

# Basic base R

## Week 2

AEM 2850: R for Business Analytics  
Cornell Dyson  
Spring 2022

Acknowledgements: Grant McDermott

# Announcements

If you are new to the class, please review Hui's canvas announcement to help get up to speed

Reminders:

- New course content will be posted regularly at [aem2850.toddgerarden.com](http://aem2850.toddgerarden.com)
- Bookmark [the schedule](#) and visit often
- Submit assignments via canvas
  - Lab 1 was due yesterday (Monday) at 11:59pm
  - Reflection - Week 2 is due Wednesday at 11:59pm
  - No penalties for late submissions this week

We will have an undergraduate TA, Sophie McComb ([sm2397@cornell.edu](mailto:sm2397@cornell.edu))

- Poll: what times do you prefer for additional office hours?

# Plan for today

The plan for this week is to introduce ourselves to **base** R

Prologue: where we're from

Introduction to base R

Object-oriented programming in R

"Everything is an object"

"Everything has a name"

Indexing

Cleaning up

# Prologue

# How we feel about the course, in a few words

| excited

| curious

| the vibes of this class are immaculate!

| nervous

# Where we're from

How can we use your 36 PDF submissions to construct our origin story without having to read them one by one?

- **Caveat:** This is based on submissions as of 10:07pm Monday night

We can use R for this!

I won't go through details now, but here is one way to approach it:

- convert PDFs to plain text using `pdfutils::pdf_text`
- use tools from `readr` and `stringr` packages to clean up text and extract relevant bits
- use `dplyr` functions to manipulate data
- make maps using `maps` and `ggplot2::ggplot`

By the end of the course you will be able to do most, if not all, of this

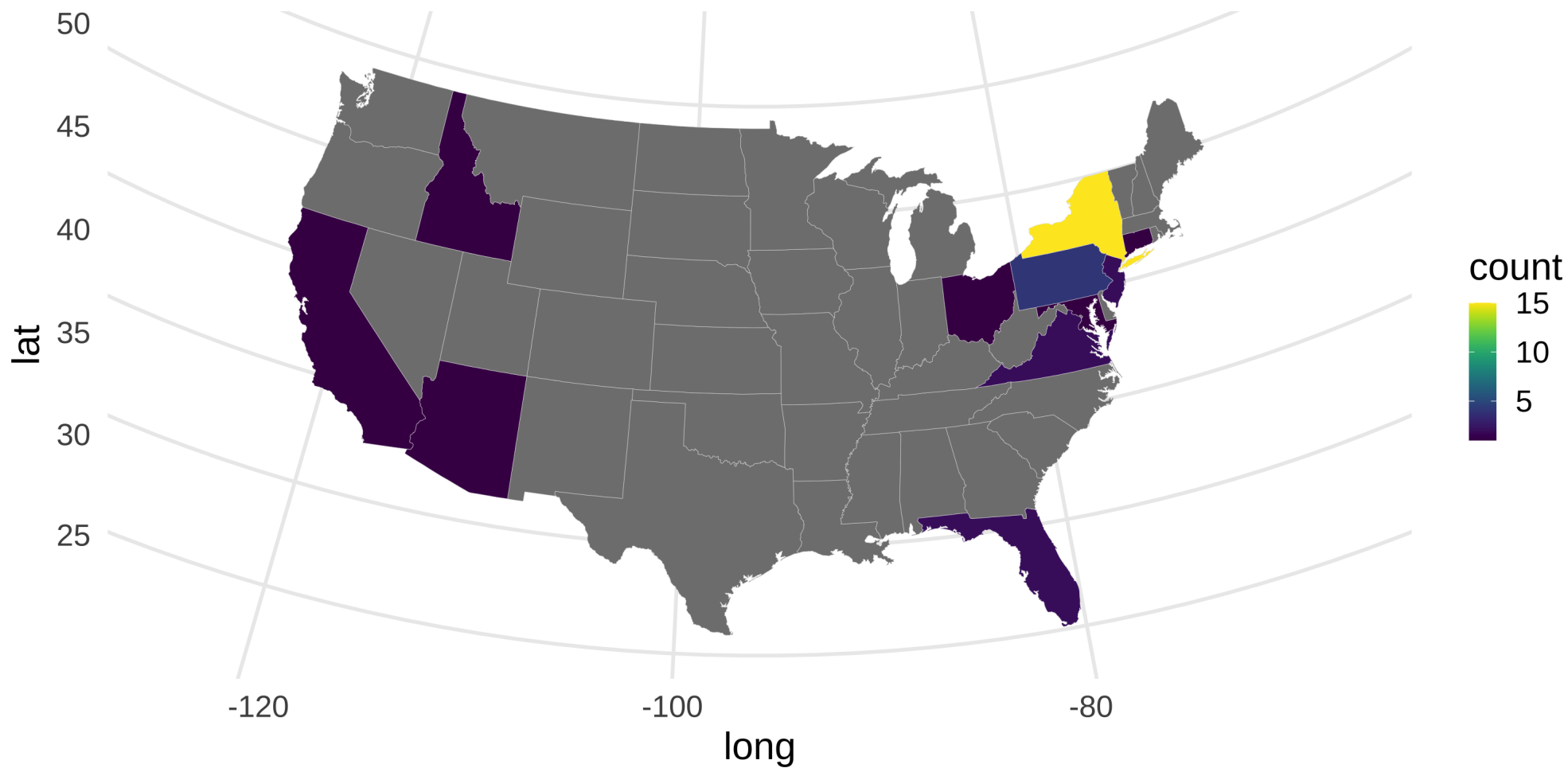
# Are we all from New York?

Are more or less than half of us from the state of New York?

```
# step 1: do a bunch of data prep that's omitted for clarity  
# step 2: show us the answer!  
ny_share
```

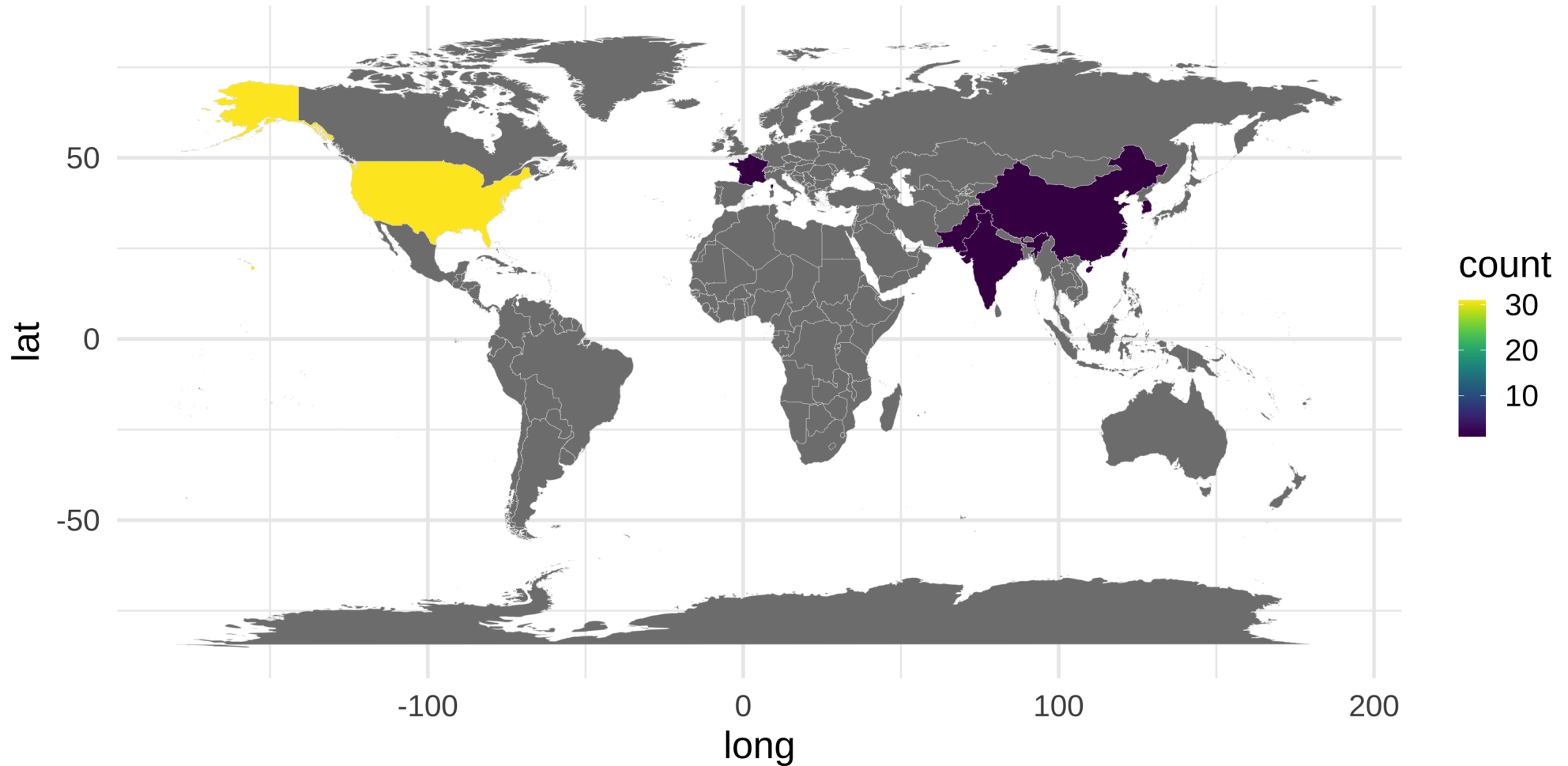
```
## # A tibble: 2 × 2  
##   origin    percent  
##   <chr>      <dbl>  
## 1 new york    40.5  
## 2 other      59.5
```

# Are we all from New York?





# Are we all from the U.S.?



# Introduction to base R

(Some of this is review from Week 1)

# Basic arithmetic

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

```
1+2 # add / subtraction
```

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

```
5/2 # divide
```

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

```
2^3 # exponentiate
```

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

```
2+4*1^3 # standard order of precedence (`*` before `+`, etc.)
```

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

# Logic

R also comes equipped with a full set of logical operators and Booleans

```
1 > 2
```

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

```
(1 > 2) & (1 > 0.5) # "&" is the "and" operator
```

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

```
(1 > 2) | (1 > 0.5) # "|" is the "or" operator
```

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

# Logic

We can negate expressions with: **!**

This is helpful for filtering data

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

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

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

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

NA means **not available** (i.e., missing)

# Logic

For value matching we can use: `%in%`

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

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

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

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

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

# Logic

To evaluate whether two expressions are equal, we need to use **two** equal signs

```
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
```

# Logic

**Evaluation caveat:** What do you think will happen if we evaluate `0.1 + 0.2 == 0.3`?

```
0.1 + 0.2 == 0.3
```

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

Uh-oh! What went wrong here?

**Problem:** Computers represent numbers as binary (i.e., base 2) floating-points. More [here](#)

- Fast and memory efficient, but can lead to unexpected behavior
- Similar to how standard decimals can't precisely capture certain fractions (e.g.,  $\frac{1}{3} = 0.3333\dots$ )

**Solution:** Use `all.equal()` for evaluating floats (i.e., fractions)

```
all.equal(0.1 + 0.2, 0.3)
```

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



# Assignment

In R, we can use either `<-` or `=` to handle assignment

## Assignment with `<-`

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

```
a <- 10 + 5  
a
```

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

# Assignment

## Assignment with =

You can also use = for assignment.

```
b = 10 + 10  
b
```

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

## Which assignment operator should you use?

Most R users prefer `<-`, which can be inserted using the keyboard shortcut Alt/Option + -

It doesn't really matter for our purposes, other languages use =

**Bottom line:** Use whichever you prefer, just be consistent

# Help

For more information on a (named) function or object in R, consult the "help" documentation using ?

For example:

```
?plot
```

# Vignettes

For some packages, `vignette()` will provide a detailed intro

```
# Try this:  
vignette("dplyr")
```

Vignettes are a great way to learn how and when to use a package

# Comments

Comments in R code are demarcated by `#`

Use comments to document your logic in `.R` scripts and within `.Rmd` code chunks

```
# THIS IS A CODE SECTION ----  
# this is a comment  
vibe <- "immaculate" # iykyk
```

Comments should be concise and used only when necessary (unlike the comments above)

Using at least four trailing dashes (`----`) creates a code section, which simplifies navigation and code folding

- Also works with trailing equals (`====`) or pound signs (`=====`)

**Keyboard shortcut:** use `Ctrl/Cmd+Shift+c` in RStudio to (un)comment whole sections of highlighted code.

# Object-oriented programming in R

# Object-oriented programming

In R:

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

**"Everything is an object"**



# What are objects?

There are many different *types* (or *classes*) of objects

Here are some objects that we'll be working with regularly:

- vectors
- matrices
- data frames
- lists
- functions

# Data frames

The most important object we will be working with is the **data frame**

You can think of it basically as an Excel spreadsheet

```
# 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
```

This is essentially just a table with columns named **x** and **y**

Each row is an observation telling us the values of both **x** and **y**

# Aside: built-in data frames

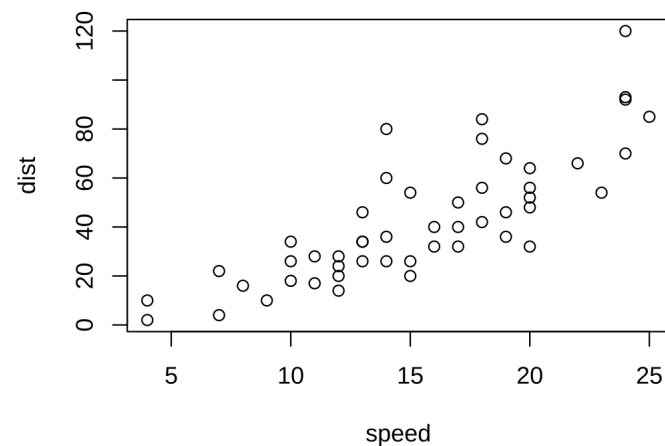
Base R and packages have a bunch of built in data frames with special names you can call on

As you may recall, we used `cars` on Thursday:

```
head(cars)
```

```
##      speed  dist
## 1         4     2
## 2         4    10
## 3         7     4
## 4         7    22
## 5         8    16
## 6         9    10
```

```
plot(cars)
```



# Back to objects

Each object class has its own set of rules for determining valid operations

```
# 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
```

At the same time, you can (usually) convert an object from one type to another

```
# Convert it to (i.e., create) a matrix call "mat"  
mat <- as.matrix(d)  
mat
```

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

# Working with multiple objects

In R we can have multiple data frames in memory at once

Even though we just made `mat`, `d` still exists:

```
d
```

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

# Ways to learn about objects

Printing an object directly in the console is often handy

- e.g., Type `d` and hit Enter

`View()` is very helpful, and has the same effect as clicking on the object in your RStudio *Environment* pane

- e.g., Type `View(d)` and hit Enter

Use the `str` command to learn about an object's **structure**

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

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

You can also use `class` to get an object's class without all the other details

# 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 do a logical comparison of these "x" and "y" variables:

```
x < y
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

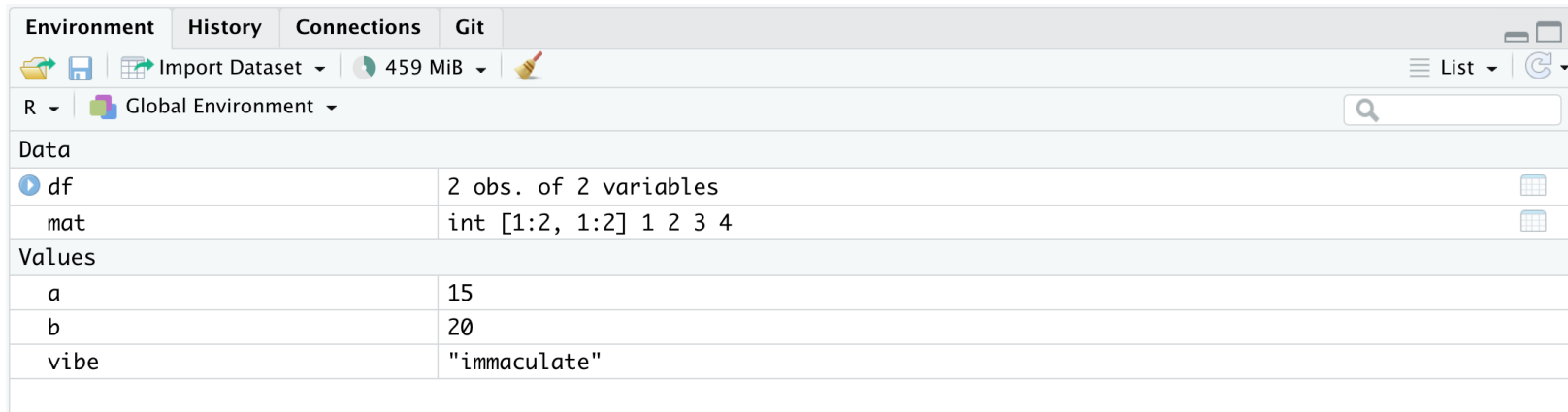
Uh-oh. What went wrong here?

# Global environment

The error message provides the answer to our question:

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

R looked in our *Global Environment* and couldn't find **x**



The screenshot shows the RStudio Global Environment pane. At the top, there are tabs for Environment, History, Connections, and Git. Below the tabs, there are icons for Import Dataset, a memory usage indicator (459 MiB), and a search icon. The main area displays the Global Environment with a search bar. Under the 'Data' section, there are two rows: 'df' with '2 obs. of 2 variables' and 'mat' with 'int [1:2, 1:2] 1 2 3 4'. Under the 'Values' section, there are three rows: 'a' with '15', 'b' with '20', and 'vibe' with '"immaculate"'. Each row has a small icon on the right side.

Global Environment	
Data	
df	2 obs. of 2 variables
mat	int [1:2, 1:2] 1 2 3 4
Values	
a	15
b	20
vibe	"immaculate"

We have to tell R that **x** and **y** belong to the object **d**

We will come back to this



**"Everything has a name"**

# Reserved words

R has a bunch of key/reserved words that serve specific functions

- You can't (re)assign these, even if you wanted to

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

```
if
else
while # looping
function
for # looping
TRUE
FALSE
NULL # null/undefined
Inf #infinity
NaN # not a number
NA # not available / missing
```

# Semi-reserved words

There are other words that are sort of reserved, in that they have a particular meaning

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

The most important example is `c()`, which binds and concatenates objects together

```
my_vector <- c(1, 2, 5)
my_vector
```

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

What do you think will happen if you type the following?

```
c <- 4
c(1, 2, 5)
```

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

In this case, R is "smart" enough to distinguish between the variable `c` and the built-in function `c()`

# Semi-reserved words (cont.)

But R won't always be able to distinguish between conflicting definitions! For example:

```
pi
```

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

```
pi <- 2  
pi
```

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

**Bottom line:** Don't use (semi-)reserved words!

# Namespace conflicts

Try loading the `dplyr` package in RStudio

```
library(dplyr)
```

What warning gets reported?

The following objects are masked from ‘package:stats’:

`filter`, `lag`

The following objects are masked from ‘package:base’:

`intersect`, `setdiff`, `setequal`, `union`

The warning *masked from 'package:X'* is about a **namespace conflict**

Here, both `dplyr` and the `stats` package (which gets loaded automatically when you start R) have functions named `filter` and `lag`

# Namespace conflicts

Whenever a namespace conflict arises, the most recently loaded package will gain preference

The `filter()` function now refers specifically to the `dplyr` variant

What if we want the `stats` variant?

1. Use `stats::filter()`
2. Assign `filter <- stats::filter`

# Solving namespace conflicts

## 1. Use `package::function()`

Explicitly call a conflicted function from a package using the `package::function()` syntax

We can also use `::` to clarify the source of a function or dataset in our code

```
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 package
```

The `::` syntax also allows us to call functions without loading the package (as long as it is installed)

# Solving namespace conflicts

## 2. Assign `function <- package::function`

A more permanent option is to assign a conflicted function name to a particular package

```
filter <- stats::filter # Note the lack of parentheses  
filter <- dplyr::filter # Change it back again
```



# User-side namespace conflicts

Namespace conflicts don't just arise from loading packages

Users like you and me can (and probably will!) create them through assignment

# Indexing

# Indexing

How do we index in R?

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

```
1+2
```

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

The `[1]` above denotes the first (and, in this case, only) element of our output

In this case, a vector of length one equal to the value "3"

# Indexing

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

```
rnorm(n = 50, mean = 0, sd = 1) # take 50 draws from the standard normal distribution
```

```
## [1] -2.24051258  0.87621667  0.02023101  0.07889824  0.67678710  1.10088875
## [7]  0.43904763  0.72103766 -1.20186185  0.95402445 -1.11378979  0.89866371
## [13] -1.53267418  0.32063190  0.23561080  0.76263056 -0.68174290 -0.26372295
## [19] -1.61103024 -0.99382381  0.50202583  0.54782405 -0.80941434  0.42823947
## [25] -1.03933217  0.17573398  0.38370213  0.05990615  0.90945020 -1.27156194
## [31]  0.09621791 -0.12100332  1.93847505  0.66398124 -0.16670568 -0.25605654
## [37] -1.52465957  0.52227681 -1.46560994  0.20889804  1.30934496 -1.88967994
## [43] -1.27385163  1.56537390 -0.24523061 -0.98211091  1.92821719  0.10089220
## [49] -0.18325608  0.27410159
```

# Option 1: []

We can 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
```

# Option 1: []

This also works on larger arrays (vectors, matrices, data frames, and lists)

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

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

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

```
## # A tibble: 3 × 1  
##   name  
##   <chr>  
## 1 Luke Skywalker  
## 2 C-3PO  
## 3 R2-D2
```

# Option 1: []

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

They can contain a collection of objects that don't share the same structure

For example, you can have lists containing:

- a scalar, a string, and a data frame
- a list of data frames
- a list of lists

# Option 1: []

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
```

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

# Option 2: \$

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: \$

The `$` form of indexing also works for other object types

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

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

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

# Option 2: \$

Finally, `$` provides another way to avoid the "object not found" problem that we ran into earlier

```
x < y # Doesn't work
```

```
## Error in eval(expr, envir, enclos): object 'x' not found
```

```
d$x < d$y # Works!
```

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

Cleaning up

# Removing objects

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

```
a <- "hello"  
b <- "world"  
rm(a, b)
```

You can use `rm(list = ls())` to remove all objects in your working environment, though this is frowned upon

- Better just to start a new R session