My first R Markdown

Rudy Rupp

21/08/2020

Data Scientist's Toolsbox Week 1

What is Data Science? 0:43 - 0:56 They state that a data scientist is broadly defined as someone who combines the skills of software programmer, statistician, and storyteller/artists to extract the nuggets of gold hidden under mountains of data.

Data Scientist's Toolsbox Week 2

R Studio tour

```
matrix(c(1, 2, 3, 4, 5, 6, 7, 8), nrow = 4, ncol = 2)
```

```
##      [,1] [,2]
## [1,]    1    5
## [2,]    2    6
## [3,]    3    7
## [4,]    4    8
```

R Packages

Installing from CRAN
If you are installing from the CRAN repository, use the *install.packages()* function, with the name of the package you want to install in quotes between the parentheses (note: you can use either single or double quotes). For example:

install.packages("ggplot2")

To install multiple packages at once, you can do so by using a character vector:
install.packages(c("ggplot2", "devtools", "lme4"))

Installing from Bioconductor
The BioConductor repository uses their own method to install packages. First, to get the basic functions required to install through BioConductor, use:

source("https://bioconductor.org/biocLite.R")

This makes the main install function of BioConductor, *biocLite()*, available to you. Following this, you call the package you want to install in quotes, between the parentheses of the biocLite command, like so:

biocLite("GenomicFeatures")

Installing from GitHub
You first must find the package you want on GitHub and take note of both the package name AND the author of the package. Check out this guide for installing from GitHub:

- *install.packages("devtools")* - only run this if you don't already have devtools installed.
- *library(devtools)* - loads the devtools package
- *install_github("author/package")* replacing "author" and "package" with their GitHub username and the name of the package.

Loading packages
Installing a package does not make its functions immediately available to you. First you must load the package into R; to do so, use the *library()* function. For example:

```r
library(ggplot2)
```

NOTE: Do not put the package name in quotes! Unlike when you are installing the packages, the *library()* command does not accept package names in quotes!

Checking what packages you have installed
installed.packages() or *library()* with nothing between the parentheses to check.

Updating packages
old.packages() identify packages that have been updated since last updated.
To update all packages, use update.packages(). If you only want to update a specific package, use install.packages("packagename")

Unloading packages
To unload a given package you can use the detach() function. For example, *detach("package:ggplot2", unload=TRUE)*

Uninstalling packages
If you no longer want to have a package installed, you can simply uninstall it using the function remove.packages(). For example, *remove.packages("ggplot2")*

How do you know what version of R you have?
Type *version* into the console and it will output information on the R version you are running.
sessionInfo() - will tell you what version of R you are running along with a listing of all of the packages you have loaded. The output of this command is a great detail to include when posting a question to forums - it tells potential helpers a lot of information about your OS, R, and the packages that you are using.

What commands are included in a package?
You can use the *help()* function to access a package's help files.

```r
help(package = "ggplot2")
```

If you still have questions about what functions within a package are right for you or how to use them, many packages include "vignettes." These are extended help files, that include an overview of the package and its functions, but often they go the extra mile and include detailed examples of how to use the functions in plain words that you can follow along with to see how to use the package. To see the vignettes included in a package, you can use the *browseVignettes()* function.
Eg: *browseVignettes("ggplot2")*

You should see that there are two included vignettes: "Extending ggplot2" and "Aesthetic specifications." Exploring the Aesthetic specifications vignette is a great example of how vignettes can be helpful, clear instructions on how to use the included functions.

Data Scientist's Toolsbox Week 3

Version control (Git & GitHub) Vocabulary

- **Repository** online folder that contains the whole, shared project Commit to save your changes locally, and add explanations of what has been changed.
- **Push** upload changes made locally to the repository, and merge them.
- **Pull** download the most recent changes from the repository and merge with your local folder.
- **Staging** to commit files separately and add separate explanations to each section edited, in order to simplify looking at the change log.
- **Branch** your local files and edits, before you push them.
- **Merge** independent branches are joined together into a single file. May need manual control if the same section of the same file was edited differently in both branches.
- **Conflict** when branches have been edited in the same section and file, making automatic merging hard.
- **Clone** make a local copy of a whole Git repository's files and all the change history.
- **Fork** make a personal copy of a repository that doesn't affect the original.

Best Practices:

- Purposeful, single issue commits
- Informative commit messages
- Pull and Push often
- Git username Rodrupp
- Git email rodrupp@gmail.com
- Git password: automatically generated on firefox.

Linking an Existing Project with GitHub Open Git Bash or Terminal and navigate to the directory containing your project files. Move around directories by typing cd ~/dir/name/of/path/to/file
Once here, type *git init* followed by *git add .* -> this initializes (init) this directory as a git repository and adds all of the files in the directory (.) to your local repository. Commit these changes to the git repository using *git commit -m "Initial commit"*

Go to GitHub.com, and again, create a new repository:

- 1) Make sure the name is the exact same as your R project;
- 2) Do NOT initialize a README file, .gitignore, or license.

Find *"...push an existing repository from the command line"* and copy to clipboard. Paste these into the Terminal, then reload the repository page.

Data Scientist’s Toolsbox Week 4

R Markdown Example

This is an R Markdown document. Markdown is a simple formatting syntax for authoring HTML, PDF, and MS Word documents. For more details on using R Markdown see http://rmarkdown.rstudio.com.

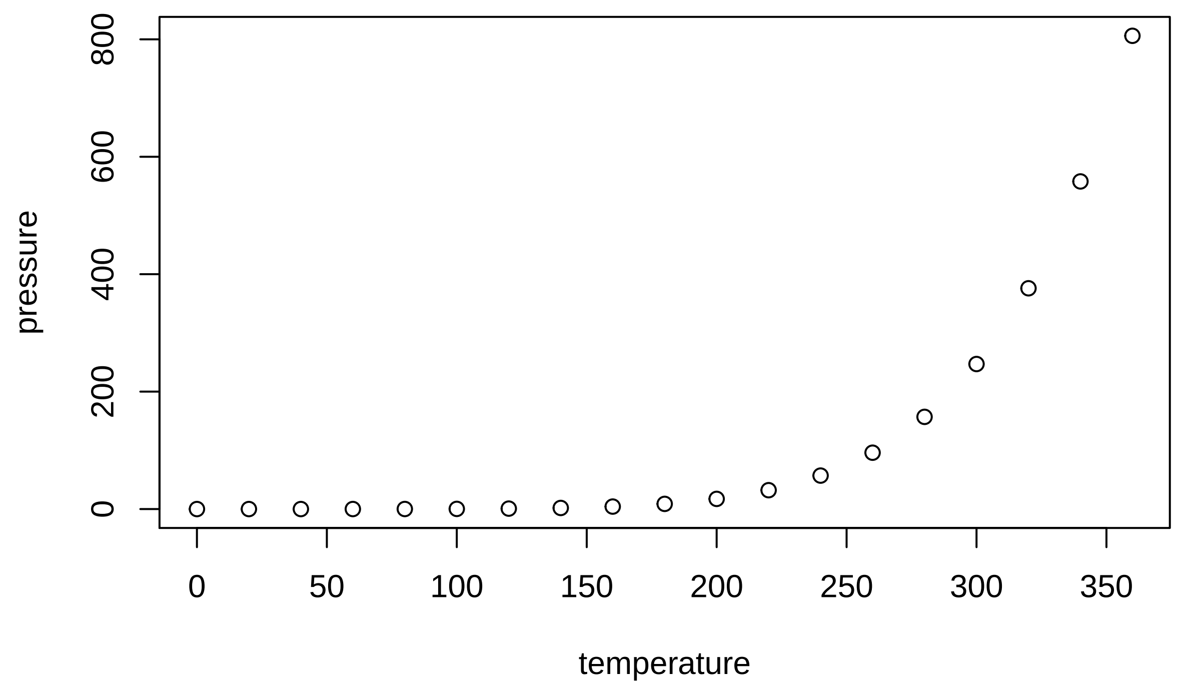
When you click the **Knit** button a document will be generated that includes both content as well as the output of any embedded R code chunks within the document. You can embed an R code chunk like this:

```r
summary(cars)
```

```
##      speed           dist       
##  Min.   : 4.0   Min.   :  2.00  
##  1st Qu.:12.0   1st Qu.: 26.00  
##  Median :15.0   Median : 36.00  
##  Mean   :15.4   Mean   : 42.98  
##  3rd Qu.:19.0   3rd Qu.: 56.00  
##  Max.   :25.0   Max.   :120.00
```

Including Plots

You can also embed plots, for example:



Note that the `echo = FALSE` parameter was added to the code chunk to prevent printing of the R code that generated the plot.

R Markdown formatting

To bold text use double asterisks **bold**.

to make it italics use single asterisks *italics*.

to strikethrough text use double tildes ~~strikethrough~~

Use # for headings: - # Biggest Header
- ## Second biggest
- ### Third heading

Code boxes created with triple backdash “‘

```
print("Testing some code here.")
```

```
## [1] "Testing some code here."
```

```
print("testing the keyboard shortcut ctrl+alt+i")
```

```
## [1] "testing the keyboard shortcut ctrl+alt+i"
```

```
print("testing the insert button on RStudio")
```

```
## [1] "testing the insert button on RStudio"
```

```
print("testing multiple lines of code.")
```

```
## [1] "testing multiple lines of code."
```

```
print("and here is the third line")
```

```
## [1] "and here is the third line"
```

Use Ctrl+enter to run a line of code in RStudio without having to knit the document.
Use Ctrl+shift+enter to run all lines in a chunk.

Making bulletted lists:

- bullets must finish with two spaces

- this is an example

- i wonder what happens if i don’t include two spaces?

- ok, here goes:
- did it work?
  – indent bullets with a double dash

This is a link to google

Types of Data Analysis

1) Descriptive

2) Exploratory

3) Inferential

4) Predictive

5) Causal

6) Mechanistic

Descriptive Describes or summarises a set of data. Often the first analysis to perform. eg mean, standard deviation, etc.

Exploratory Analysis Examine data and find new relationships. Help formulate hypotheses and drive future studies.

Inferential Analysis Use a small sample of data do infer something about a larger population provides estimate and uncertainty. Depends on your sampling.

Predictive Analysis Use current data and historical data to make predictions. Dependent on measuring the right variables. More data and simple model is usually better.

Causal Analysis See what happens to one variable when we manipulate another. Gold standard in data analysis. Often applied to randomized studies designed to identify causation. Often challenging to obtain data.

Mechanistic Analysis Understant the exact changes in variables that lead to exact changes in other variables. Less common and more complex. Applied to simple situation or those that are nicely modeled by deterministic equations (eg physics or engineering).

Experimental Design

Experimental design is organizing an experiment, so that you have the correct data and enough of it to clearly and effectively answer your data science question.

In this order:

1) Formulate your question before collecting any data

2) Design the experiment.

3) Identify problems.

4) Collect data.

And then a bunch of stuff I already know…

Big Data

Volume, Velocity, Variety, Unstructured

Examples: Text files and documents, Websites and applications, Sensor data, Image Files, Audio files, Video files, Email Data, Social Media data.

Before you can start looking for answers, you need to turn your unstructured data into a format that you can analyze.

R Programming Week 1

Learning Objectives
- Install R Stoduio and Packages, including Swirl
- History of R and S
- describe differences between atomic data types
- Subset R objects using "[", "[[" and "$"
- Understand coersion, implicit and explicit
- Remove NAs from objects

Setting values

```r
x <- 1  ##sets the value of x to 1
x  ## prints the value of x
```

```
## [1] 1
```

```r
msg <- "hello"  ##gives msg the text "hello"
msg ## prints the text set to msg
```

```
## [1] "hello"
```

You can also set calculations into the value

```r
x <- x+1 ##Increases the value of x by one.
y <- 5
z <- x+y
print(z)
```

```
## [1] 7
```

The result of the expression may be auto printed.

```r
x <- 5 ## nothing pinter
x  ## auto-printing occurs
```

```
## [1] 5
```

```r
print(x)  ## explicit printing
```

```
## [1] 5
```

A vector may also be set for a variable

```
x <- 1:20  ## creates an integer sequence from 1 to 20
x
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
```

Calculations can also be applied on vectors

```
x <- 1:20
y <- x*3
y
```

```
##  [1]  3  6  9 12 15 18 21 24 27 30 33 36 39 42 45 48 51 54 57 60
```

Objects and Attributes

R has 5 basic or "atomic"classes of objects
- character (text)
- numeric (real numbers, ie decimal numbers)
- integer
- complex
- logical (true/false)

The most basic object is a vector.
- Vectors can contain only objects of the same class
- except *lists*

Empty vectors can be created with the vector() function. It has 2 arguments: the class of the object and the length of the vector.

Numbers

Typically treated as numeric objects (Even if "1" is entered, it is understood as "1.0000000000"). You can enter 1L to explicitly make it an integer.

Inf can be used as a number for infinity. *-Inf* also possible
NaN can be used as a missing or undefined value, e.g 0/0

Attributes

Objects can have attributes such as:
- names, dimnames
- dimensions (eg matrices and arrays)
- class
- length (the length of a vector is the number of elements in the vector)
- other, user-defined attributes

Attributes can be accessed using the *attributes()* function.

Vectors and Lists

The c() function can be used to create vectors. c can stand for concatonate.

```r
x <- c(0.5, 0.6)  ## numeric
y <- c(T, F)   ## logical
z <- c("a", "b", "c")  ## character
w <- 9:29  ## integer
v <- c(1+0i, 2+4i)  ## complex

print(c(x,y,z,w,v))
```

```
##  [1] "0.5"   "0.6"   "TRUE"  "FALSE" "a"     "b"     "c"     "9"     "10"
## [10] "11"    "12"    "13"    "14"    "15"    "16"    "17"    "18"    "19"
## [19] "20"    "21"    "22"    "23"    "24"    "25"    "26"    "27"    "28"
## [28] "29"    "1+0i"  "2+4i"
```

Or use vector function:

```r
x <- vector("numeric", length = 10)
x
```

```
##  [1] 0 0 0 0 0 0 0 0 0 0
```

By default for numeric vectors, the value is 0.

Mixing Objects

Objects can be forced into a different class. You won't get an error message.

```r
x <- c(1.7, "a")  ## all will be treated as characters / text
x
```

```
## [1] "1.7" "a"
```

```r
y <- c(TRUE, 2)  ## all will be treated as numeric. T = 1, F = 0
y
```

```
## [1] 1 2
```

```r
z <- c("a", TRUE)  ## all will be treated as characters / text
z
```

```
## [1] "a"    "TRUE"
```

You can explicitly coerce objects into another class.

```r
x <- 0:6
class(x)
```

```
## [1] "integer"
```

```r
as.numeric(x)
```

```
## [1] 0 1 2 3 4 5 6
```

```r
as.logical(x)
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE  TRUE  TRUE
```

```r
as.character(x)
```

```
## [1] "0" "1" "2" "3" "4" "5" "6"
```

```r
as.complex(x)
```

```
## [1] 0+0i 1+0i 2+0i 3+0i 4+0i 5+0i 6+0i
```

Nonsensical coercion results in NAs

```r
x <- c("a", "b", "c")
as.numeric(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

```r
as.logical(x)
```

```
## [1] NA NA NA
```

```r
as.complex(x)
```

```
## Warning: NAs introduced by coercion
```

```
## [1] NA NA NA
```

Lists

Can contain objects of different classes. Elements are encased in double brackets.

```r
x <- list(1, "a", TRUE, 1+4i)
x
```

```
## [[1]]
## [1] 1
##
## [[2]]
## [1] "a"
##
## [[3]]
## [1] TRUE
##
## [[4]]
## [1] 1+4i
```

Matrices

Are vectors with a *dimension* attribute.

```
m <- matrix(1:6, nrow = 2, ncol = 3)
m
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
dim(m)    ## gives dimensions of matrix, rows and columns
```

```
## [1] 2 3
```

```
attributes(m)   ## shows attribute dim
```

```
## $dim
## [1] 2 3
```

Matrices get constructed column-wise, top to bottom.

Matrices can be made from normal vectors

```
m <- 1:10
m
```

```
##  [1]  1  2  3  4  5  6  7  8  9 10
```

```
dim(m) <- c(2,5)  ## adds the dimension attribute to it, while assigning a value to
 ## the dim attribute of m: the number of rows and columns.
m
```

```
##      [,1] [,2] [,3] [,4] [,5]
## [1,]    1    3    5    7    9
## [2,]    2    4    6    8   10
```

Making matrices by column binding or row binding:

```
x <- 1:3
y <- 10:12
cbind(x, y)
```

```
##      x  y
## [1,] 1 10
## [2,] 2 11
## [3,] 3 12
```

```
rbind(x, y)
```

```
##   [,1] [,2] [,3]
## x    1    2    3
## y   10   11   12
```

Factors

Used to represent categorical data. Ordered or unordered. You could think of a factor as an integer vector where each integer has a label. Eg: 1 - high; 2 - medium; 3 - low. They are treated specially by modeling functions like *lm()* and *glm()*. They are better than using integers because they are self-descriptive.

```
x <- factor(c("yes", "yes", "no", "yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: no yes
```

```
table(x)  ## counts how many of each level
```

```
## x
##  no yes
##   2   3
```

```
unclass(x)  ## removes the class of the vector, shows integer values
```

```
## [1] 2 2 1 2 1
## attr(,"levels")
## [1] "no"  "yes"
```

The order of the levels of the factor is set in alphabetical order. between "yes" and "no", No will be the baseline level. You can explicitly change this with the *level()* function.

```
x <- factor(c("yes", "yes", "no", "yes", "no"),
            levels = c("yes", "no"))
x
```

```
## [1] yes yes no  yes no
## Levels: yes no
```

Missing Values

Denoted by NA "Not Available" (or NaN "Not a Number", for undefined mathematical operations).
- *is.na()* and *is.nan()* used to test objects.
- NA values can have classes also, eg Integer NA, character NA, etc.
- NaN value is also NA, but converse is not true

```r
x <- c(1, 2, NA, 10, 3)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

```r
is.nan(x)
```

```
## [1] FALSE FALSE FALSE FALSE FALSE
```

```r
x <- c(1, 2, NaN, NA, 4)
is.na(x)
```

```
## [1] FALSE FALSE  TRUE  TRUE FALSE
```

```r
is.nan(x)
```

```
## [1] FALSE FALSE  TRUE FALSE FALSE
```

Data Frames

Used to store tabular data. - represented as a special type of list, but each element of the list has to have the same length.
- each element of the list is a column, and the length of each element is the number of rows.
- they can store different classes of objects in each column, which makes them different from matrices.
- special attribute called *row.names*
- usually created with *read.table()* or *read.csv()* or *data.frame()*
- can be converted into matrices with *data.matrix()* (will coerce objects into being the same class)

```r
x <- data.frame(foo = 1:4, bar = c(T, T, F, F))
x
```

```
##   foo   bar
## 1   1  TRUE
## 2   2  TRUE
## 3   3 FALSE
## 4   4 FALSE
```

```r
nrow(x)
```

```
## [1] 4
```

```r
ncol(x)
```

```
## [1] 2
```

Names Attribute

Important for writing readable code and self describing objects.

```
x <- 1:3
names(x)
```

```
## NULL
```

```
names(x) <- c("foo", "bar", "norf")
x
```

```
##  foo  bar norf
##    1    2    3
```

```
names(x)
```

```
## [1] "foo"  "bar"  "norf"
```

List names and matrix names:

```
x <- list(a = 1, b = 2, c = 3)
x
```

```
## $a
## [1] 1
##
## $b
## [1] 2
##
## $c
## [1] 3
```

```
m <- matrix(c("BB", "Bb", "Bb", "bb"), nrow = 2, ncol = 2)
dimnames(m) <- list(c("B", "b"), c("B", "b"))  ##row names, then column names
m
```

```
##   B    b
## B "BB" "Bb"
## b "Bb" "bb"
```

Reading Tabular Data

Principal functions
- *read.table()*, *read.csv()* read text files in rows and columns and return a data frame in R
- *readLines()* reads lines of a text file, gives a character vector
- *source()* read R code files
- *dget()* read R files: objects that are "deparsed"
- *load()* reading binary objects into R
- *unserialize()* also for binary objects

Analogous functions for writing data, in order:
- *write.table()*
- *writeLines()*
- *dump()*
- *dput()*
- *save()*
- *serialize()*

Reading files with read.table()

Examine *?read.table* help file. Creates a **data frame** from a file in table format. Common arguments:

- *file* name of file / path to file
- *header* indicates if the first line is a header or not
- *sep* indicates how columns are separated
- *colClasses* character vector of the same length as number of columns. Indicates class of each column (can be different for each column).
- *nrows* number of rows (not required)
- *comment.char* indicates what character indicates comments
- *skip* skips a number of lines
- *stringsAsFactors* default=true, are character variables encoded as factors?

for moderately sized datasets, you can call read.table without any arguments other than *file*. R will automatically:
- skip lines that begin with #
- find number of rows
- classify variables

However, giving R these arguments will make it runs more smoothly and predictably. *read.csv()* is an identical command except that the default separator is a comma instead of a space. CSV files are common outputs from excel files and many other programs.

Reading Larger Datasets with read.table

Things to do to facilitate reading.
- Read the Help page carefully for read.table - Make a rough calculation of the memory required to store your dataset. Is it larger than the RAM on your computer? If so don't proceed.
- Set *comment.char = " "* if there are no commented lines in your file.
- use the colClasses argument. Specifying this option instead of using the default. R won't need to try to figure it out. If all classes are the same class you can set them all with *colClasses = "numeric"* for example. Alternatively, use the first 100 rows and let read.table find the classes, then apply them to reading the whole text:

```
initial <- read.table("datatable.txt", nrows = 100)
classes <- sapply(initial, class)  ## S apply calls the class of each column
tabAll <- read.table("datatable.txt", colClasses = classes)
```

- set *nrows* to help reduce memory usage. a mild overestimate is okay.

It's useful to know a few things:
- How much memry is available?
- what other applications are in use?
- are other users using resources of the computer?
- what operating system are you using? is it 32 or 64 bit?

Calculating Memory Requirements

Consider a data frame with 1,500,00 rows and 120 columns, all of which are numeric data. How much memory is required?

each numeric object required 8 bytes, stored in 64 bits of data.

1,500,00 x 120 x 8 bytes/numeric = 1,440,000,000 bytes
1,440,000,000 / (2^20 bytes / MB) = 1,373.29 MB
= 1.34 GB

This is the RAW data. A bit more is required to read the data: rule of thumb is you require double the memory as the raw data.

Other Text Formats

Not in form of tables. Used to store more metadata, for example class of the data in each column. Work better in version control, as text files are more easily read by git. It can be easier to fix corruption in the file.
Downside: less space-efficient.
dump() and *dput()* to write, *source()* and *dget()* to read.

```
y <- data.frame(a = 1, b = "a")
dput(y)
```

```
## structure(list(a = 1, b = "a"), class = "data.frame", row.names = c(NA,
## -1L))
```

```
dput(y, file = "y.R")
new.y <- dget("y.R")
new.y
```

```
##   a b
## 1 1 a
```

dput can only be used on one R object, but *dump* can be used in multiple.

```
x <- "foo"
y <- data.frame(a = 1, b = "a")
dump(c("x", "y"), file = "data.R")
rm(x, y)   ## removes the objects x and y
source("data.R")   ## reconstructs the objects from the file
y
```

```
##   a b
## 1 1 a
```

```
x
```

```
## [1] "foo"
```

Interfaces with other things

Establish connections with files, compressed files or other more exotic things.
- *file()*
- *gzfile()* file compressed with gzip
- *bzfile()* file compressed with bzip2
- *url()* opens a connection to a webpage

file() function has the arguments:
- *description* name of the file
- *open* has a flag to describe:
- "r" read only.
- "w" writing (and initializing a new file)
- "a" appending
- "rb", "wb", "ab", as above but in binary mode (windows)

Connections let you navigate files, too. Often connections happen behind the scenes.

```
con <- file("datatable.txt", "r")
data <- read.csv(con)
close(con)

 ## these three lines are equivalent to simply using:

data <- read.csv("datatable.txt")
```

But it can be useful to do so explicitly, such as when reading lines of a text file:

```
con <- file("datatable.txt")
x <- readLines(con, 10)
x
```

```
##  [1] "1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 "
##  [2] "9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 "
##  [3] "7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 "
##  [4] "5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 "
##  [5] "3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 "
##  [6] "1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 "
##  [7] "9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 "
##  [8] "7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 "
##  [9] "5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 "
## [10] "1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 "
```

```
close(con)
```

writeLines() take a character vector and writes each element one line at a time to a text file.
readLines() can also be used to read from a website, after establishing a connection to a non-file object.

```
con <- url("http://www.jhsph.edu", "r")
x <- readLines(con)
head(x)
```

```
## [1] "<!DOCTYPE html>"
## [2] "<html lang=\"en\">"
## [3] ""
## [4] "<head>"
## [5] "<meta charset=\"utf-8\" />"
## [6] "<title>Johns Hopkins Bloomberg School of Public Health</title>"
```

```r
close(con)
```

This is another way to read data other than *read.Table* and *read.csv*.

Subsetting

extracting subsets of R objects
- "[" always returns an object of the same class as the original. e.g. subsetting a vector makes a vector. Can be used to select more than one element.
- "[[" used to extract elements of a list or dataframa. Can only extract a single element. Returned object won't necessarily be of the same class.
- "$" is used to extract elements of a list or dataframe by name. Similar to using "[["

```r
x <- c("a", "b", "c", "c", "d", "a")
x[1]
```

```
## [1] "a"
```

```r
x[2]
```

```
## [1] "b"
```

```r
x[1:4]   ## subsetting using a numerical index
```

```
## [1] "a" "b" "c" "c"
```

```r
x[x > "a"]   ## subsetting using a logical index
```

```
## [1] "b" "c" "c" "d"
```

```r
u <- x > "a"   ## created logical vector u
u
```

```
## [1] FALSE  TRUE  TRUE  TRUE  TRUE FALSE
```

```r
x[u]   ## subsetting using a logical vector as an index
```

```
## [1] "b" "c" "c" "d"
```

Subsetting Lists

```r
x <- list(foo = 1:4, bar = 0.6, baz = "hello")
x
```

```
## $foo
## [1] 1 2 3 4
##
## $bar
## [1] 0.6
##
## $baz
## [1] "hello"
```

```r
x[1]  ## extract 1st element, same class as original (a list)
```

```
## $foo
## [1] 1 2 3 4
```

```r
class(x[1])
```

```
## [1] "list"
```

```r
x[[1]] ## extract 1st element as just a sequence
```

```
## [1] 1 2 3 4
```

```r
class(x[[1]])
```

```
## [1] "integer"
```

```r
x$bar ## gives element associated with name "bar"
```

```
## [1] 0.6
```

```r
x[["bar"]] ## same as above
```

```
## [1] 0.6
```

```r
x["bar"] ## same as x[1], returns it as a list
```

```
## $bar
## [1] 0.6
```

```r
x[c(1,3)]  ## extract multiple elements, returns as same class (list)
```

```
## $foo
## [1] 1 2 3 4
##
## $baz
## [1] "hello"
```

```r
name <- "foo" ## the name of an element can be saved as an object, too
x[[name]]  ## useful to look up a name that is automatically computated
```

```
## [1] 1 2 3 4
```

```r
x$name  ## doesn't work as [[ ]]. Only looks for the literal name
```

```
## NULL
```

```r
x$foo  ## works because "foo" is a valid name
```

```
## [1] 1 2 3 4
```

The [[can take an integer sequence:

```r
x <- list(a = list(10, 12, 14), b = c(3.14, 2.81))
x[[c(1,3)]]  ## extracts the 3rd element of the 1st list
```

```
## [1] 14
```

```r
             ## inception!
x[[1]][[3]]  ## double subsetting, same result as above
```

```
## [1] 14
```

```r
x[[c(2,1)]]  ## one more example
```

```
## [1] 3.14
```

Subsetting Matrices

Can be subset with a row index and a column index.

```r
x <- matrix(1:6, 2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
x[1, 2]
```

```
## [1] 3
```

```r
x[2, 1]
```

```
## [1] 2
```

Indices can also be missing, which will return a full row or column.

```r
x[1, ]
```

```
## [1] 1 3 5
```

```r
x[, 2]
```

```
## [1] 3 4
```

By default, when a single element of a matrix is retrieved, it's returned as a vector of length 1, rather than a 1 x 1 matrix. this can be turned off by setting *drop = FALSE*.

```r
x <- matrix(1:6, 2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```r
x[1, 2]
```

```
## [1] 3
```

```r
x[1, 2, drop = FALSE]
```

```
##      [,1]
## [1,]    3
```

```r
## A similar behaviour occurs when subsetting a single column
## or row.

x[1, ]  ## returns a vector
```

```
## [1] 1 3 5
```

```r
x[1, , drop = FALSE]  ## returns a matrix
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
```

Partial Matching

Partial matching names is allowed with "[[" and "$"

```r
x <- list(aardvark = 1:5, peter = 0.6, paul = 3.14)
x$a  ## by default looks for a name that matches "a".
```

```
## [1] 1 2 3 4 5
```

```r
x$p  ## fails when there are multiple matches
```

```
## NULL
```

```r
x[["a"]]  ## looks for an exact match, fails
```

```
## NULL
```

```r
x[["a", exact = FALSE]]  ## default behaviour can be turned off
```

```
## [1] 1 2 3 4 5
```

Removing NA values

A common task

```r
x <- c(1, 2, NA, 4, NA, 5)  ## an object that requires cleaning
bad <- is.na(x)  ## logical vector testing for NA
bad
```

```
## [1] FALSE FALSE  TRUE FALSE  TRUE FALSE
```

```r
x[!bad]  ## "!" used to invert the logical vector.
```

```
## [1] 1 2 4 5
```

```r
         ## The logical vector determines which elements to return.
```

You can take the subset of different object that have no missing values:

```r
x <- c(1, 2, NA, 4, NA, 6, 7, 8, NA, 10, 11)
y <- c("a", "b", NA, "d", NA, "f", "g", "h", "i", NA, "k")
good <- complete.cases(x, y)
good
```

```
##  [1]  TRUE  TRUE FALSE  TRUE FALSE  TRUE  TRUE  TRUE FALSE FALSE  TRUE
```

```r
x[good]  ## note some data is lost because of missing values in y
```

```
## [1]  1  2  4  6  7  8 11
```

```r
y[good]  ## note some data is lost because of missing values in x
```

```
## [1] "a" "b" "d" "f" "g" "h" "k"
```

Removing NA values from dataframes. Also removes whole rows or columns where NAs are present.

```
airquality[1:6, ]  ## example data frame
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```
good <- complete.cases(airquality)
               ## logical vector tells me which rows are complete.
airquality[good, ][1:6, ]
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 7    23     299  8.6   65     5   7
## 8    19      99 13.8   59     5   8
```

```
        ## subset the airquality matrix using the logical vector
```

Vectorised Operations

Many operations in R are *vectorised*, making code more efficient, concise and easier to read.

```
x <- 1:4; y <- 6:9
x + y  ## add 1st element of x to 1st element of y, 2nd to 2nd, etc.
```

```
## [1]  7  9 11 13
```

```
x > 2  ## compares all numbers to 2
```

```
## [1] FALSE FALSE  TRUE  TRUE
```

```
x >= 2
```

```
## [1] FALSE  TRUE  TRUE  TRUE
```

```
y == 8  ## == tests for equality
```

```
## [1] FALSE FALSE  TRUE FALSE
```

```
x * y  ## also works with other operations
```

```
## [1]  6 14 24 36
```

```
x / y
```

```
## [1] 0.1666667 0.2857143 0.3750000 0.4444444
```

Similarly with matrices:

```
x <- matrix(1:4, 2, 2); y <- matrix(rep(10, 4), 2, 2)
x
```

```
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
```

```
y
```

```
##      [,1] [,2]
## [1,]   10   10
## [2,]   10   10
```

```
x * y  ## element-wise multiplication
```

```
##      [,1] [,2]
## [1,]   10   30
## [2,]   20   40
```

```
x / y
```

```
##      [,1] [,2]
## [1,]  0.1  0.3
## [2,]  0.2  0.4
```

```
x %*% y  ## true matrix multiplication
```

```
##      [,1] [,2]
## [1,]   40   40
## [2,]   60   60
```

Week 1 Quiz Questions

```
## Extract the subset of rows of the data frame where Ozone
## values are above 31 and Temp values are above 90. What is the
## mean of Solar.R in this subset?

quiz <- read.csv(file = "hw1_data.csv")
Ozone <- quiz[ , 1]  ## creating a list of column 1, Ozone
Temp <- quiz[ , 4]  ## creating a list of column 4, Temp
Solar.R <- quiz[ , 2]  ## creating a list of column 2, Solar.R
```

```r
Truncated <- cbind(Ozone, Temp, Solar.R)  ## merge into new matrix

Cleanrows <- complete.cases(Truncated)
                 ## logical vector for complete rows

ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

CleanOzone <- ValidRows124[ , 1]  ## List of column 1 of the new matrix
OzoneTest <- CleanOzone > 31  ## logical vector for Ozone column > 31

BigOzone <- ValidRows124[OzoneTest, ]
           ## matrix with rows where ozone is bigger than 31 extracted

CleanTemp <- BigOzone[ , 2]
           ## List of column 2 of the BigOzone matrix
TempTest <- CleanTemp > 90  ## logical vector for Temp column > 90
BigTemp <- BigOzone[TempTest, ]
        ## matrix with rows where Ozone is bigger than 31
        ## and Temp is bigger than 90
BigTemp
```

```
##       Ozone Temp Solar.R
##  [1,]    97   92     267
##  [2,]    97   92     272
##  [3,]    76   97     203
##  [4,]   118   94     225
##  [5,]    84   96     237
##  [6,]    85   94     188
##  [7,]    96   91     167
##  [8,]    78   92     197
##  [9,]    73   93     183
## [10,]    91   93     189
```

```r
mean(BigTemp[ , 3])
```

```
## [1] 212.8
```

A more elegant solution:

```r
## Extract the subset of rows of the data frame where Ozone
## values are above 31 and Temp values are above 90. What is the
## mean of Solar.R in this subset?

quiz <- read.csv(file = "hw1_data.csv")  ##reading file
Ozone <- quiz[ , 1]  ## list of column 1, Ozone
Temp <- quiz[ , 4]  ## list of column 4, Temp
Solar.R <- quiz[ , 2]  ## list of column 2, Solar.R

Truncated <- cbind(Ozone, Temp, Solar.R)  ## merge into new matrix

Cleanrows <- complete.cases(Truncated)
```

```
                 ## logical vector for complete rows

ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

OandTTest <- ValidRows124[ , 1] > 31 & ValidRows124[ , 2] > 90
        ## execute both tests and save as a logical vector

ValidRows124[OandTTest, ]  ## This is the resulting matrix
```

```
##       Ozone Temp Solar.R
##  [1,]    97   92     267
##  [2,]    97   92     272
##  [3,]    76   97     203
##  [4,]   118   94     225
##  [5,]    84   96     237
##  [6,]    85   94     188
##  [7,]    96   91     167
##  [8,]    78   92     197
##  [9,]    73   93     183
## [10,]    91   93     189
```

```
mean(ValidRows124[OandTTest, ][ , 3])
```

```
## [1] 212.8
```

```
      ## taking the mean of the third column of the resulting matrix
```

Event Better

```
## Extract the subset of rows of the data frame where Ozone
## values are above 31 and Temp values are above 90. What is the
## mean of Solar.R in this subset?

quiz <- read.csv(file = "hw1_data.csv")  ##reading file
x <- quiz[quiz$Ozone > 31 & quiz$Temp > 90,]
   ## Subset of data frame where row elements labeled "Ozone"
   ## (column label) are bigger than 31, and rows labeled
   ## "Temp" (column label) are bigger than 90.
x[!is.na(x$Ozone), ]
```

```
##     Ozone Solar.R Wind Temp Month Day
## 69     97     267  6.3   92     7   8
## 70     97     272  5.7   92     7   9
## 120    76     203  9.7   97     8  28
## 121   118     225  2.3   94     8  29
## 122    84     237  6.3   96     8  30
## 123    85     188  6.3   94     8  31
## 124    96     167  6.9   91     9   1
## 125    78     197  5.1   92     9   2
## 126    73     183  2.8   93     9   3
## 127    91     189  4.6   93     9   4
```

```r
   ## subset of rows, in which th elements labeled "Ozone"
   ## are NOT NAs
mean(x[!is.na(x$Ozone),"Solar.R"]) ## mean of column "Solar.R"
```

```
## [1] 212.8
```

```r
 ## What is the mean of "Temp" when "Month" is equal to 6?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file
Temp <- quiz[ , 4]  ## list of column 4, Temp
Month <- quiz[ , 5]  ## list of column 5, Month

Truncated <- cbind(Temp, Month)  ## merge into new matrix
Cleanrows <- complete.cases(Truncated)
                 ## logical vector for complete rows
ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

June <- ValidRows124[ , 2] == 6  ## Logical vector for month = 6
ValidRows124[June, ]  ## the matrix with extracted rows only
```

```
##       Temp Month
##  [1,]   78     6
##  [2,]   74     6
##  [3,]   67     6
##  [4,]   84     6
##  [5,]   85     6
##  [6,]   79     6
##  [7,]   82     6
##  [8,]   87     6
##  [9,]   90     6
## [10,]   87     6
## [11,]   93     6
## [12,]   92     6
## [13,]   82     6
## [14,]   80     6
## [15,]   79     6
## [16,]   77     6
## [17,]   72     6
## [18,]   65     6
## [19,]   73     6
## [20,]   76     6
## [21,]   77     6
## [22,]   76     6
## [23,]   76     6
## [24,]   76     6
## [25,]   75     6
## [26,]   78     6
## [27,]   73     6
## [28,]   80     6
## [29,]   77     6
## [30,]   83     6
```

```r
mean(ValidRows124[June, ][ , 1])
```

```
## [1] 79.1
```

```r
    ## mean of the first column of the matrix with extracted rows only
```

Better

```r
 ## What is the mean of "Temp" when "Month" is equal to 6?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file

quiz[quiz$Month == 6, "Temp"]
```

```
##  [1] 78 74 67 84 85 79 82 87 90 87 93 92 82 80 79 77 72 65 73 76 77 76 76 76 75
## [26] 78 73 80 77 83
```

```r
  ## Subset elements of column "Temp", on rows where elements
  ## labeled "Month" are equal to 6.
mean(quiz[quiz$Month == 6, "Temp"])
```

```
## [1] 79.1
```

```r
## What was the maximum ozone value in the month of May
## (i.e. Month is equal to 5)?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file
Ozone <- quiz[ , 1]  ## list of column 1, Ozone
Month <- quiz[ , 5]  ## list of column 5, Month

Truncated <- cbind(Ozone, Month)  ## merge into new matrix
Cleanrows <- complete.cases(Truncated)
                 ## logical vector for complete rows
ValidRows124 <- Truncated[Cleanrows, ]  ## extract complete rows

May <- ValidRows124[ , 2] == 5  ## Logical vector for month = 5
ValidRows124[May, ]  ## the matrix with extracted rows only
```

```
##       Ozone Month
##  [1,]    41     5
##  [2,]    36     5
##  [3,]    12     5
##  [4,]    18     5
##  [5,]    28     5
##  [6,]    23     5
##  [7,]    19     5
##  [8,]     8     5
##  [9,]     7     5
## [10,]    16     5
## [11,]    11     5
## [12,]    14     5
## [13,]    18     5
## [14,]    14     5
## [15,]    34     5
## [16,]     6     5
## [17,]    30     5
## [18,]    11     5
## [19,]     1     5
## [20,]    11     5
## [21,]     4     5
## [22,]    32     5
## [23,]    23     5
## [24,]    45     5
## [25,]   115     5
## [26,]    37     5
```

```
max(ValidRows124[May, ][ , 1])
```

```
## [1] 115
```

```
    ## mean of the first column of the matrix with extracted rows only
```

Better

```
## What was the maximum ozone value in the month of May
## (i.e. Month is equal to 5)?
quiz <- read.csv(file = "hw1_data.csv")  ##reading file

quiz[quiz$Month == 5, "Ozone"]
```

```
##  [1]  41  36  12  18  NA  28  23  19   8  NA   7  16  11  14  18  14  34   6  30
## [20]  11   1  11   4  32  NA  NA  NA  23  45 115  37
```

```
  ## extracts the column "Ozone" from the rows where elements
  ## labeled "Month" are equal to 5
  ## NAs are still present

quiz[quiz$Month == 5 & !is.na(quiz$Ozone), "Ozone"]
```

```
##  [1]  41  36  12  18  28  23  19   8   7  16  11  14  18  14  34   6  30  11   1
## [20]  11   4  32  23  45 115  37
```

```
  ## add onto that a logical vector to remove NAs

max(quiz[quiz$Month == 5 & !is.na(quiz$Ozone), "Ozone"])
```

```
## [1] 115
```

Swirl Exercises Week 1

Swirl lesson 1: Basic Building Blocks

New commands:

- *ls()* displays a list of all variables saved in the workspace
- *getwd()* displays the path t the working directory
- *sqrt()* square root
- pressing the up arrow on the console shows the previous command
- *abs()* absolute value
- *c()* can be used to merge vectors or lists, eg:

```
x <- c(2, 23, 1)
c(x, 21, x)
```

```
## [1]  2 23  1 21  2 23  1
```

Swirl lesson 2: Workspace and files

- *list.files()* displays a character vector with the names of files in the directory.
  – *path =* argument to choose directory. Default is *getwd()*
  – *?list.files* help page has useful information on arguments. – *file.info()*, *file.access()*, *files()*, *file.choose()* and *choose.files()* could be useful.
- *args()* with the function name in the brackets, shows all arguments a function can take, and their default values.
- *dir.create("testdir")* to create a new folder.
- *setwd("testdir")* sets a new working directory.
- *file.create("mytest.R")* creates new file.
- *file.exists("mytest.R")* logical vector, tests if file exists.
- *file.info("mytest.R")* returns a data frame with details of size, directory, mode, and time stamps.
  – *file.info("mytest.R")$mode* retrieves specific items from the data frame
- *file.rename("mytest.R","mytest2.R")*
- *file.copy("mytest2.R", "mytest3.R")*
- *file.path("folder1", mytest3.R")* Character vector that holds a path to a file from components, in a platform-independent way. Example returns *"folder1/mytest3.R"*
  – for example *dir.create(file.path("testdir2", "testdir3"), recursive = TRUE)* creates folder"testdir3" inside folder "testdir2".

Swirl lesson 3: Sequences of Numbers

- *1:20* gives a numerical vector of length 20. Numbers start at 1 and finish at 20, increments of 1.
  – *pi:10* also works. value of 10 is not reached.

- *seq(1, 10)* does the same, but has more arguments.
  - *seq(1, 10, by=0.5)* gives increments of 0.5
  - *seq(5, 10, length=30)* calculates even intervals to match the length.
  - *seq(along.with = my_seq)* along.with take the length from the length of this argument.

- *length()* gives the length of a vector.

- *rep(0, times = 40)* replicates a value to create a vector.
  - can also take a vector value. *rep(c(0, 1, 2), times = 10)*
  - alternatively, repeat each value with *rep(c(0, 1, 2), each = 10)*

Swirl lesson 4: Vectors

- logical operators >, <, >=, <=, == and !=

- "A | B" means A OR B. "A & B" means A AND B

- !A is the negation of A and is TRUE when A is FALSE and vice versa.

- *paste(my_char, collapse = " ")* merges the elements of a vector, separating them with a space.

- can also be used to join multiple character vectors:

```
 paste(1:3, c("X", "Y", "Z"), sep = "")
```

```
## [1] "1X" "2Y" "3Z"
```

```
        ## the numeric vector 1:4 gets 'coerced' into
        ## a character vector by the paste() function.
paste(LETTERS, 1:4, sep = "-")
```

```
##  [1] "A-1" "B-2" "C-3" "D-4" "E-1" "F-2" "G-3" "H-4" "I-1" "J-2" "K-3" "L-4"
## [13] "M-1" "N-2" "O-3" "P-4" "Q-1" "R-2" "S-3" "T-4" "U-1" "V-2" "W-3" "X-4"
## [25] "Y-1" "Z-2"
```

Swirl lesson 5: Missing Values

- *rnorm(1000)* to create a vector with 1000 draws from a standard normal distribution.

- *sample(y, 100)* takes 100 values at random from vector y.

Swirl lesson 6: Subsetting Vectors

- *x[!is.na(x)]* shorthand to remove NAs

- *x[c(3, 20, 21)]* picks the 3rd, 20th and 21st elements of x

- *x[-c(3, 20, 21)]* picks all elements of x, except the 3rd, 20th and 21st.

- **Named elements in vectors**

```r
c(foo = 11, bar = 2, norf = NA)  ## named from start
```

```
##  foo  bar norf
##   11    2   NA
```

```r
vect <- c(11, 2, NA)
names(vect) <- c("foo", "bar", "norf")  ## named after existing
vect
```

```
##  foo  bar norf
##   11    2   NA
```

- *identical (x, y)* gives a logical vector of length 1

Swirl lesson 7: Matrices and Data Frames

- *dim(x)* gives the dimensions of an object. For a vector, that's NULL. Rows, then Columns.
- *length(x)* works for vectors.
- giving dimensions to a vector will make it into a matrix
- *colnames()* can call or set the names of columsn in a data frame
- *colnames(my_data) <- c("patient", "age", "weight", "bp", "rating", "test")*

R Programming Week 2: Programming with R

Learning Objectives

- write if-else expressions.
- write "for", "while" and "repeat" loops
- define a function and specify its return value
- describe how R binds a value to a symbol via the search list
- explain lexical scoping with respect to how the value of free variables are resolved.
- describe the difference between lexical scoping and dynamic scoping rules.
- convert character strings representing date & time to an R datetime object.

Control Structures

allow you to control the flow of execution of a program. For example:

- *if*, *else* tests a condition.
- *for* repeat a fixed number of times.
- *while* repeat while a condition is true.
- *repeat* repeats indefinitely.
- *break* stops the execution of a loop.
- *next* skips one iterations.
- *return* exists a function.

These are used when writing programs. When writing on the command line, there are a separate set of functions with *apply* which are more useful for loops.

if, else

The "else" argument is not necessary.

```
x <- rnorm(1, mean = 3)
x
```

```
## [1] 4.328598
```

```
if(x>3) {
        y <- 10
} else {
        y <- 0
}
y
```

```
## [1] 10
```

Can also be written as:

```
x <- rnorm(1, mean = 3)
x
```

```
## [1] 4.068946
```

```
y <- if(x>3) {
             10
} else {
             0
}
y
```

```
## [1] 10
```

"For" loops

for loops take an interator variable and assign it successive values from a sequence of a vector. Most commonly used for iterating over the elements of an object.

```
for(i in 1:10) {
                print(i*2)
}  ## a simple example
```

```
## [1] 2
## [1] 4
## [1] 6
## [1] 8
## [1] 10
## [1] 12
## [1] 14
## [1] 16
## [1] 18
## [1] 20
```

```
x <- c("a", "b", "c", "d", "e")
for (i in 1:4) {
                print(x[i])
  }  ## subsetting x with i to give the first 4 elements of x
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

```
for(i in seq_along(x)) {
                        print(x[i])
}  ## seq_along gives all the elements
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

```
for(letter in x) {
                  print(letter)
}  ## here "letter" is just an object. Gives all values.
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
## [1] "e"
```

```
for(i in 1:4) print(x[i]) ## in the same line, the "{" is not needed.
```

```
## [1] "a"
## [1] "b"
## [1] "c"
## [1] "d"
```

They can also be nested:

```
x <- matrix(1:6, 2, 3)
x
```

```
##      [,1] [,2] [,3]
## [1,]    1    3    5
## [2,]    2    4    6
```

```
for(i in seq_len(nrow(x))) {
     ## seq_len creates an integer sequence up to the length of the
     ## object. In this case, the number of rows of x, so the
     ## sequence is 1 2.
          for(j in seq_len(ncol(x))) {
            ## here seq_len will create the sequence 1 2 3,
            ## as there are 3 columns.
                  print(x[i, j])
            ## as i and j change, here they are the same as the row
            ## and column from the matrix. the subsetting will
            ## extract only one element.
          }
}
```

```
## [1] 1
## [1] 3
## [1] 5
## [1] 2
## [1] 4
## [1] 6
```

Nesting loops beyond 2 or 3 levels can be very difficult to read.

"While" Loops

The loop begins by testing a condition, and if it's true, runs the code. It then tests the condition again before each iteration.

```
count <- 0
while(count < 10) {
                  print(count)
                  count <- count + 1
}
```

```
## [1] 0
## [1] 1
## [1] 2
## [1] 3
## [1] 4
## [1] 5
## [1] 6
## [1] 7
## [1] 8
## [1] 9
```

Can become infinite if not well written. Be careful.

There can be more than one condition in the test

```
z <- 5
while(z >= 3 && z <= 10) {
          coin <- rbinom(1, 1, 0.5)  ## set to 1 or 0 randomly
          print(c(z, coin)) ## print value and coin result
          if(coin == 1) {
                    z <- z + 1
          } else {
                    z <- z - 1
          }     ## step up or down at random until a value is reached
                ## that doesn't meet the conditions.
}
```

```
## [1] 5 1
## [1] 6 0
## [1] 5 1
## [1] 6 0
## [1] 5 0
## [1] 4 1
## [1] 5 0
## [1] 4 1
## [1] 5 1
## [1] 6 0
## [1] 5 0
## [1] 4 0
## [1] 3 1
## [1] 4 1
## [1] 5 0
## [1] 4 0
## [1] 3 1
## [1] 4 1
## [1] 5 0
## [1] 4 0
## [1] 3 0
```

Conditions are evaluated left to right.

"Repeat" Loops, "Next" and "Break"

Repeat is not frequently used because it continues to run forever, or until *break* is called within it (typically inside an if function).

```r
x0 <- 10000
tol <- 1

repeat {
        x1 <- sample(-x0:x0, 1)
        print(c("x is ", x1), quote = FALSE)
        if(abs(x1 - x0) <= tol) {
           ## breaks the loop in the two values are within a
           ## tolerance limit set as tol
                break
        } else {
        x0 <- x1
        }
}
```

```
## [1] x is  9747
## [1] x is  6962
## [1] x is  5775
## [1] x is  -2026
## [1] x is  796
## [1] x is  329
## [1] x is  65
## [1] x is  -52
## [1] x is  43
## [1] x is  -19
## [1] x is  4
## [1] x is  3
```

It's safer to use a *for* loop with many iterations, then report whether a break was achieved or not.

```r
x0 <- 10000
tol <- 1
conv <- "Convergeance has NOT been achieved"

for(i in 1:10) {
        x1 <- sample(-x0:x0, 1)
        print(c("x is ", x1), quote = FALSE)
        if(abs(x1 - x0) <= tol) {
                print(c(i, "iterations"), quote = FALSE)
                conv <- "Convergeance HAS been achieved"
                break
        } else {
        x0 <- x1
        }
}
```

```
## [1] x is  -4499
## [1] x is  -95
## [1] x is  -85
## [1] x is  -26
## [1] x is  -26
## [1] 5          iterations
```

```r
print(conv)
```

```
## [1] "Convergeance HAS been achieved"
```

This way, if there's a problem with the code, the program will still finish and give you an end. *return* can also finish a loop, but also finishes any functions. Will return to this argument later.

Next can skip one iteration of a loop. E.g.:

```r
for(i in 1:30) {
            if(i <= 10) {
                    next
                    ## skips first 10 iterations
            }
            print(i)
}
```

```
## [1] 11
## [1] 12
## [1] 13
## [1] 14
## [1] 15
## [1] 16
## [1] 17
## [1] 18
## [1] 19
## [1] 20
## [1] 21
## [1] 22
## [1] 23
## [1] 24
## [1] 25
## [1] 26
## [1] 27
## [1] 28
## [1] 29
## [1] 30
```

Functions

Functions are typically held in a separate text file.

```r
source("My_Functions.R")
```

```
## [1] "Code read"
```

```r
## a simple function that adds 2 numbers together
add2(286, 365)
```

```
## [1] 651
```

```r
## a simple function that gives all elements of a vector that are
## higher than a threshold. Default threshold is 10.
above(1:20)
```

```
##  [1] 11 12 13 14 15 16 17 18 19 20
```

```r
## This function gives the mean value of each column in a matrix
## or data frame. NAs are removed by default, but this can be turned
## off
columnmean(airquality)
```

```
## [1]  42.129310 185.931507   9.957516  77.882353   6.993464  15.803922
```

Functions are stored as R objects, with class “function”. They are “first class objects”, which means they can be treated similarly to other objects. They can be passed as arguments in other functions and they can be nested.
The return value of a function is the last expression in its body to be evaluated.

Not all arguments of a function must be included. They may have default values, and they may be called by name or position. E.g.:

```r
mydata <- rnorm(100)
## consider the sd function, that calculates standard deviations.
sd(mydata)
```

```
## [1] 1.080959
```

```r
sd(x = mydata)
```

```
## [1] 1.080959
```

```r
sd(x = mydata, na.rm = FALSE)
```

```
## [1] 1.080959
```

```r
sd(na.rm = FALSE, x = mydata)
```

```
## [1] 1.080959
```

```r
sd(na.rm = FALSE, mydata)
```

```
## [1] 1.080959
```

```r
   ## in th last example, the order is changed. it might work but
   ## is not recommended.
```

Positional matching and matching by name can be mixed, which is useful for long list of arguments. E.g.:

```r
mydata <- data.frame(x = 1:50, y = sequence(50, from = 27, by = 2))

args(lm)
```

```
## function (formula, data, subset, weights, na.action, method = "qr",
##     model = TRUE, x = FALSE, y = FALSE, qr = TRUE, singular.ok = TRUE,
##     contrasts = NULL, offset, ...)
## NULL
```

```r
y <- 10
x <- 2
  ## the lm() functions has many arguments, some of which
  ## have defaults
lm(y ~ x, mydata, 1:100, model = FALSE)
```

```
##
## Call:
## lm(formula = y ~ x, data = mydata, subset = 1:100, model = FALSE)
##
## Coefficients:
## (Intercept)            x
##          25            2
```

```r
lm(data = mydata, y ~ x, model = FALSE, 1:100)
```

```
##
## Call:
## lm(formula = y ~ x, data = mydata, subset = 1:100, model = FALSE)
##
## Coefficients:
## (Intercept)            x
##          25            2
```

```r
  ## both commands give the same result. But mixing matching is
  ## not recommended.
```

Function arguments can also be partially matched for a named argument. When an argument is given, the program will, in order, attempt to match:
1. an exact name
2. a partial match
3. a positional match

In addition, you can also set an argument to *NULL*. It can mean a variety of things.

```r
source("My_Functions.R")
```

```
## [1] "Code read"
```

```r
args(above)
```

```
## function (x, n = 10)
## NULL
```

```r
above(1:20, n = NULL)
```

```
## integer(0)
```

The arguments to a function are only evaluated as they are needed.

```r
f <- function(a, b) {
        a^2
}
f(2)
```

```
## [1] 4
```

```r
        ## even though argument b is undefined and has no default,
        ## it still gives a logical value, because argument b is
        ## not used.

f <- function(a, b) {
        print(a)
        print(b)
}
f(2)
```

```
## [1] 2
```

```
## Error in print(b): argument "b" is missing, with no default
```

```r
        ## error only occurs when b is needed. the function
        ## works correctly up to then.
```

The "…" Argument

Indicates a variable number of arguments which are usually passed onto another function. Often used to extend another function, without copying the entire list of arguments from that function.

```r
myplot <- function(x, y, type = "l", ...) {
            plot(x, y, type = type, ...)
}  ## imitates plot functions, but sets type to "l" by default.
```

Generic functions use … so that extra arguments can be passed to methods (more on this later).

It is also necessary when the number of arguments cannot be known in advance.

```r
args(paste)
```

```
## function (..., sep = " ", collapse = NULL, recycle0 = FALSE)
## NULL
```

```
x <- "Hello"
y <- "world!"
z <- "I'm"
w <- "R!"
paste(x, y, z, w, sep = " ", collapse = NULL, recycle0 = FALSE)
```

```
## [1] "Hello world! I'm R!"
```

```
      ## several arguments can be placed in the ... position.

args(cat)  ## similar function
```

```
## function (..., file = "", sep = " ", fill = FALSE, labels = NULL,
##     append = FALSE)
## NULL
```

You **CANNOT** use positional matching nor partial matching for arguments that come after the "..."

Scoping Rules

Symbol Binding

What happens if you assign a function string that already exists?

```
lm <- function(x) x * x
lm(5)
```

```
## [1] 25
```

the *lm()* function already exists and is defined in the stats package. But R will first search for the meaning of lm in the *global environment* (the workspace and objects set by the user). After that it works through a list of other environments in order, until it find it. The list can be viewed with the *search()* function.

```
search()
```

```
##  [1] ".GlobalEnv"        "package:ggplot2"   "package:stats"
##  [4] "package:graphics"  "package:grDevices" "package:utils"
##  [7] "package:datasets"  "package:methods"   "Autoloads"
## [10] "package:base"
```

Order matters! Can be configured by the user. When loading a package with the *library()* function, that package gets put to position 2 in the list, after the global environment, and everything else is pushed down one level.
Note that R has separate namespaces for functions and non-functions. So it is possible to have an object named *c* and a function named *c*. (Though on the global environment there can only be one object named *c*)

R Scoping Rules

- Determine how a value is associated with a free variable in a function
- *lexical* aka *static* scoping. (an alternative is *dynamic* scoping)
- R uses the *search* list to bind a value to a symbol
- *lexical scoping* is particularly useful for sumplifying statistical computations.

```
f <- function(x, y) {
            x^2 + y / z
}
```

How does R assign a value to *z*, if it's not defined in the function header?

- *the values of free variables are searched for in the environment in which the function was defined.*
  - An environment is a collection of (symbol, value) pairs
  - every environment has a parent environment, and can have multiple children environments.
  - the only environment without a parent is the empty environment
  - a function + an environment = a closure (or function closure)
- First R searches for the free variable in the environment of the function. It then continues searching in the parent environment.
- the search continues down the sequence until it reaches the top-level environment.
- after that, it continues down the *search* list until it hits the empty environment. If no value is found, an error is shown.

```
make.power <- function(n) {
                pow <- function(x) {
                         x^n
                }
                pow
}  ## this function returns another function as its value.
   ## so i can use:
cube <- make.power(3)
square <- make.power(2)
   ## to make functions "cube" and "square" that call "pow"
   ## with a value
cube(5)
```

```
## [1] 125
```

```
square(5)
```

```
## [1] 25
```

- *ls(environment())* can be used to call the environment of a function. (*ls()* lists objects.)

- *get()* returns the value of an object, and can be used to find the value of arguments of a function, e.g.:

```
ls(environment(cube))
```

```
## [1] "n"   "pow"
```

```
get("n", environment((cube)))
```

```
## [1] 3
```

```
ls(environment(square))
```

```
## [1] "n"   "pow"
```

```
get("n", environment((square)))
```

```
## [1] 2
```

What does this mean?

```
y <- 10
f <- function(x) {
          y <- 2
          y^2 + g(x)
}

g <- function(x) {
          y * x
}
f(3)
```

```
## [1] 34
```

```
        ## results in 34 because y is set to 2 in the function f,
        ## but not in the function g
g(3)
```

```
## [1] 30
```

```
        ## results in 30 because the environment of g doesn't
        ## include setting y to 2.
```

- in *lexical* scoping the value of y is looked up in the environment where the function is defined.
- in *dynamic* scoping the value of y is looked up in the environment where the function is **called** (known as "calling environment")
  – in R this is called the "parent frame"

Consequences of Lexical Scoping

- In R all objects must be stored in memory.
- functions must carry a pointer to their respective environments, which could be anywhere.
- In S-PLUS, for comparison, free variables are always looked up in the global workspace. So everything can be stored in the disk, because the "defining environment" of all functions was the same.

Coding Standards

Making code readable.

- write your code using a text editor and save it as a text file.
  – these can be read by any editing program.
- indent your code. (8 spaces per tab?)
- limit the width of your code.(80 columns?).
  – forces you to limit nesting loops, which is not very readable.
- limit the length of individual functions: one basic activity.
  – helps you debug functions and which function is malfunctioning.

Dates and Times in R

Represented using special object classes:

- Dates are represented by the *Date* class
  – Dates are stored internally as the number of days since 1970-01-01
- Times are represented by the *POSIXct* or the *POSIXlt* class
  – Times are stored internally as the number of seconds since 1970-01-01
  – *POSIXct* is stored internally as a large integer, and is useful for storing times in something like a data frame
  – *POSIXlt* is stored internally as a list, and includes other information such as day of the week, day of the year, day of the month.
  – Functions that work on dates and times:
  — *weekdays*
  — *months*
  — *quarters*

```
x <- as.Date("1970-01-01")
x
```

```
## [1] "1970-01-01"
```

```
unclass(x)
```

```
## [1] 0
```

```
unclass(as.Date("1970-01-02"))
```

```
## [1] 1
```

```
unclass(as.Date("1969-12-31"))
```

```
## [1] -1
```

```
y <- Sys.time()
y
```

```
## [1] "2020-10-09 13:54:41 NZDT"
```

```
p <- as.POSIXlt(y)
names(unclass(p))
```

```
##  [1] "sec"    "min"    "hour"   "mday"   "mon"    "year"   "wday"   "yday"
##  [9] "isdst"  "zone"   "gmtoff"
```

```
p$sec
```

```
## [1] 41.23495
```

```
p$yday
```

```
## [1] 282
```

```
y  ## Sys.time() is already in POSIXct format
```

```
## [1] "2020-10-09 13:54:41 NZDT"
```

```
unclass(x)
```

```
## [1] 0
```

```
x$sec  ## returns error since it is stored as only an integer
```

```
## Error in x$sec: $ operator is invalid for atomic vectors
```

The *strptime* function helps convert dates and times written in different formats.

```
datestring <- c("January 10, 2020 16:45", "September 26, 1986 03:21")
x <- strptime(datestring, "%B %d, %Y %H:%M")
x
```

```
## [1] "2020-01-10 16:45:00 NZDT" "1986-09-26 03:21:00 NZST"
```

```r
class(x)
```

```
## [1] "POSIXlt" "POSIXt"
```

```r
## use the help page to remember what the strings are
```

You can use data and time objects to perform calculations, comparisons, etc. But they must be the same class. The operators keep track of tricky things like leap years and time-zones. Plotting functions also recognise date & time objects and interpret them in unique ways.

Week 2 Quiz Questions

```r
f <- function(x) {
        g <- function(y) {
                y + z
        }
        z <- 4
        x + g(x)
}
z <- 10
f(3)
```

```
## [1] 10
```

Week 2 Programming Assignment

Part 1

```r
## Write a function named 'pollutantmean' that calculates the mean of a
## pollutant (sulfate or nitrate) across a specified list of monitors. The
## function 'pollutantmean' takes three arguments: 'directory', 'pollutant',
## and 'id'. Given a vector monitor ID numbers, 'pollutantmean' reads that
## monitors' particulate matter data from the directory specified in the
## 'directory' argument and returns the mean of the pollutant across all of
## the monitors, ignoring any missing values coded as NA. A prototype of
## the function is as follows

##  pollutantmean <- function(directory, pollutant, id = 1:332) {
                ## 'directory' is a character vector of length 1 indicating
                ## the location of the CSV file.

                ## 'pollutant' is a character value of length 1 indicating
                ## the name of the pollutant for which we will calculate the
                ## mean; either "sulfate" or "nitrate".

                ## 'id' is an integer vector indicating the monitor ID
                ## numbers to be used

                ## Return the mean of the pollutant across all monitors list
                ## in the 'id' vector (ignoring NA values_
                ## NOTE: Do not round the result)
##  }

## You can see some example output from this function below. The function
## that you write should be able to match this output. Please save your code
## to a file named pollutantmean.R.

## https://d3c33hcgiwev3.cloudfront.net/_3b0da118473bfa0845efddcbe29cc336_pollutantmean-demo.html?Expir

source("pollutantmean.R")
a <- pollutantmean("specdata", "sulphate", 1:10)
b <- pollutantmean("specdata", "nitrate", 70:72)
c <- pollutantmean("specdata", "nitrate", 23)
c(a, b, c)
```

```
## [1] 4.064128 1.706047 1.280833
```

Part 2

```
## Write a function that reads a directory full of files and reports the
## number of completely observed cases in each data file. The function
## should return a data frame where the first column is the name of the
## file and the second column is the number of complete cases. A
## prototype of this function follows:

## complete <- function(directory, id = 1:322) {
        ## 'directory' is a character vector of length 1 indicating
        ## the location of the CSV files

        ## 'id' is an integer vector indicating the monitor ID numbers
        ## to be used

        ## Return a data frame of the form:
        ## id nobs
        ## 1 117
        ## 2 1041
        ## ...
        ## where 'id' is the monitor ID number and 'nobs' is the
        ## number of complete cases.
## }

## You can see some example output from this function below. The function
## that you write should be able to match this output. Please save your
## code to a file named complete.R. To run the submit script for this part,
## make sure your working directory has the file complete.R in it.
##
## https://d3c33hcgiwev3.cloudfront.net/_3b0da118473bfa0845efddcbe29cc336_complete-demo.html?Expires=16
```

```r
source("complete.R")
complete("specdata", 1)
```

```
##   id nobs
## 1  1  117
```

```r
complete("specdata", c(2, 4, 8, 10, 12))
```

```
##   id nobs
## 1  2 1041
## 2  4  474
## 3  8  192
## 4 10  148
## 5 12   96
```

```r
complete("specdata", 30:25)
```

```
##   id nobs
## 1 30  932
## 2 29  711
## 3 28  475
## 4 27  338
## 5 26  586
## 6 25  463
```

```r
complete("specdata", 3)
```

```
##   id nobs
## 1  3  243
```

Part 3

```r
## Write a function that takes a directory of data files and a threshold for
## complete cases and calculates the correlation between sulfate and
## nitrate for monitor locations where the number of completely observed
## cases (on all variables) is greater than the threshold. The function
## should return a vector of correlations for the monitors that meet the
## threshold requirement. If no monitors meet the threshold requirement,
## then the function should return a numeric vector of length 0. A prototype
## of this function follows

## corr <- function(directory, threshold = 0) {
## 'directory' is a character vector of length 1 indicating the location of
## the CSV files

## 'threshold' is a numeric vector of length 1 indicating the number of
## completely observed observations (on all variables) required to compute
## the correlation between nitrate and sulfate; the default is 0
## return a numeric vector of correlations
## NOTE: Do not round the result!
## }

## For this function you will need to use the 'cor' function in R which
## calculates the correlation between two vectors. Please read the help
## page for this function via '?cor' and make sure that you know how to
## use it.

## You can see some example output from this function below. The function
## that you write should be able to approximately match this output. Note
## that because of how R rounds and presents floating point numbers, the
## output you generate may differ slightly from the example output. Please
## save your code to a file named corr.R. To run the submit script for this
## part, make sure your working directory has the file corr.R in it.

## https://d3c33hcgiwev3.cloudfront.net/_e92e575b8e62dcb1e3a086d2ff0d5a1e_corr-demo.html?Expires=160099

source("corr.R")
source("complete.R")
cr <- corr("specdata", 150)
head(cr)
```

```
## [1] -0.01895754 -0.14051254 -0.04389737 -0.06815956 -0.12350667 -0.07588814
```

```r
summary(cr)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.21057 -0.05147  0.09333  0.12401  0.26836  0.76313
```

```r
cr <- corr("specdata", 400)
head(cr)
```

```
## [1] -0.01895754 -0.04389737 -0.06815956 -0.07588814  0.76312884 -0.15782860
```

```r
summary(cr)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -0.17623 -0.03109  0.10021  0.13969  0.26849  0.76313
```

```r
cr <- corr("specdata", 5000)
summary(cr)
```

```
##    Min. 1st Qu.  Median    Mean 3rd Qu.    Max.
##
```

```r
length(cr)
```

```
## [1] 0
```

```
cr <- corr("specdata")
summary(cr)
```

```
##     Min.  1st Qu.   Median     Mean  3rd Qu.     Max.
## -1.00000 -0.05282  0.10718  0.13684  0.27831  1.00000
```

```
length(cr)
```

```
## [1] 323
```

Swirl Exercises Week 2

Swirl Lesson 8: Logic

- Logical operators ==, *!=*, >, >=, <, <=
  - *%%* is used to find the remainder.
- *"!"* in front of a *TRUE* returns a *FALSE* and vice-versa

```
15 %% 7
```

```
## [1] 1
```

```
TRUE == TRUE
```

```
## [1] TRUE
```

```
(FALSE == TRUE) == FALSE
```

```
## [1] TRUE
```

```
5 != 7
```

```
## [1] TRUE
```

```
! 5 == 7
```

```
## [1] TRUE
```

```
!(0 >= -1)
```

```
## [1] FALSE
```

```
(TRUE != FALSE) == !(6 == 7)
```

```
## [1] TRUE
```

- *"&"* and *"&&"* return TRUE only if both parts are TRUE
  - *"&"* evaluates each element in a vector (recycling may happen).
  - *"&&"* evaluates only the first element of vectors.

- *"|"* and *"||"* return TRUE if either parts are TRUE
  - same differences as for *"&"* and *"&&"*

- All **AND** operators are evaluated before **OR** operators.

```
TRUE & TRUE
```

```
## [1] TRUE
```

```
FALSE & TRUE
```

```
## [1] FALSE
```

```
FALSE & FALSE
```

```
## [1] FALSE
```

```
TRUE & c(TRUE, FALSE, FALSE)
```

```
## [1]  TRUE FALSE FALSE
```

```
TRUE && c(TRUE, FALSE, FALSE)
```

```
## [1] TRUE
```

```
6 != 10 && FALSE && 1 >= 2
```

```
## [1] FALSE
```

```
5 > 8 || 6 != 8 && 4 > 3.9
```

```
## [1] TRUE
```

- *isTRUE()* returns TRUE only if the argument inside is *TRUE*, otherwise it returns *FALSE*.

- *identical()* returns TRUE is both arguments are identical.

- *xor()* is the exclusive **OR**. It returns TRUE if only one element is TRUE.

- *which()* takes a logical vector as argument and returns the indeces of TRUE elements.

- *any()* and *all()* return TRUE if any or all elements of a logical vector are TRUE, respectively.

```r
isTRUE("egg salad")
```

```
## [1] FALSE
```

```r
isTRUE(5 != 4)
```

```
## [1] TRUE
```

```r
identical("hello", "Hello")
```

```
## [1] FALSE
```

```r
xor(5 == 6, !FALSE)
```

```
## [1] TRUE
```

```r
xor(5 <= 6, 4*3 == 12)
```

```
## [1] FALSE
```

```r
which(c(TRUE, FALSE, TRUE))
```

```
## [1] 1 3
```

```r
any(c(1, 2, 3, 4, 5, 6) < 0)
```

```
## [1] FALSE
```

```r
all(c(1, 2, 3, 4, 5, 6) > 0)
```

```
## [1] TRUE
```

Swirl Lesson 9: Functions

Functions as arguments

You can pass functions as arguments to other functions just like you can pass data to functions. Let's say you define the following functions:

```r
add_two_numbers <- function(num1, num2){
    num1 + num2
}

multiply_two_numbers <- function(num1, num2){
  num1 * num2
}
```

```
some_function <- function(func){
    func(2, 4)
}

some_function(add_two_numbers)
```

```
## [1] 6
```

```
some_function(multiply_two_numbers)
```

```
## [1] 8
```

```
some_function(paste)
```

```
## [1] "2 4"
```

As you can see we use the argument name "func" like a function inside of "some_function()."

Finish the function definition below so that if a function is passed into the "func" argument and some data (like a vector) is passed into the dat argument the evaluate() function will return the result of dat being passed as an argument to func.

```
evaluate <- function(func, dat){
  func(dat)
}

evaluate(sum, c(2, 4, 6))
```

```
## [1] 12
```

```
evaluate(median, c(7, 40, 9))
```

```
## [1] 9
```

```
evaluate(floor, 11.1)
```

```
## [1] 11
```

Ellipses as arguments

The ellipses can be used to pass on arguments to other functions that are used within the function you're writing. Usually a function that has the ellipses as an argument has the ellipses as the last argument. The usage of such a function would look like: *ellipses_func(arg1, arg2 = TRUE, . . .)*
In the above example arg1 has no default value, so a value must be provided for arg1. arg2 has a default value, and other arguments can come after arg2 depending on how they're defined in the ellipses_func() documentation. Interestingly the usage for the paste function is as follows:
paste (. . . , sep = " ", collapse = NULL)
Notice that the ellipses is the first argument, and all other arguments after the ellipses have default values. This is a strict rule in R programming: all arguments after an ellipses must have default values. Take a look at the simon_says function below:

```
simon_says <- function(...){
   paste("Simon says:", ...)
}
simon_says("put your hands on your head!")
```

```
## [1] "Simon says: put your hands on your head!"
```

Telegrams used to be peppered with the words START and STOP in order to demarcate the beginning and end of sentences. Write a function below called telegram that formats sentences for telegrams. For example the expression `telegram("Good", "morning")` should evaluate to: *"START Good morning STOP"*

```
telegram <- function(...){
  paste("START", ..., "STOP")
}
telegram("Good", "morning")
```

```
## [1] "START Good morning STOP"
```

Unpacking Elipses

Let's explore how to "unpack" arguments from an ellipses when you use the ellipses as an argument in a function. Below I have an example function that is supposed to add two explicitly named arguments called alpha and beta.

```
add_alpha_and_beta <- function(...){
   # First we must capture the ellipsis inside of a list
   # and then assign the list to a variable. Let's name this
   # variable `args`.

   args <- list(...)

   # We're now going to assume that there are two named arguments within args
   # with the names `alpha` and `beta.` We can extract named arguments from
   # the args list by using the name of the argument and double brackets. The
   # `args` variable is just a regular list after all!

   alpha <- args[["alpha"]]
   beta  <- args[["beta"]]

   # Then we return the sum of alpha and beta.

   alpha + beta
}

add_alpha_and_beta(alpha = 100, beta = 20)
```

```
## [1] 120
```

Have you ever played Mad Libs before? The function below will construct a sentence from parts of speech that you provide as arguments. We'll write most of the function, but you'll need to unpack the appropriate arguments from the ellipses.

```r
mad_libs <- function(...){
  # Do your argument unpacking here!

  args <- list(...)
  place <- args["place"]
  adjective <- args["adjective"]
  noun <- args["noun"]

  # Don't modify any code below this comment.
  # Notice the variables you'll need to create in order for the code below to
  # be functional!
  paste("News from", place, "today where", adjective, "students took to the streets in protest of the n
}
mad_libs(place = "Christchurch", adjective = "uncool", noun = "Ski ramp")
```

```
## [1] "News from Christchurch today where uncool students took to the streets in protest of the new Sk
```

Creating Binary Operators

The syntax for creating new binary operators in R is unlike anything else in R, but it allows you to define a new syntax for your function. I would only recommend making your own binary operator if you plan on using it often!
User-defined binary operators have the following syntax:
%[whatever]%
where [whatever] represents any valid variable name.
Let's say I wanted to define a binary operator that multiplied two numbers and then added one to the product. An implementation of that operator is below:

```r
"%mult_add_one%" <- function(left, right){ # Notice the quotation marks!
   left * right + 1
}
4 %mult_add_one% 5
```

```
## [1] 21
```

Write your own binary operator below from absolute scratch! Your binary operator must be called %p% so that the expression:
"Good" %p% "job!"
will evaluate to: "Good job!"

```r
"%p%" <- function(left, right){
  paste(left, right)
}
"Hello" %p% "world!"
```

```
## [1] "Hello world!"
```

Swirl Lesson 14: Dates and Times

- *Sys.Date()* gets a date object with the current date

- *as.Date("1986-09-26")* to create a custom date object
  – use *unclass(Sys.Date())* to see that dates are stored as days since 1970-01-01.
  – dates before 1970 are stored as negative numbers.

- *Sys.time()* is a POSIXct object with the current time.
  – time is stored internally as the number of seconds since 1970-01-01 00:00:00

- *as.POSIXlt(Sys.time())* coerces it into a POSIXlt object
  – it is stored internally as a list, with elements representing seconds, minutes, hours, day of month, month, year, day of week, day of year, daylight savings, time zone, and two more time zone related elements.

```
t1 <- as.POSIXlt("1986-09-26 12:45:00 BRT")
str(unclass(t1))  ## str() Compactly displays structure of an object.
```

```
## List of 11
##  $ sec   : num 0
##  $ min   : int 45
##  $ hour  : int 12
##  $ mday  : int 26
##  $ mon   : int 8
##  $ year  : int 86
##  $ wday  : int 5
##  $ yday  : int 268
##  $ isdst : int 0
##  $ zone  : chr "NZST"
##  $ gmtoff: int NA
```

```
t1$min  ## easily extracted elements
```

```
## [1] 45
```

- *weekdays()* returns day of the week for any date or time object.

- *months()* returns the month.

- *quarters()* returns the quarter of th year.
  – there functions work on date, POSIXct and POSIXlt objects.

```
t1 <- as.POSIXlt("1986-09-26 12:45:00 BRT")
weekdays(t1)
```

```
## [1] "Friday"
```

```
months(t1)
```

```
## [1] "September"
```

```
quarters(t1)
```

```
## [1] "Q3"
```

- *strptime()* can extract dates and times from formats not recognised by R.
  – it converts a character vector into POSIXlt.

```
t1 <- "October 17, 1986 08:24"
strptime(t1, "%B %d, %Y %H:%M")
```

```
## [1] "1986-10-17 08:24:00 NZST"
```

- arithmetic operations + and -, and comparisons <, > and == can be performed on date and time objects.

- *difftime()* allows for more precise control of units when performing subtractions.

R Programming Week 3: Loop Functions and Debugging

Writing *for* and *while* loops in the command line is cumbersome. The loop functions that contain the word *"apply"* are more convenient. We'll look into several variants.

Learning Objectives
- Use the loop functions *apply*, *lapply*, *tapply*, *mapply* and *split*.
- Define an anonymous function and describe its use in loop functions.
- Describe how to start the R debugger for an arbitrary R function.
- Describe what the traceback() function does and what is the function call stack.

Loop Functions - *lapply*

Loop over a list of objects and apply a function to every element. *sapply* does the same and tries to simplify the results.

```
lapply
```

```
## function (X, FUN, ...)
## {
##     FUN <- match.fun(FUN)
##     if (!is.vector(X) || is.object(X))
##         X <- as.list(X)
##     .Internal(lapply(X, FUN))
## }
## <bytecode: 0x0000000012aa7da8>
## <environment: namespace:base>
```

- *"X"* must be a list or it will be coerced into a list.

- *"FUN"* is a function to apply and "..." are arguments of that function.

- the actual looping is done internally through C code, to make it faster.

- always returns a list. Names are preserved.

```
## Example
x <- list(a = 1:5, b = rnorm(10), c = rnorm(5, 10))
lapply(x, mean)
```

```
## $a
## [1] 3
##
## $b
## [1] -0.1743012
##
## $c
## [1] 10.12245
```

```
## runif() function generates "n" random numbers between 0 and 1
lapply(1:4, runif)
```

```
## [[1]]
## [1] 0.2593938
##
## [[2]]
## [1] 0.4172592 0.9962143
##
## [[3]]
## [1] 0.80678051 0.05278674 0.63111060
##
## [[4]]
## [1] 0.8576799 0.3805782 0.8052955 0.6845491
```

```
## optional arguments of runif() can be passed into the "..." argument
lapply(1:4, runif, min = 0, max = 10)
```

```
## [[1]]
## [1] 4.069254
##
## [[2]]
## [1] 8.4599014 0.6985098
##
## [[3]]
## [1] 5.586301 7.582894 8.288604
##
## [[4]]
## [1] 1.476704 1.519734 2.562934 1.527662
```

Using loop functions with anonymous functions

Anonymous functions can be very useful in combination with *apply* loops.

```
## Consider a list of matrices
x <- list(a = matrix(1:4, 2, 2), b = matrix(1:6, 3, 2))
x
```

```
## $a
##      [,1] [,2]
## [1,]    1    3
## [2,]    2    4
##
## $b
##      [,1] [,2]
## [1,]    1    4
## [2,]    2    5
## [3,]    3    6
```

```r
## we can create an anonymous function to extract the first column of each
## matrix
lapply(x, function(arg) arg[ ,1])
```

```
## $a
## [1] 1 2
##
## $b
## [1] 1 2 3
```

sapply will try to simplify the result of lapply

- If the result is a list where every element is length 1, then a vector is returned.
- If the result is a list where every element is a vector with the same length, a matrix is returned.
- If sapply can't simplify the output, a list is returned.

```r
x <- list(a = 1:5, b = rnorm(10), c = rnorm(5, 10))
sapply(x, mean)  ##returns a vector instead of a list
```

```
##           a           b           c
##  3.00000000 -0.05766038  9.79864125
```

```r
mean(x) ## doesn't work because mean doesn't apply to lists.
```

```
## Warning in mean.default(x): argument is not numeric or logical: returning NA

## [1] NA
```

```r
x <- list(a = 1:5, b = rnorm(10), c = rnorm(5, 10), d = c("a", "b", "c"))
y <- sapply(x, mean)  ## calculates means where possible and returns NA
```

```
## Warning in mean.default(X[[i]], ...): argument is not numeric or logical:
## returning NA
```

```r
y                      ## otherwise
```

```
##           a           b           c           d
## 3.000000000 0.002920501 9.906119165          NA
```

```
class(y)
```

```
## [1] "numeric"
```

Loop functions - *apply*

Used to evaluate a function over the margins of an array (matrices or 3d arrays). It used to be faster than using a a for loop, but that's no longer the case. It just requires less typing.

```
args(apply)
```

```
## function (X, MARGIN, FUN, ...)
## NULL
```

- *"X"* is an array (e.g. a matrix).
- *"MARGIN"* is an integer vector indicating which margins should be retained.
- *"FUN"* is a function.
- *". . . "* is for arguments to be passed to the function.

```
x <- matrix(rnorm(200), 20, 10)  ## an example matrix

## Calculate the mean of each column
## dimension 1 has 20 rows and dimension 2 has 10 columns.
apply(x, 2, mean) ## preserve columns, colapse rows
```

```
##  [1]  0.06301671  0.13747918  0.23665856  0.32512812  0.16434357 -0.11346865
##  [7] -0.17936281 -0.01775737 -0.23231724  0.09590819
```

```
## Calculate the sum of each row
apply(x, 1, sum) ## preserve rows, collapse columns.
```

```
##  [1] -3.2851222 -0.3428815 -2.7114357  5.5307796 -5.2923320  1.1462172
##  [7]  6.8314410  4.6236969  6.5513020 -4.2053624  1.2119266 -5.8492805
## [13]  5.0950744  5.3751881  3.3585426  2.0111818 -5.3794222 -3.4950357
## [19] -1.4077275 -0.1741852
```

For sums and means of matrix dimensions, however, there are faster, optimized functions:
- *rowSums()* = apply(x, 1, sum)
- *rowMeans()* = apply(x, 1, mean)
- *colSums()* = apply(x, 2, sum)
- *colMeans()* = apply(x, 2, mean)

For functions that return a vector of length > 1, *apply()* can create a matrix.

```
## Another example
## signif() is used to round to significant figures.
x <- matrix(signif(rnorm(50, 10, 5),3), 5, 10)
x
```

```
##       [,1]  [,2]  [,3]  [,4]  [,5]  [,6]  [,7]  [,8]  [,9] [,10]
## [1,]  1.11  6.62 17.10  3.86  8.97  4.27  9.47 10.50  1.92  5.76
## [2,] 15.50  9.27 14.20  5.94 14.50 16.40 19.60  5.17  6.50  9.41
## [3,] 10.40 15.10 16.50 14.60 10.50  9.77 11.70 12.00 12.40  7.78
## [4,] 13.50 10.10 10.10  4.61 14.80  6.07  8.12  8.16 11.10 10.50
## [5,]  8.08 14.00  6.32  9.04 17.40  5.89 14.80 13.00 12.00  7.90
```

```
## the quantile() function calculates percentiles in a population.
apply(x, 1, quantile, probs = c(0, 0.5, 1))
```

```
##       [,1]   [,2]  [,3]  [,4]  [,5]
## 0%    1.11  5.170  7.78  4.61  5.89
## 50%   6.19 11.805 11.85 10.10 10.52
## 100% 17.10 19.600 16.50 14.80 17.40
```

Each row of the matrix is analysed by the quantile function, which gives a vector of length 3. So *apply()* builds a matrix with the rows of "x" as the column names and the quantiles that the *quantile()* function calculates as the rows.

Using *apply()* in an array:

```
## a is an array of dimensions 2 by 2 by 10
a <- array(signif(rnorm(2 * 2 * 10), 2), c(2, 2, 10))

## preserve dimensions 1 and 2, operate the function of the 3rd dimension.
apply(a, c(1, 2), mean)
```

```
##         [,1]   [,2]
## [1,] -0.4620 0.2400
## [2,]  0.4816 0.0971
```

```
## can also be achieved by:
rowMeans(a, dims = 2)
```

```
##         [,1]   [,2]
## [1,] -0.4620 0.2400
## [2,]  0.4816 0.0971
```

Loop functions - *mapply*

is a multivariate apply. It can apply a function in parallel over a set of arguments. It can have multiple lists as arguments.

```
args(mapply)
```

```
## function (FUN, ..., MoreArgs = NULL, SIMPLIFY = TRUE, USE.NAMES = TRUE)
## NULL
```

- *"FUN"* is a function

- *"... "* contains arguments to apply to the function. (coerced to list)
- *"MoreArgs"* is a list of other arguments to FUN.
- *"SIMPLIFY"* indicates whether the result should be simplified.

```
list(rep(1, 4), rep(2, 3), rep(3, 2), rep(4, 1))
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

```
## instead, we can have mapply use the rep() function
mapply(rep, 1:4, 4:1)
```

```
## [[1]]
## [1] 1 1 1 1
##
## [[2]]
## [1] 2 2 2
##
## [[3]]
## [1] 3 3
##
## [[4]]
## [1] 4
```

```
noise <- function(n, mean = 5, sd = 2) {
        signif(rnorm(n, mean, sd), 2)
}

## If I wanted to create sets of 6 to 10 random numbers with noise(), each
## set having average 2, 4, 6, 8 and 10 respectively.
## noise(6:10, c(2, 4, 6, 8, 10), 2) won't work because the arguments must be single elements.
## Using mapply, we can do this:
mapply(noise, n = 6:10, mean = c(2, 4, 6, 8, 10), sd = 2)
```

```
## [[1]]
## [1] 2.80 5.30 2.70 0.88 2.60 0.45
##
## [[2]]
## [1] 2.2 3.0 4.3 3.4 5.0 2.7 4.7
##
## [[3]]
## [1] 5.9 6.9 7.3 3.7 4.7 2.9 8.7 2.3
##
## [[4]]
## [1]  7.3  6.7 10.0 11.0 12.0  9.8  4.3 10.0  7.0
##
## [[5]]
##  [1]  9.6 10.0 11.0 13.0  8.6 11.0 10.0 11.0  6.0 11.0
```

Use a function that doesn't allow vector inputs and vectorise it.

Loop Functions - *tapply*

used to apply a function over subsets of a vector.

```
args(tapply)
```

```
## function (X, INDEX, FUN = NULL, ..., default = NA, simplify = TRUE)
## NULL
```

- *"X"* is a vector.
- *"Index"* is a vector of same length or a list of factors (coerced into list).
- *"FUN"* is a function.
- *". . . "* contains other arguments of *FUN*.
- *"simplify"* to simplify the result.

```
x <- c(rnorm(10), runif(10), rnorm(10,1)) ## runif is random of uniform dist.
f <- gl(3, 10)  ## generates a numeric vector, elements are 1 repeated k times,
                ## then 2 repeated k times, then 3, until the number is equal
                ## to n.
f
```

```
##  [1] 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 3 3 3 3 3 3 3 3 3 3
## Levels: 1 2 3
```

```
tapply(x, f, mean)
```

```
##          1          2          3
## 0.08514136 0.58167538 0.70835088
```

- the factor variable determines which elements of the vector will be grouped together.

```
tapply(x, f, range)
```

```
## $`1`
## [1] -1.150176  1.944931
##
```

```
## $`2`
## [1] 0.2817242 0.9211490
##
## $`3`
## [1] -0.1729652  2.1925351
```

- Here a minimum and a maximum are given for each group, so the result of *tapply* is a list where each element is a vector of length 2.

Loop Functions - *split*

Works very similarly to *tapply*, but it doesn't apply a function to the elements.

```r
args(split)
```

```
## function (x, f, drop = FALSE, ...)
## NULL
```

- useful for using before *lapply* or *mapply*
- can be used to split much more complicated objects.
- always results in a list.

```r
x <- c(rnorm(10), runif(10), rnorm(10,1))
f <- gl(3, 10)
split(x, f)
```

```
## $`1`
##  [1]  1.1584690  0.4130293 -1.4944173  0.2341887 -1.1568217 -0.3673109
##  [7] -1.1656069  0.7499407  0.3579714  0.2189300
##
## $`2`
##  [1] 0.3068854 0.9314319 0.8774141 0.7182688 0.1571962 0.2207047 0.4192154
##  [8] 0.7844402 0.8508573 0.9400721
##
## $`3`
##  [1]  1.0276878  0.4908139  2.6268009  1.9066116  1.8974786  3.2735570
##  [7]  2.1389448  1.4279899  1.7336307 -0.2690090
```

```r
lapply(split(x, f), mean)  ## exactly same thing as the tapply function
```

```
## $`1`
## [1] -0.1051628
##
## $`2`
## [1] 0.6206486
##
## $`3`
## [1] 1.625451
```

```r
library(datasets)
head(airquality)
```

```
##   Ozone Solar.R Wind Temp Month Day
## 1    41     190  7.4   67     5   1
## 2    36     118  8.0   72     5   2
## 3    12     149 12.6   74     5   3
## 4    18     313 11.5   62     5   4
## 5    NA      NA 14.3   56     5   5
## 6    28      NA 14.9   66     5   6
```

```r
## For example if I wanted to calculate the mean of Ozone, solar radiation,
## wind and temperature within each month. I can split the data frame into
## monthly pieces.
s <- split(airquality, airquality$Month)
lapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
## $`5`
##    Ozone  Solar.R     Wind
##       NA       NA 11.62258
##
## $`6`
##     Ozone   Solar.R      Wind
##        NA 190.16667  10.26667
##
## $`7`
##      Ozone    Solar.R       Wind
##         NA 216.483871   8.941935
##
## $`8`
##    Ozone  Solar.R     Wind
##       NA       NA 8.793548
##
## $`9`
##    Ozone  Solar.R     Wind
##       NA 167.4333  10.1800
```

```r
## includes NAs
```

I can then also use *sapply* to give the results a a matrix instead:

```r
sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")]))
```

```
##                5         6          7        8        9
## Ozone         NA        NA         NA       NA       NA
## Solar.R       NA 190.16667 216.483871       NA 167.4333
## Wind    11.62258  10.26667   8.941935 8.793548  10.1800
```

```r
## and I can remove NAs with the 'na.rm' argument of colMeans

sapply(s, function(x) colMeans(x[, c("Ozone", "Solar.R", "Wind")], na.rm = TRUE))
```

```
##                 5         6          7          8         9
## Ozone    23.61538  29.44444  59.115385  59.961538  31.44828
## Solar.R 181.29630 190.16667 216.483871 171.857143 167.43333
## Wind     11.62258  10.26667   8.941935   8.793548  10.18000
```

Splitting on more than one level

Using two different factors for splitting.

```
x <- rnorm(10)
f1 <- gl(2, 5)
f2 <- gl(5, 2)

## For example if I have a population of measurement and they are tagged
## by both gender and age group.
cbind(x, f1, f2)
```

```
##                 x f1 f2
##  [1,] -1.46454169  1  1
##  [2,]  0.15220458  1  1
##  [3,]  1.77388875  1  2
##  [4,] -0.93897323  1  2
##  [5,]  1.04726254  1  3
##  [6,] -0.08435072  2  3
##  [7,]  0.12174599  2  4
##  [8,]  1.23003647  2  4
##  [9,]  0.12317980  2  5
## [10,]  0.37841770  2  5
```

```
interaction(f1, f2)  ## Combines the levels of both factors. There's now 10.
```

```
##  [1] 1.1 1.1 1.2 1.2 1.3 2.3 2.4 2.4 2.5 2.5
## Levels: 1.1 2.1 1.2 2.2 1.3 2.3 1.4 2.4 1.5 2.5
```

```
str(split(x, list(f1, f2)))
```

```
## List of 10
##  $ 1.1: num [1:2] -1.465 0.152
##  $ 2.1: num(0)
##  $ 1.2: num [1:2] 1.774 -0.939
##  $ 2.2: num(0)
##  $ 1.3: num 1.05
##  $ 2.3: num -0.0844
##  $ 1.4: num(0)
##  $ 2.4: num [1:2] 0.122 1.23
##  $ 1.5: num(0)
##  $ 2.5: num [1:2] 0.123 0.378
```

```
## the split function doesn't require the use of the interaction function.
## Not all levels have observations, they result in an empty vector.

str(split(x, list(f1, f2), drop = TRUE))  ## drops empty levels from the result
```

```
## List of 6
##  $ 1.1: num [1:2] -1.465 0.152
##  $ 1.2: num [1:2] 1.774 -0.939
##  $ 1.3: num 1.05
##  $ 2.3: num -0.0844
##  $ 2.4: num [1:2] 0.122 1.23
##  $ 2.5: num [1:2] 0.123 0.378
```

```r
## very useful for later using an sapply or lapply.
```

Debugging Tools - Diagnosing the Problem

Indication R will give that something's wrong:

- **message**: an expected notification/diagnostic produced by the *message* function; execution of the function continues. Message appears in the middle.
- **warning**: an indication of something wrong, but not fatal. execution of the function continues. generated by the *warning* function. Message appears after the function.
- **error**: a fatal problem has occurred. execution stops. produced by the *stop* function.
- **condition**: a generic concept for indicating that something unexpected can occur. programmer can create their own conditions.

```r
log(-1)  ## Basic warning.
```

```
## Warning in log(-1): NaNs produced
```

```
## [1] NaN
```

```r
printmessage <- function(x) {
        if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than of equal to zero")
        invisible(x)  ## prevents auto-printing
}
printmessage(1)
```

```
## [1] "x is greater than zero"
```

```r
printmessage(NA)  ## error produced!
```

```
## Error in if (x > 0) print("x is greater than zero") else print("x is less than of equal to zero"): mi
```

```r
printmessage2 <- function(x) {
        if(is.na(x))
                print("x is a missing value!")
        else if(x > 0)
                print("x is greater than zero")
        else
                print("x is less than of equal to zero")
        invisible(x)
}
x <- log(-1)
```

```
## Warning in log(-1): NaNs produced
```

```r
printmessage2(x) ## no error produced, but unexpected.
```

```
## [1] "x is a missing value!"
```

How do you know that something is wrong with your function?

- What was your input? How did you call the function? Arguments, etc?
- What were you expecting? Output, messages, other results?
- What did you get?
- How does it differ from what you were expecting?
- Were your expectations correct in the first place?
- Can you reproduce the problem?

Debugging Tools - Basic Tools

The primary tools for debugging functions in R:

- *traceback*: prints out the function call stack after an error occurs: does nothing if there is no error.
- *debug*: flags a function for "debug mode". when the function is next called, it will freeze and you can step through the function line-by-line.
- *browser*: similar to *debug*. when read, execution will suspend and you can go line-by-line. Can stop the function in the middle, instead of the start.
- *trace*: allows you to insert debugging code into a function without changing the function.
- *recover*: allows you to modify the default error behaviour, so you can browse the function call stack.

Debugging Tools - Using the Tools (examples)

Traceback

```
mean(xxx)
```

```
## Error in mean(xxx): object 'xxx' not found
```

```
## traceback()  ## has to be called immediately after error occurs
```

```
lm(xxx - yyy)
```

```
## Error in lm(xxx - yyy): object 'xxx' not found
```

```
## traceback()  ## error actually occurred 4 levels deep.
```

Useful for giving details of errors to others, when asking for help.

Debug

debug is hard to see in static form, but here's an example:

```
# debug(lm)  ## prints out the full body of the next function,
           ## then opens the browser.
lm(xxx - yyy)
```

```
## Error in lm(xxx - yyy): object 'xxx' not found
```

The browser acts just like a separate workspace. The environment of this workspace is the environment of the function. Type *n* and enter to run the next line. You can then use debug on a function inside that function to browse that function.

Recover

```
# options(error = recover)
  ## Sets a global option for this R session.

read.csv("nosuchfile")
```

```
## Warning in file(file, "rt"): cannot open file 'nosuchfile': No such file or
## directory
```

```
## Error in file(file, "rt"): cannot open the connection
```

Instead of getting the console back, you get a menu. This is similar to the output of the traceback function. Error occurred at 3rd level, so you can type 3 to browse the environment of that function.

Swirl Exercises Week 3

Swirl Lesson 10: lapply and sapply

These functions offer a concise and convenient means of implementing the **Split-Apply-Combine strategy** for data analysis. That means, the function will *split* up the data into smaller pieces, *apply* a function to each piece, then *combine* the results.

Consider a data frame with several columns representing measured variables and rows representing observations (for example countries where the measurements were taken). Suppose we want to know the class of object of each column. We could individually look it up with a subsetting function, but since we need to do this for many columns, we can use *lapply* to do it as a loop for all of them. *lapply(dataframe, class)* command will give a list of all the column names and the class of the object. *sapply* does the same, but, if possible, simplifies the result into a vector or matrix instead of a list.

Swirl Lesson 11: vapply and tapply

vapply works similarly to *sapply*, however you specify the object class you want as an output. If the function cannot convert the object into that class, it throws an error. This can prevent significant problems in code, that might be caused by getting an unexpected return value from *sapply*.

```
x <- data.frame(name = c("a", "b", "c", "d", "e"), min = runif(5, 0, 10), max = runif(5, 10, 200), tree
x
```

```
##   name       min       max  tree num
## 1    a 0.8108841 161.77691  TRUE   1
## 2    b 9.2864959  39.53735 FALSE   2
## 3    c 1.4237410 151.26999 FALSE   3
## 4    d 8.4710626  95.88576 FALSE   4
## 5    e 6.5007502  95.16610 FALSE   5
```

```
sapply(x, class)
```

```
##        name         min         max        tree         num
## "character"   "numeric"   "numeric"   "logical"   "integer"
```

```
vapply(x, class, character(1))
```

```
##        name         min         max        tree         num
## "character"   "numeric"   "numeric"   "logical"   "integer"
```

```
vapply(x, class, numeric(1))  ## error produced because output is not numeric
```

```
## Error in vapply(x, class, numeric(1)): values must be type 'double',
##  but FUN(X[[1]]) result is type 'character'
```

tapply splits the data up into groups, based on the value of a variable vector, then applies a function to each group.

```r
x <- data.frame(fus = runif(30), roh = runif(30), dah = runif(30), dborn = gl(5, 6))
x
```

```
##           fus         roh        dah dborn
## 1  0.46680871 0.332666193 0.03532268     1
## 2  0.23553302 0.278080031 0.78182949     1
## 3  0.17421557 0.155778546 0.13477902     1
## 4  0.16916596 0.272749214 0.22951251     1
## 5  0.88023538 0.829216284 0.54923975     1
## 6  0.46067965 0.495685180 0.47886587     1
## 7  0.30730464 0.716341048 0.39937819     2
## 8  0.29591218 0.403606669 0.58256979     2
## 9  0.40027474 0.665255382 0.51143737     2
## 10 0.12102152 0.802446470 0.70563794     2
## 11 0.33785305 0.023107972 0.97569323     2
## 12 0.75412421 0.765980775 0.68492046     2
## 13 0.93120310 0.651170495 0.27826806     3
## 14 0.91561122 0.001457154 0.41021162     3
## 15 0.86324379 0.818511414 0.31949719     3
## 16 0.45216872 0.924479598 0.62971360     3
## 17 0.18163758 0.617131021 0.13254293     3
## 18 0.99181859 0.366522645 0.84079761     3
## 19 0.31042690 0.702102630 0.69324942     4
## 20 0.83983150 0.195511940 0.37970855     4
## 21 0.48538414 0.511670478 0.92752889     4
## 22 0.66193810 0.033542307 0.36341692     4
## 23 0.04463682 0.132201801 0.05887245     4
## 24 0.62640325 0.823139876 0.81414666     4
## 25 0.19679845 0.208414745 0.24732158     5
## 26 0.85617976 0.273137556 0.15943863     5
## 27 0.93598739 0.895340025 0.80445904     5
## 28 0.68022610 0.696389204 0.55716895     5
## 29 0.07741567 0.583340711 0.16285882     5
## 30 0.61491767 0.219186367 0.01329829     5
```

```r
tapply(x$fus, x$dborn, mean)
```

```
##         1         2         3         4         5
## 0.3977730 0.3694151 0.7226138 0.4947701 0.5602542
```

```r
  ## takes the mean of column "fus", but separated into the groups set
  ## by "dborn"
```