Introduction to R

Jared Berry

September 12, 2019

My goals

To convince you that:

- R is widely used for a reason and a powerful asset to you in this field
- R is incredibly flexible and not so difficult to learn
- You can do just about anything in R that you can in other programming languages/statistical software (with some limitations)
- For early-stage data cleaning/manipulation, R is incredibly useful
- It can be a powerful tool for automating mundane tasks
- R is worth continuing to learn after we're done here

Some caveats

- Time
- Learning curve/limitations of teaching programming in a classroom setting

Tons of resources for continuing to master this after we're done here

(Rough) Plan for this course

Today

- Learn what R is and the basics/idiosyncrasies of the language
- Become familiar with the primary data types (objects) and how to work with them
- Using libraries/packages
- The basics of data I/O, plotting, and modeling

Beyond

- Conditionals and control flow
- The basics of loops and user-defined functions
- Some useful functions/tips/tricks for automation
- Data cleaning and manipulation
- `dplyr` (and other bits from the Tidyverse)
- Some more advanced plotting in `ggplot`
- Working through an extended example

If I'm going too slow or too fast, tell me

What is R?

R is an Open Source Statistical Package

- As an open source package, code is free for anyone to view, use, or modify
- Since no one owns it, no one can profit from it
- Means that, since there is a communal effort and communal ownership, it's free
- Designed by statisticians, for statisticians

R (along with Python) is fairly ubiquitous in this field

- Moving toward the industry standard for this work
- Incredibly flexible
- Extensive documentation
- A vibrant online community for support and resources
  - stack overflow
  - CRAN

RStudio

RStudio is an integrated development environment (IDE) that builds on base R

- More intuitive/user-friendly than base R
- Makes it much easier to see what you're doing
- Has support for Git, R Markdown, local job management, and more

Your window

- Script (upper-left)
- Console (bottom-left)
- Data overview/Environment (top-right)
- Multifunction (bottom-right)

R Basics

R can function, at its most basic level, as a calculator

- Simple arithmetic operations (e.g. $+$, -, *, $/$, ^,.)

```
3+5
```

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

- Somewhat more advanced operations (e.g. logs, trigonometric operations,.)

```
log(27)
```

```
## [1] 3.295837
```

R Basics

Variable assignment with either <- or =

```r
x <- 4*8
x
```

```
## [1] 32
```

Some notes on style

- Spacing
  - Leave space between operands
- Naming variables
  - Using names like "data1", "data2", "myData", "dframe", "df", etc. is bad and only bad people do it
- Commenting
  - Well commented code can save you literally hours of work

Data types in R

R is *vectorized* and, loosely speaking *functional*

- Much like MATLAB, R is a vectorized language, which adds a tremendous amount of power
- We can think of everything in terms of vectors and matrices
    - No scalars!
    - Operations are vectorized as well

Most common data types (or objects) include:

- Vectors
- Matrices
- Data Frames
- Lists

We operate primarily by applying functions to objects that achieve a specific outcome, rather than relying on the attributes and methods those objects 'have'

Vectors in R

Most everything in R is built from vectors

```r
# Create a vector with 'c'
x <- c(1,2,3)
x
```

```
## [1] 1 2 3
```

```r
typeof(x)
```

```
## [1] "double"
```

```r
length(x)
```

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

```r
str(x)
```

```
##  num [1:3] 1 2 3
```

Vectors in R

Typical flavors are numeric, integer, character, logical, date, and factor (there are many, many more)

- Vectors are flat: check length with `length` and type with `typeof` (or `class`)
- Check types with `is.numeric`, `is.character`, `is.logical`, etc.
- Coerce types with `as.numeric`, `as.character`, `as.Date` etc.

```
# Create a vector with 'c'
y <- c("one", "two", "three", "4")
is.character(y)
```

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

```
as.numeric(y)
```

```
## Warning: NAs introduced by coercion
```

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

Vectors in R

Operations are vectorized and elementwise (unless specified)

```
#Vectorized operations
x <- c(3:5) # Note this is the same as c(3,4,5)
y <- seq(from=2,to=6,by=2) #Note this is the same as seq(2,6,2)

x / 2
```

```
## [1] 1.5 2.0 2.5
```

```
sqrt(x)
```

```
## [1] 1.732051 2.000000 2.236068
```

```
x - y
```

```
## [1]  1  0 -1
```

Vectors in R

To subset a vector we use [] notation, and specify an index

```r
x <- c(1,10,8,5,2)
x[1]
```

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

```r
x[3]
```

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

Matrices in R

The most common structure in R (the dataframe) is an extension of the matrix

```r
# Assign a rote matrix in R
z <- matrix(1:9,3,3)
dim(z)
```

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

```r
z
```

```
##      [,1] [,2] [,3]
## [1,]    1    4    7
## [2,]    2    5    8
## [3,]    3    6    9
```

Matrices in R

Again, R uses `[]` notation

- *i*,*j* matrix notation
- Empty rows (or columns) return all
  - `X[2,]`; `X[,3]`; `X[2,3]`
- `c()` can be used to select multiple rows or columns
  - `X[c(1,2,3,4,5),]`
  - `X[,c(1:3)]`
- Always: `X[selection of rows, selection of columns]`

Matrices in R

```r
z <- matrix(seq(1,12),3,4)
z[2:3, 3:4]
```

```
##      [,1] [,2]
## [1,]    8   11
## [2,]    9   12
```

```r
z[,2:3]
```

```
##      [,1] [,2]
## [1,]    4    7
## [2,]    5    8
## [3,]    6    9
```

```r
z[,1]
```

```
## [1] 1 2 3
```

Dataframes in R

Dataframes are the most commonly used data object in R

- Essentially a souped-up matrix with i,j notation
- Each column is one type of data (i.e. character, numeric, date, logical, etc.)
- Use [] notation to index in, can use column names or integer indexes
- Most all data read into R will be in the form of a dataframe

```
str(df)
```

```
## 'data.frame':    3 obs. of  3 variables:
##  $ nums_1: num  1 2 3
##  $ nums_2: num  6 5 4
##  $ strs  : chr  "a" "b" "c"
```

```
df
```

```
##   nums_1 nums_2 strs
## 1      1      6    a
## 2      2      5    b
## 3      3      4    c
```

Dataframes in R

```r
df[2,]
```

```
##   nums_1 nums_2 strs
## 2      2      5    b
```

```r
df[,3]
```

```
## [1] "a" "b" "c"
```

```r
df[2,1]
```

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

```r
df[c(1:3),]
```

```
##   nums_1 nums_2 strs
## 1      1      6    a
## 2      2      5    b
## 3      3      4    c
```

Subsetting dataframes in R

$ notation is preferred for selecting/working with columns

- Requires the column name
- $ returns the column as a vector, `[]` returns a `data.frame`
- Less ambiguous, less error-prone
- Can combine $ notation with `[]` for readability and automation
- `[[]]` can be used to similar effect with integer indexes

There is a `subset()` function - I'd encourage you not to use it

Dataframes in R

```
df$nums_1
```

```
## [1] 1 2 3
```

```
df$nums_1[1:2]
```

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

```
df[[1]]
```

```
## [1] 1 2 3
```

```
df[c("nums_1", "nums_2")]
```

```
##   nums_1 nums_2
## 1      1      6
## 2      2      5
## 3      3      4
```

Lists in R

Objects which can act as a collection of varied data types

- A list can contain any number of elements of vectors, matrices, dataframes, etc.
- Uses `[[]]` and `$` notation for accessing elements, much like the `data.frame`
- Very useful for storing multifaceted data and for automation purposes
- Families of R functions work off of the list structure

Lists in R

```r
my_list <- list()

my_list$a_vector <- c(1,2,3)
my_list$a_matrix <- matrix(seq(1,9),3,3)
my_list$a_dataframe <- df

my_list$a_vector
```

```
## [1] 1 2 3
```

```r
my_list[[1]]
```

```
## [1] 1 2 3
```

```r
my_list[[1]][1:2]
```

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

Reading in data in R

There are a lot of ways to read in data, and just as many places to get it

- Use the `setwd()` function to set the working directory (or see the 'Files' tab)
- We'll begin with the most commonly used: `read.csv`
- `variable <- read.csv(file.path, ...)`

We can pull in other types of data (dta, sas7bdat, SQL, etc.) with the help of packages/libraries

- `install.packages` and `library` commands
- If you can think it, there is probably a package that can do it
- More on this in a moment

Once data is in memory, the obvious next step is to inspect it

- `head`, `tail`, `str`, `names`, `nrow`, `ncol`, `dim`, `summary`, `table`, `unique`, etc.
- We can also grab summary statistics ad hoc

Reading in data in R

```
salaries_data <- read.csv("Salaries.csv", stringsAsFactors = F)
dim(salaries_data)
```

```
## [1] 397   7
```

```
str(salaries_data)
```

```
## 'data.frame':    397 obs. of  7 variables:
##  $ X            : int  1 2 3 4 5 6 7 8 9 10 ...
##  $ rank         : chr  "Prof" "Prof" "AsstProf" "Prof" ...
##  $ discipline   : chr  "B" "B" "B" "B" ...
##  $ yrs.since.phd: int  19 20 4 45 40 6 30 45 21 18 ...
##  $ yrs.service  : int  18 16 3 39 41 6 23 45 20 18 ...
##  $ sex          : chr  "Male" "Male" "Male" "Male" ...
##  $ salary       : int  139750 173200 79750 115000 141500 97000 17
```

Reading in *more* data in R

In order to pull in less 'traditional' data, we need to rely on functions outside the scope of base R

- Think of packages like apps, DLC, game expansions, etc.
- Packages make R extensible, and give you access to a multitude of functions 'off the shelf'
- Free to download and use (open-source!)
- Again, if you think it, there is probably a package that can do it
- https://cran.r-project.org/web/packages/available_packages_by_name.html

To access new packages

- `install.packages("haven")`
- `library(haven)`
- `help(package = "haven")`
- To access a specific function in an installed package, *without* loading it, use `package::function`

Reading in *more* data in R

```r
# install.packages("haven")
library(haven)
wage_data <- data.frame(read_dta("MROZ.dta"))
head(wage_data[1:9])
```

```
##   inlf hours kidslt6 kidsge6 age educ   wage repwage hushrs
## 1    1  1610       1       0  32   12 3.3540    2.65   2708
## 2    1  1656       0       2  30   12 1.3889    2.65   2310
## 3    1  1980       1       3  35   12 4.5455    4.04   3072
## 4    1   456       0       3  34   12 1.0965    3.25   1920
## 5    1  1568       1       2  31   14 4.5918    3.60   2000
## 6    1  2032       0       0  54   12 4.7421    4.70   1040
```

Writing data out in R

A typical workflow also involves performing analysis in R, and writing the data out for use in other programs

- Most commonly, use `write.csv` much the same as `read.csv`
- `writeRDS` creates R-specific files with better compression
- The packages above (and others) allow for writing out to more niche/complicated file formats
  - `readxl` for writing out to Excel spreadsheets
  - `haven` for writing out to Stata, SAS, etc.

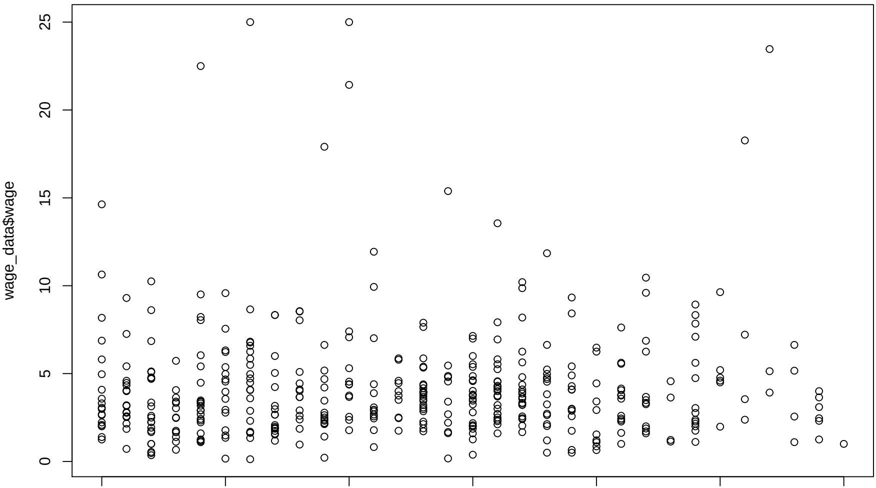
Basics of plotting R

R is *especially* powerful for data visualization

- Base R plots are extremely customizable, and can get you a long way (`?plot`)
- `hist()` and `boxplot()` are also available 'off the shelf'
- There are tons of great packages available, particularly `ggplot2`, to take you even further
- As always documentation is your best friend
- https://www.r-graph-gallery.com/

Basics of plotting R

```r
plot(wage_data$age, wage_data$wage)
```



Basics of modeling in R

Simple linear models (OLS) with `lm`

- `lm(y ~ x1 + x2 + x3 + ... + xn, data = data)`
- R is, first and foremost, a statistical computing language, so it's modeling capabilities can't be understated
- Unfortunately, we won't delve much into this here

Basics of modeling in R

```r
# Simple regression
fit <- lm(wage~educ, wage_data)
fit
```

```
##
## Call:
## lm(formula = wage ~ educ, data = wage_data)
##
## Coefficients:
## (Intercept)         educ
##     -2.0924       0.4953
```

Conditionals

- `==`, `!=`
- `<`, `>`
- `<=`, `>=`
- `%in%`
- `is.family`
- `|` and `&`
- `||` and `&&`

Conditionals

```
# Conditionals
1 < 2
```

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

```
x < y
```

```
## Warning in x < y: longer object length is not a multiple of short
## length
```

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

```
x == y
```

```
## Warning in x == y: longer object length is not a multiple of shor
## length
```

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

```
1 != 1
```

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

```
x %in% z
```

Conditionals

By themselves, conditionals seem boring/useless - used in control flow and for subsetting, they are incredibly useful

- The booleans generated from conditionals can be used for filtering data
- `TRUE` and `FALSE` values deterime what is kept and what is dropped
- Can be combined with `which` to return indices

Conditionals

```r
# Using conditionals
x <- c(10,90,2,0,7,10,4)
x >= 10
```

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

```r
which(x >= 10)
```

```
## [1] 1 2 6
```

```r
# Using conditionals in subsetting
dim(salaries_data)
```

```
## [1] 397   7
```

```r
dim(salaries_data[salaries_data$salary > 100000,])
```

```
## [1] 256   7
```

Practice

Using the 'Salaries' dataset:

- Using subsetting, drop the `X` index column
- Create both a histogram and a boxplot of the `salary` variable
- What proportion of the professors in the dataset are Female?
- Conduct a simple linear regression of `yrs.service` on `salary`
- Report the coefficients, standard errors, and confidence interval for the regression specified above
- Create a simple plot of `yrs.service` and `salary`
- Using `?plot`, create properly formatted labels and titles for the plot above
- What is the average salary of the `AssocProf` rank?
- Compute the standard deviations in salary for male and female professors, separately
- Which `discipline` has the higher median salary?
- How many years of service does the 200th individual in this dataset have?

Warm-up

- Read the `state_unemp_clean.csv` data into memory and assign it to a variable of your choosing
- Convert the `date` column to the date type (Hint: Use `as.Date` and reassign it to the `date` variable)
- Which state has the highest unemployment rate in the sample?
- In what year was that rate reached?
- What is the average household income across all the states in the sample?
- Create a time-series plot of the unemployment rate in the state with the lowest unemployment rate in 2016
- Change the `x`- and y-labels and plot title to descriptive names
- Using `?plot` for help, change the type of the plot to a line graph

Control flow

Determine the behavior of your program based on a specified condition

```
if (condition) {
  true_action
} else {
  false_action
}
```

Control flow

```
a <- 7

if(a%%2 == 0) {
  print("even")
} else {
  print("odd")
}
```

```
## [1] "odd"
```

```
ifelse(a%%2 == 0, "even", "odd")
```

```
## [1] "odd"
```

Functions

When what you need isn't available in base R or a package - write it!

User defined functions:

- Make code easier to read
- Reduce the possibility of human error
- Can be called where/whenever once defined
- Again, make automation/reproducibility easier

Rule of thumb: If you have to cut and paste the same complicated block of code more than twice, it might be helpful to define a function that can do it for you

Functions

```r
# A *very* simple function
square <- function(x) {
  return(x**2)
}

square(3)
```

```
## [1] 9
```

```r
square(x)
```

```
## [1]  100 8100    4    0   49  100   16
```

Loops

Two flavors

- `for`
- `while`
- Critically important for automation tasks/reproducibility

We will loop across (unsurprisingly) *vectors* in R

- This means we are not constrained to looping over indexes
- Can loop over indexes, but also vectors of strings (i.e. IDs)

Loops

General structure

- `for/while (object in (vector of things to loop over)) {`
- code that is executed over each element of the vector of things to loop over
- }

```
# Simple loop
for(i in 1:5){
  print(i)
}
```

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

Loops

```r
#Loops - extending our mix of control flow and conditionals above
a_vector <- c(1,6,7,8,8293,21,888,3,-2)

for(i in 1:length(a_vector)){
  if(a_vector[i]%%2 == 0) {
    print("even")
  } else {
    print("odd")
  }
}
```

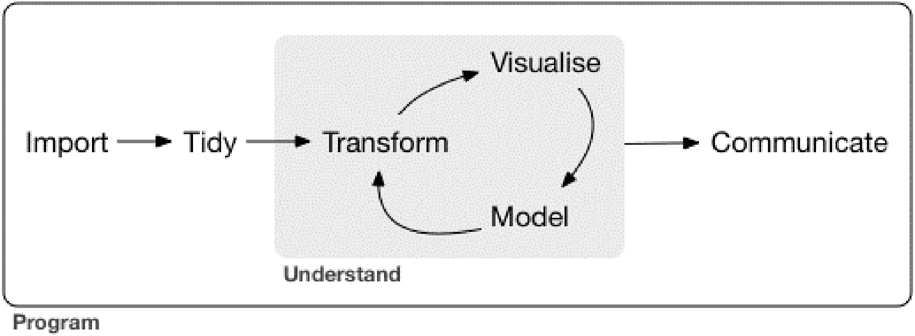
```
## [1] "odd"
## [1] "even"
## [1] "odd"
## [1] "even"
## [1] "odd"
## [1] "odd"
## [1] "even"
## [1] "odd"
## [1] "even"
```

Practice

Specify a random vector using the following syntax `rand_vect <- round(100*runif(1000),0)`

- Write a function `cube`, which takes a value and returns that value cubed; write a loop to apply this function to all the elements of the vector; print the cubed values
- Using a loop and control flow, check if each element of the vector is a perfect square, if it is return the index, i, and print "Perfect square!"
    - Hint: Use `x%%1` to check if your number is a whole number
- Load the 'salaries_list.Rds' object into memory (`readRDS()`)
- Inspect the list; what does each element contain? What is distinct about them?
- Loop through the list, and return the average of the `yrs.since.phd` variable for each element

Data analysis



Data cleaning and manipulation

Likely a big part of your next job

Typically involves. . .

- Data aggregation
- Merging together datasets from multiple sources
- Dealing with missing values
- Reshaping data
- And more

'Tidy' data

"Like families, tidy datasets are all alike but every messy dataset is messy in its own way." - Hadley Wickham has sought to advance a "standard" of sorts for what constitutes "tidy" data:

- Each variable forms a column.
- Each observation forms a row.
- Each type of observational unit forms a table.

These principles are adopted with the intent of reducing the time spent on data wrangling

- More time for modeling/the fun stuff
- Many packages to do this
- Base R does just fine, but we'll start working with `dplyr` here

Dealing with missing values

Missing values are 'contagious' and will interfere with summary functions

- Generally, it's okay to remove these, and we can do so by setting `na.rm = TRUE` in summary functions

How to deal with missing values depends on your particular research question

- `is.na()` (also good for 'special' missing values)
- `complete.cases()`
- `na.omit()`
- `df[is.na(df)] <- replacement`

dplyr basics

Why `dplyr`?

- Provides a grammar of data manipulation
- Standardized - most all `dplyr` commands follow the same, general, structure for arguments
  - `newdata <- (data, selection/condition/formula/etc.)`
- Human-readable - it's easier for those less familiar to follow code written with human english
  - 'Does what it says on the tin'
- Compatibility - works seemlessly with other `tidyverse` packages, and extends base R
- Fast, expressive, and agnostic about the format of your data
- `glimpse()`

dplyr basics

5 simple commands to make your life easier

- `select()` :: `$` and `[]`
- `filter()` :: `subset()`
- `arrange()` :: `sort()`
- `mutate()` :: `data$new_var <- var`
- `summarise()` :: `aggregate()`

dplyr basics

`select()`

- `newdata <- select(data, column1, column2,.)`
  - Note the difference between `-` and `!`
  - `!` is used for negating rows and conditionals
  - `-` is used for negating column selection
- Helper functions
  - `starts_with`, `ends_with`, `contains`, `matches`, `everything()`

`filter()`

- `newdata <- filter(data, conditions)`
- Remember your conditionals!

`arrange()`

dplyr basics

`mutate()`

`summarise()`

Pipes

One of the most powerful elements of `dplyr` programming is the pipe - `%>%`

- "Pipes" data into a function
- Defaults to first argument, taking advantage of `dplyr`'s standardization
- Sends output from the function to the next
- `data1 %>% stuff is done to data1 and becomes data2 %>% stuff is done to data2 etc.`
- Especially powerful when using `group_by` for subsetting

Merges and joins

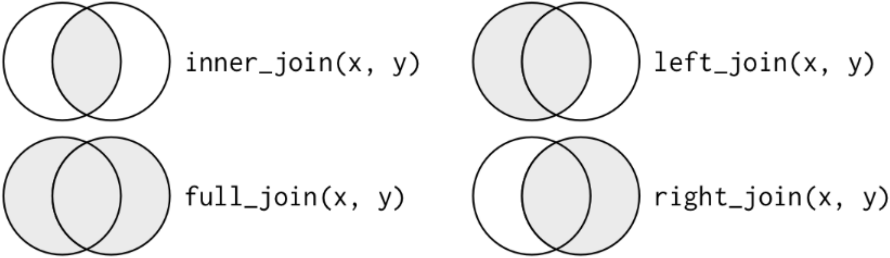
Base R

- `cbind()` binds data frames/matrices column-wise
- `rbind()` binds data frames/matrices row-wise (requires columns share the same name)
    - Requires equal numbers of columns/rows, respectively
    - *Not a merge* - does not involve keys
- `merge()`

`dplyr`

- `bind_rows()` and `bind_cols()` are better variants of the above
- `left_join()`
    - One of many dplyr merge commands-most commonly used
    - mutating and filtering joins

Merges and joins



https://r4ds.had.co.nz/relational-data.html#understanding-joins

Practice

- Merge in the World Development Indicators indicators data with the WEO data
- Report countries that did not receive valid `region` identifers and remove them
- Using `dplyr` and pipes (if you can!), in one chained command, create a subset of the data that:
  - Has only those observations from the `Europe & Central Asia` region from 2016
  - Has only the `country`, `gdp_cp`, `unemployment_rate`, and `curr_acc_bal` values
- Change the units of `unemployment_rate` to reflect a percent with a mutate command (divide by 100)
- Find the average unemployment rate for this group
- Create a time-series plot of Danish `unemployment` over the sample period
- Find the average level of GDP for each income group
- Which region has the largest within-region disparity in GDP per capita, as measured by standard deviation?

Dates

`as.Date(string, format)`

Symbol	Meaning	Example
%d	day as a number (0-31)	01-31
%a	abbreviated weekday	Mon
%A	unabbreviated weekday	Monday
%m	month (00-12)	00-12
%b	abbreviated month	Jan
%B	unabbreviated month	January
%y	2-digit year	07
%Y	4-digit year	2007

`lubridate`

Useful functions for cleaning and automation

Regular expressions

- `gsub`, `sub`, `grep`, `grepl`
- Paste commands (string manipulation)
  - `paste` and `paste0`
  - `substr`
  - `stringr`

The `apply` family

- Speedier loops in disguise
  - i.e. work through each element of a list or vector, each column of a dataframe, etc.
  - `lapply` and `sapply` (with others)
  - `lapply(data, function)`

`Reduce()` and `do.call()` for condensing lists

- `do.call(list, function)`

More tidying functions

There will be times where you confront data in the wrong ‘format’

- Most commonly, time series data that is ‘wide’ (i.e. each variable is a year)
- `tidyr` is a lightweight package to address this, without cumbersome loops
- `gather()` will gather ‘wide’ data
- `spread()` will spread ‘long’ data

A gentle introduction to ggplot2

Plots are built in layers of 'geom' functions

- `ggplot(data = mpg, aes(x = cty, y = hwy)) + geom_point()`
- `qplot` is a quick interface to work with `ggplot`, relying on useful defaults

ggplot2 layering

How does the `ggplot()` function work? By adding layers

- Specify an input data set
- Specify the columns to be used for x and y variables
- Specify the type of plot

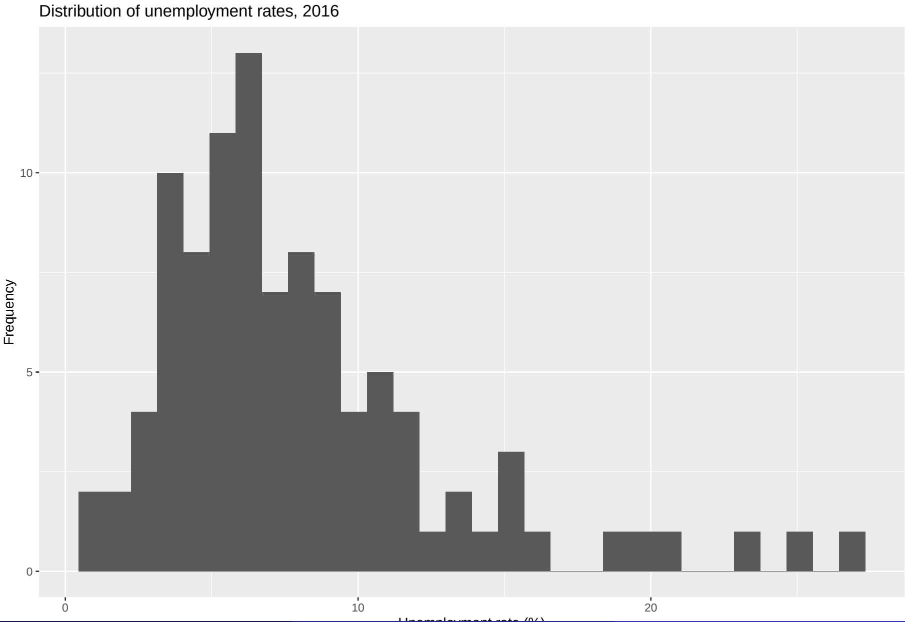
ggplot2

```r
# Give ggplot an input dataset, and a variable to plot
ggplot(weo_2016, aes(x=unemployment_rate)) +

# Pick the shape
  geom_histogram(bins=30) +

# Set some labels
  labs(x = "Unemployment rate (%)",
       y = "Frequency",
       title = "Distribution of unemployment rates, 2016")
```

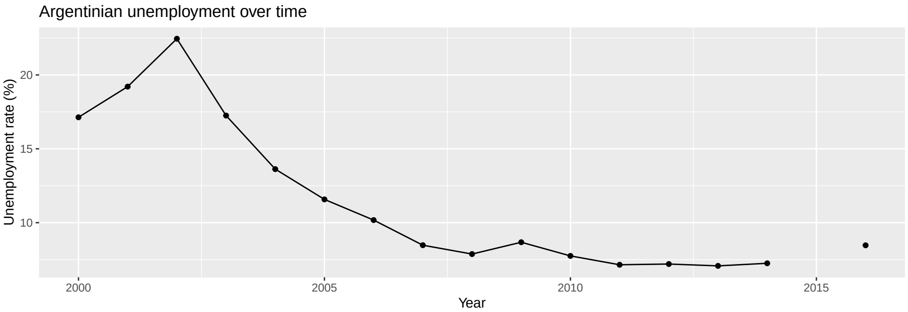
ggplot2



Distribution of unemployment rates, 2016
Frequency
10
5
0
0
10
20

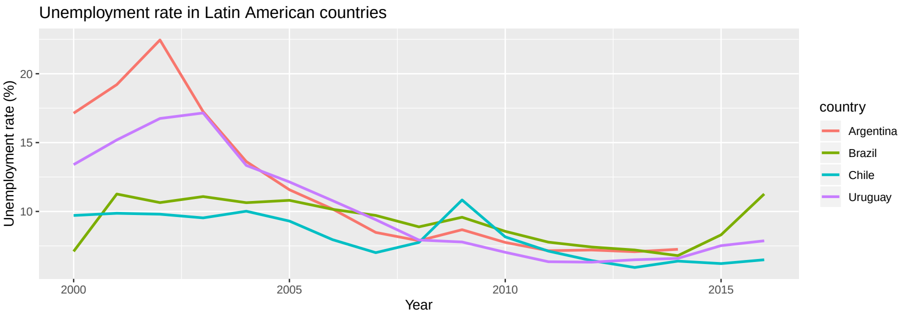
ggplot2

```r
# More geom layers
ggplot(weo_country, aes(x=year, y=unemployment_rate)) +
  geom_point() +
  geom_line() +
  labs(x = "Year", y = "Unemployment rate (%)",
       title = "Argentinian unemployment over time")
```



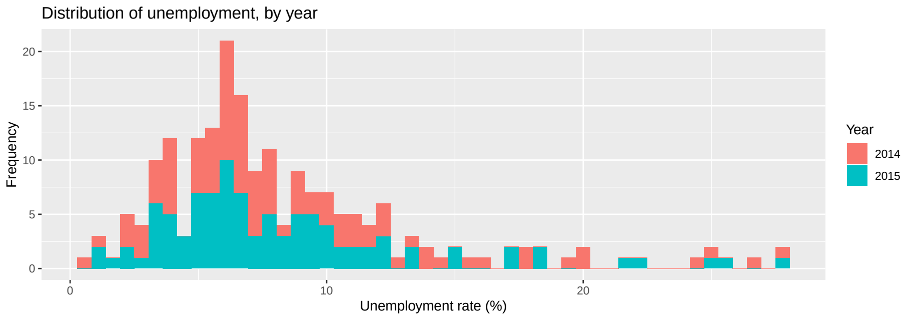
ggplot2

```
# Assign the 'country' variable to the 'col' argument
ggplot(weo_countries, aes(x=year, y=unemployment_rate, col=country))
  geom_line(size=1) +
  labs(x = "Year", y = "Unemployment rate (%)",
       title = "Unemployment rate in Latin American countries")
```



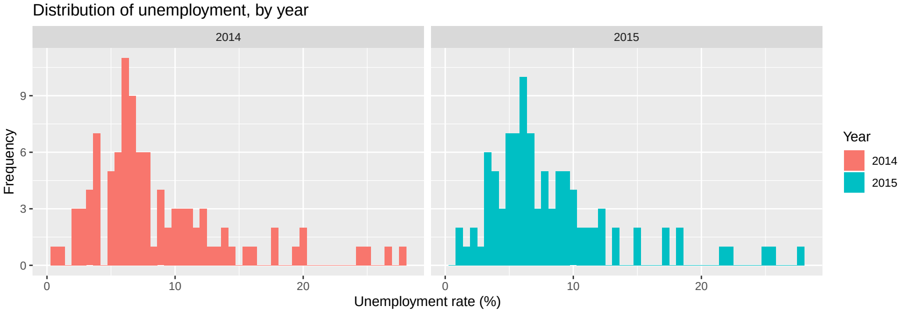
ggplot2

```r
# Assign the 'year' variable to the 'fill' argument
ggplot(weo_years, aes(x=unemployment_rate, fill=factor(year))) +
  geom_histogram(bins=50) +
  labs(x = "Unemployment rate (%)", y = "Frequency",
       title = "Distribution of unemployment, by year") +
  scale_fill_discrete(name = "Year")
```



ggplot2

```r
# Side-by-side using the 'facet_wrap' argument
ggplot(weo_years, aes(x=unemployment_rate, fill=factor(year))) +
  geom_histogram(bins=50) +
  facet_wrap(~factor(year)) +
  labs(x = "Unemployment rate (%)", y = "Frequency",
       title = "Distribution of unemployment, by year") +
  scale_fill_discrete(name = "Year")
```



ggplot2 practice

- Ensure you have `weo_full` in memory - if not, revisit the code from the `session_2.R` file
- Plot the GDP per capita values for the `Europe & Central Asia` region over time, with each country as a separate color; label accordingly
- Using `dplyr` commands (and pipes, if possible), plot the average `unemployment_rate` over time, with each region as its own color; label accordingly
- *Bonus*: Try to replicate both of these plots using the `qplot` function

R Markdown

We've been working exclusively with R scripts (`.R` files)

- R scripts are standard for writing reproducible code
- R scripts are not particularly high-quality for presentation purposes

R Markdown wraps a markup language (think LaTeX, XML, etc.) around the R programming language

- R Markdown can embed all of your code into a 'prettified' HTML, PDF, or Word file
- With very little effort, you can put together high-quality reports and presentations with embedded code and data visualizations

Resources

- https://r4ds.had.co.nz/
- http://adv-r.had.co.nz/
- https://www.rstudio.com/resources/cheatsheets/
- https://www.datacamp.com/
- https://lagunita.stanford.edu/courses/HumanitiesSciences/StatLearning/Winter2016/about