Introduction to R

Guy J. Abel

R Console Start Up

```
R version 4.0.2 (2020-06-22) -- "Taking Off Again"
Copyright (C) 2020 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

```
> 3 - 1
[1] 2
```

- When R is running it stores everything in the computer's active memory.
  - R knows 1 and 3, 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 ("$\rightarrow$") and down ("$\downarrow$") 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.

```
> 3 -
  +
```

- 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:

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

- The continuation line also commonly occurs with
  - Un-closed parenthesis ()
  - Un-closed 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 raising to the power (e.g. $10^2 = 100$)
- 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.

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 # some additions
[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.

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

- Misplaced comments can break your code.

```
> 3 - # 1
  +
```

Spacing

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

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

- For character strings spaces matter...

```
> message("Strings  obey  spacing.")
Strings  obey  spacing.
> message(" Strings    obey         spacing   .    ")
 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"
```

```
> 3 - 1   5 * 9
Error: unexpected numeric constant in " 1 + 2   5"
```

```
>  3 - 1; 5 * 9
[1] 2
[1] 45
```

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
`factorial()`	returns the factorial
`sign()`	returns the sign (negative or positive)
`round()`	rounds the input to the desired digit

Remember we can use `?` to view the help file of any function, 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 (the factorial of 8)

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

# 6. The logarithm of 100 to the base 10

# 7. The exponential of 4

# 8. Print your name using the message function
```

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

Logic Operators

```
> pi
[1] 3.141593
>
> pi > 5
[1] FALSE
>
> pi > 3 & pi > 4
[1] FALSE
>
> pi > 5 | pi < 4
[1] TRUE
```

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
```

- 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")
> readxl::read_excel
function (path, sheet = NULL, range = NULL, col_names = TRUE,
    col_types = NULL, na = "", trim_ws = TRUE, skip = 0, n_max = Inf,
    guess_max = min(1000, n_max), progress = readxl_progress(),
    .name_repair = "unique")
```

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)
> v1
 [1]  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"  "1"
[16] "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(from = 0, to = 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 vector 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
 [1]  1  2  3  4  5  6  7  8  9 10
> 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(x = 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. Test to see where 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
`ifelse()`	returns different values depending on logical `test`

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`

```
> v1
 [1]  1  2  3  4  5  6  7  8  9 10
>
> # logical test
> v1 > 5
 [1] FALSE FALSE FALSE FALSE FALSE  TRUE  TRUE  TRUE  TRUE  TRUE
>
> # display a character vector for results
> ifelse(test = v1 > 8, yes = "High", no = "Low")
 [1] "Low"  "Low"  "Low"  "Low"  "Low"  "Low"  "Low"  "Low"  "High" "High"
>
> # can return numeric values as well
> ifelse(test = v1 > 8, yes = 1, no = 0)
 [1] 0 0 0 0 0 0 0 0 1 1
```

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 tabulation of the elements in a4

# 11. Create a vector from a4 which recodes elements above zero as 1 and below zero

# 12. Create a vector from a4 which recodes elements above zero with label "positive
#     and below zero or zero with label "negative or zero"
```

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
`cov()`	the co-variance (takes two inputs `cov(x,y)`)
`cor()`	the correlation coefficient (takes two inputs `cor(x,y)`)
`IQR()`	Inter Quartile Range
`quantile()`	returns values corresponding to the given probabilities (takes input `probs`)
`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] 1 7 3 8 5
> sample(x = v1, size = 15, replace = TRUE)
 [1] 10  6 10  5  3 10  3  8  3  6  1  2  8  7  8
```

Sampling

- Sample from many different types of statistical distributions

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

```
> # normal
> rnorm(n = 5, mean = 1, sd = 2)
[1] -1.5948555  2.4919915  0.8964033  3.5129072 -1.1273850
> # binomial
> rbinom(n = 8, size = 10, prob = 0.3)
[1] 3 3 1 2 2 2 2 4
> # poisson
> rpois(n = 8, lambda = 2)
[1] 2 0 1 0 0 2 2 0
> # uniform
> runif(n = 5, min = 5, max = 8)
[1] 7.875833 6.655136 7.239813 7.612478 6.281276
```

Exercise 6 (ex16.R)

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

a6
# 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
#    sequence of integers between 1 to 100

a7
# 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" "R" "S"
[20] "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: 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
Cossonay          61.7        69.3          22         5     2.82
```

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" "R" "S"
[20] "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: Lists

- Lists are used in R for regression results and many other complex outputs:
- We can check the structure of objects using the `str()` function:

```
> m <- lm(formula = Fertility ~ Education + Catholic, data = swiss)
> str(m)
List of 12
 $ coefficients : Named num [1:3] 74.234 -0.788 0.111
  ..- attr(*, "names")= chr [1:3] "(Intercept)" "Education" "Catholic"
 $ residuals    : Named num [1:47] 14.32 6.55 11.85 13.34 13.92 ...
  ..- attr(*, "names")= chr [1:47] "Courtelary" "Delemont" "Franches-Mnt" "Moutier"
 $ effects      : Named num [1:47] -480.9 -56.2 -31 11.5 13.2 ...
  ..- attr(*, "names")= chr [1:47] "(Intercept)" "Education" "Catholic" "" ...
 $ rank         : int 3
 $ fitted.values: Named num [1:47] 65.9 76.5 80.7 72.5 63 ...
  ..- attr(*, "names")= chr [1:47] "Courtelary" "Delemont" "Franches-Mnt" "Moutier"
 $ assign       : int [1:3] 0 1 2
 $ qr           :List of 5
  ..$ qr   : num [1:47, 1:3] -6.856 0.146 0.146 0.146 0.146 ...
  .. ..- attr(*, "dimnames")=List of 2
  .. .. ..$ : chr [1:47] "Courtelary" "Delemont" "Franches-Mnt" "Moutier" ...
  .. .. ..$ : chr [1:3] "(Intercept)" "Education" "Catholic"
  .. ..- attr(*, "assign")= int [1:3] 0 1 2
  ..$ qraux: num [1:3] 1.15 1.03 1.17
  ..$ pivot: int [1:3] 1 2 3
  ..$ tol  : num 1e-07
```

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 - helps select or adjust values of your R object based on **position** integer(s)
- 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 61.7
[16] 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 58.3 65.4
[31] 75.5 69.3 77.3 70.5 79.4 65.0 92.2 79.3 70.4 65.7 72.7 64.4 77.6 67.6 35.0
[46] 44.7 42.8
```

Indexing

- Indexing also works for lists
- Can use the $ or two square brackets:

```
> m$coefficients
(Intercept)   Education    Catholic
 74.2336892  -0.7883293   0.1109210
> m[["coefficients"]]
(Intercept)   Education    Catholic
 74.2336892  -0.7883293   0.1109210
```

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 third and sixth element of a8 to missing values

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

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 actions 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

```
> my_square <- function(x = NULL){
+   y <- x ^ 2
+   return(y)
+ }
> my_square(x = 2)
[1] 4
> my_square(x = 10)
[1] 100
>
> # add a p argument to raise x to any power
> my_power <- function(x = NULL, p = 2){
+   y <- x ^ p
+   return(y)
+ }
> my_power(x = 2)
[1] 4
> my_power(x = 2, p = 4)
[1] 16
```

Functions

```
> my_square <- function(x = NULL){
+   y <- x ^ 2
+   return(y)
+ }
> my_square(x = 2)
[1] 4
> my_square(x = 10)
[1] 100
>
> # add a p argument to raise x to any power
> my_power <- function(x = NULL, p = 2){
+   y <- x ^ p
+   return(y)
+ }
> my_power(x = 2)
[1] 4
> my_power(x = 2, p = 4)
[1] 16
```

Loops

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

```
> # create empty vector to save action
> v8 <- rep(x = NA, times = 12)
> v8
 [1] NA NA NA NA NA NA NA NA NA NA NA NA
>
> # fill the vector with random numbers from a poisson distribution
> # with increasing lambda mean based on i value
> for(i in 1:10){
+   x <- rpois(n = 1, lambda = i)
+   message(x)
+   v8[i] <- x
+ }
4
0
4
3
8
1
5
5
9
8
> v8
 [1]  4  0  4  3  8  1  5  5  9  8 NA NA
```

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"))
+ }
```

Loops

- Example of a basic population projection based on a simple model:

$$p_{t+1} = p_t(1 + r_t)$$

```
> # initial population of 100 and growth rate of five percent per year
> p <- rep(NA, times = 10)
> p[1] <- 100
> for(i in 1:9){
+   p[i+1] <- p[i] * (1 + 0.05)
+ }
>
> p
 [1] 100.0000 105.0000 110.2500 115.7625 121.5506 127.6282 134.0096 140.7100
 [9] 147.7455 155.1328
```

Functions

```
> # can wrap for loops in a function
> 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 = 100, r = 0.05, n = 5)
[1] 100.0000 105.0000 110.2500 115.7625 121.5506 127.6282
>
> 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 8 (ex18.R)

```
# 1. Create a function called tfr with inputs
#    a) asfr with NULL default
#    b) period with 5 as the default value
#    that calculates the total fertility rate (tfr) based on a vector
#    of age specific fertility rates (asfr) over a give period.
tfr <- function(##### = NULL, period = #####){
  x <-  period * sum(asfr)
  return(x)
}
# 2. Run your function with the two sets of asfr values
#    a) 0.05, 0.1, 0.12, 0.09, 0.06 and 0.04
#    b) 0.1, 0.18, 0.16, 0.14, 0.12 and 0.02
tfr(asfr = c(0.05, #####, 0.12, 0.09, 0.06, #####))
#####(asfr = c(0.1, 0.18, 0.16, 0.14, #####, 0.02))
# 3. Create a function called rpois_add with inputs
#    a) n with default value 10
#    b) lambda1
#    c) lambda2
#    that adds together n random numbers from Poisson distributions with inputs from
rpois_add <- #####(n = 10, #####  = NULL, lambda2 = NULL){
  ##### <- rpois(n = #####, lambda = lambda1)
  r2 <- rpois(n = n, ##### = lambda2)
  #####(r1 + #####)
}
# 4. Run your function with n = 20, lambda1 = 5, lambda2 = 8
```

Working Directory

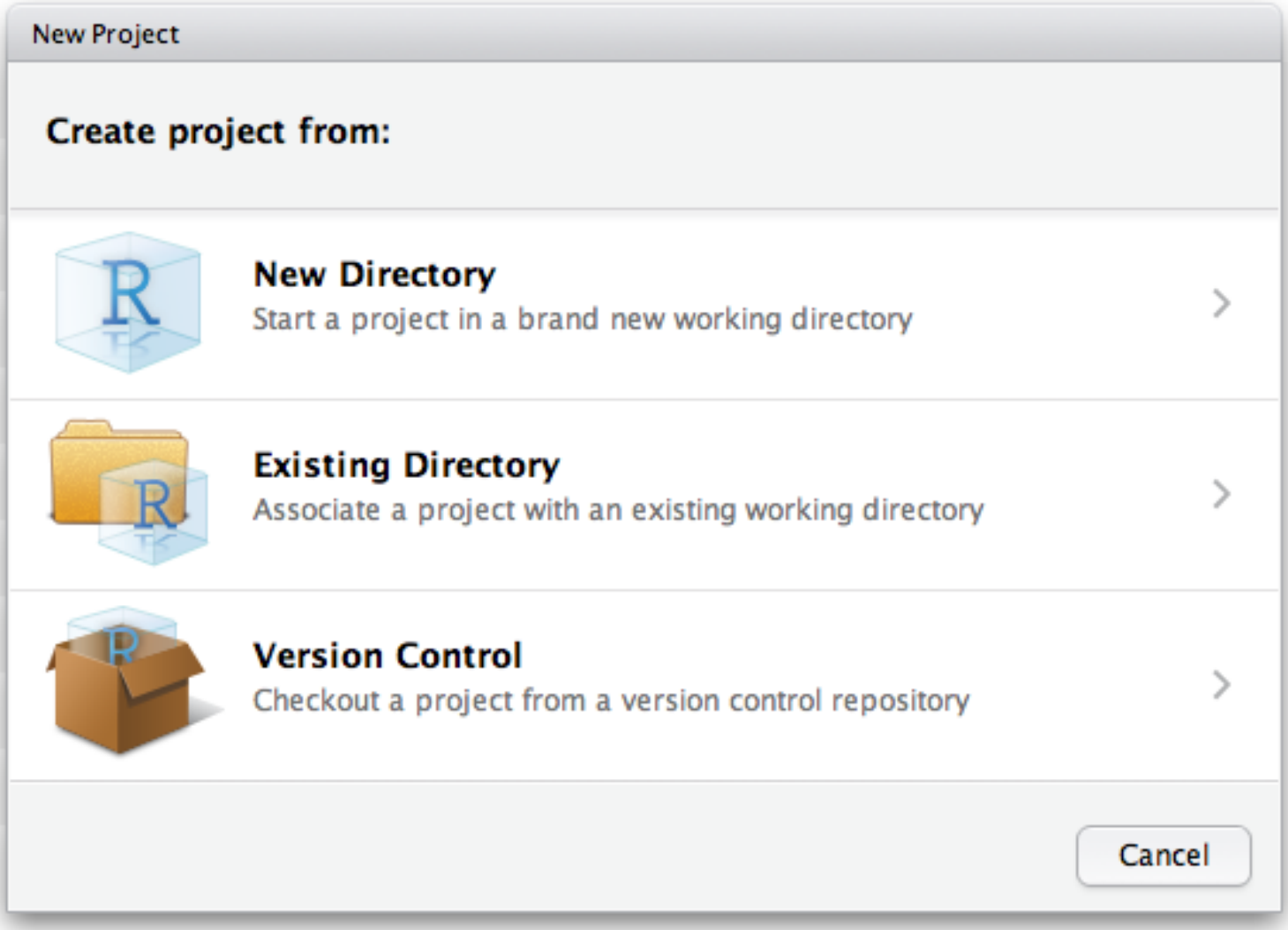
- R works in a particular location on your computer.
- The `getwd()` tells you the current R is working directory

```
> getwd()
"C:/Users/Guy/Teaching"
```

- 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).

Working Directory

- RStudio project files tells R which folder to work in.
  - Load the project file to get R set to the directory that you want to work in.



RStudio Projects

- At the start of your R session use `File | Open Project`
  - Can also select from the Project drop down in the top right hand corner.
- Once loaded or created you R session will be in the location of the `.Rproj` file.
- Can load data, save plots or source R scripts using full or relative paths
- For full paths R uses / or \\ instead of \ when setting directories:

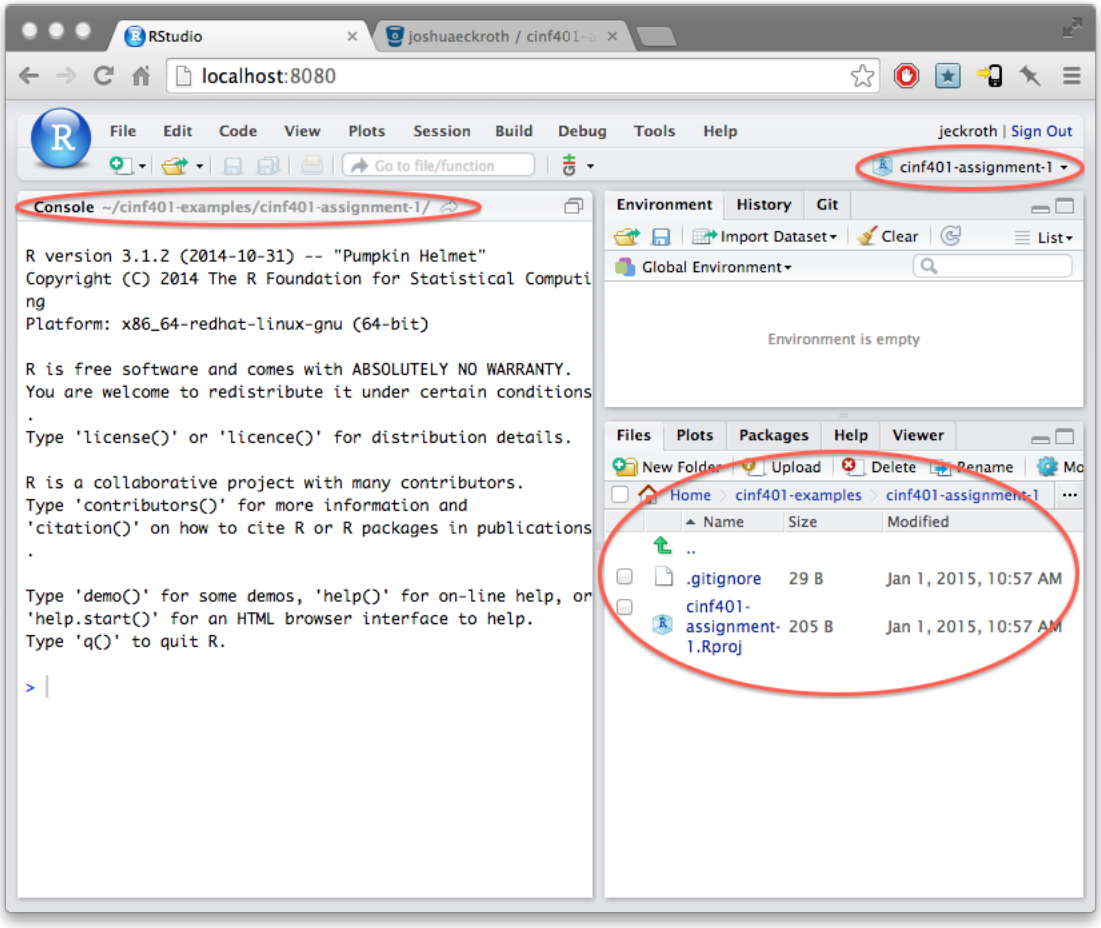
```
# read data from data folder
> d0 <- read_csv("C:/Users/Guy/Teaching/data/survey.csv")
# same as
> d0 <- read_csv("C:\\Users\\Guy\\Teaching\\data\\survey.csv")
# will not work
> d0 <- read_csv("C:\\Users\\Guy\\Teaching\\data\survey.csv")
Error: '\T' is an unrecognized escape in character string starting ""E:\T"
```

Relative paths

- For relative paths
  - / is the root folder of the file system.
  - ./ usually denotes the current folder that your program or script is in, usually the same one with the file you run.
  - ../ denotes the folder above the current one.

```
# read from data folder in current directory - C:/Users/Guy/Teaching
> d0 <- read_csv("./data/survey.csv")
# read from another directory - C:/Users/Guy/Research
> d0 <- read_csv("../Research/data-raw/SurveyResults.csv")
```

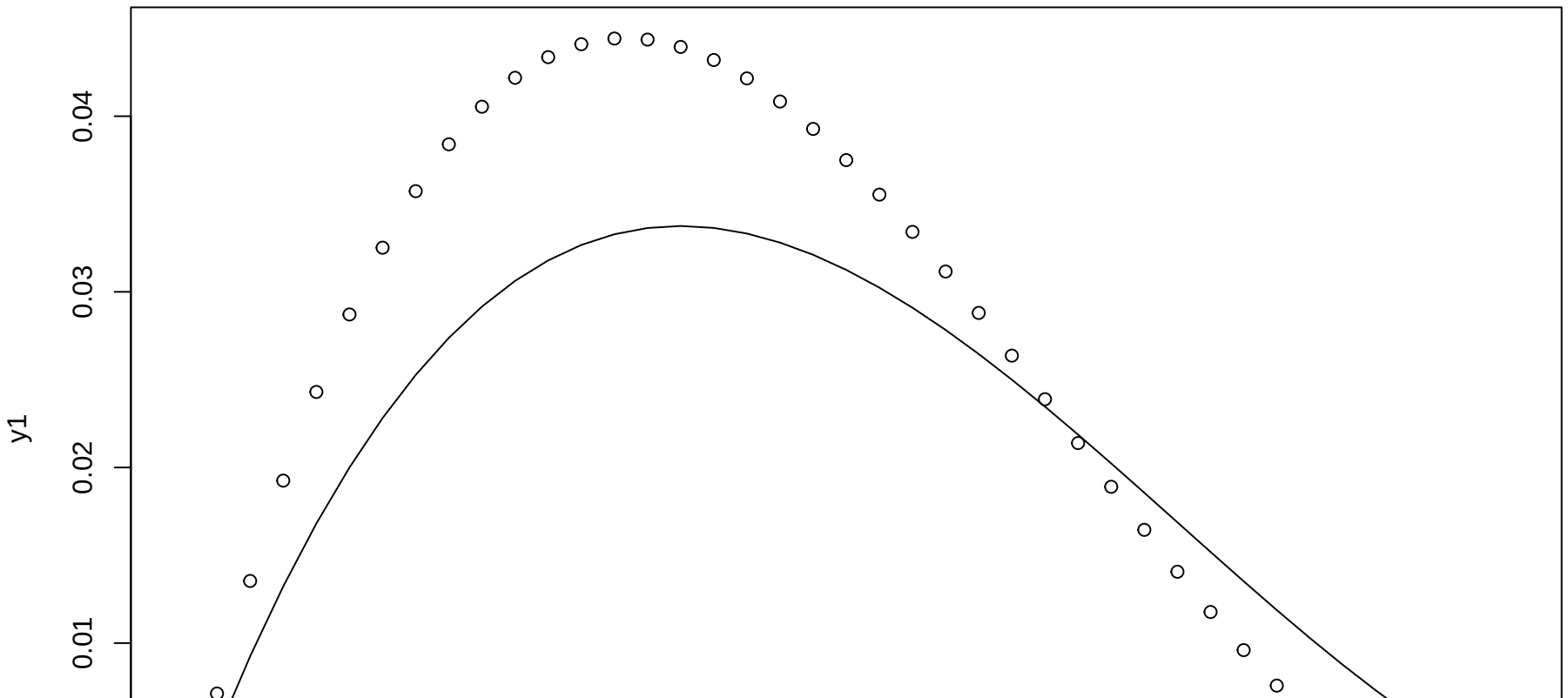
RStudio Projects



Sourcing a Script

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

```
> # remove all object
> rm(list = ls())
> # run code in ex19.R using relative path:
> source(file= "../exercise-solutions/ex18.R")
```



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 work flow
  - Keeps the number of objects to a minimum
  - Quickly identify a particular line of code for revision.