Basics

Guy J. Abel

R Console Start Up

```
R version 3.4.3 (2017-11-30) -- "Kite-Eating Tree"
Copyright (C) 2017 The R Foundation for Statistical Computing
Platform: x86_64-w64-mingw32/x64 (64-bit)

R is free software and comes with ABSOLUTELY NO WARRANTY.
You are welcome to redistribute it under certain conditions.
Type 'license()' or 'licence()' for distribution details.

R is a collaborative project with many contributors.
Type 'contributors()' for more information and
'citation()' on how to cite R or R packages in publications.

Type 'demo()' for some demos, 'help()' for on-line help, or
'help.start()' for an HTML browser interface to help.
Type 'q()' to quit R.

>
```

R Console Start Up

- Note the > at the bottom.
- Whenever you see this symbol, it means that R is not doing anything and just waiting for your input.
- It's called the prompt.

Very Basics

- We can enter R commands at the prompt

```
> 1 + 2
[1] 3
```

- When R is running it stores everything in the computer's active memory.
  - R knows `1` and `2`, as well as the basic operator `+`.
  - It creates an object in the active memory containing the results of the computation.
  - R prints the content of the results object to the screen.
- R let's you know it has finished when displaying in the square brackets `[]`
  - The square brackets `[]` tells you how many results were calculated.
  - In our case this object contains only one value, i.e. R has finished and printed the first and only value `[1]` of the computation.
- You can scroll through previous commands you've entered by using the up ("→") and down ("↓") keys on your keyboard.

R Basics

- R may sometimes display a + (also known as the continuation line).
  - R will not reporting the [], as it is not displaying any content from an object stored in memory.
  - Entered an incomplete command
  - R is waiting for more input.

```
> 1 +
  +
```

- If you are not sure what's going on hit `Esc` or `Ctrl-C`.
  - It will tell R to forget it and bring back the prompt.

R Basics

- If you know what R wants, then you can complete the computation:

```
> 1 +
  + 2
 [1] 3
```

- The continuation line also commonly occurs with
  - Unclosed parenthesis ()
  - Unclosed quotes ""

```
> 7 / (1 + 3
+   )
[1] 1.75
> "Guy
+ "
[1] "Guy\n"
```

- Remember if you are in a syntactic hole... `Esc`.

Calculator

- R understands the following basic operators:
  1. + and - for addition and subtraction
  2. * and / for multiplication and division
  3. ^ for exponents
- R observes standard rules of operator precedence.
- You can use brackets () if you are not sure, e.g.

```
> 7 / (1 + 3)
[1] 1.75
```

is not the same as this:

```
> 7 / 1 + 3
[1] 10
```

Character Strings

- R also allows you to type in stings of characters (letters, words, phrases)
- If you do not use quotation marks R will think you are asking for an object

```
> "Guy"
[1] "Guy"
> 'Guy'
[1] "Guy"
```

```
> Guy
Error: object 'Guy' not found
```

- We will go into depth on objects a little later.
- For now remember text must go in quotation.

Sending Messages

- The `print()` function prints output in the R console.
  - Often do not need to use `print()` except when creating functions (more on creating functions later)

```
> 0.2
[1] 0.2
> print(0.2)
[1] 0.2
>
> "Guy"
[1] "Guy"
> print("Guy")
[1] "Guy"
```

- The `paste()` function can link together character strings.
  - Also allows non-characters as inputs, converts output to one character string.

```
> paste("I ate", "a bar of chocolate")
[1] "I ate a bar of chocolate"
> paste("I ate", 1, "bar of chocolate")
[1] "I ate 1 bar of chocolate"
> paste("I ate", 1, "bar of chocolate", sep = "  .  ")
[1] "I ate  .  1  .  bar of chocolate"
```

Comments

- The comment operator # will tell R to ignore everything printed after it (in the current line).
- Extremely useful to annotate your code.

```
> 1 + 2 + 3 # Here R does some serious Maths
[1] 6
```

- It is good practice to annotate your code.
  - Within a surprisingly short period of time you will forget what each bit of code is trying to do.

Comments

- Be careful.

```
> 1 + 2 # + 20
[1] 3
```

- Misplaced comments can break your code.

```
> 1 + # 2
+
```

Spacing

- For the most part, R does not care about spacing.

```
> 1 + 2
[1] 3
>
> # same as
> 1     +           2
[1] 3
```

- For character strings spaces matter...

```
> print("Strings  obey  spacing.")
[1] "Strings  obey  spacing."
> print(" Strings    obey          spacing    .     ")
[1] " Strings    obey          spacing    .     "
```

Semi-Colons

- R evaluates code line by line.
- A line break tells R that a statement is to be evaluated.
- Instead of a line break, you can use a semicolon (;) to tell R where statements end.

```
> "Guy"
[1] "Guy"
```

```
> "Guy" "Abel"
Error: unexpected string constant in ""Guy" "Abel""
```

```
> "Guy"; "Abel"
[1] "Guy"
[1] "Abel"
```

Exercise 1 (ex11.R)

- Open `ex11.R` and complete the following exercises:

```
# 1. 5 plus 6

# 2. 2 multiplied by 8

# 3. 8 divided by 3

# 4. 909 minus 506

# 5. 5 to the power of 10

# 6. Tell R to say your first name

# 7. Q: Which symbol does R use to ignore all code after?
#    A:
# 8. On one line of code with two R print commands write
#    your first and last name.
```

Basic Functions

- R comes with a many many pre-installed functions.
  - Installed as part of the `base` package which is located in your library directory.
  - Functions have names and take arguments in parentheses: `function(...)`
- Takes the form

```
function(argument1, argument2)
```

  - Arguments are options for the function.
  - Each function has different arguments.
  - If there are multiple arguments, use a comma `,` to separate.

Basic Functions

- We can find out what arguments (options) are for a function using ?

```
> ?log
```

- In RStudio you can also use `Tab` once you have typed the function name and opened the parenthesis
- The help file reports that the function takes two arguments, separated by a comma.
  - `x` a number you want to take the logarithm of (no default)
  - `base` the base system for the logarithm (default `base = exp(1)`)

```
> log(x = 10)
[1] 2.302585
> # same as
> log(x = 10, base = exp(1))
[1] 2.302585
> # change the base argument...
> log(x = 10, base = 10)
[1] 1
```

Basic Functions

- R knows what arguments are supplied to the function and their order.

```
> log(x = 10)
[1] 2.302585
> # same as
> log(10)
[1] 2.302585
>
> # knows the order as well (i.e. second input is the `base`)
> log(10, 10)
[1] 1
> # same as
> log(x = 10, base = 10)
[1] 1
```

- Whilst this reduces your typing, I advise not to do it.
  - Difficult to remember which inputs for which arguments.
  - Even more difficult for others looking at your code.
  - Using `Tab` and `Enter` in RStudio means its easy to write argument names fully.

Basic Functions

Basic mathematical functions in `base` R

Function	Description
`log()`	computes natural logarithms
`log10()`	computes logarithm to the base 10
`exp()`	computes the exponential function
`sqrt()`	takes the square root
`abs()`	returns the absolute value
`sin()`	returns the sine
`factorial()`	returns the factorial
`sign()`	returns the sign (negative or positive)
`round()`	rounds the input to the desired digit

Remember we can use `?` to understand how any function works, e.g. `?round`

Exercise 2 (ex12.R)

```
# 1. Natural logarithm of 5

# 2. Square root of 121

# 3. Absolute value of 10 and -11

# 4. 8 x 7 x 6 x 5 x 4 x 3 x 2 x 1

# 5. Round pi (3.141593) to 3 decimal places

# 6. The sin angle of 60

# 7. e to the power of 4

# 8. Print your name
```

Logic Operators

- Among the most used features of R are logical operators.
- You will use these throughout your code and they are crucial for all sorts of data manipulation.
- When R evaluates statements containing logical operators it will return either `TRUE` or `FALSE`

Function	Description
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
==	equal
!=	not equal
&	and
|	or

Exercise 3 (ex13.R)

```
# 1. Test if 3 is larger than pi

# 2. Test if pi is equal to 3.141593

# 3. Test if 7 divided by 3 is less than or equal to 3

# 4. Test if 5 times 2 is greater than or equal to 10

# 5. Test if 1 plus 5 is not equal to 7

# 6. Test if logarithm of 1000 is larger 7

# 7. Test if pi is greater than 3 and 4

# 8. Test if pi is less than  3 or 4
```

Assignment and Reference

- Instead of recalculating everything over and over again we can give things names and recall them later.
- Before we implicitly relied on and then manipulated objects and R implicitly printed these objects to the screen.

```
> 1 + 2
[1] 3
```

- We can assign and recall names using the assignment operator <-.
    - Think of this as the M+ button on your calculator.

```
> a  <- 1 + 2
```

- Above, R no longer gave the answer to our problem. It just returns the prompt.

Assignment and Reference

- We can print the results to the screen by typing the name of our new object

```
> a
[1] 3
```

- If you give R the name of some object it knows you don't even have to use the `print()` function.
- Just type in the name and R will give you the result.

```
> a
[1] 3
```

- It does not do the calculation again when you print the object.

Beware

- We can use just about any name we like except
  - Cannot be a number, e.g. `3 <- 1 + 2`
  - Cannot start with with a number, e.g. `3a <- 1 + 2`
- You can break R code by creating objects that already are existing functions.

```
> exp
function (x)  .Primitive("exp")
> exp <- 1 + 3
> exp
[1] 4
```

- The exp function still works, but is lower down in the search environment than the new object.
  - We could create our own function exp that does something completely different.
  - Directly get a function in a specific package using `::`

```
> base::exp
function (x)  .Primitive("exp")
```

Beware

- R is case-sensitive.

```
> A
Error: object 'A' not found
```

```
> a
[1] 3
```

- If you are not sure, check the potential object name by printing it before assigning it!

```
> e
Error in eval(expr, envir, enclos): object 'e' not found
```

Playing with Trivial Vectors

- Named objects behave just like the ones R already knows.
- This is very useful when things get more complicated:

```
> a
[1] 3
> a * 2
[1] 6
>
> b  <- a * sqrt(a)
> b
[1] 5.196152
```

Keeping Track of Objects

- To see what objects you have created (the ones R stored in active memory) you can use the `ls()` function.

```
> ls()
[1] "a"   "b"   "exp"
```

- RStudio also shows each created object in the Environment tab.
- If you want to remove an object from memory use the `rm()` function.
- Be very careful. This will delete thing permanently.

```
> rm(b, exp)
> ls()
[1] "a"
```

- If you want to remove all objects from active memory can also use the button with a brush on it above the object list in RStudio or directly in the console:

```
> rm(list = ls())
```

Real Vectors

- So far we have only created trivial vectors of length 1. Let's assign some longer ones.
- To do this you will use the `c()` function.
  - The `c` stands for concatenate,
  - Combines a bunch of elements together, separated by commas.
  - Make sure you never call an object `c` (like we just created objects `a` and `b`)

```
> v1  <- c(1,2,3,4,5,6,7,8,9,10)
```

How about a character vector?

```
> v2  <- c("a", "b", "c", "d")
> v2
[1] "a" "b" "c" "d"
```

Or . . .

```
> # will convert everything to characters if there is at least one
> v3  <- c(1, "two", 3, 4)
> v3
[1] "1"   "two" "3"   "4"
```

Real Vectors

- You can also string multiple vectors together with the `c()` function.

```
> v4  <- c(v1 , v2 , v1)
> v4
 [1] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10" "a"  "b"  "c"  "d"
[15] "1"  "2"  "3"  "4"  "5"  "6"  "7"  "8"  "9"  "10"
```

Simplifying Vector Creation

- Using the `c()` function can be tedious
  - Have to manually type all elements of a vector.
  - Will make mistakes
- Use the colon (`:`) to tell R to create an integer vector.

```
> 1:20
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20
> # backwards
> 10:-5
 [1] 10  9  8  7  6  5  4  3  2  1  0 -1 -2 -3 -4 -5
```

- Use the `seq()` function for more general sequences.

```
> seq(from = 0, to = 10)
 [1]  0  1  2  3  4  5  6  7  8  9 10
> # the 'by' argument let's you set the increments
> seq(from = 0, to = 10, by = 2)
[1]  0  2  4  6  8 10
```

Simplifying Vector Creation

```
> # the length.out argument specifies the
> # length of the vector and R figures out
> # the increments itself
> seq(0, 10, length.out = 25)
 [1]  0.0000000  0.4166667  0.8333333  1.2500000  1.6666667  2.0833333
 [7]  2.5000000  2.9166667  3.3333333  3.7500000  4.1666667  4.5833333
[13]  5.0000000  5.4166667  5.8333333  6.2500000  6.6666667  7.0833333
[19]  7.5000000  7.9166667  8.3333333  8.7500000  9.1666667  9.5833333
[25] 10.0000000
```

Simplifying Vector Creation

- Another useful function is `rep()` which allows you to repeat things.

```
> rep(x = 0, times = 10)
 [1] 0 0 0 0 0 0 0 0 0 0
> # as always you can drop the argument name
> rep(x = "Hello", times = 3)
[1] "Hello" "Hello" "Hello"
> # repeating vec 1 twice
> rep(x = v1 , times = 2)
 [1]  1  2  3  4  5  6  7  8  9 10  1  2  3  4  5  6  7  8  9 10
> # we can repeat each element as well
> rep(x = v2 , each = 2)
[1] "a" "a" "b" "b" "c" "c" "d" "d"
```

Vector Operations

- R is very powerful when working with vectors.
- Most standard mathematical functions work with vectors.

```
> v1 + v1
 [1]  2  4  6  8 10 12 14 16 18 20
> v1 ^ 2
 [1]   1   4   9  16  25  36  49  64  81 100
> log(v1)
 [1] 0.0000000 0.6931472 1.0986123 1.3862944 1.6094379 1.7917595 1.9459101
 [8] 2.0794415 2.1972246 2.3025851
```

Exercise 4 (ex14.R)

```
# 1. Create a vector called a1 for the 1, 4, 9, 16, 25

# 2. Create a vector called a2 from 1 to 100

# 3. Create a vector called a3 from -8 to 20 in steps of 4

# 4. Create a vector called a4 that combines a1 and a3

# 5. Create a vector called a5 that has your first and last name repeated 3 times

# 6. Find the square root of each element in a1

# 7. Which elements in a3 are positive (greater than zero)

# 8. Divide a2 by a1. Then divide a2 by a3. What is going on?
```

Selected Functions for Vectors

- R has many functions in the `base` package that work with vectors.

Function	Description
`sum()`	sums of the elements of the vector
`prod()`	product of the elements of the vector
`min()`	minimum of the elements of the vector
`max()`	maximum of the elements of the vector
`range()`	the range of the vector
`sort()`	sorts the vector (argument: `decreasing = FALSE`)
`rev()`	Reverse the order of the vector
`length()`	returns the length of the vector
`which()`	returns the index after evaluating a logical statement
`unique()`	returns a vector of all the unique elements of the input
`table()`	returns tabulation count for each unique element

Exercise 5 (ex15.R)

```
# 1. Find the sum of all elements in a1

# 2. What is the largest element in a4

# 3. Multiply all elements of a2 together

# 4. How many elements are there in a5

# 5. Reverse the order of a2

# 6. Which elements of a1 are greater than 10

# 7. What is the range of values in a4

# 8. What are the unique elements of a4

# 9. Sort the values in a4

# 10. Create a tabultion of the elements in a4
```

Selected Statistical Functions for Vectors

- R has many statistical functions in the `stats` package that work with vectors.

Function	Description
`mean()`	mean of the elements
`median()`	median of the elements
`sd()`	the standard deviation
`var()`	the variance (on
`cov()`	the covariance (takes two inputs `cov(x,y)`)
`cor()`	the correlation coefficient (takes two inputs `cor(x,y)`)
`IQR()`	Inter Quartile Range
`quantile()`	Sample quantiles corresponding to the given probabilities
`summary()`	returns summary statistics

Sampling

- R can generate many different types of random numbers.
- The `sample()` function draws randomly from a given vector

```
> v1
 [1]  1  2  3  4  5  6  7  8  9 10
> sample(x = v1, size = 5)
[1] 10  1  3  2  8
> sample(v1, size = 15, replace = TRUE)
 [1]  8 10  4  5 10  4  9  6  8  7  5  7  2  5  5
```

Function	Description
`rnorm()`	random generation for the normal distribution (`mean`, `sd`)
`rbinom()`	random generation for the binomial distribution (`size`, `prob`)
`rpois()`	random generation for the Poisson distribution (`lambda`)
`runif()`	random generation for the uniform distribution (`min`, `max`)

Exercise 6 (ex16.R)

```
# 1. Create a vector a6 that is a sequence from 1 to 25 of length 8

# 2. What is the mean of a6

# 3. what is the standard deviation of a6

# 4. Create a vector a7 that is a random sample of size 8 from a2

# 5. What is the correlation of a6 and a7

# 6. Create a vector a8 of length 8 of random numbers from a normal
#    distribution with mean 10 and standard deviation 2

# 7. What is Inter Quartile Range of a8

# 8. Create a summary of a8
```

Objects

- R is an object oriented language.
    - Everything in R is an object.
    - When R does anything, it creates and manipulates objects.
    - Objects are a bit like the memory button on the calculator, but with much much more freedom.
- R objects come in different types and flavors.

Basic Objects: Vectors

- One-dimensional sequences of elements of the same mode. (More on modes later)
- For example, this could be vector of length 26 (i.e. one containing 26 elements) where each element is a letter in the alphabet.
- R has some built in vectors

```
> pi
[1] 3.141593
> LETTERS
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
[18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"
```

Basic Objects: Matrices

- Two dimensional rectangular objects (matrices) or
- All elements of matrices have to be of the same mode.
- Can build matrices using the `matrix()` function

```
> matrix(data = 1:12, nrow = 4)
     [,1] [,2] [,3]
[1,]    1    5    9
[2,]    2    6   10
[3,]    3    7   11
[4,]    4    8   12
```

Basic Objects: Lists

- Like vectors but they do not have to contain elements of the same mode.
- The first element of a list could be a vector of the 26 letters of the alphabet.
- The second element could contain a vector of numbers.
- A third could be a 2 by 3 matrix.

```
> list(LETTERS, 1:10, matrix(1:6, nrow = 2))
[[1]]
 [1] "A" "B" "C" "D" "E" "F" "G" "H" "I" "J" "K" "L" "M" "N" "O" "P" "Q"
[18] "R" "S" "T" "U" "V" "W" "X" "Y" "Z"

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

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

Basic Objects: Data Frames

- Data frames are best understood as special matrices (technically they are a type of list).
- They are two dimensional containers with
  - Rows corresponding to observations
  - Columns corresponding to vectors
- R has some built in data frames

```
> swiss
             Fertility Agriculture Examination Education Catholic
Courtelary        80.2        17.0          15        12     9.96
Delemont          83.1        45.1           6         9    84.84
Franches-Mnt      92.5        39.7           5         5    93.40
Moutier           85.8        36.5          12         7    33.77
Neuveville        76.9        43.5          17        15     5.16
Porrentruy        76.1        35.3           9         7    90.57
Broye             83.8        70.2          16         7    92.85
Glane             92.4        67.8          14         8    97.16
Gruyere           82.4        53.3          12         7    97.67
Sarine            82.9        45.2          16        13    91.38
Veveyse           87.1        64.5          14         6    98.61
Aigle             64.1        62.0          21        12     8.52
Aubonne           66.9        67.5          14         7     2.27
Avenches          68.9        60.7          19        12     4.43
```

Modes

- All objects have a certain mode.
- Some objects can only deal with one mode at a time, others can store elements of multiple modes.
- Some basic modes include:

1. `integer`: integers (e.g. 1, 2 or -1000)
2. `numeric`: real numbers (e.g 2.336, -0.35)
3. `character`: elements made up of text-strings (e.g. `"text"`, `"Hello World!"`, or `"123"`)
4. `logical`: data containing logical constants (i.e. `TRUE` and `FALSE`)

Indexing

- Sometimes you do not want to print or manipulate an entire vector.
- This is where indexing comes in.
- Indexing vectors is done with `[]`.

```
> v4 <- c("I", "really", "like", "chocolate", "ice cream.")
> # with the bracket we reference the third element
> v4[3]
[1] "like"
> # we can  reference a sequence of elements
> v4[2:4]
[1] "really"    "like"      "chocolate"
> # or any elements we like
> v4[c(1,3,4)]
[1] "I"         "like"      "chocolate"
```

Indexing

```
> # all except the 2nd element
> v4[-2]
[1] "I"          "like"       "chocolate"  "ice cream."
> # and we can change elements
> v4[4]  <- "strawberry"
> v4
[1] "I"          "really"     "like"       "strawberry" "ice cream."
```

Indexing

- Indexing also works with matrices
- Using a `,` to separate rows and columns `[row, column]`

```
> # create a matrix
> m0 <- matrix(data = 1:12, nrow = 3)
> m0
     [,1] [,2] [,3] [,4]
[1,]    1    4    7   10
[2,]    2    5    8   11
[3,]    3    6    9   12
> # bottom right element
> m0[3, 4]
[1] 12
> # third column
> m0[, 3]
[1] 7 8 9
```

Indexing

- The `[row, column]` indexing also works with data frames
- R users usually prefer to use the name of the column
- The `dataframe$column` selects the entire column

```
> # can use the raw row and column coordinates
> swiss[38,1]
[1] 79.3
> # the row and column names
> swiss["Sion","Fertility"]
[1] 79.3
> # the column name and row coordinates
> # this is useful if there are no row names, which can often be the case.
> swiss$Fertility[38]
[1] 79.3
> # the whole column
> swiss$Fertility
 [1] 80.2 83.1 92.5 85.8 76.9 76.1 83.8 92.4 82.4 82.9 87.1 64.1 66.9 68.9
[15] 61.7 68.3 71.7 55.7 54.3 65.1 65.5 65.0 56.6 57.4 72.5 74.2 72.0 60.5
[29] 58.3 65.4 75.5 69.3 77.3 70.5 79.4 65.0 92.2 79.3 70.4 65.7 72.7 64.4
[43] 77.6 67.6 35.0 44.7 42.8
```

Missing Values

- NA is used to represent missing data
- Different functions treat missing data differently

```
> v5  <- c(10, 7, NA, NA, 0, NA, -2, 8)
> v5
[1] 10  7 NA NA  0 NA -2  8
> # this won't work by default
> max(v5)
[1] NA
> # there is usually an option to allow functions to omit NA
> max(v5, na.rm = TRUE)
[1] 10
> # test for missing values using is.na
> is.na(v5)
[1] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE
> # test for non missing values using !is.na
> !is.na(v5)
[1]  TRUE  TRUE FALSE FALSE  TRUE FALSE  TRUE  TRUE
> # this works (does not have na.rm argument)
> summary(v5)
   Min. 1st Qu.  Median    Mean 3rd Qu.    Max.    NA's
   -2.0     0.0     7.0     4.6     8.0    10.0       3
```

Exercise 7 (ex17.R)

```
# 1. Print out the second row of the m1 matrix, where
# m1 <- matrix(1:28, nrow = 4, ncol = 7)

# 2. Print out the top right value of the m1 matrix

# 3. Print out the first column of the m1 matrix

# 4. Print out the whole of the Infant.Mortality column in the swiss data

# 5. What is the maximum fertility in the swiss data

# 6. Print out the Catholic value for Glane in the swiss data

# 7. Set the thrid and sixth element of a8 to missing values

# 8. What is the variance of the remaining numbers in a8
```

Conditionals

- R is programmable language.
- Conditionals are one of the fundamentals of basic programming
  - Evaluating conditional statements (returns `TRUE` or `FALSE`)
  - Performing some action based on the evaluation.
- The `if()` function is used when you want action only when a statement is `TRUE`.
- Takes format

```
if(statement){
  action
}
```

  - The `statement` must one logical value (`TRUE` or `FALSE`)

Conditionals

```
> v6 <- rnorm(n = 10)
> v6
 [1] -0.6264538  0.1836433 -0.8356286  1.5952808  0.3295078 -0.8204684
 [7]  0.4874291  0.7383247  0.5757814 -0.3053884
>
> # test to see if the 2nd element is positive
> if(v6[2] > 0){
+   print("My number is positive")
+ }
[1] "My number is positive"
>
> # test to see if the 3rd element is positive
> if(v6[3] > 0){
+   print("My number is positive")
+ }
>
> # paste() function to join together numbers and character stings
> if(v6[2] > 0){
+   print(paste(v6[2], "is positive"))
+ }
[1] "0.183643324222082 is positive"
```

Conditionals

- The `ifelse()` function is used when you want one action when statement `TRUE` and different action when `FALSE`.
- Very useful when creating variables in data frames. Has three arguments
  - `test` the statement
  - `yes` action if statement is `TRUE`
  - `no` action if statement is `FALSE`

```
> # logical test
> v6 > 0
 [1] FALSE  TRUE FALSE  TRUE  TRUE FALSE  TRUE  TRUE  TRUE FALSE
>
> # display a character vector for results
> ifelse(test = v6 > 0, yes = "Positive", no = "Negative")
 [1] "Negative" "Positive" "Negative" "Positive" "Positive" "Negative"
 [7] "Positive" "Positive" "Positive" "Negative"
>
> # can return numeric values as well
> ifelse(test = v6 > 0, yes = 0, no = 1)
 [1] 1 0 1 0 0 1 0 0 0 1
```

Conditionals

```r
> # remember v5
> v5
[1] 10  7 NA NA  0 NA -2  8
>
> # remember is.na()
> is.na(v5)
[1] FALSE FALSE  TRUE  TRUE FALSE  TRUE FALSE FALSE
>
> # can use any test that returns logical
> ifelse(test = is.na(v5), yes = "Missing", no = "Observed")
[1] "Observed" "Observed" "Missing"  "Missing"  "Observed" "Missing"
[7] "Observed" "Observed"
```

Loops

- Loops are functions that carry out repetitive actions.
- The `for()` function in R is used to repeat actions a given number of times
- Takes format:

```
for(element in vector){
  action
}
```

- Within the `{}` we put the actions we want repeated multiple times.
  - General code, using `element` in place of the parts of the code that are non-general.
- Within the `()` we put an instruction.
  - The `element` values are taken from `vector` in each round of the loop.
  - Typically people use a single letter such as `i`.
  - The `vector` contains all the possible `element` values to be used.

Loops

- For each `i` in the vector `v7` we calculate `x` and ask R to print to the screen.
- Or put differently, for each `i` element in a vector we do the thing inside the curly braces.

```
> v7 <- rnorm(n = 5)
> v7
[1]  1.5117812  0.3898432 -0.6212406 -2.2146999  1.1249309
> for(i in v7){
+   x <- i * 2
+   print(x)
+ }
[1] 3.023562
[1] 0.7796865
[1] -1.242481
[1] -4.4294
[1] 2.249862
```

Loops

```
> v7
[1]  1.5117812  0.3898432 -0.6212406 -2.2146999  1.1249309
> # basic action with no print
> for(i in v7){
+   x <- i * 2
+ }
> # still does the work just not printed.
> # only saves the last version of x
> x
[1] 2.249862
```

Loops

```
> # create empty vector to save action
> n <- length(v7)
> n
[1] 5
> v8 <- rep(NA, times = n)
> v8
[1] NA NA NA NA NA
> for(i in 1:n){
+   v8[i] <- v7[i] * 2
+ }
> v8
[1]  3.0235623  0.7796865 -1.2424812 -4.4293998  2.2498618
```

Loops

- We cannot assign new objects in a `for` loop. Can create a new object set to `NULL` to represents an empty object.

```
> for(i in 1:n){
+   v9[i] <- v7[i] * 3
+ }
Error in eval(expr, envir, enclos): object 'v9' not found
> # create a empty object
> v9 <- NULL
> v9
NULL
> # build up a vector
> for(i in 1:n){
+   v9[i] <- v7[i] * 3
+ }
> v9
[1]  4.535344  1.169530 -1.863722 -6.644100  3.374793
```

Loops

```
> # simple projections
> p <- rep(NA, times = 10)
> p[1] <- 100
> r <- 0.05
> for(i in 1:9){
+   p[i+1] <- p[i] * (1 + r)
+ }
> p
 [1] 100.0000 105.0000 110.2500 115.7625 121.5506 127.6282 134.0096
 [8] 140.7100 147.7455 155.1328
```

Loops

- Loops can be really useful for repetitive tasks

```
> for(i in 1:10){
+   # read in different excel sheets
+   d <- read_excel(path = "myexcelfile.xlsx", sheet = i)
+   # run regression models using data from excel sheet
+   m <- lm(formula = y ~ x1 + x2, data = d)
+   # save regression model coefficients
+   write_csv(x = m$coefficients, path = paste0("model", i, ".csv"))
+ }
```

Exercise 8 (ex18.R)

```
# 1. Using the runif() function create an object called r1 based on one random numbe

# 2. Write an if statment to return "Big" if r1 is greater than 0.5

# 3. Write an ifelse statment to return "BIG" if r1 is greater than 0.8 and "not big

# 4. Using the runif() function create an object called r2 based on 20 random number

# 5. Write an ifelse statment to return 1 if r2 is greater than 0.4 and less than 0.

# 6. Create a for loop that prints your name 100 times


# 7. In a for loop print the cumulative sum of r2



# 8. Create a vector r3 of 20 missing values

# 9. In a for loop replace the NA's or r3 with the value of  r2 times 2




# 10. Create a NULL
```

Functions

- R is a programming language. We can create (program) our own functions using the `function()` function.
  - Wrap up code for repetitive tasks
  - Simplifies complex tasks
  - Creates user friendly access to your R code.
- Functions takes the format

```
function(arguments, ...){
  actions
  return(value)
}
```

  - Wrap all sorts of functions, loops, and statements inside of a function to simplify repetitive tasks.
  - Specify as many arguments as you like.
  - Set argument(s) to `NULL` if we do not want a default value. Forces the user to set.
  - The `return()` function is not essential.
    - The actions will take place but nothing is outputted unless we print the object on the last line of the function.

Functions

```
> sp_rate <- function(p0 = 5, r = 0.05){
+   p <- p0
+   for(i in 1:5){
+     p[i+1] <- p[i] * (1 + r)
+   }
+   return(p)
+ }
> sp_rate()
[1] 5.000000 5.250000 5.512500 5.788125 6.077531 6.381408
> sp_rate(p0 = 20, r = 0.1)
[1] 20.0000 22.0000 24.2000 26.6200 29.2820 32.2102
>
> # add a n argument for the number of projection periods
> sp_rate <- function(p0 = NULL, r = NULL, n = NULL){
+   p <- p0
+   for(i in 1:n){
+     p[i+1] <- p[i] * (1 + r)
+   }
+   return(p)
+ }
> sp_rate(p0 = 20, r = 0.1, n = 20)
 [1]  20.00000  22.00000  24.20000  26.62000  29.28200  32.21020  35.43122
 [8]  38.97434  42.87178  47.15895  51.87485  57.06233  62.76857  69.04542
[15]  75.94997  83.54496  91.89946 101.08941 111.19835 122.31818 134.55000
```

Exercise 9 (ex19.R)

```
# 1. Create a function called my_name with inputs
#    a) name
#    b) n
#    that prints your name n times
#    (Hint: Use a for loop)
##### <- function(name, n){
  #####(i in 1:#####){
    #####(name, n)
  }
}
# 2. Run your function with your name and n = 10

# 3. Create a function called rpois_add with inputs
#    a) n
#    b) lambda1
#    c) lambda2
#    that adds together n random numbers from poisson distrubions with mean lambda1
rpois_add <- #####(n, #####, lambda2){
  ##### <- rpois(n = #####, lambda = lambda1)
  r2 <- rpois(n = n, ##### = lambda2)
  #####(r1 + #####)
}
# 4. Run your function with n = 20, lambda1 = 5, lambda2 = 8
```

Working Directory

- R works in a particular location on your computer.
- We can change the location to make it easier to
  - Read in data (more on this later)
  - Run R scripts
  - Save data, plots, other outputs (more on this later).
- The `getwd()` function shows the current folder (directory) R is working in.

```
> getwd()
[1] "C:/Users/Guy/Dropbox/SHU2017-3XS371026/slides-md"
```

- The `setwd()` function allows you to set the directory of your R session.
  - Note: R uses / or \\ instead of \ when setting directories:

```
> setwd(dir = "C:/Users/Guy/Dropbox/SHU2017-3XS371026/exercise-solutions")
> # on mac would be
> # setwd(dir = "/Users/Guy/Dropbox/SHU2017-3XS371026/exercise-solutions")
> getwd()
[1] "C:/Users/Guy/Dropbox/SHU2017-3XS371026/exercise-solutions"
> setwd(dir = "C:/Users/Guy/Dropbox/SHU2017-3XS371026/")
> getwd()
[1] "C:/Users/Guy/Dropbox/SHU2017-3XS371026"
```

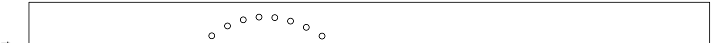
RStudio Projects

- With RStudio you can use a project file and never have to worry about `setwd()`
  - `File | New Project`
  - Select `New Directory` if you do not have a folder already
  - Select `Existing Directory` if you do not have a folder with some saved files.
- At the start of your R session use `File | Open Project`
  - Can also select from the Project dropdown in the top right hand corner.

Sourcing a Script

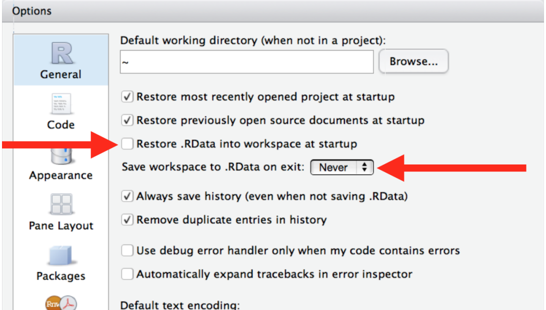
- The source() function runs R code in a file.

```
> # remove all objecst
> rm(list = ls())
> # set working directory to soloution folder
> setwd(dir = "C:/Users/Guy/Dropbox/SHU2017-3XS371026/exercise-solutions/")
> # run code in ex19.R, could run from anywhere with the full path:
> # source(file= "C:/Users/Guy/Dropbox/SHU2017-3XS371026/exercise-solutions/ex19.R")
> source(file = "ex19.R")
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
[1] "Guy"
```



Sourcing a Script

- Using R code scripts (.R files) allows you to save and share your code.
    - Can break big project into multiple scripts.
    - Keeps a tidy workflow
    - Keeps the number of objects to a minimum
    - Quickly identify a particular line of code for revision.
- Instruct RStudio not to preserve your workspace between sessions:
    - Will encourage use of scripts.
    - Easy to run a script to create workspace objects.
    - Difficult to re-create workspace objects without scripts



Assignment 1 (assign1.R)

```
##
## Assignment 1
##

## Create two questions, with answers below, based on Exercise 1 (ex11.R)

# 1.

# 2.

## Create two questions, with answers below, based on Exercise 2 (ex12.R)

# 1.

# 2.

## Create two questions, with answers below, based on Exercise 3 (ex13.R)

# 1.

# 2.

## Create two questions, with answers below, based on Exercise 4 (ex14.R)

# 1.
```

Homework

- For next lesson install the `tidyverse` set of packages and others using:

```
> install.packages("tidyverse")
> install.packages("foreign")
> install.packages("readstata13")
> install.packages("openxlsx")
```