

R for newbies

Learning Objectives

- What is R
- Why R
- Find your way around RStudio
- R building blocks
- Libraries
- Using Help Pages

Lesson

What is R?

R is a free software environment for statistical computing and graphics. It works with a wide variety of UNIX platforms, Windows and MacOS. It has numerous advantages over other computer languages that make it particularly well suited to statistics.

Why R?

This is the long-story short version, but see [here](#) for more detail.

Why do people think R is hard? Because it's not a graphical user interface or GUI (pronounced gooey). No more point and click. You have to describe what tasks you want the computer to complete in text, using the language of R. Sounds like a disadvantage? We want to convince you otherwise.

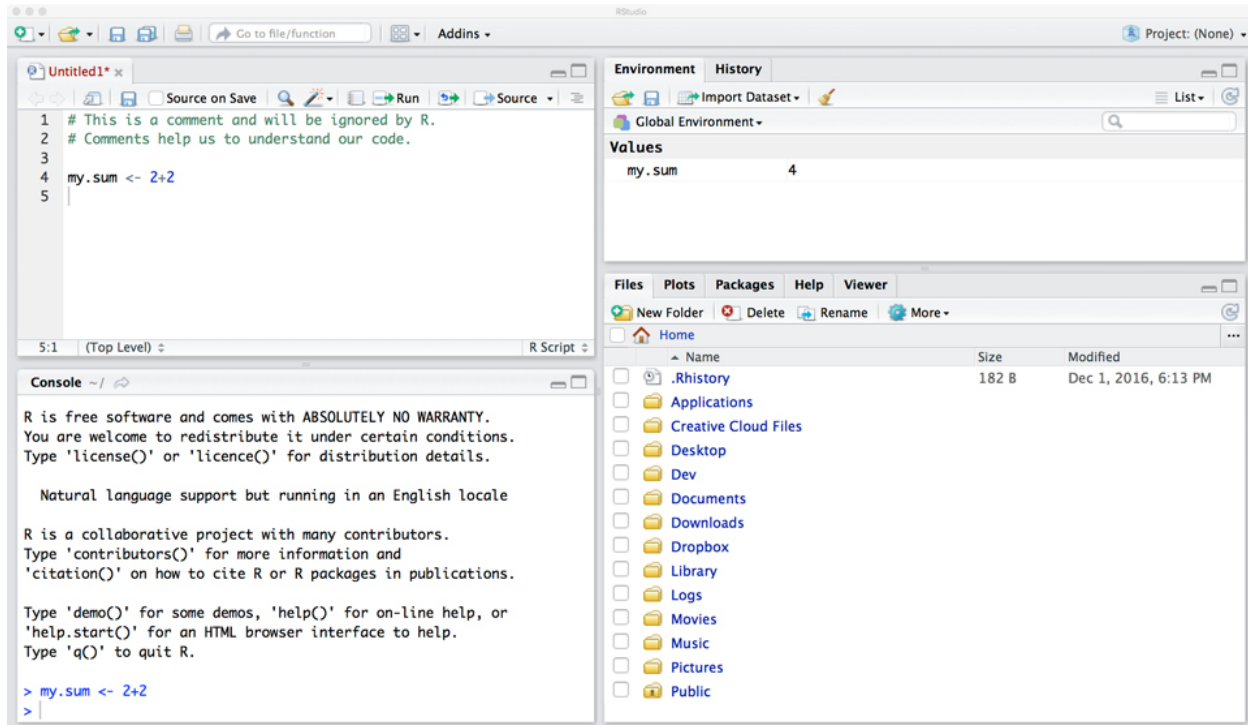
Three good reasons:

- You can do **anything** in R. If R is an entire language, we want to teach you how to become fluent. Once versed in this language you can tell the computer to do anything you can think of. In a GUI environment, someone has had to anticipate what you will want ahead of time, and translate a button push into a specific command. It's a little like going abroad armed with only a few flash cards to interact with the locals. Wouldn't you rather want to have a conversation in their native language?
- **Science should be reproducible.** You can't record point and clicks. But you can re-run your **code**. How many times have you trawelled through your data to produce a graph, only to return to it months later and have no idea how you produced it? This isn't an issue in R, as your code tells a permanent story as to how you arrived at your results.
- **Long term gain from upfront investment.** If you have ever worked hard preparing a table or a graph, and then discovered that either you have a new data point to add, or the next month's results are available then the GUI approach means starting all the pointing-and-clicking from scratch. But write it down once, re-run as often as you wish.

The underlying principle of what we want to show you is how to build a **data pipeline**. You start with your data in a spreadsheet. You write a bunch of instructions (a script) using the R language. Your script produces an output: a table, plot or entire manuscript. You can change your data, or add new data, and run the *same* script. Instantly regenerating an updated output.

RStudio - Making R a Breeze

RStudio is an application layer that sits on top of R known as an integrated development environment (IDE). It gives you some of the convenience of having a GUI, but with full access to the power of R. RStudio is a set of integrated tools designed to help you be more productive with R. It includes, to name a few: - a console (runs R) - a syntax-highlighting editor (helps make R more readable to humans) - tools for working with plots - a history of your actions - workspace management - debugging tools



The screen is divided into four panes. The two most important are labelled **Source** (*usually* top left), and **Console** (*usually* bottom left). On the right are supporting panes with tabs for finding help, and inspecting your files. If you can't see four panes then try clicking the 'minimise'/'maximise' icons in the top right corners of each pane.

There's a handy cheat sheet for R studio available [here](#).

Exercise Find the files tab and navigate to your project. If not already present, set up your project folder structure (I use folders for each of plots, data, reports and code). Navigate to your 'code' folder and then click on the 'more' button and select 'Set as working directory'.

Console (bottom left)

The console *is* R! Type anything here, and it will be interpreted by R.

Try typing 2+2

```
> 2+2
[1] 4
```

There are 4 things to explain in the little code snippet above.

1. The command prompt ">" (or greater than sign to you and me) is simply R prompting you to enter some text
2. The expression 2+2 is the sum that we asked R to perform.
3. We'll come back to the [1] at the beginning of the next line in a moment.

4. R prints the answer 4

Re-assuring as it is that R knows that $2+2=4$, you were probably hoping for a little more. Typing directly into R is a start, but we want to teach you *reproducible* research. The scientific method requires that we document our work, but we can't reproduce your typing unless we record it somewhere.

TIP: Use the up and down arrow keys to go backwards and forwards in your 'command' history to quickly fix typos.

Source (top left)

The solution is to create a file, write your commands in that file, and then tell R to work through the commands in that file. Switch to the pane labelled source, and this time type 2-2.

Now typing **enter** brings you to a new line. But go back to the line with 2-2 and hit **command-enter** (on Windows **control-enter**) instead. This sends the last line you wrote from the 'source' document, to the console. You should now see that R can add and subtract!

```
> 2+2
[1] 4
> 2-2
[1] 0
```

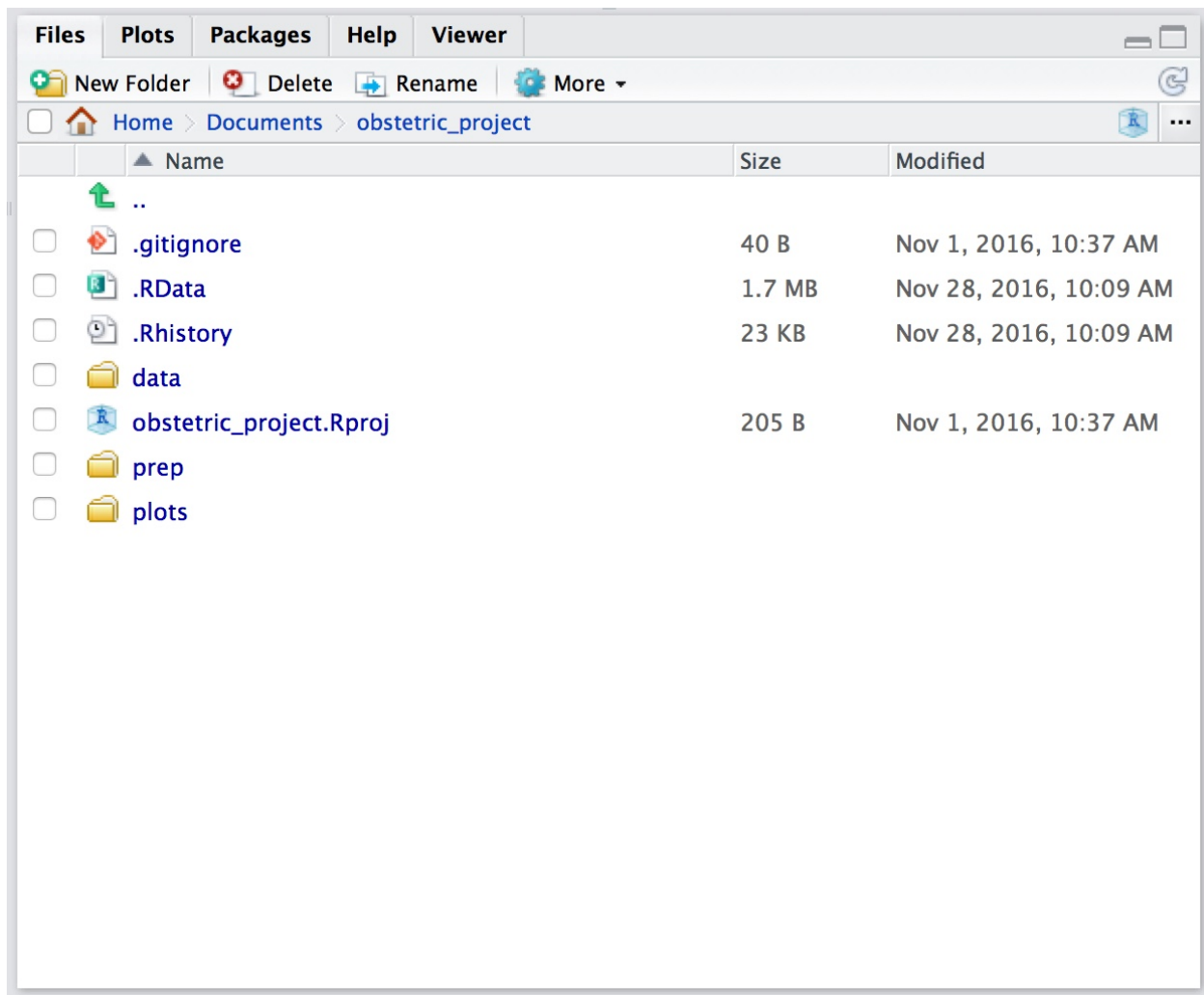
Now save the file you have written as `labbook_YYMMDD.R` (replace YYMMDD with today's date e.g. `labbook_160103.R`). You must use the `.R` extension to indicate that this is an R script, but you can, of course, choose any name you wish.

The console is also called an *interactive* R session. What you type here is gone when you close down R. The source pane is simply a view into a text file (with the `.R` extension) that allows you to document, edit, correct and *most importantly* save your work. The next time you come back to R, you re-open the file, and replay your commands. Got it? Then you're now starting to follow the principles of reproducible research.

Files and directories

It's going to be helpful to have an understanding of how files and folders (aka directories) are named on your computer because unlike your usual habit of pointing and clicking to open something, we will need to start writing things down. This will come in useful later when you want to tell R where to find an excel spreadsheet to import.

- Directories (folders) contain files and other directories.
- Files have a 'name', and an 'extension' (traditionally a three letter code telling the computer what sort of file it was). Here are some examples:
 - `cover letter.doc`
 - `my spreadsheet.xls`
 - `some_cool_data.RData`
- The 'root' directory is the start of the tree of folders that live on your hard drive.
 - On Windows machines you'll be used to thinking of the root as the `C:\` drive (a back slash)
 - On a Unix machine (Mac OS X and Linux), the root is just `/` (a forward slash)
- You can write the *absolute* 'path' to any file as if you walked along the branch of the tree from the 'root' to the file. For example, if you saved a Word document on your desktop called `readme.docx`:
 - Windows: `C:\Users\steve\desktop\readme.docx`
 - Unix: `/Users/steve/desktop/readme.docx`
- You can write the *relative* 'path' to any file using your current position as the starting point. This is called your 'working directory'. To move up a directory level you type `..`. You can see RStudio demonstrate this for you. Have a look at the image of RStudio's file browser below:



As you can see, to move up a level, RStudio has given you a graphical representation of `..`. You'll be using relative paths more often than absolute paths. So let's look at them in more detail.

- For example, if your working directory is your **documents** folder which is at
 - Windows: `C:\Users\steve\documents`
 - Unix: `/Users/steve/documents`
- And you want to go to the same `readme.docx` file as before. You would type
 - Windows: `..\desktop\readme.docx`
 - Unix: `../desktop/readme.docx`
- which reads as 'go up one level' (the `..`), then go down into a folder called **desktop**, and then to the file called `readme.docx`.
- Finally, the `.` (single not double dot) is just the short cut for the *current* directory. So `./readme.docx` simply means the file called `readme.docx` in *this* directory. Seems pedantic but that's computers for you!

You can see the path to any file or directory by right clicking and looking for properties (Windows), or using the 'Get Info' menu item (Mac OS X).

R building blocks

I want to argue that there are only three building blocks within R.

- names
- data
- functions

Names

The easiest way to work with something in R, be it data, numbers, functions, or anything else you can think of is to label it with a specific name. We use the `<-` assignment operator to assign a name. Just think of it as labelling whatever it is you are working on.

TIP: Use *option* (Mac OS X) or *alternate* (Windows) and the `-` (minus) key as a short cut to type this in RStudio.

Let `answer` equal 42

```
answer <- 42
```

Let `msg` equal Hello World!

```
msg <- `Hello World!`
```

Once assigned to that name, we can call it back again and again.

Data

We are about to use a bunch of very ‘math-y’ words. Please do not be put off!

R is a *statistical computing* language, and statistics uses vectors and matrices. These are *not(!)* complicated things.

Vectors

In R, a vector is a list of things of the same type.

A list of numbers (a number vector):

```
1, 4, 9, 16, 25
```

A list of characters (a character vector):

```
"Q", "W", "E", "R", "T", "Y"
```

A TRUE or FALSE list (a logical vector):

```
TRUE, FALSE, TRUE, TRUE, FALSE
```

We *index* the list by counting from left to right starting with 1 (not 0).

To make these vectors in R, we *combine* the listed elements using `c()`.

```
x <- c("Q", "W", "E", "R", "T", "Y")
```

Now if you want the 3rd element, simply type `x[3]`.

```
x <- c("Q", "W", "E", "R", "T", "Y")
x[3]
[1] "E"
```

Because R *always* ‘thinks’ in vectors, even the answer is ‘indexed’ hence the leading `[1]`.

TRY THIS: Try typing `1:100`. This is a shorthand way of writing all the numbers between 1 and 100. Look below, and see that the number in square brackets is just R keeping count of the numbers it shows you.

```
R> 1:100
 [1]  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19 20 21 22 23
[24] 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46
[47] 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69
[70] 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92
[93] 93 94 95 96 97 98 99 100
```

Data frames

Data frames are many vectors combined in a grid. It's analogous to storing data in an excel spread sheet. So if I have age, sex, and weight data in 3 vectors:

```
age <- c(5,7,41,41)
sex <- c("M", "M", "F", "M")
weight <- c(16,30,55,78)
```

Then I can combine them into a *data frame*.

```
class_demographics <- data.frame(age,sex,weight)
```

Have a look at `class_demographics`.

```
R> class_demographics
  age sex weight
1   5  M    16
2   7  M    30
3  41  F    55
4  41  M    78
```

What's nice is that the `data.frame` *numbers* the rows, and *names* the columns.

Moreover, rather than having to use numerical addresses to look at data, you can use the `$` operator to access the columns by name.

EXERCISE 1. Try typing `class_demographics$age`. What is the output? 2. What happens if you try to add a value `TRUE` to a vector containing just integers like `c(1, 4, 7, 9)`? 3. Now what happens if you add a string to your vector? Why is this happening?

Functions

Functions are abstract black boxes that take an input, and give you an output. They perform tasks. To perform a task, they need an input, and the result of their work is the output. A simple function might be to tell us the time. `sys.time()` is a function built into `r` that tells us the time. Try it yourself by typing it into the console.

Let's try a more visual example. We've written a custom function that helps navigate our character across the screen.

```
ed.moveforward()
```



The function `ed.moveforward` is something we've pre-written to get our character Ed to move forward across the screen. The `()` is the instruction to the computer to use that function. It's our 'go' command. Any statement followed by `()` is a function. This function takes no arguments at present. Arguments are additional pieces of information we can give to our function, to make its behaviour more specific. At the moment `ed.moveforward()` moves our character forward across the screen at a *really* slow pace. Lets see if we can pass an argument into our function to speed him up.

```
ed.moveforward(speed = "fast")
```



Now we've given the `ed.moveforward` function it's first argument. This argument tells the function the speed at which Ed should move forward. We have another function called `ed.turnright()` which turns Ed through a quarter turn to his right. We can chain these two together and get him to return to his starting position.

```
ed.moveforward(speed = "fast")
ed.turnright()
ed.turnright()
ed.moveforward(speed = "fast")
```




As you get used to `r` you'll want to start writing your own functions. Functional programming abstracts away difficult to read concepts, and helps you focus on the data. For now, just be aware that functions are an important part of using any computer language and will help you do anything. Try some of the following functions on your data, can you identify what they are doing?

```
head()
tail()
summary()
mean()
```

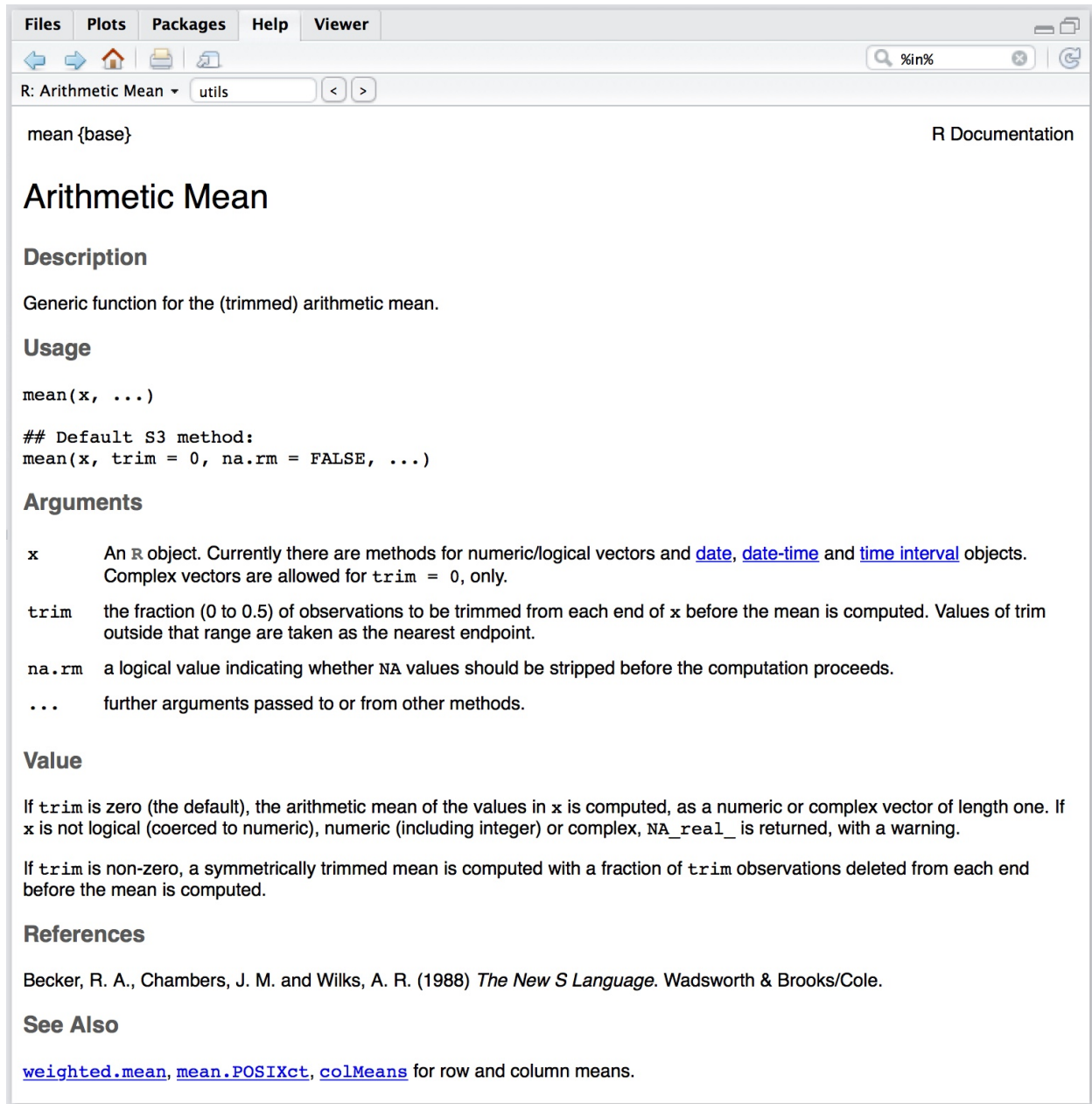
Using Help Pages

Functions in `r` are extensively documented. If you type any function, preceded by a `?` you will be taken to a help page. These are technical documents, but once you understand how to read them, it will open the door to understanding how any function works in `r`. Lets use the function `mean()` as an example.

```
?mean
```

Look through the help page. The information displayed is usually given in the same form: 1. Brief description 2. an example of usage 3. all the possible arguments that can be passed to the function (and whether they have default values) - you can see an example here with `na.rm = FALSE` this shows you that the argument `na.rm` (or, should the function remove missing values) is automatically passed to the function with a value of `FALSE`, unless you specify otherwise 4. A detailed breakdown of each argument 5. comprehensive details of

how the function works and what to expect as an output 6. An example of use, typically using manufactured data so you can use these examples yourself verbatim.



R: Arithmetic Mean ▾ utils < >

mean {base} R Documentation

Arithmetic Mean

Description

Generic function for the (trimmed) arithmetic mean.

Usage

```
mean(x, ...)
```

Default S3 method:
mean(x, trim = 0, na.rm = FALSE, ...)

Arguments

x An R object. Currently there are methods for numeric/logical vectors and [date](#), [date-time](#) and [time interval](#) objects. Complex vectors are allowed for `trim = 0`, only.

trim the fraction (0 to 0.5) of observations to be trimmed from each end of `x` before the mean is computed. Values of `trim` outside that range are taken as the nearest endpoint.

na.rm a logical value indicating whether NA values should be stripped before the computation proceeds.

... further arguments passed to or from other methods.

Value

If `trim` is zero (the default), the arithmetic mean of the values in `x` is computed, as a numeric or complex vector of length one. If `x` is not logical (coerced to numeric), numeric (including integer) or complex, `NA_real_` is returned, with a warning.

If `trim` is non-zero, a symmetrically trimmed mean is computed with a fraction of `trim` observations deleted from each end before the mean is computed.

References

Becker, R. A., Chambers, J. M. and Wilks, A. R. (1988) *The New S Language*. Wadsworth & Brooks/Cole.

See Also

[weighted.mean](#), [mean.POSIXct](#), [colMeans](#) for row and column means.

Exercises

Questions

1. Can you explain the difference between the console and the source panes in R studio?
2. In RStudio, have a look in the *Environment* tab of the pane on the top right? What do you think is shown here?

3. Try using the help function to find out what `ls()` does. Hint: try typing this in the search box of the *Help* pane. Don't worry if the 'help' doesn't make much sense! Or just type `?ls` in the console.

Home work

Type the following (you'll need a working internet connection).

```
install.packages("swirl")
library(swirl)
swirl()
```

This brings up an interactive R lesson. Give your name and proceed through the menu choices till you get to the course repository.

```
| Please choose a course, or type 0 to exit swirl.
```

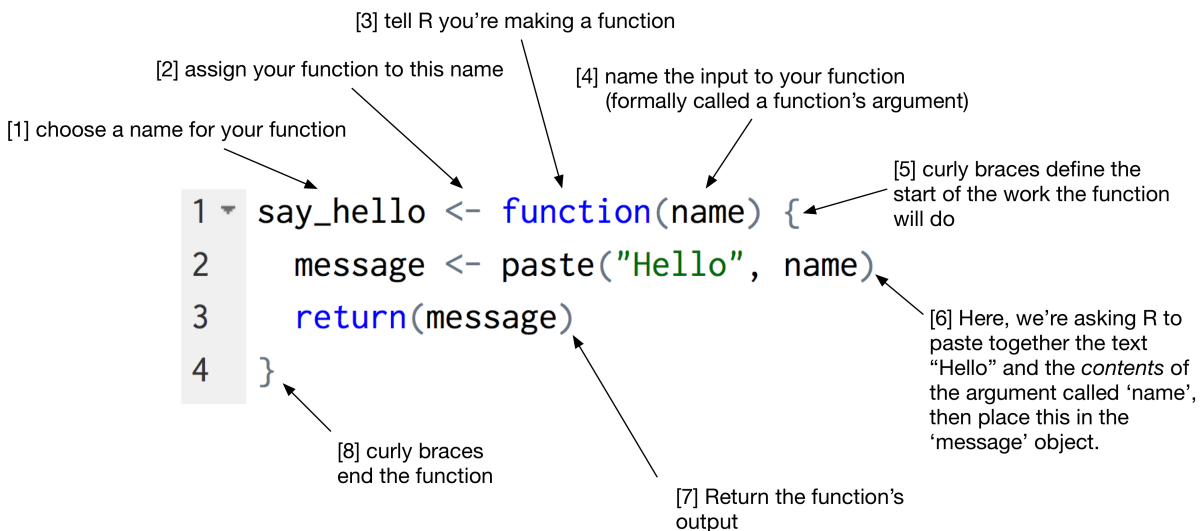
```
1: R Programming
2: Take me to the swirl course repository!
```

```
Selection: 1
```

```
| Please choose a lesson, or type 0 to return to course menu.
```

```
1: Basic Building Blocks      2: Workspace and Files      3: Sequences of Numbers
4: Vectors                   5: Missing Values          6: Subsetting Vectors
7: Matrices and Data Frames  8: Logic                   9: Functions
10: lapply and sapply        11: vapply and tapply      12: Looking at Data
13: Simulation               14: Dates and Times        15: Base Graphics
```

Extra Credit - Writing Functions



To write a function, we need to use the `function` keyword and assign it a name. The image above details this process.

Let's do a quick clinical example to help consolidate the lesson. We want to work out BMI from height and weight. We could write a function called `BMI` and pass it the arguments `height` and `weight`. It should end

up looking something like this when called:

```
BMI(height = 1.72, weight = 80)
R> [1] 27.04164
```

Ok, lets write this function

```
BMI <- function(height, weight) {
  x = weight / height^2
  return(x)
}
```

And let's break down each element:

BMI	# This is the name for our function
<-	# The assignment operator, assigning anything on the right hand side, to our chosen name
function	# the function keyword, indicating that we are going to write a function
(height, weight)	# the parenthesis contains the arguments that we are going to pass to the function
{	# the opening curly brace shows where the contents of the function start
x = ...	# everything in between is the contents of our function
}	# the closing curly brace shows where the contents of the function end

[Previous topic](#) — [Next topic](#)