Big Data and Economics

Lecture 4: R language basics

Grant McDermott, adapted by Kyle Coombs
Bates College | EC/DCS 368

Table of contents

1. Prologue
2. Introduction
3. Object-oriented programming in R
4. "Everything is an object"
5. "Everything has a name"
6. Indexing[1]
7. Cleaning up[1]

[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

Housekeeping

- Full version of these slides available on Grant McDermott's website
- Project proposals looked good:
  - Note: Some of these papers have proprietary data components, I need you to confirm that you can replicate sufficient parts of their analysis
  - Note 2: Some of you picked experimental data (cool), I want you to spend some time stress-testing to confirm the data are "real"
  - Unless I *specifically* say otherwise, you should not upload to Lyceum
- Please navigate to these slides on your desktop to follow along
- Try the code interactively on your laptop (with a Codespace or by forking/cloning the repository to your computer)

Checklist

☑ Fork repo class material if you haven't done so

☑ Clone repo if you haven't done so

☑ Pull from the lecture repo to get the latest slides.

☑ Update your R packages. Do this regularly as a matter of good habit. (`update.packages()` should do the trick, or click `Update` under the Packages tab (lower right-hand side))

Agenda

- We're going to spent a lot of time live coding together. This is deliberate.
- I want you to type 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 class, 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

(Some important R concepts)

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.

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

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

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

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

Logic

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

```
1 > 2
```

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

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

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

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

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

```
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:

```
1 > 2
```

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

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

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

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

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

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

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

You can read more about logical operators and types here and here. In the next few slides, however, I want to emphasise some special concepts and gotchas...

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
```

Much like standard arithmetic, 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?

```
1 > 0.5 & 2
```

Much like standard arithmetic, 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
```

Much like standard arithmetic, 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] .

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

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

```
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
```

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:

```
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.)

```
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"
```

Assignment

In R, we can use either `<-` or `=` to handle assignment.[1]

Assignment

In R, we can use either `←` or `=` to handle assignment.[1]

Assignment with `←`

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

```
a ← 10 + 5
a
```

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

Assignment

In R, we can use either `←` or `=` to handle assignment.[1]

Assignment with `←`

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

```
a ← 10 + 5
a
```

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

[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.

Assignment (cont.)

Assignment with `=`

You can also use `=` for assignment.

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

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

Assignment (cont.)

Assignment with `=`

You can also use `=` for assignment.

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

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

Which assignment operator to use?

Most R users (purists?) seem to prefer `←` for assignment, since `=` also has specific role for evaluation *within* functions.

- We'll see lots of examples of this later.
- But I don't think it matters; `=` is quicker to type and is more intuitive if you're coming from another programming language. (More discussion here and here.)

Bottom line: Pick one and be consistent.

Challenge: Detect government payments

Imagine that you have a large bank transaction dataset. You know government payments are only divisible by $100, but never divisible by $4 or $400.

Write code to check if a transaction is possibly a government payment.

- Implement the modulo operator `%`
- Employ the corder order of logical operators (`&`, `|`)
- Utilize any arithmetic operations you need

```
transaction ← 9400
# Your code here
```

- I had to do a slightly more complicated verison of this for my dissertation.

Help

For more information on a (named) function or object in R, consult the "help" documentation. For example:

```
help(plot)
```

Or, more simply, just use `?`:

```
?plot # This is what most people use.
```

Help

For more information on a (named) function or object in R, consult the "help" documentation. For example:

```
help(plot)
```

Or, more simply, just use `?`:

```
?plot # This is what most people use.
```

Aside 1: Comments in R 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.

Help

For more information on a (named) function or object in R, consult the "help" documentation. For example:

```
help(plot)
```

Or, more simply, just use `?`:

```
?plot # This is what most people use.
```

Aside 1: Comments in R 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.

Aside 2: See the *Examples* section at the bottom of the help file?

- You can run them with the `example()` function. Try it: `example(plot)`.

Help (cont.)

Vignettes

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

- Try running `vignette("dplyr")` in your console now.

Help (cont.)

Vignettes

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

- Try running `vignette("dplyr")` in your console now.

I highly encourage reading package vignettes if they are available.

- They are often the best way to learn how to use a package.

Help (cont.)

Vignettes

For many packages, you can also try the `vignette()` function, which will provide an introduction to a package and it's purpose through a series of helpful examples.

- Try running `vignette("dplyr")` in your console now.

I highly encourage reading package vignettes if they are available.

- They are often the best way to learn how to use a package.

One complication is that you need to know the exact name of the package vignette(s).

- E.g. The `dplyr` package actually has several vignettes associated with it: "dplyr", "window-functions", "programming", etc.
- You can run `vignette()` (i.e. without any arguments) to list the available vignettes of every *installed* package installed on your system.
- Or, run `vignette(all = FALSE)` if you only want to see the vignettes of any *loaded* packages.

Help (cont.)

Demos

Similar to vignettes, many packages come with built-in, interactive demos.

To list all available demos on your system:[1]

```
demo(package = .packages(all.available = TRUE))
```

[1] How would you limit the demos to one particular package?

Help (cont.)

Demos

Similar to vignettes, many packages come with built-in, interactive demos.

To list all available demos on your system:[1]

```
demo(package = .packages(all.available = TRUE))
```

To run a specific demo, just tell R which one and the name of the parent package. For example:

```
demo("graphics", package = "graphics")
```

[1] How would you limit the demos to one particular package?

Object-oriented programming in R

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.

"Everything is an object"

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

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

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

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

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

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

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

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

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.

Global environment

Let's go back to the simple data frame that we created a few slides earlier.

```
df1
```

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

Global environment

Let's go back to the simple data frame that we created a few slides earlier.

```
df1
```

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

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 simple data frame that we created a few slides earlier.

```
df1
```

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

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:

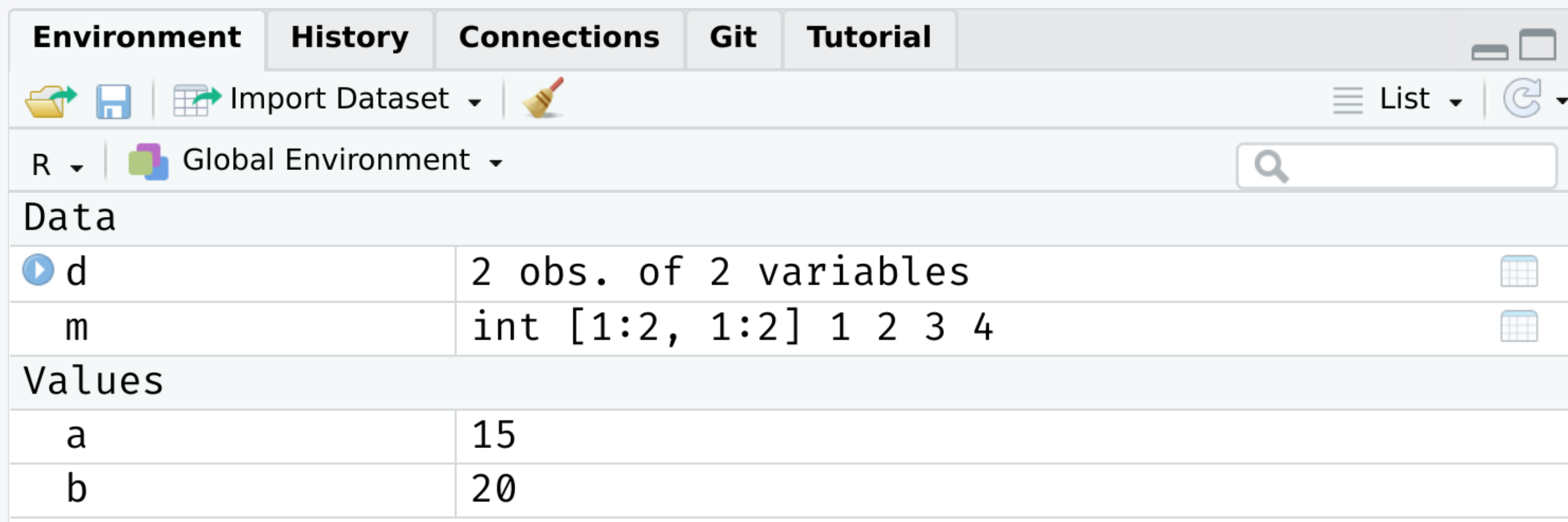
```
## Error in eval(predvars, data, env): object 'y' not found
```

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:

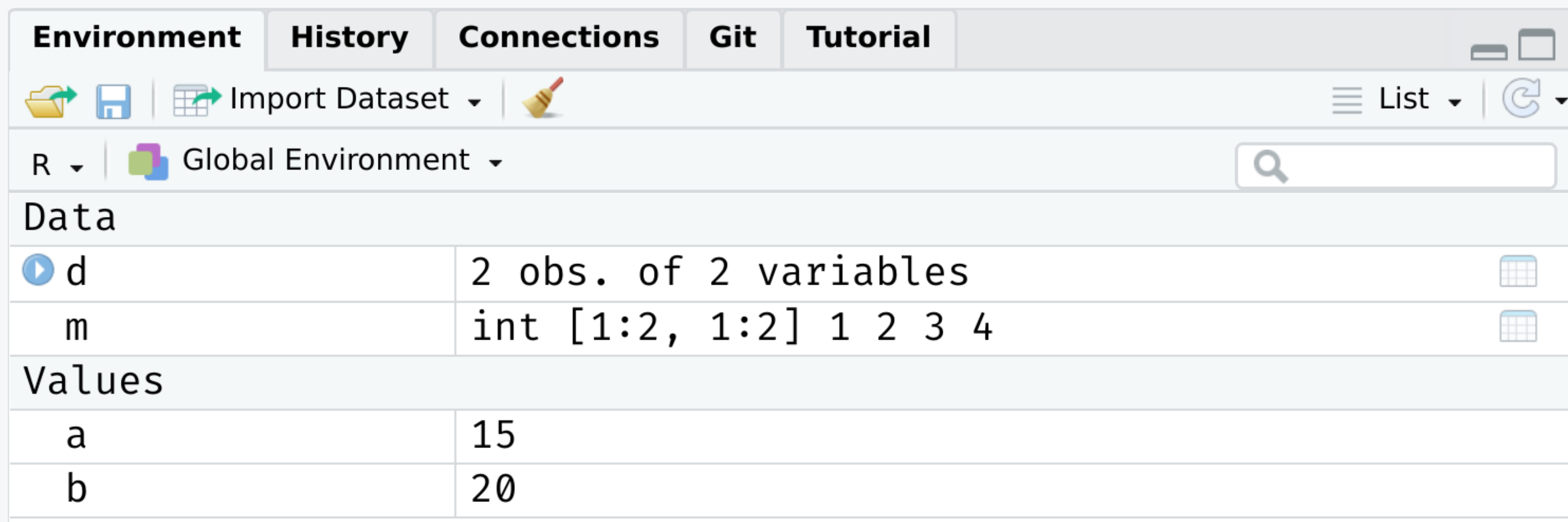


Global environment (cont.)

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: Because the variables "x" and "y" live as separate objects in the global environment, 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
##           2            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
##           2            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.

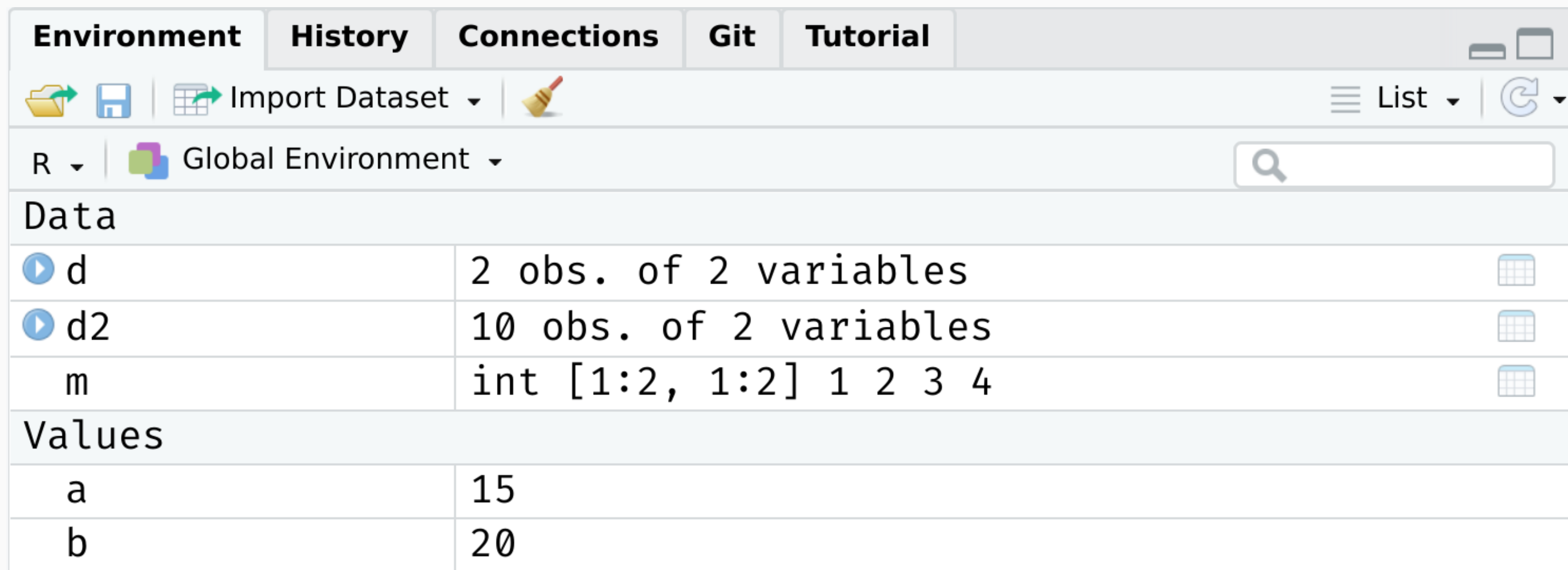
- In Stata, the entire workspace essentially consists of one (and only one) data frame meaning no ambiguity where variables are coming from.
- That "convenience" has a high price -- literally you need to buy Stata 16 or higher to use `frames` to open multiple data frames with less flexibility.
- Speaking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

Working with multiple objects

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

- E.g. We can copy an exiting data frame, or create new one entirely from scratch. Either will exist happily with our existing objects in the global environment.

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



Working with multiple objects (cont.)

Again, however, it does mean that you have to pay attention to the names of those distinct data frames and be specific about which objects you are referring to.

- Do we want to run a regression of "y" on "x" from data frame `df1` or data frame `df2`?

"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.

```
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

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.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

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.

```
library(dplyr)
```

```
##
## Attaching package: 'dplyr'

## The following objects are masked from 'package:stats':
##
##     filter, lag

## The following objects are masked from 'package:base':
##
##     intersect, setdiff, setequal, union
```

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:

```
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 scale
```

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.
```

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:

```
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:

```
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.

```
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
```

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:

```
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"
```

```
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.

```
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.

```
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
```

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] -0.8884779
```

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] -0.8884779
```

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.

```
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:

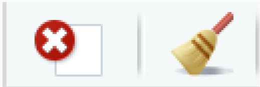
```
plot(1:10)
dev.off()
```

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:



Next lecture(s): Data wrangling and cleaning

Appendix

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