : int  1 2 3 4 5 6 7 8 9 10 ...
 $ Speed: int  850 740 900 1070 930 850 950 980 980 880 ...
>
> tapply(data$Speed, data$Expt, mean)
    1     2     3     4     5
909.0 856.0 845.0 820.5 831.5
>
```

In the above, `tapply` takes the Speed values, splits them up into a list of vectors by value of Expt, and then applies mean to each vector.

tapply/split

Try playing with 'split' to understand splitting:

```
>
> split(data$Speed, data$Expt)
```

tapply/split

Try playing with 'split' to understand splitting:

```
>
> split(data$Speed, data$Expt)
$`1`
[1]  850  740  900 1070  930  850  950  980  980  880 1000  980  930  650  760  810 1000
[18] 1000  960  960

⋮

$`5`
[1]  890  840  780  810  760  810  790  810  820  850  870  870  810  740  810  940  950  800
[19]  810  870
>
```

Pop quiz!

Consider the "DNase" dataset, which contains data obtained during the development of an ELISA assay for the recombinant protein DNase.

```
>
> str(DNase)
Classes nfnGroupedData, nfGroupedData, groupedData and 'data.frame': 176 obs.
of 3 variables:
$ Run    : Ord.factor w/ 11 levels "10"<"11"<"9"<..:  4 4 4 4 4 4 4 4 4 4 ...
$ conc   : num 0.0488 0.0488 0.1953 0.1953 0.3906 ...
$ density: num 0.017 0.018 0.121 0.124 0.206 0.215 0.377 0.374 ...
:
>
```

Using `tapply`, calculate the mean and standard deviation of the optical density measurements for the given values of protien concentration.

Pop quiz!, continued

Using `tapply`, calculate the mean and standard deviation of the optical density measurements for the given values of protien concentration.

Pop quiz!, continued

Using `tapply`, calculate the mean and standard deviation of the optical density measurements for the given values of protien concentration.

```
>
> tapply(DNase$density, DNase$conc, mean)
 0.04882812  0.1953125   0.390625    0.78125     1.5625      3.125
 0.05331818 0.15095455 0.23972727 0.40677273 0.66631818 1.03772727
       6.25       12.5
 1.42859091 1.76986364
>
> tapply(DNase$density, DNase$conc, sd)
 0.04882812  0.1953125   0.390625    0.78125     1.5625      3.125
 0.02756358 0.02429085 0.02628268 0.02755243 0.03057604 0.03789985
       6.25       12.5
 0.06917036 0.08462258
>
```

Saving your results

Inevitably you will need to save the results of your analysis. Let's take a look at the humble ".csv" file.

```
[USERNAME@gra-loginX myUSER]$ ls -sh1 airOT2010.*
151M airOT2010.RDS
154M airOT2010.Rdata
1.4G airOT2010.csv
[USERNAME@gra-loginX myUSER]$
```

```
[USERNAME@gra-loginX myUSER]$ Rscript timeexamples.R
[1] "Reading Rdata file"
  user   system  elapsed
 8.564    0.634    9.225
[1] "Reading RDS file"
   user   system  elapsed
 11.045    0.598   11.685
[1] "Reading CSV file"
    user   system  elapsed
 141.143    3.167  144.440
[USERNAME@gra-loginX myUSER]$
```

A note on file formats

CSV (Comma Separated Value) – or any text-based format – is the worst possible format for quantitative data. It manages the trifecta of being:

- Slow to read.
- Huge in size.
- Inaccurate.

Converting floating point numbers back and forth between internal representations and strings is slow and prone to truncation errors.

Use binary formats whenever possible. The ".Rdata" format is a bit prone to change; ".RDS" is modestly better. Portable formats like HDF5 (for data frames) or NetCDF4 (for matrices and arrays) are compact, accurate, fast (though not as fast as .Rdata/.RDS), portable, and can be read by tools other than R.

Using 'save' and 'load'

The simplest way to save and retrieve data:

- You can save variables using the 'save' function.
- To load saved data, use 'load'.
- Note that your loaded data will overwrite any existing variables of the same name.

```
>
> save(label.names, days.of.week, file = "mydata.Rdata")
>

exit and come back

>
> load("mydata.Rdata")
> print(label.names)
[1] "1" "2" "3" "4" "5" "6" "7"
>
```

Using sink

Sink allows you to redirect the output of R to a file:

- This is handy to save development steps.
- Use `sink('filename')` to start the sink.
- Use `sink(NULL)` to stop it.
- You can use the "`split = T`" option to send the output to both the file and the terminal.
- Note that sink only saves the output of the commands, not the commands themselves.

```
> 2 + 5
[1] 7
> sink('output2.txt')
> print('Hello')
> print('Do a bunch of stuff')
> 2 + 5
> sink(NULL)
>
> 2 + 5
[1] 7
> quit()
myUSER@mycomp ~>
myUSER@mycomp ~> cat output2.txt
[1] "Hello"
[1] "Do a bunch of stuff"
[1] 7
myUSER@mycomp ~>
```

Workspace management

You can also save the state of your R session:

- You are working in a "workspace". To save your workspace for next time, use save.image(). This will put your image in a file named ".Rdata".
- To load a previous workspace, use 'load'.
- Note that your loaded image will append to, and overwrite, you current workspace.
- R will ask if you want to save your workspace when you try to exit.

```
> # do a bunch of stuff
> save.image()
> # or alternatively
> save.image(file = 'myimage.Rdata')
>
> load("myimage.Rdata")
>
```

Installing R packages

R makes it crazy-easy to install packages which you don't already have.

- To load a non-default library, use the 'library' function.
- To install non-standard packages, use 'install.packages'.

```
> library('fun')
Error in library("fun") : there is no package called fun
> 
> install.packages("fun")
Installing package into '/home/s/scinet/myUSER/R'
(as 'lib' is unspecified)
--- Please select a CRAN mirror for use in this session ---
.
.
.
** testing if installed package can be loaded
* DONE (fun)
> 
> library('fun')
> 
```

Plotting in R

R has a tonne of plotting options. Let's start with 'plot':

- If one argument is specified, it plots the argument against value index. With two arguments it takes the first as the 'x' axis.
- By default, every time 'plot' is run it writes over the previous plot.
- To add a line to an existing plot, use "lines".
- If you want customized axis labels, they can add them after you make the plot, or at plot time.

```
>
> x <- 1:100 / 100.
>
> plot(sin(x * 4 * pi))
>
> plot(x, sin(x * 4 * pi))
>
> plot(x, sin(x * 4 * pi),
+ type = "o", col = "blue",
+ axes = F)
>
> lines(x, sin(x * 2 * pi),
+ col = 'red', type = 'o')
>
> title(xlab = "time [s]")
>
```

Plotting in R, continued

The plotting parameters can be set before plotting using the 'par' function. Some useful options include

- par(new = TRUE). This will keep R from overwriting your previous plot, which is the default behaviour.
- par(family = 'HersheySans'), change the font family to a vector-drawn font. Always use vector-drawn fonts for publication-quality plots.
- par(font = 2), change the font format. 1 - default, 2 - bold, 3 - italic, 4 - bold italic.
- par(ann = FALSE), do not annotate the plot. In this case you must label your axes after the plotting function is called.
- par(mfrow = c(2,2)), make a 2 x 2 plots. Allows you to put several plots together.
- Many other options.

Plotting in R, continued more

Name it and you can plot it:

- hist: plot a histogram.
- pie: plot a pie chart.
- Note that 'hist' will bin your data, but 'pie' won't.

- The 'table' function creates a table, which gives counts of entries.
- scatterplots, 3D plots, box plots, log plots, contour plots...

```
> data <- airquality
> hist(data$Ozone)
>
> bad.days <- data$Ozone > 20.0
> table(data$Month[bad.days])
 5  6  7  8  9
11  6 22 23 17
> bad.months <- names(table(data$Month[bad.days]))
> bad.month.counts <- as.vector(table(data$Month[bad.days]))
> pie(bad.month.counts, label = bad.months)
```

Saving your plots

Once you have your plot, you will eventually need to save it. The commands for saving your files depends upon the type of file you want:

- bmp(filename = 'myfile.bmp').
- jpeg(filename = 'myfile.jpeg').
- png(filename = 'myfile.jpeg').
- tiff(filename = 'myfile.tiff').
- pdf(file = 'myfile.pdf').

The problem with these functions is that you need to invoke them before you start making your plot. You then invoke "dev.off()" to stop saving the image.

Alternatively, you can create your image and then use one of the dev.copy(...), dev.copy2pdf(...), etc. commands.

R data analysis, an example

Let's use an R package to build a decision tree. Let's use the Iris dataset.

- We first randomly split the iris dataset, 70/30, into training and test datasets.
- We then separate the relevant values from the dataset.

```
>
> str(iris)
'data.frame':  150 obs.  of 5 variables:
 $Sepal.Length: num 5.1 4.9 4.7 4.6 5 ...
 $Sepal.Width : num 3.5 3 3.2 3.1 3.6 ...
 $Petal.Length: num 1.4 1.4 1.3 1.5 1.4 ...
 $Petal.Width : num 0.2 0.2 0.2 0.2 0.2 ...
 $Species     : Factor w/ 3 levels ...
>
> ind <- sample(1:2, nrow(iris),
+ replace = T, prob = c(0.7, 0.3))
> trainData <- iris[ind == 1,]
> testData <- iris[ind == 2,]
>
```

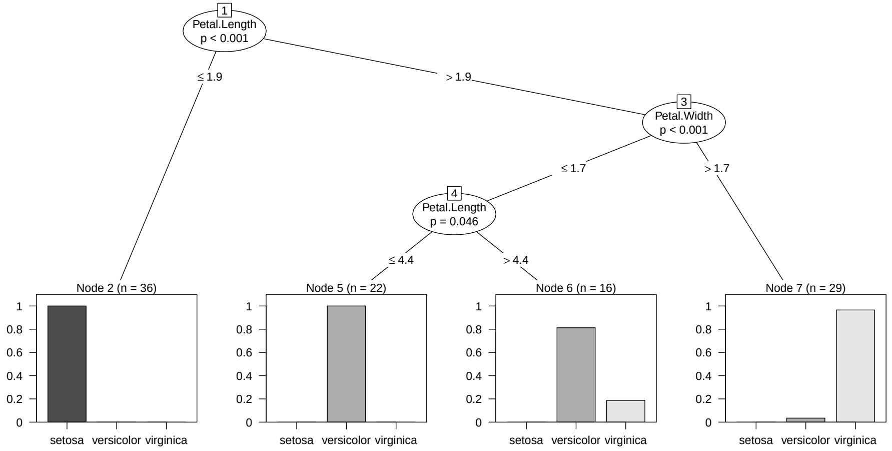
R analysis, an example, continued

Now that the data's split up, we're ready to generate the tree.

- Load the 'party' library.
- Create our 'formula'.
- Generate the decision tree.
- Check the result against the training data.
- Pretty good fit!
- Plot the result.

```
> library(party)
>
> myFormula <- Species ~ Sepal.Length +
+ Sepal.Width + Petal.Length + Petal.Width
>
> iris_tree <- ctree(myFormula,
+ data = trainData)
>
> table(predict(iris_tree),
+ trainData$Species)
            setosa versicolor virginica
 setosa         36          0         0
 versicolor      0         35         3
 virginica       0          1        28
>
> plot(iris_tree)
>
```

Our decision tree



R analysis, an example, continued more

Ok, but how does the decision tree do on the test data?

- Test the built tree with the test data.
- Print out the table of results.
- Not bad!

```
>
> testPred <- predict(iris_tree,
+ newdata = testData)
>
> table(testPred, testData$Species)
            setosa versicolor virginica
 setosa         14          0         0
 versicolor      0         14         2
 virginica       0          0        17
>
```

Using R scripts

Once you have a series of commands which you will need to run repeatedly you should save them in a script.

- A script is just a list of commands that you want the R interpreter to execute.
- It's as if you are running the commands at the command line yourself.
- By convention R script files have a ".R" file extension.

```r
# myscript.R

# Create the matrix.
A <- matrix(rnorm(9), nrow = 3,
  ncol = 3)

# Create b.
b <- 1:3

# Solve.
x <- solve(A, b)

cat("the answer is", x, "\n")
```

By saving your commands in a script, you'll remember what you did six months from now.

Running R scripts

Once you have a script, how do you run it? There are two command line options for running your script:

- `R CMD BATCH myscript.R`
  Note that by default this will generate a file called "myscript.Rout", which is what would have been seen on the screen had you run the commands by hand.
- `Rscript myscript.R`
  This is a better option, as it runs as a proper script.

```
[USERNAME@gra-loginX ~]$
[USERNAME@gra-loginX ~]$ R CMD BATCH myscript.R
[USERNAME@gra-loginX ~]$
[USERNAME@gra-loginX ~]$ Rscript myscript.R
The answer is -1.820291 -0.9132152 0.3703467
[USERNAME@gra-loginX ~]$
```

Command line arguments

Running a script usually involves passing parameters to the script, so that you don't need to change the script every time it's run.

If you don't put "`trailingOnly = T`" you'll get the full Rscript command as the first command line argument.

```
# myscript2.R
args <- commandArgs(trailingOnly = TRUE)
cat("The command line arguments are", args, "\n")
```

```
[USERNAME@gra-loginX ~]$
[USERNAME@gra-loginX ~]$ Rscript myscript2.R a 3.2
The command line arguments are a 3.2
[USERNAME@gra-loginX ~]$
```

Where am I?

R has functions for dealing with where things get done. R refers to this as your 'working directory':

- getwd(): get the working directory.
- setwd('somedir'): set the working directory to "somedir".
- dir(): list the contents of the working directory.
- ls(): list the existing variables (you won't see builtin variables, such as pi).

```
>
> getwd()
[1] "/home/myUSER/summer_school"
>
> setwd('..')
>
> getwd()
[1] "/home/myUSER"
>
> dir()
summer_school
>
> ls()
"days.of.week"      "label.names"
>
```

Enough to get started

There's obviously a lot more to learn about using R.
Nonetheless, this is enough to get you started.