Data Science for Economists

Lecture 4: R language basics

Kyle Coombs
Bates College | EC/DCS 368

Table of contents

1. Prologue
2. RStudio Layout
3. Object-oriented programming in R

[1] Items 6 and 7 are less critical to complete today. 6 is helpful to understand, but we can come back to it later as needed. 7 is just tips on how to refresh your R session without restarting.

Prologue

Agenda

- We're going to spent a lot of time live coding together. This is deliberate.
- You're also going to try code I will/have not covered in class -- also deliberate
- I want you to try R commands — and navigate RStudio — without copy+paste.
  - Slightly more painful in the beginning, but much better payoff in the long-run.

Goal:

- Today's goal is to make sure you know how to do basic skills in R
- These skills may seem simple, but are a critical foundation for the rest of the course
- There will also be some review from PS1
  - This is intentional: it is easier to understand a concept you see in multiple contexts
  - It also proves to you that you can figure a lot out on your own

Not the goal: Fluency

- My goal is not that you leave this lecture, or even this course, fluent in R
  - That's outside the scope of 80 minutes, let alone 12 weeks

While I still have your attention

- Guess what? Everything I'm teaching you is summarized in a cheatsheet
- Posit, the parent organization of R, hosts loads of cheatsheets
- I link to them on the course website
- I cannot stress enough how useful these are when trying to figure out how to write code
- It will save you time painstakingly searching Google, StackOverflow, scouring help files, and bickering with ChatGPT/GitHub CoPilot, etc.

Introduction

The RStudio Panes

- Console
- Environment Pane
- Browser Pane
- Source Editor

Console

- Typically bottom-left
- This is where you can type in code and have it run immediately
- Or, when you run code from the Source Editor, it will show up here
- It will also show any output or errors

Console Example

- Let's copy/paste some code in there to run

```r
#Generate 500 heads and tails
dat <- sample(c("Heads","Tails"),500,replace=TRUE)
#Calculate the proportion of heads
mean(dat=="Heads")
#This line should give an error - it didn't work!
dat <- sample(c("Heads","Tails"),500,replace=BLUE)
#This line should give a warning
#It did SOMETHING but maybe not what you want
mean(dat)
#This line won't give an error or a warning
#But it's not what we want!
mean(dat=="heads")
```

What We Get Back

- We can see the code that we've run
- We can see the output of that code, if any
- We can see any errors or warnings (in red). Remember - errors mean it didn't work. Warnings mean it *maybe* didn't work.
- Just because there's no error or warning doesn't mean it DID work! Always think carefully
- Specific note: Warning: (package name) was built in R version (version number) just means that your R installation isn't fully updated. Usually not a problem, but you can update R at R-project.org to make this go away

Environment Tab

- Environment tab shows us all the objects we have in memory
- For example, we created the `dat` object, so we can see that in Environment
- It shows us lots of handy information about that object too
  - (we'll get to that later)
- You can erase everything with that little broom button (technically this does `rm(list=ls())`)

Browser Pane

- Bottom-right
- Lots of handy stuff here!
- Mostly, the *outcome* of what you do will be seen here
- Plots you make will show up here
- Some functions create tables or output that show up in Viewer
- Packages tab - avoid for loading, but the update button is nice!

Files Tab

- Basic file browser
- Handy for opening up files
- Can also help you set the working directory:
    - Go to folder
    - In menu bar, Session
    - Set Working Directory
    - To Files Pane Location

Help Tab

- This is where help files appear when you ask for them
- You can use the search bar here, or

```r
help(plot)
?plot # This is what most people use
```

- In additon to documentation there's:
  - Vignettes (more detailed documentation), type `vignette("packagename")`
  - Demos (interactive examples), type `demo("packagename")`
  - Examples (examples of how to use the function), type `example("functionname")`
- Of course, plenty of materials also available across the internet!
  - And Gen AI can be helpful, but remember it's not always right

Source Pane

- You should be working with code FROM THIS PANE, not the console!
- Why? Replicability!
- Also, COMMENTS! USE THEM! PLEASE! `#` lets you write a comment.
- Switch between tabs like a browser

Aside: Comments in R files are demarcated by `#`.

- Hit `Ctrl+Shift+c` (`Cmd+Shift+c` on Macs) in RStudio to (un)comment whole sections of highlighted code.

- In Rmarkdown files, `<!--- --->` is the equivalent syntax for comments. NOT `#`.

    - Yes, that's confusing. Yes, I expect you to use the syntax correctly.

Running Code from the Source Pane

- Select a chunk of code and hit the "Run" button
- Click on a line of code and do Ctrl/Cmd-Enter to run just that line and advance to the next <- Super handy!
- Going one line at a time lets you check for errors more easily
- Let's try some!

```r
data(mtcars)
mean(mtcars$mpg)
mean(mtcars$wt)
372+565
log(exp(1))
2^9
(1+1)^9
```

More fun comamnds and tricks in the appendix

Autocomplete

- RStudio comes with autocomplete!
- Typing in the Source Pane or the Console, it will try to fill in things for you
    - Command names (shows the syntax of the function too!)
    - Object names from your environment
    - Variable names in your data
- Let's try redoing the code we just did, typing it out
- It also pairs with GitHub CoPilot if you have successfully gotten GitHub Education access

Help

- Autocomplete is one way that RStudio tries to help you out
- The way that R helps you out is with the documentation
- When you start doing anything serious with a computer, like programming, the most important skills are:
    - Knowing to read documentation
    - Knowing to search the internet for help (always!)

help()

- You can get the documentation on most R objects using the `help()` function
- `help(mean)`, for example, will show you:
  - What the function is
  - The "syntax" for the function and the order the arguments go in
  - The available options for the function
  - Other, related functions, like `weighted.mean`
  - Ideally, some examples of proper use
- Not just for functions/commands - some data sets will work too! Try `help(mtcars)`

Packages

- R runs on user-contributed packages that contain functions you can use
- Packages stored on CRAN can be installed with `install.packages('packagename')`
- Once a package is installed you don't need to install it again (except to update it)
- But every time you open R you'll need to load it in again with `library(packagename)` if you want to use its functions
- *Please don't* include package installation in your code itself; this will make you re-install the package every time you run!

```r
# If we haven't installed it yet
# install.packages('vtable')
library(vtable)
vtable(iris)
```

Rmarkdown

- Go File $\rightarrow$ New File $\rightarrow$ RMarkdown to create a new RMarkdown document[1]
- RMarkdown is a blend of an R file and a markdown file
  - Markdown is a simple way to format text
  - R is a programming language
  - RMarkdown lets you write text and code in the same document
- A text document with some basic layout, for example hashtags for sectioning
- Include code chunks with three backticks which execute when you Render to PDF, HTML, MD, etc.
- Include code in-line with single backticks and an `r`

[1] You can also make Quarto documents, which work across languages, but we're sticking with RMarkdown for now.

Object-oriented programming in R

Working in R

- Everything in R is an object
- We can only really do three things in R:
  - Create objects with `←` or `=`
  - Send objects through functions to manipulate them
  - Look at objects

Appendix: More on objects

Objects

- Let's create a basic object

```
a <- 1
```

- We've taken `1` and stored it inside the `a` object
- Now if we just type `a` by itself, it will show us the `1` we stored inside
- This is a numeric object. We could also have `'strings'` or logicals: `TRUE` or `FALSE` or factors

Objects

- We can manipulate objects

```r
a + 1
```

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

- "create a new object that takes `a` (1) and adds 1 to it (`1+1=2`)
- Notice that `a` itself doesn't change until we *reassign it*

```r
a
```

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

```r
a = a + 1
a
```

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

Assignment

Assignment with `←`[1]

`←` is normally read aloud as "gets". You can think of it as a (left-facing) arrow.[2]

```
b ← 10 + 5
b
```

```
## [1] 15
```

Assignment with `=`

```
b = 10 + 10 ## Note that the assigned object *must* be on the left with "=".
b
```

```
## [1] 20
```

R purists insist on `←`, but just pick one and be consistent

Assignment

Assignment with `←`[1]

`←` is normally read aloud as "gets". You can think of it as a (left-facing) arrow.[2]

```
b ← 10 + 5
b
```

```
## [1] 15
```

Assignment with `=`

```
b = 10 + 10 ## Note that the assigned object *must* be on the left with "=".
b
```

```
## [1] 20
```

R purists insist on `←`, but just pick one and be consistent

[1] The `←` is really a `<` followed by a `-`. It just looks like one thing b/c of the font I'm using here.

[2] An arrow can point in the other direction too (i.e. `→`). So, `10 + 5 → a` following code chunk is equivalent, although used much less frequently.

Vectors

- A vector is a collection of objects of the same type
- While lots of functions create vectors, we can also make them ourselves with `c()` (concatenate)

```
my_vector ← c(1,8,2,4,3)
```

- We can refer to a certain element of a vector with square brackets `[]` with a single number or a range `start:end`

```
my_vector[3:4]
```

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

- Or using another vector of logicals (`TRUE` and `FALSE`) to pick elements (handy when we get to data!)

```
my_vector[c(TRUE, FALSE, TRUE, TRUE, FALSE)]
```

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

More on indexing

Data Frames

- A `data.frame` is what we'll be working with most of the time.
- It's a collection of vectors of the same length
- We can create them ourselves, but often we will read in a file or use `data()` to get a data set

```r
df1 = data.frame(x = 1:5, y = 6:10)

data(mtcars)
mtcars
```

```
##                      mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Mazda RX4           21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
## Mazda RX4 Wag       21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
## Datsun 710          22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Hornet 4 Drive      21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
## Hornet Sportabout   18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
## Valiant             18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
## Duster 360          14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
## Merc 240D           24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
## Merc 230            22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
## Merc 280            19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
## Merc 280C           17.8   6 167.6 123 3.92 3.440 18.90  1  0    4    4
## Merc 450SE          16.4   8 275.8 180 3.07 4.070 17.40  0  0    3    3
## Merc 450SL          17.3   8 275.8 180 3.07 3.730 17.60  0  0    3    3
```

Data Frames

- We can get a vector back out of the data frame with `$` or `[[]]`

```
mtcars$cyl[1:5]
```

```
## [1] 6 6 4 6 8
```

```
mtcars[['cyl']][1:5]
```

```
## [1] 6 6 4 6 8
```

Data Frames

- Which we might want to do to send it to a function like `mean` that takes a vector!

```
mean(mtcars$cyl)
```

```
## [1] 6.1875
```

Logic

R also comes equipped with a full set of logical operators and Booleans, which follow standard programming protocol. For example:

```r
1 > 2
```

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

```r
1 > 2 & 1 > 0.5 ## The "&" stands for "and"
```

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

```r
1 > 2 | 1 > 0.5 ## The "|" stands for "or" (not a pipe a la the shell)
```

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

```r
isTRUE (1 < 2)
```

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

Logic

R also comes equipped with a full set of logical operators and Booleans, which follow standard programming protocol. For example:

```r
1 > 2
```

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

```r
1 > 2 & 1 > 0.5 ## The "&" stands for "and"
```

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

```r
1 > 2 | 1 > 0.5 ## The "|" stands for "or" (not a pipe a la the shell)
```

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

```r
isTRUE (1 < 2)
```

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

You can read more about logical operators and types here and here. I also summarise more in the appendix.

if/else

The `if/else` statement is a fundamental building block of programming logic. It allows us to evaluate a logical statement and then execute a particular command if that statement is TRUE. For example:

```r
x = 5
if (1>x) {
  print("x is greater than 1")
} else {
  print("x is less than or equal to 1")
}
```

```
## [1] "x is less than or equal to 1"
```

R's `ifelse` collapses this into one line. (Try it yourself.)

```r
ifelse(1>x, "x is greater than 1", "x is less than or equal to 1")
```

```
## [1] "x is less than or equal to 1"
```

- `case_when()` in the `dplyr` package is a more flexible version of `ifelse()` that can handle multiple conditions. (It is on the problem set.)

Global environment

Let's go back to the df1 data frame we made

```
df1
```

```
##   x  y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

Global environment

Let's go back to the df1 data frame we made

```
df1
```

```
##    x  y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

Now, let's try to run a regression[1] on these "x" and "y" variables:

```
lm(y ~ x) ## The "lm" stands for linear model(s)
```

```
## Error in eval(predvars, data, env): object 'y' not found
```

[1] Yes, this is a dumb regression with perfectly co-linear variables. Just go with it.

Global environment

Let's go back to the df1 data frame we made

```
df1
```

```
##   x  y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

Now, let's try to run a regression[1] on these "x" and "y" variables:

```
lm(y ~ x) ## The "lm" stands for linear model(s)
```

```
## Error in eval(predvars, data, env): object 'y' not found
```

Uh-oh. What went wrong here? (Answer on next slide.)

[1] Yes, this is a dumb regression with perfectly co-linear variables. Just go with it.

Global environment (cont.)

The error message provides the answer to our question:

```
## Error in eval(predvars, data, env): object 'y' not found
```

Global environment (cont.)

The error message provides the answer to our question:

```
## Error in eval(predvars, data, env): object 'y' not found
```

R can't find the variables that we've supplied in our Global Environment:

Global environment (cont.)

The error message provides the answer to our question:

```
## Error in eval(predvars, data, env): object 'y' not found
```

R can't find the variables that we've supplied in our Global Environment:

Put differently: We have to tell R that they belong to the object `df1`.

- Think about how you might do this before clicking through to the next slide.

Global environment (cont.)

There are a various ways to solve this problem. One is to simply specify the datasource:

```
lm(y ~ x, data = df1) ## Works when we add "data = df1"!
```

```
##
## Call:
## lm(formula = y ~ x, data = df1)
##
## Coefficients:
## (Intercept)            x
##           5            1
```

Global environment (cont.)

There are a various ways to solve this problem. One is to simply specify the datasource:

```
lm(y ~ x, data = df1) ## Works when we add "data = df1"!
```

```
##
## Call:
## lm(formula = y ~ x, data = df1)
##
## Coefficients:
## (Intercept)            x
##           5            1
```

I want to emphasize this global environment issue, because it is something that Stata users (i.e. many economists) struggle with when they first come to R.

- In Stata, the entire workspace essentially consists of one (and only one) data frame meaning no ambiguity where variables are coming from.
- That "convenience" has a high price -- literally you need to buy Stata 16 or higher to use `frames` to open multiple data frames with less flexibility.
- Speaking of which...

Working with multiple objects

R's ability to keep multiple objects in memory at the same time is a huge plus for data work.

- E.g. We can copy an existing data frame, or create new one entirely from scratch. Either will exist happily with our existing objects in the global environment.
- Just make sure to give them distinct names and be specific about which objects you are referring to.
- More on names

```
df2 = data.frame(x = rnorm(10), y = runif(10))
```

dplyr

- In this course, we'll be working with data largely using the **dplyr** package, which is a part of the **tidyverse**
- **dplyr** is a collection of verbs for manipulating data, all strung together with the pipe `%>%` (which technically **dplyr** just borrows from **magrittr**)
- **dplyr** has lots of functions in it. Today we'll cover just five: `pull()`, `filter()`, `%>%`, `select()`, and `mutate()`.
- For all of these, don't forget that if you want to take the data frame you've changed and *keep* that change, you have to re-assign the object with `<-` or `=`!
- If you already know a different way of doing things in R, I **strongly recommend** using the **dplyr** commands, if only for this class, as it will make it much easier to follow along with the material, and it's good to learn to pick up new things (and also in my experience you guys end up making things waaaay harder on yourselves by avoiding the switch)

pull()

- Pull just takes a vector back out of a data set, just like `$` or `[[]]`
- So why use it? It plays well with the pipe (upcoming)
- Also, it's flexible. It takes a variable name, a variable name as a string, or a column number

```
mtcars$cyl
pull(mtcars, cyl)
pull(mtcars, 'cyl')
pull(mtcars, 2)
mtcars %>% pull(cyl)
```

filter()

- `filter()` picks just some of the *rows* of your data
- Important for analyzing a subset of your data!
- Give it a logical statement that's `TRUE` for the rows you want. `==` checks for equality!

```r
filter(mtcars, cyl == 4 & am == 1)
```

```
##                 mpg cyl  disp  hp drat    wt  qsec vs am gear carb
## Datsun 710     22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
## Fiat 128       32.4   4  78.7  66 4.08 2.200 19.47  1  1    4    1
## Honda Civic    30.4   4  75.7  52 4.93 1.615 18.52  1  1    4    2
## Toyota Corolla 33.9   4  71.1  65 4.22 1.835 19.90  1  1    4    1
## Fiat X1-9      27.3   4  79.0  66 4.08 1.935 18.90  1  1    4    1
## Porsche 914-2  26.0   4 120.3  91 4.43 2.140 16.70  0  1    5    2
## Lotus Europa   30.4   4  95.1 113 3.77 1.513 16.90  1  1    5    2
## Volvo 142E     21.4   4 121.0 109 4.11 2.780 18.60  1  1    4    2
```

The pipe

- The pipe `%>%` is important for making your code readable, and minimizing balanced-parentheses errors
- It takes whatever is on its left and makes it the first argument of the function on the right
- So whatever object you're working with you take, ship it along to the next function, process, then ship along again, then ship along again! Like a conveyer belt
- Notice that all **dplyr** functions take the data frame as the first argument, making it easy to chain them
- "Ships along" anything, including vectors or single numbers, not just data frames! Track what the object being shipped is in each step.

The pipe

- See how clean it can make the code!

```r
mean(mtcars[mtcars$am == 1,]$cyl, na.rm = TRUE)
```

```
## [1] 5.076923
```

vs.

```r
mtcars %>%
  filter(am == 1) %>%
  pull(cyl) %>%
  mean(na.rm = TRUE)
```

```
## [1] 5.076923
```

select()

- `select()` is like `filter()`, but instead of picking rows it picks columns
- Unlike `pull()` it doesn't give you a vector - it gives you back a data frame with fewer columns

```
mtcars %>%
  select(mpg, cyl) %>%
  summary()
```

```
##       mpg             cyl       
##  Min.   :10.40   Min.   :4.000  
##  1st Qu.:15.43   1st Qu.:4.000  
##  Median :19.20   Median :6.000  
##  Mean   :20.09   Mean   :6.188  
##  3rd Qu.:22.80   3rd Qu.:8.000  
##  Max.   :33.90   Max.   :8.000  
```

mutate()

- `mutate()` creates a new column using the old ones. You can assign multiple columns at once!
- The syntax is `NewVariableName = FunctionOfOldVariables`
- Don't forget to save the object!

```
mtcars <- mtcars %>%
  mutate(high_mpg = mpg > median(mpg))
mtcars %>%
  pull(high_mpg) %>%
  table()
```

```
## .
## FALSE  TRUE
##    17    15
```

group_by()

- The `group_by()` command can be used to group data by one or more variables
- This is useful for calculating statistics by group

```
mtcars %>%
  group_by(cyl) %>%
  mutate(mean_mpg_by_cyl = mean(mpg))
```

```
## # A tibble: 32 × 13
## # Groups:   cyl [3]
##      mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb high_mpg
##    <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <lgl>
##  1  21       6  160    110  3.9   2.62  16.5     0     1     4     4 TRUE
##  2  21       6  160    110  3.9   2.88  17.0     0     1     4     4 TRUE
##  3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1 TRUE
##  4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1 TRUE
##  5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2 FALSE
##  6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1 FALSE
##  7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4 FALSE
##  8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2 TRUE
##  9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2 TRUE
## 10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4 FALSE
## # ℹ 22 more rows
## # ℹ 1 more variable: mean_mpg_by_cyl <dbl>
```

summarise()

- `summarise` is useful for collapsing data
- It can be used to calculate summary statistics for each group

```
mtcars %>%
  group_by(cyl) %>%
  summarise(mean_mpg = mean(mpg), sd_mpg = sd(mpg))
```

```
## # A tibble: 3 × 3
##     cyl mean_mpg sd_mpg
##   <dbl>    <dbl>  <dbl>
## 1     4     26.7   4.51
## 2     6     19.7   1.45
## 3     8     15.1   2.56
```

Basics of R

- This has been a lot of information!
- To really learn it you'll have to get used to applying it yourself
- For that we will be using the **swirl** package which will walk us through using these commands!

Swirl

- Open up R and install Swirl with `install.packages('swirl')`
- Then, load up swirl with `library(swirl)`
- Download the Swirl course for this class (the first time you use it) with `install_course_github('big-data-and-economics','ECON368-R-Swirls')`
- Then start swirl with `swirl()` and pick ECON368-R-Swirls, and the R Basics lesson. Let's work through this!
- This is your in-class assignment, due before Thursday

Next lecture(s): Working with data!

Appendix

Basic arithmetic

R is a powerful calculator and recognizes all of the standard arithmetic operators:

```
1+2 ## Addition
```

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

```
6-7 ## Subtraction
```

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

```
5/2 ## Division
```

```
## [1] 2.5
```

```
2^3 ## Exponentiation
```

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

```
2+4*1^3 ## Please Excuse My Dear Aunt Sally (PEMDAS)
```

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

Basic arithmetic (cont.)

We can also invoke modulo operators (integer division & remainder).

- Very useful when dealing with time, for example.

```r
100 %/% 60 ## How many whole hours in 100 minutes?
```

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

```r
100 %% 60 ## How many minutes are left over?
```

```
## [1] 40
```

Logic (cont.)

Order of precedence

Is this statement TRUE or FALSE?

```
1 > 0.5 & 2
```

Logic (cont.)

Order of precedence

Is this statement TRUE or FALSE?

```
1 > 0.5 & 2
```

Logic statements follow a strict order of precedence. Logical operators (`>`, `==`, etc) are evaluated before Boolean operators (`&` and `|`). Failure to recognise this can lead to unexpected behaviour...

Logic (cont.)

Order of precedence

Is this statement TRUE or FALSE?

```r
1 > 0.5 & 2
```

Logic statements follow a strict order of precedence. Logical operators (`>`, `==`, etc) are evaluated before Boolean operators (`&` and `|`). Failure to recognise this can lead to unexpected behaviour...

What's happening here is that R is evaluating two separate "logical" statements:

- `1 > 0.5`, which is is obviously TRUE.
- `2`, which is TRUE(!) because R is "helpfully" converting it to `as.logical(2)==TRUE`.

Logic (cont.)

Order of precedence

Is this statement TRUE or FALSE?

```
1 > 0.5 & 2
```

Logic statements follow a strict order of precedence. Logical operators (`>`, `==`, etc) are evaluated before Boolean operators (`&` and `|`). Failure to recognise this can lead to unexpected behaviour...

What's happening here is that R is evaluating two separate "logical" statements:

- `1 > 0.5`, which is is obviously TRUE.
- `2`, which is TRUE(!) because R is "helpfully" converting it to `as.logical(2)==TRUE`.

Solution: Be explicit about each component of your logic statement(s).

```
1 > 0.5 & 1 > 2
```

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

Logic (cont.)

Negation: `!`

We use `!` as a short hand for negation. This will come in very handy when we start filtering data objects based on non-missing (i.e. non-NA) observations.

```
is.na(1:10)
```

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

```
!is.na(1:10)
```

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

```
# Negate(is.na)(1:10) ## This also works. Try it yourself.
```

Logical operators (cont.)

Value matching: `%in%`

To see whether an object is contained within (i.e. matches one of) a list of items, use `%in%`[1] .

```r
4 %in% 1:10
```

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

```r
4 %in% 5:10
```

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

[1] There's no equivalent "not in" command, but how might we go about creating one? See here.

Logical operators (cont.)

Evaluation

We'll get to assignment shortly. However, to preempt it somewhat, we use two equal signs for logical evaluation.

```
1 = 1 ## This doesn't work
```

```
## Error in 1 = 1: invalid (do_set) left-hand side to assignment
```

```
1 == 1 ## This does.
```

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

```
1 != 2 ## Note the single equal sign when combined with a negation.
```

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

Not in

There's no equivalent "not in" command, but how might we go about creating one?

- Hint: Think about negation...

Not in

There's no equivalent "not in" command, but how might we go about creating one?

- Hint: Think about negation...

```
`%ni%` = Negate(`%in%`) ## The backticks (`) help to specify functions.
4 %ni% 5:10
```

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

Back

"Everything is an object"

Motivation

R is an object-oriented programming (OOP)[1] , which is often summarised as:

> **"Everything is an object and everything has a name."**

Motivation

R is an object-oriented programming (OOP)[1] , which is often summarised as:

> **"Everything is an object and everything has a name."**

In the next two sections, I want to dive into this idea a little more. I also want to preempt some issues that might trip you up if you new to R or OOP in general.

- At least, they were things that tripped me up at the beginning (and still do)

The good news is that avoiding and solving these issues is pretty straightforward.

- Not to mention: A very small price to pay for the freedom and control that R offers us.

[1] Technically, there are actually *multiple* OOP frameworks in R (**S3**, **S4**, **R6**). Hadley Wickham's "Advanced R" provides a very thorough overview of the main ones. Read his book sometime if you're into this stuff, it is superbly helpful.

What are objects?

It's important to emphasise that there are many different *types* (or *classes*) of objects.

We'll revisit the issue of "type" vs "class" in a slide or two. For the moment, it is helpful simply to name some objects that we'll be working with regularly:

- vectors
- matrices
- data frames
- lists
- functions
- etc.

What are objects?

It's important to emphasise that there are many different *types* (or *classes*) of objects.

We'll revisit the issue of "type" vs "class" in a slide or two. For the moment, it is helpful simply to name some objects that we'll be working with regularly:

- vectors
- matrices
- data frames
- lists
- functions
- etc.

Most likely, you already have a good idea of what distinguishes these objects and how to use them.

- However, there are subtleties that may confuse while you're still getting used to R.
- E.g. There are different kinds of data frames. "tibbles" and "data.tables" are enhanced versions of the standard data frame in R.

Object class, type, and structure

```r
df1 = data.frame(x = 1:2, y = 3:4)  ## Create a small data frame called "df1".
```

Use the `class`, `typeof`, and `str` commands to understand more about a particular object.

```r
class(df1) ## Evaluate its class.
```

```
## [1] "data.frame"
```

```r
typeof(df1) ## Evaluate its type.
```

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

```r
str(df1) ## Show its structure.
```

```
## 'data.frame':    2 obs. of  2 variables:
##  $ x: int  1 2
##  $ y: int  3 4
```

Object class, type, and structure

```r
df1 = data.frame(x = 1:2, y = 3:4)  ## Create a small data frame called "df1".
```

Use the `class`, `typeof`, and `str` commands to understand more about a particular object.

```r
class(df1) ## Evaluate its class.
```

```
## [1] "data.frame"
```

```r
typeof(df1) ## Evaluate its type.
```

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

```r
str(df1) ## Show its structure.
```

```
## 'data.frame':    2 obs. of  2 variables:
##  $ x: int  1 2
##  $ y: int  3 4
```

PS — Confused why `typeof(df1)` returns "list"? See **here**.

PPS — Convert classes with `as.[class]()`. e.g. `as.matrix(df1)` makes a matrix.

Object class, type, and structure (cont.)

Of course, you can always just inspect/print an object directly in the console.

- E.g. Type `df1` and hit Enter.

```
df1
```

```
##   x y
## 1 1 3
## 2 2 4
```

The `View()` function is also very helpful. This is the same as clicking on the object in your RStudio *Environment* pane. (Try both methods now.)

- E.g. `View(df1)`.
- Why is it important to know how to inspect objects?

Object class, type, and structure (cont.)

Of course, you can always just inspect/print an object directly in the console.

- E.g. Type `df1` and hit Enter.

```
df1
```

```
##   x y
## 1 1 3
## 2 2 4
```

The `View()` function is also very helpful. This is the same as clicking on the object in your RStudio *Environment* pane. (Try both methods now.)

- E.g. `View(df1)`.
- Why is it important to know how to inspect objects?
- R is open source and you will often be working with functions that you did not write which return objects that you are unfamiliar with.

"Everything has a name"

Reserved words

We've seen that we can assign objects to different names. However, there are a number of special words that are "reserved" in R.

- These are are fundamental commands, operators and relations in base R that you cannot (re)assign, even if you wanted to.
- We already encountered examples with the logical operators.

See here for a full list, including (but not limited to):

```
if
else
while
function
for
TRUE
FALSE
NULL
Inf
NaN
NA
```

Semi-reserved words

In addition to the list of strictly reserved words, there is a class of words and strings you might call "semi-reserved".

- These are named functions or constants (e.g. `pi`) that you can re-assign if you really wanted to... but already come with important meanings from base R.

Arguably the most important semi-reserved character is `c()`, which we use for concatenation; i.e. creating vectors and binding different objects together.

```r
my_vector = c(1, 2, 5)
my_vector
```

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

Semi-reserved words

In addition to the list of strictly reserved words, there is a class of words and strings you might call "semi-reserved".

- These are named functions or constants (e.g. `pi`) that you can re-assign if you really wanted to... but already come with important meanings from base R.

Arguably the most important semi-reserved character is `c()`, which we use for concatenation; i.e. creating vectors and binding different objects together.

```
my_vector = c(1, 2, 5)
my_vector
```

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

What happens if you type the following? (Try it in your console.)

```
c = 4
c(1, 2 ,5)
```

Semi-reserved words (cont.)

(Continued from previous slide.)

In this case, thankfully nothing. R is "smart" enough to distinguish between the variable `c = 4` that we created and the built-in function `c()` that calls for concatenation.

Semi-reserved words (cont.)

(Continued from previous slide.)

In this case, thankfully nothing. R is "smart" enough to distinguish between the variable `c = 4` that we created and the built-in function `c()` that calls for concatenation.

However, this is still *extremely* sloppy coding. R won't always be able to distinguish between conflicting definitions. And neither will you. For example:

```
pi
```

```
## [1] 3.141593
```

```
pi = 2
pi
```

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

Semi-reserved words (cont.)

(Continued from previous slide.)

In this case, thankfully nothing. R is "smart" enough to distinguish between the variable `c = 4` that we created and the built-in function `c()` that calls for concatenation.

However, this is still *extremely* sloppy coding. R won't always be able to distinguish between conflicting definitions. And neither will you. For example:

```
pi
```

```
## [1] 3.141593
```

```
pi = 2
pi
```

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

Two fixes:

1. `rm(pi)`
2. Restart your RStudio session

Both a pain. **Bottom line:** Don't use (semi-)reserved characters!

Namespace conflicts

A similar issue crops up when we load two packages, which have functions that share the same name. E.g. Look what happens we load the `dplyr` package.

```r
library(dplyr)
```

Namespace conflicts

A similar issue crops up when we load two packages, which have functions that share the same name. E.g. Look what happens we load the `dplyr` package.

```r
library(dplyr)
```

The messages that you see about some object being *masked from 'package:X'* are warning you about a namespace conflict.

- E.g. Both `dplyr` and the `stats` package (which gets loaded automatically when you start R) have functions named "filter" and "lag".

Namespace conflicts (cont.)

The potential for namespace conflicts is a result of the OOP approach.[1]

- Also reflects the fundamental open-source nature of R and the use of external packages. People are free to call their functions whatever they want, so some overlap is only to be expected.

[1] Similar problems arise in virtually every other programming language (Python, C, etc.)

Namespace conflicts (cont.)

The potential for namespace conflicts is a result of the OOP approach.[1]

- Also reflects the fundamental open-source nature of R and the use of external packages. People are free to call their functions whatever they want, so some overlap is only to be expected.

Whenever a namespace conflict arises, the most recently loaded package will gain preference. So the `filter()` function now refers specifically to the `dplyr` variant.

But what if we want the `stats` variant? Well, we have two options:

1. Temporarily use `stats::filter()`
2. Permanently assign `filter = stats::filter`

[1] Similar problems arise in virtually every other programming language (Python, C, etc.)

Solving namespace conflicts

1. Use `package::function()`

We can explicitly call a conflicted function from a particular package using the `package::function()` syntax. For example:

```r
stats::filter(1:10, rep(1, 2))
```

```
## Time Series:
## Start = 1
## End = 10
## Frequency = 1
##  [1]  3  5  7  9 11 13 15 17 19 NA
```

Solving namespace conflicts

1. Use `package::function()`

We can explicitly call a conflicted function from a particular package using the `package::function()` syntax. For example:

```
stats::filter(1:10, rep(1, 2))
```

```
## Time Series:
## Start = 1
## End = 10
## Frequency = 1
##  [1]  3  5  7  9 11 13 15 17 19 NA
```

We can also use `::` for more than just conflicted cases.

- E.g. Being explicit about where a function (or dataset) comes from can help add clarity to our code. Try these lines of code in your R console.

```
dplyr::starwars ## Print the starwars data frame from the dplyr package
scales::comma(c(1000, 1000000)) ## Use the comma function, which comes from the scales pack
```

Solving namespace conflicts (cont.)

2. Assign `function = package::function`

A more permanent solution is to assign a conflicted function name to a particular package. This will hold for the remainder of your current R session, or until you change it back. E.g.

```r
filter = stats::filter ## Note the lack of parentheses.
filter = dplyr::filter ## Change it back again.
```

Solving namespace conflicts (cont.)

2. Assign `function = package::function`

A more permanent solution is to assign a conflicted function name to a particular package. This will hold for the remainder of your current R session, or until you change it back. E.g.

```
filter = stats::filter ## Note the lack of parentheses.
filter = dplyr::filter ## Change it back again.
```

General advice

I would generally advocate for the temporary `package::function()` solution.

Another good rule of thumb is that you want to load your most important packages last. (E.g. Load the tidyverse after you've already loaded any other packages.)

Other than that, simply pay attention to any warnings when loading a new package and `?` is your friend if you're ever unsure. (E.g. `?filter` will tell you which variant is being used.)

- In truth, problematic namespace conflicts are rare. But it's good to be aware of them.

User-side namespace conflicts

A final thing to say about namespace conflicts is that they don't only arise from loading packages. They can arise when users create their own functions with a conflicting name.

- E.g. If I was naive enough to create a new function called `c()`.

User-side namespace conflicts

A final thing to say about namespace conflicts is that they don't only arise from loading packages. They can arise when users create their own functions with a conflicting name.

- E.g. If I was naive enough to create a new function called `c()`.

In a similar vein, one of the most common and confusing errors that even experienced R programmers run into is related to the habit of calling objects "df" or "data"... both of which are functions in base R![1]

- See for yourself by typing `?df` or `?data`.

Again, R will figure out what you mean if you are clear/lucky enough. But, much the same as with `c()`, it's relatively easy to run into problems.

- Case in point: Triggering the infamous "object of type closure is not subsettable" error message. (See from 1:45 here.)

[1] Guess who has two thumbs and keeps making this mistake? This guy.

Indexing

Option 1: []

We've already seen an example of indexing in the form of R console output. For example:

```
1+2
```

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

The `[1]` above denotes the first (and, in this case, only) element of our output.[1] In this case, a vector of length one equal to the value "3".

Option 1: []

We've already seen an example of indexing in the form of R console output. For example:

```r
1+2
```

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

The `[1]` above denotes the first (and, in this case, only) element of our output.[1] In this case, a vector of length one equal to the value "3".

Try the following in your console to see a more explicit example of indexed output:

```r
rnorm(n = 100, mean = 0, sd = 1)
# rnorm(100) ## Would work just as well. (Why? Hint: see ?rnorm)
```

[1] Indexing in R begins at 1. Not 0 like some languages (e.g. Python, JavaScript, or my problem sets).

Option 1: [] (cont.)

More importantly, we can also use `[]` to index objects that we create in R.

```r
a = 1:10
a[4] ## Get the 4th element of object "a"
```

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

```r
a[c(4, 6)] ## Get the 4th and 6th elements
```

```
## [1] 4 6
```

It also works on larger arrays (vectors, matrices, data frames, and lists). For example:

```r
df1[1, 1] ## Show the cell corresponding to the 1st row & 1st column of the data frame.
```

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

Option 1: [] (cont.)

More importantly, we can also use `[]` to index objects that we create in R.

```
a = 1:10
a[4] ## Get the 4th element of object "a"
```

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

```
a[c(4, 6)] ## Get the 4th and 6th elements
```

```
## [1] 4 6
```

It also works on larger arrays (vectors, matrices, data frames, and lists). For example:

```
df1[1, 1] ## Show the cell corresponding to the 1st row & 1st column of the data frame.
```

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

What does `df2[1:3, 1]` give you?

Option 1: [] (cont.)

We haven't covered them yet, but **lists** are a more complex type of array object in R.

- They can contain an assortment of objects that don't share the same class, or have the same shape (e.g. rank) or common structure.
- E.g. A list can contain a scalar, a string, and a data frame. Or you can have a list of data frames, or even lists of lists.

Option 1: [] (cont.)

We haven't covered them yet, but **lists** are a more complex type of array object in R.

- They can contain an assortment of objects that don't share the same class, or have the same shape (e.g. rank) or common structure.
- E.g. A list can contain a scalar, a string, and a data frame. Or you can have a list of data frames, or even lists of lists.

The relevance to indexing is that lists require two square brackets `[[]]` to index the parent list item and then the standard `[]` within that parent item. An example might help to illustrate:

```r
my_list = list(a = "hello", b = c(1,2,3), c = data.frame(x = 1:5, y = 6:10))
my_list[[1]] ## Return the 1st list object
```

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

```r
my_list[[2]][3] ## Return the 3rd element of the 2nd list object
```

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

Option 2: $

Lists provide a nice segue to our other indexing operator: `$`.

- Let's continue with the `my_list` example from the previous slide.

```r
my_list
```

```
## $a
## [1] "hello"
##
## $b
## [1] 1 2 3
##
## $c
##   x  y
## 1 1  6
## 2 2  7
## 3 3  8
## 4 4  9
## 5 5 10
```

Option 2: $

Lists provide a nice segue to our other indexing operator: `$`.

- Let's continue with the `my_list` example from the previous slide.

```r
my_list
```

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

Notice how our (named) parent list objects are demarcated: "$a", "$b" and "$c".

Option 2: $ (cont.)

We can call these objects directly by name using the dollar sign, e.g.

```
my_list$a ## Return list object "a"
```

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

```
my_list$b[3] ## Return the 3rd element of list object "b"
```

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

```
my_list$c$x ## Return column "x" of list object "c"
```

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

Option 2: $ (cont.)

We can call these objects directly by name using the dollar sign, e.g.

```
my_list$a ## Return list object "a"
```

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

```
my_list$b[3] ## Return the 3rd element of list object "b"
```

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

```
my_list$c$x ## Return column "x" of list object "c"
```

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

Aside: Typing `View(my_list)` (or, equivalently, clicking on the object in RStudio's environment pane) provides a nice interactive window for exploring the nested structure of lists.

Option 2: $ (cont.)

The `$` form of indexing also works (and in the manner that you probably expect) for other object types in R, like `data.frame`s.

In some cases, you can also combine the two index options.

- E.g. Get the 1st element of the "name" column from the our data frame.

```
df2$x[1]
```

```
## [1] 0.1295931
```

Option 2: $ (cont.)

The `$` form of indexing also works (and in the manner that you probably expect) for other object types in R, like `data.frame`s.

In some cases, you can also combine the two index options.

- E.g. Get the 1st element of the "name" column from the our data frame.

```
df2$x[1]
```

```
## [1] 0.1295931
```

However, note some key differences between the output from this example and that of our previous `df2[1, 1]` example. What are they?

- Hint: Apart from the visual cues, try wrapping each command in `str()`.

Option 2: $ (cont.)

The last thing that I want to say about `$` is that it provides another way to avoid the "object not found" problem that we ran into with our earlier regression example.

```
lm(y ~ x) ## Doesn't work
```

```
## Error in eval(predvars, data, env): object 'y' not found
```

```
lm(df1$y ~ df1$x) ## Works!
```

```
##
## Call:
## lm(formula = df1$y ~ df1$x)
##
## Coefficients:
## (Intercept)        df1$x
##           2            1
```

Cleaning up

Removing objects (and packages)

Use `rm()` to remove an object or objects from your working environment.

```
a = "hello"
b = "world"
rm(a, b)
```

You can also use `rm(list = ls())` to remove all objects in your working environment (except packages), but this is frowned upon.

- Better just to start a new R session.

Removing objects (and packages)

Use `rm()` to remove an object or objects from your working environment.

```r
a = "hello"
b = "world"
rm(a, b)
```

You can also use `rm(list = ls())` to remove all objects in your working environment (except packages), but this is frowned upon.

- Better just to start a new R session.

Detaching packages is more complicated, because there are so many cross-dependencies (i.e. one package depends on, and might even automatically load, another.) However, you can try, e.g. `detach(package:dplyr)`

- Again, better just to restart your R session.

Removing plots

You can use `dev.off()` to removing any (i.e. all) plots that have been generated during your session. For example, try this in your R console:

```r
plot(1:10)
dev.off()
```

Removing plots

You can use `dev.off()` to removing any (i.e. all) plots that have been generated during your session. For example, try this in your R console:

```
plot(1:10)
dev.off()
```

You may also have noticed that RStudio has convenient buttons for clearing your workspace environment and removing (individual) plots. Just look for these icons in the relevant window panels: