Data Science for Economists

Lecture 3: Objects and the Environment

Alex Marsh
University of North Carolina | ECON 390

Table of contents

1. Introduction

2. Object-oriented programming in R

3. "Everything is an object"

4. "Everything has a name"

5. Indexing

6. Cleaning up

✱ Slides adapted from Grant McDermott's EC 607 at University of Oregon.

Introduction

(Some important R concepts)

Agenda

Today will be more high level than the last few lectures.

- While there will be some coding, this will mostly be solidifying and clarifying certain concepts.

I want everyone to feel comfortable in RStudio and to understand how to make scripts and RMarkdown documents.

- I want you to get comfortable typing R commands yourself — and navigating the RStudio IDE — without resorting to copy+paste.
- Slightly more painful in the beginning, but much better payoff in the long-run.

Object-oriented programming in R

Motivation

In our very first lecture, I mentioned R's approach to object-oriented programming (OOP), which is often summarized as:

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

Motivation

In our very first lecture, I mentioned R's approach to object-oriented programming (OOP), which is often summarized 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.

Motivation

In our very first lecture, I mentioned R's approach to object-oriented programming (OOP), which is often summarized 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.

The good news, as well see, 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.

Disclaimer

Okay, this slide is just to let you know that I'm being a little fast and loose with terms.

Most obviously, 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.

But for our purposes, I think it is much more helpful to think about (a) the shared characteristics of these different systems and (b) the broad implications of OOP in R.

- What we lose in detail, we hopefully gain in perspective.
- But do read Hadley's book if you get the chance. It's incredibly helpful (as are all his books).

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

What are objects? (cont.)

Each object class has its own set of rules ("methods") for determining valid operations.

- For example, you can perform many of the same operations on matrices and data frames. But there are some operations that only work on a matrix, and vice versa.
- At the same time, you can (usually) convert an object from one type to another.

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

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

```
## Convert it to (i.e. create) a matrix call "m".
m = as.matrix(d)
m
```

```
##      x y
## [1,] 1 3
## [2,] 2 4
```

Object class, type, and structure

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

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

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

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

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

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

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

Object class, type, and structure

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

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

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

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

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

```
str(d) ## Show its structure.
```

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

PS — Confused by the fact that `typeof(d)` returns "list"? See here.

Object class, type, and structure (cont.)

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

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

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(d)`.

Global environment

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

```
d
```

```
##   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.

```
d
```

```
##   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.

```
d
```

```
##   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:

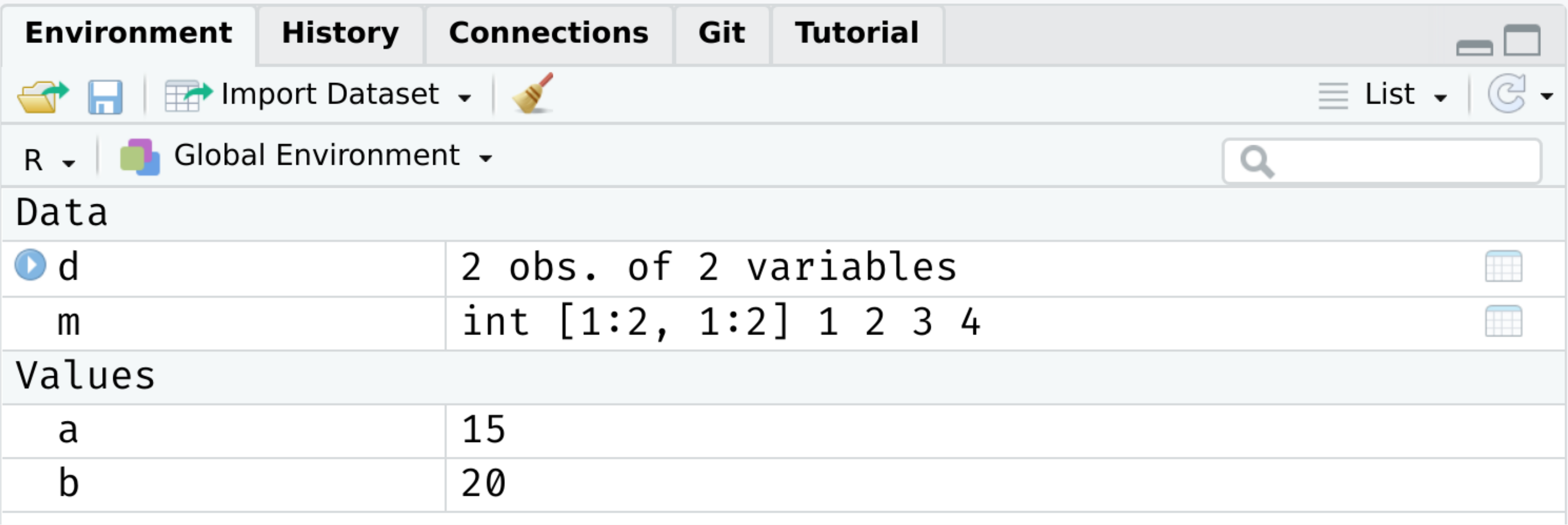
```
## Error in eval(predvars, data, env): object 'y' not found
```

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:



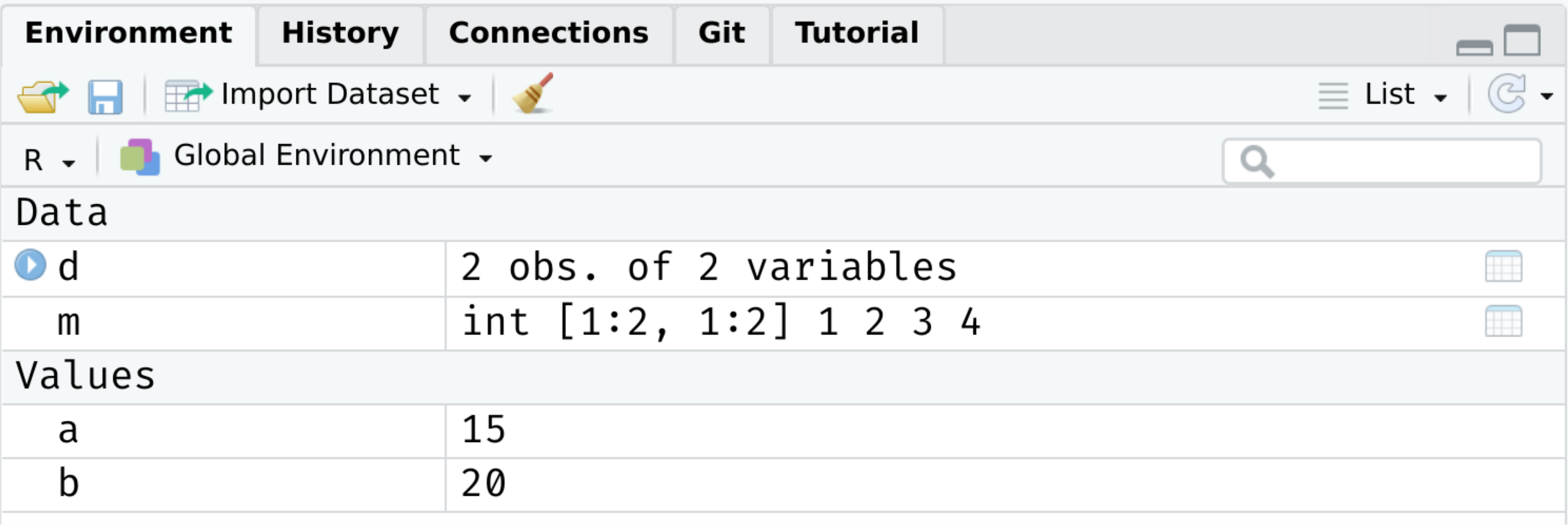
```
Data
d      2 obs. of 2 variables
m      int [1:2, 1:2] 1 2 3 4
Values
a      15
b      20
```

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:



Data	
d	2 obs. of 2 variables
m	int [1:2, 1:2] 1 2 3 4
Values	
a	15
b	20

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 `d`.

- 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 = d) ## Works when we add "data = d"!
```

```
##
## Call:
## lm(formula = y ~ x, data = d)
##
## 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 = d) ## Works when we add "data = d"!
```

```
##
## Call:
## lm(formula = y ~ x, data = d)
##
## Coefficients:
## (Intercept)            x
##           2            1
```

I wanted 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.

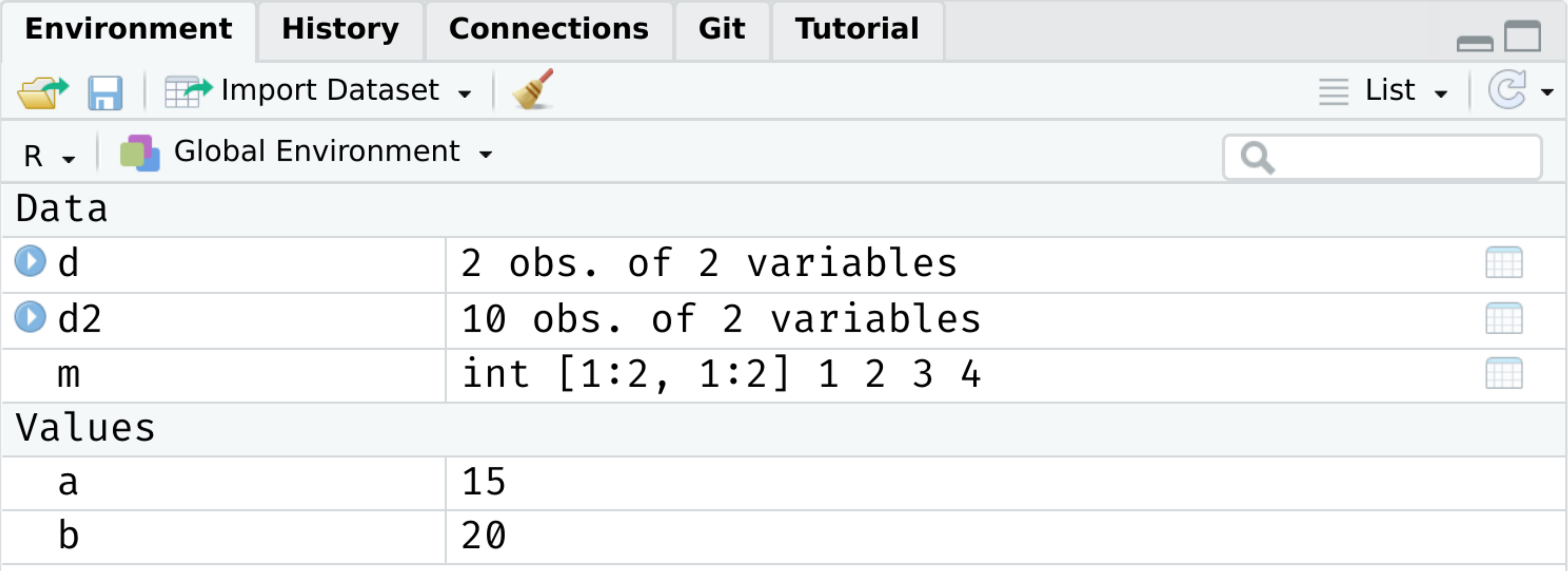
- In Stata, the entire workspace essentially consists of one (and only one) data frame. So there can be no ambiguity where variables are coming from.
- However, that "convenience" comes at a really high price IMO. You can never read more than two separate datasets (let alone object types) into memory at the same time, have to resort all sorts of hacks to add summary variables to your dataset, etc.
- Speaking of which...

Working with multiple objects

As I keep saying, 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.

```r
d2 = data.frame(x = rnorm(10), y = runif(10))
```



Data	
d	2 obs. of 2 variables
d2	10 obs. of 2 variables
m	int [1:2, 1:2] 1 2 3 4
Values	
a	15
b	20

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 `d` or data frame `d2`?

"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 that I am going to 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 that I am going to 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:

```r
pi
```

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

```r
pi = 2
pi
```

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

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)
```

```
## 
## 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:

```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:

```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
```

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.

```r
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
```

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.

```r
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!

- 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.)

Indexing

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".

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 and JavaScript).

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:

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

```
## # A tibble: 1 x 1
##   name
##   <chr>
## 1 Luke Skywalker
```

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:

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

```
## # A tibble: 1 x 1
##   name
##   <chr>
## 1 Luke Skywalker
```

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

Option 1: [] (cont.)

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

- They can contain a random 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 properly yet (patience), but **lists** are a more complex type of array object in R.

- They can contain a random 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.

```
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.

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

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

```r
starwars$name[1]
```

```
## [1] "Luke Skywalker"
```

Option 2: $ (cont.)

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

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

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

```
starwars$name[1]
```

```
## [1] "Luke Skywalker"
```

However, note some key differences between the output from this example and that of our previous `starwars[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(d$y ~ d$x) ## Works!
```

```
## 
## Call:
## lm(formula = d$y ~ d$x)
## 
## Coefficients:
## (Intercept)          d$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:

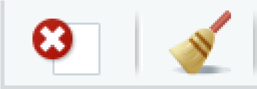
```
plot(1:10)
dev.off()
```

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:

```r
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:



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

Next lecture(s): Logic and loops