

# IQSS Workshops

October 2019



# Contents

<b>Introduction</b>	<b>7</b>
Table of Contents . . . . .	7
Authors and Sources . . . . .	7
 <b>I R</b>	 <b>9</b>
 <b>1 R Introduction</b>	 <b>11</b>
1.1 Setup . . . . .	11
1.2 R Interfaces . . . . .	12
1.3 Exercise 0 . . . . .	13
1.4 R basics . . . . .	13
1.5 Getting data into R . . . . .	14
1.6 Exercise 1 . . . . .	15
1.7 Popularity of your name . . . . .	16
1.8 Exercise 2.1 . . . . .	17
1.9 Pipe operator in R . . . . .	18
1.10 Exercise 2.2 . . . . .	18
1.11 Plotting baby name trends over time . . . . .	18
1.12 Exercise 3 . . . . .	19
1.13 Finding the most popular names . . . . .	19
1.14 Exercise 4 . . . . .	20
1.15 Percent choosing one of the top 10 names . . . . .	21
1.16 Exercise 5 . . . . .	22
1.17 Saving our Work . . . . .	22

1.18	Exercise solutions . . . . .	23
1.19	Wrap-up . . . . .	27
<b>2</b>	<b>R Regression Models</b>	<b>29</b>
2.1	Setup . . . . .	29
2.2	Before fitting a model . . . . .	30
2.3	Regression with continuous outcomes . . . . .	31
2.4	Exercise 0 . . . . .	34
2.5	Interactions and factors . . . . .	34
2.6	Exercise 1 . . . . .	35
2.7	Regression with binary outcomes . . . . .	36
2.8	Exercise 2 . . . . .	37
2.9	Multilevel modeling . . . . .	37
2.10	Exercise 3 . . . . .	40
2.11	Exercise solutions . . . . .	41
2.12	Wrap-up . . . . .	44
<b>3</b>	<b>R Graphics</b>	<b>45</b>
3.1	Setup . . . . .	45
3.2	Why <code>ggplot2</code> ? . . . . .	46
3.3	Geometric objects and aesthetics . . . . .	49
3.4	Exercise 0 . . . . .	52
3.5	Statistical transformations . . . . .	52
3.6	Exercise 1 . . . . .	53
3.7	Scales . . . . .	54
3.8	Exercise 2 . . . . .	57
3.9	Faceting . . . . .	57
3.10	Themes . . . . .	58
3.11	The #1 FAQ . . . . .	59
3.12	Putting it all together . . . . .	60
3.13	Exercise solutions . . . . .	66
3.14	Wrap-up . . . . .	68

<i>CONTENTS</i>	5
-----------------	---

## 4 R Data Wrangling 71

4.1 Setup . . . . .	71
4.2 Example project . . . . .	72
4.3 Exercise 0 . . . . .	73
4.4 Useful data manipulation packages . . . . .	73
4.5 Working with Excel worksheets . . . . .	73
4.6 Reading Excel data files . . . . .	75
4.7 Exercise 1 . . . . .	76
4.8 Data cleanup . . . . .	76
4.9 Exercise 2 . . . . .	79
4.10 Exercise 3 . . . . .	79
4.11 Data organization and storage . . . . .	79
4.12 Exercise 4 . . . . .	80
4.13 Exercise solutions . . . . .	80
4.14 Wrap-up . . . . .	83

## II Python 85

### 5 Python Introduction 87

5.1 Setup . . . . .	87
5.2 What is Python? . . . . .	88
5.3 How can I interact with Python? . . . . .	88
5.4 Reading the text of Alice in Wonderland from a file . . . . .	90
5.5 Counting chapters, lines, and words . . . . .	91
5.6 Exercise: Reading text from a file and splitting . . . . .	92
5.7 Exercise: count the number of main characters . . . . .	95
5.8 Working with nested structures: words within paragraphs within chapters . . . . .	95
5.9 Exercise: Iterating and counting things . . . . .	97
5.10 Importing numpy and calculating simple statistics . . . . .	98
5.11 Wrap-up . . . . .	98

<b>6</b>	<b>Python Web-Scraping</b>	<b>101</b>
6.1	Setup . . . . .	101
6.2	Preliminary questions . . . . .	102
6.3	Example project . . . . .	102
6.4	Take shortcuts if you can . . . . .	103
6.5	Parse html if you have to . . . . .	107
6.6	Use Scrapy for large or complicated projects . . . . .	112
6.7	Use a browser driver as a last resort . . . . .	113
6.8	Wrap-up . . . . .	113
<b>III</b>	<b>Stata</b>	<b>115</b>
<b>7</b>	<b>Stata Introduction</b>	<b>117</b>
7.1	Setup . . . . .	117
7.2	Why stata? . . . . .	118
7.3	Getting data into Stata . . . . .	120
7.4	Statistics and graphs . . . . .	121
7.5	Basic data management . . . . .	123
7.6	Exercise 2: Variable labels and value labels . . . . .	123
7.7	Exercise 3: Manipulating variables . . . . .	125
7.8	Wrap-up . . . . .	125

# Introduction

## Table of Contents

Materials for the software workshops held at the Institute for Quantitative Social Science and Harvard Business School at Harvard.

1. R Introduction
2. R Regression Models
3. R Graphics
4. R Data Wrangling
5. Python Introduction
6. Python Web-Scraping
7. Stata Introduction

These workshops are a work-in-progress, please provide feedback! Email: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

## Authors and Sources

The contents of these workshops are the result of a collaborative effort from members of the Data Science Services team at IQSS and the Research Computing Services team at HBS. The main contributors are: Ista Zahn, Steve Worthington, Bob Freeman, Jinjie Liu, Yihan Wang, and Victoria Liublinska.





# Part I

## R



# Chapter 1

## R Introduction

### Topics

- Assignment
- Function arguments
- Finding help
- Reading data
- Filtering rows, selecting columns, and arranging data
- Conditional operations
- Saving data

### 1.1 Setup

#### 1.1.1 Software & materials

You should have R and RStudio installed — if not:

- Download and install R: <http://cran.r-project.org>
- Download and install RStudio: <https://www.rstudio.com/products/rstudio/download/#download>

Download materials:

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/R/Rintro.zip>
- Extract materials from the zipped directory **Rintro.zip** (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

Start RStudio and create a new project:

- On Windows click the start button and search for RStudio. On Mac RStudio will be in your applications folder.
- In Rstudio go to **File -> New Project**.
- Choose **Existing Directory** and browse to the **Rintro** directory.
- Choose **File -> Open File** and select the blank version of the **.Rmd** file.

### 1.1.2 Goals

Class Structure and Organization:

- Ask questions at any time. Really!
- Collaboration is encouraged - please spend a minute introducing yourself to your neighbors!

This is an introductory R course:

- Assumes no prior knowledge of R
- Relatively slow-paced
- The workshop covers R basics, the R package ecosystem, and practice reading files and manipulating data in R
- A more general goal is to get you comfortable with R so that it seems less scary and mystifying than it perhaps does now. Note that this is by no means a complete or thorough introduction to R! It's just enough to get you started.

As an example project we will analyze the popularity of baby names in the US from 1960 through 2017. Among the questions we will answer using R are:

- In which year did your name achieve peak popularity?
- How many children were born each year?
- What are the most popular names overall? For girls? For Boys?

## 1.2 R Interfaces

There are many different ways you can interact with R. See the Data Science Tools workshop notes for details.

For this workshop we will use RStudio; it is a good R-specific integrated development environment (IDE) that mostly just works.

We will also use Rmarkdown — a type of R file that allows you to combine plain text with R code. It is easy to later convert Rmarkdown into MS Word, LaTeX, a pdf, or webpage.

## 1.3 Exercise 0

The purpose of this exercise is to give you an opportunity to explore the interface provided by RStudio. You may not know how to do these things; that's fine! This is an opportunity to figure it out.

Also keep in mind that we are living in a golden age of tab completion. If you don't know the name of an R function, try guessing the first two or three letters and pressing TAB. If you guessed correctly the function you are looking for should appear in a pop up!

- 
1. Try to get R to add 2 plus 2.

```
##
```

2. Try to calculate the square root of 10.

```
##
```

3. R includes extensive documentation, including a manual named “An introduction to R”. Use the RStudio help pane. to locate this manual.

## 1.4 R basics

### 1.4.1 Function calls

The general form for calling R functions is

```
# FunctionName(arg.1 = value.1, arg.2 = value.2, ..., arg.n = value.n)
```

Arguments can be matched by name; unnamed arguments will be matched by position.

```
values <- c(1.45, 2.34, 5.68)
round(x = values, digits = 1) # match by name
round(values, 1) # match by position
round(1, values) # be careful when matching by position!
round(digits = 1, x = values) # matching by name is safer!
```

### 1.4.2 Assignment

Values can be assigned names and used in subsequent operations

- The **gets** `<-` operator (less than followed by a dash) is used to save values
- The name on the left **gets** the value on the right.

```
sqrt(10) ## calculate square root of 10; result is not stored anywhere
x <- sqrt(10) # assign result to a variable named x
```

Names should start with a letter, and contain only letters, numbers, underscores, and periods.

### 1.4.3 Asking R for help

You can ask R for help using the **help** function, or the `?` shortcut.

```
help(help)
?help
?sqrt
```

The **help** function can be used to look up the documentation for a function, or to look up the documentation to a package. We can learn how to use the **stats** package by reading its documentation like this:

```
help(package = "stats")
```

## 1.5 Getting data into R

R has data reading functionality built-in – see e.g., **help**(`read.table`). However, faster and more robust tools are available, and so to make things easier on ourselves we will use a *contributed package* instead. This requires that we learn a little bit about packages in R.

### 1.5.1 Installing and using R packages

A large number of contributed packages are available. If you are looking for a package for a specific task, <https://cran.r-project.org/web/views/> and <https://r-pkg.org> are good places to start.

You can install a package in R using the **install.packages()** function. Once a package is installed you may use the **library()** function to attach it so that it can be used.

While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called

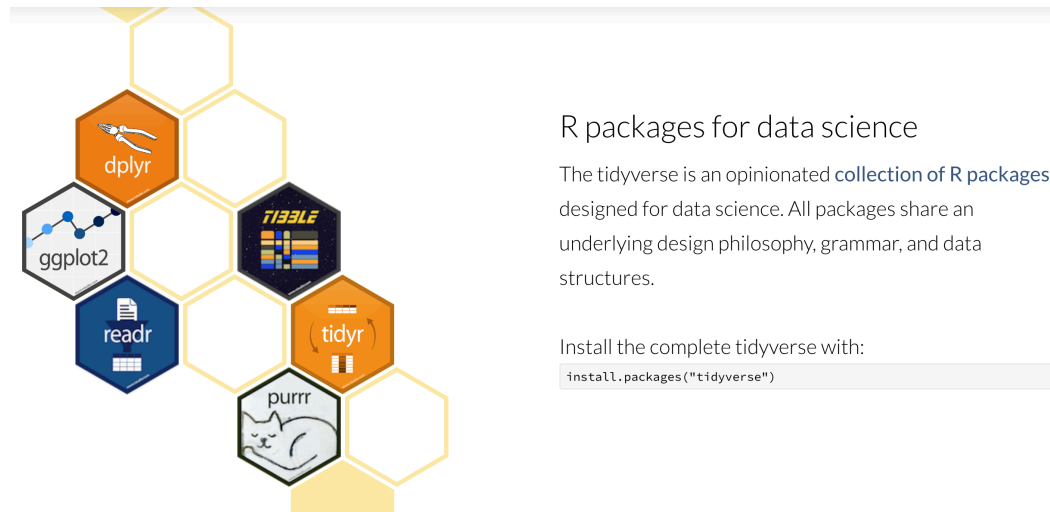


Figure 1.1

`tidyverse` have been designed specifically for data science. All packages included in `tidyverse` share an underlying design philosophy, grammar, and data structures. We will use `tidyverse` packages throughout the workshop, so let's install them now:

```
# install.packages("tidyverse")
library(tidyverse)
```

### 1.5.2 Readers for common file types

To read data from a file, you have to know what kind of file it is. The table below lists functions from the `readr` package, which is part of `tidyverse`, that can import data from common plain-text formats.

Data Type	Function
comma separated	<code>read_csv()</code>
tab separated	<code>read_delim()</code>
other delimited formats	<code>read_table()</code>
fixed width	<code>read_fwf()</code>

**Note** You may be confused by the existence of similar functions, e.g., `read.csv` and `read.delim`. These are legacy functions that tend to be slower and less robust than the `readr` functions. One way to tell them apart is that the faster more robust versions use underscores in their names (e.g., `read_csv`) while the older functions use dots (e.g., `read.csv`). My advice is to use the more robust newer versions, i.e., the ones with underscores.

### 1.5.3 Baby names data

The examples in this workshop use US baby names data retrieved from <https://catalog.data.gov/dataset/baby-names-from-social-security-card-applications-national-level-data>

A cleaned and merged version of these data is available at <http://tutorials.iq.harvard.edu/data/babyNames.csv>

## 1.6 Exercise 1

2. Read the baby names data using the `read_csv` function and assign the result with the name `baby_names`.

```
##
```

## 1.7 Popularity of your name

In this section we will pull out specific names and examine changes in their popularity over time.

The `baby_names` object we created in the last exercise is a `data.frame`. There are many other data structures in R, but for now we'll focus on working with `data.frames`.

R has decent data manipulation tools built-in – see e.g., `help(Extract)`. But, `tidyverse` packages often provide more intuitive syntax for accomplishing the same task. In particular, we will use the `dplyr` package from `tidyverse` to filter, select, and arrange data.

### 1.7.1 Filtering, selecting, and arranging data

One way to find the year in which your name was the most popular is to filter out just the rows corresponding to your name, and then arrange (sort) by Count.

To demonstrate these techniques we'll try to determine whether “Alex” or “Mark” was more popular in 1992. We start by filtering the data so that we keep only rows where Year is equal to 1992 and Name is either “Alex” or “Mark”.

```
baby_names_alexmark <- filter(baby_names,
                             Year == 1992 & (Name == "Alex" | Name == "Mark"))
baby_names_alexmark
```

Notice that we can combine conditions using `&` (AND) and `|` (OR).

In this case it's pretty easy to see that “Mark” is more popular, but to make it even easier we can arrange the data so that the most popular name is listed first.

```
arrange(baby_names_alexmark, Count)
```

```
arrange(baby_names_alexmark, desc(Count))
```

We can also use the `select()` function to subset the `data.frame` by columns. We can then assign the output to a new object.

```
baby_names_subset <- select(baby_names, Name, Count)
head(baby_names_subset)
```



### 1.7.2 Other logical operators

In the previous example we used `==` to filter rows. Other relational and logical operators are listed below.

Operator	Meaning
<code>==</code>	equal to
<code>!=</code>	not equal to
<code>&gt;</code>	greater than
<code>&gt;=</code>	greater than or equal to
<code>&lt;</code>	less than
<code>&lt;=</code>	less than or equal to
<code>%in%</code>	contained in

These operators may be combined with `&` (and) or `|` (or).

## 1.8 Exercise 2.1

### Peak popularity of your name

In this exercise you will discover the year your name reached its maximum popularity.

Read in the “babyNames.csv” file if you have not already done so, assigning the result to `baby_names`. The file is located at “<http://tutorials.iq.harvard.edu/data/babyNames.csv>”

Make sure you have installed the `tidyverse` suite of packages and attached them with `library(tidyverse)`.

1. Use `filter` to extract data for your name (or another name of your choice).

```
##
```

2. Arrange the data you produced in step 1 above by `Count`. In which year was the name most popular?

```
##
```

3. BONUS (optional): Filter the data to extract *only* the row containing the most popular boys name in 1999.

```
##
```

## 1.9 Pipe operator in R

There is one very special operator in R called a “pipe” operator that looks like this: `%>%`. It allows us to “chain” several function calls and, as each function returns an object, feed it into the next call in a single statement, without needing extra variables to store the intermediate results. The point of the pipe is to help you write code in a way that is easier to read and understand as we will see below.

There is no need to load any additional packages as the operator is made available via the `magrittr` package installed as part of `tidyverse`. Let’s rewrite the sequence of commands to output ordered counts for names “Alex” or “Mark”.

```

baby_names %>%
  filter(Year == 1992 & (Name == "Alex" | Name == "Mark")) %>%
  arrange(desc(Count))

```

Hint: try pronouncing “then” whenever you see `%>%`. Notice that we avoided creating an intermediate variable `baby_names_alexmark` and performed the entire task in just “one line”!

## 1.10 Exercise 2.2

Rewrite the solution to Exercise 2.1 using pipes. Remember that we were looking for the year your name reached its maximum popularity. For that, we filtered the data and then arranged by `Count`.

```
##
```

## 1.11 Plotting baby name trends over time

It can be difficult to spot trends when looking at summary tables. Plotting the data makes it easier to identify interesting patterns.

R has decent plotting tools built-in – see e.g., `help(plot)`. However, again, we will make use of an excellent *contributed package* from `tidyverse` called `ggplot2`.

For quick and simple plots we can use the `qplot()` function. For example, we can plot the number of babies given the name “Diana” over time like this:

```

baby_names_diana <- filter(baby_names, Name == "Diana")

```

```

qplot(x = Year, y = Count,
      data = baby_names_diana)

```

Interestingly, there are usually some gender-atypical names, even for very strongly gendered names like “Diana”. Splitting these trends out by Sex is very easy:

```
qplot(x = Year, y = Count, color = Sex,  
      data = baby_names_diana)
```

## 1.12 Exercise 3

### Plotting peak popularity of your name

Make sure the `tidyverse` suite of packages is installed, and that you have attached them using `library(tidyverse)`.

1. Use `filter` to extract data for your name (same as previous exercise)

```
##
```

2. Plot the data you produced in step 1 above, with `Year` on the x-axis and `Count` on the y-axis.

```
##
```

3. Adjust the plot so that it shows boys and girls in different colors.

```
##
```

4. BONUS (Optional): Adjust the plot to use lines instead of points.

```
##
```

## 1.13 Finding the most popular names

Our next goal is to find out which names have been the most popular.

### 1.13.1 Computing better measures of popularity

So far we've used `Count` as a measure of popularity. A better approach is to use proportion or rank to avoid confounding popularity with the number of babies born in a given year.

The `mutate()` function makes it easy to add or modify the columns of a `data.frame`. For example, we can use it to rescale the count of each name in each year:

```
baby_names <- mutate(baby_names, Count_1k = Count/1000)  
baby_names # same as print(baby_names)
```

Notice that executing the second line led to printing all 1 million rows in our `data.frame`! If we would just like to glance at the first 6 lines we can use the `head()` function:

```
head(baby_names)
```

If we like, we can also `select()` a subset of columns from the baby names data:

```
baby_names %>%
  select(Name, Sex, Year, Count_1k) %>%
  head()
```

### 1.13.2 Operating by group

Because of the nested nature of our data, we want to compute rank or proportion within each `Sex` by `Year` group. The `dplyr` package makes this relatively straightforward.

```
baby_names <-
  baby_names %>%
  group_by(Year, Sex) %>%
  mutate(Rank = rank(Count_1k)) %>%
  ungroup()

head(baby_names)
```

Note that the data remains grouped until you change the groups by running `group_by()` again or remove grouping information with `ungroup()`.

## 1.14 Exercise 4

### Most popular names

In this exercise your goal is to identify the most popular names for each year.

1. Use `mutate()` and `group_by()` to create a column named “Proportion” where `Proportion = Count/sum(Count)` for each `Year X Sex` group. Use pipes wherever it makes sense.

```
##
```

2. Use `mutate()` and `group_by()` to create a column named “Rank” where `Rank = rank(-Count)` for each `Year X Sex` group.

##

3. Filter the baby names data to display only the most popular name for each **Year X Sex** group. Keep only the columns: Name, Sex, and Proportion.

##

4. Plot the data produced in step 4, putting **Year** on the x-axis and **Proportion** on the y-axis. How has the proportion of babies given the most popular name changed over time?

##

5. BONUS (optional): Which names are the most popular for both boys and girls?

##

## 1.15 Percent choosing one of the top 10 names

You may have noticed that the percentage of babies given the most popular name of the year appears to have decreased over time. We can compute a more robust measure of the popularity of the most popular names by calculating the number of babies given one of the top 10 girl or boy names of the year.

To compute this measure we need to operate within groups, as we did using `mutate()` above, but this time we need to collapse each group into a single summary statistic. We can achieve this using the `summarize()` function.

First, let's see how this function works without grouping. The following code outputs the total number of girls and boys in the data:

```

baby_names %>%
  summarize(Girls_n = sum(Sex=="Girls"),
            Boys_n = sum(Sex=="Boys"))

```

Next, using `group_by()` and `summarize()` together, we can calculate the number of babies born each year:

```

bn_by_year <-
  baby_names %>%
  group_by(Year) %>%
  summarize(Total = sum(Count))

head(bn_by_year)

```

## 1.16 Exercise 5

### Popularity of the most popular names

In this exercise we will plot trends in the proportion of boys and girls given one of the 10 most popular names each year.

1. Filter the `baby_names` data, retaining only the 10 most popular girl and boy names for each year.

```
##
```

2. Summarize the data produced in step one to calculate the total Proportion of boys and girls given one of the top 10 names each year.

```
##
```

3. Plot the data produced in step 2, with year on the x-axis and total proportion on the y axis. Color by sex and notice the trend.

```
##
```

## 1.17 Saving our Work

Now that we have made some changes to our data set, we might want to save those changes to a file.

### 1.17.1 Saving individual datasets

You might find functions `write_csv()` and `write_rds()` from package `readr` handy!

```
# write data to a .csv file
write_csv(baby_names, "babyNames.csv")
```

```
# write data to an R file
write_rds(baby_names, "babyNames.rds")
```

### 1.17.2 Saving multiple datasets

```
ls() # list objects in our workspace
save(girls_and_boys, bn_by_year, most_popular, file="myDataFiles.RData")
```

```
## Load the "myDataFiles.RData"  
## load("myDataFiles.RData")
```

### 1.17.3 Saving and loading R workspaces

In addition to importing individual datasets, R can save and load entire workspaces

```
ls() # list objects in our workspace  
save.image(file="myWorkspace.RData") # save workspace  
rm(list=ls()) # remove all objects from our workspace  
ls() # list stored objects to make sure they are deleted
```

```
## Load the "myWorkspace.RData" file and check that it is restored  
load("myWorkspace.RData") # load myWorkspace.RData  
ls() # list objects
```

## 1.18 Exercise solutions

### 1.18.1 Ex 0: prototype

```
## 1. 2 plus 2  
2 + 2  
## or  
sum(2, 2)
```

```
## 2. square root of 10:  
sqrt(10)  
## or  
10^(1/2)
```

```
## 3. Find "An Introduction to R".
```

```
## Go to the main help page by running 'help.start()' or using the GUI  
## menu, find and click on the link to "An Introduction to R".
```

```
##
```

### 1.18.2 Ex 1: prototype

```
## read ?read_csv
```

```
baby_names <- read_csv("http://tutorials.iq.harvard.edu/data/babyNames.csv")
```

### 1.18.3 Ex 2.1: prototype

*# 1. Use `filter` to extract data for your name (or another name of your choice).*

```
baby_names_george <- filter(baby_names, Name == "George")
```

*# 2. Arrange the data you produced in step 1 above by `Count`.  
# In which year was the name most popular?*

```
arrange(baby_names_george, desc(Count))
```

*# 3. BONUS (optional): Filter the data to extract \_only\_ the  
# row containing the most popular boys name in 1999.*

```
baby_names_boys_1999 <- filter(baby_names,  
                               Year == 1999 & Sex == "Boys")
```

```
filter(baby_names_boys_1999, Count == max(Count))
```

### 1.18.4 Ex 2.2: prototype

```
baby_names %>%  
  filter(Name == "George") %>%  
  arrange(desc(Count))
```

### 1.18.5 Ex 3: prototype

*# 1. Use `filter()` to extract data for your name (same as previous exercise)*

```
baby_names_george <- filter(baby_names, Name == "George")
```



*# 2. Plot the data you produced in step 1 above, with `Year` on the x-axis and `Count` on the y-axis.*

```
qplot(x = Year, y = Count, data = baby_names_george)
```

*# 3. Adjust the plot so that it shows boys and girls in different colors.*

```
qplot(x = Year, y = Count, color = Sex, data = baby_names_george)
```

*# 4. BONUS (Optional): Adjust the plot to use lines instead of points.*

```
qplot(x = Year, y = Count, color = Sex, data = baby_names_george, geom = "line")
```

### 1.18.6 Ex 4: prototype

## 1. Use `mutate()` and `group\_by()` to create a column named "Proportion" where `Proportion = Count/sum(Count)` for each `Year X Sex` group.

```
baby_names <-  
  baby_names %>%  
  group_by(Year, Sex) %>%  
  mutate(Proportion = Count/sum(Count)) %>%  
  ungroup()  
  
head(baby_names)
```

## 2. Use `mutate()` and `group\_by()` to create a column named "Rank" where `Rank = rank(-Count)` for each `Year X Sex` group.

```
baby_names <-  
  baby_names %>%  
  group_by(Year, Sex) %>%  
  mutate(Rank = rank(-Count)) %>%  
  ungroup()  
  
head(baby_names)
```

## 3. Filter the baby names data to display only the most popular name for each `Year X Sex` group.

```
top1 <-
  baby_names %>%
  filter(Rank == 1) %>%
  select(Name, Sex, Proportion)

head(top1)
```

```
## 4. Plot the data produced in step 3, putting `Year` on the x-axis
##    and `Proportion` on the y-axis. How has the proportion of babies
##    given the most popular name changed over time?
```

```
qplot(x = Year,
      y = Proportion,
      color = Sex,
      data = top1,
      geom = "line")
```

```
## 5. BONUS (optional): Which names are the most popular for both boys
##    and girls?
```

```
bn_girls <- baby_names %>%
  filter(Sex == "Girls") %>%
  select(Name, Year, Count)

bn_boys <- baby_names %>%
  filter(Sex == "Boys") %>%
  select(Name, Year, Count)

girls_and_boys <- inner_join(bn_girls,
                             bn_boys,
                             by = c("Year", "Name"))

head(girls_and_boys)
```

```
girls_and_boys <-
  girls_and_boys %>%
  mutate(Product = Count.x * Count.y,
         Rank = rank(-Product)) %>%
  filter(Rank == 1)

head(girls_and_boys)
```

### 1.18.7 Ex 5: prototype

```
## 1. Filter the baby_names data, retaining only the 10 most
##     popular girl and boy names for each year.
```

```
most_popular <-
  baby_names %>%
  group_by(Year, Sex) %>%
  filter(Rank <= 10)

head(most_popular, n = 10)
```

```
## 2. Summarize the data produced in step one to calculate the total
##     Proportion of boys and girls given one of the top 10 names
##     each year.
```

```
top10 <-
  most_popular %>% # it is already grouped by Year and Sex
  summarize(TotalProportion = sum(Proportion))
```

```
## 3. Plot the data produced in step 2, with year on the x-axis
##     and total proportion on the y axis. Color by sex.
```

```
qplot(x = Year,
      y = TotalProportion,
      color = Sex,
      data = top10,
      geom = "line")
```

## 1.19 Wrap-up

### 1.19.1 Feedback

These workshops are a work-in-progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 1.19.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>
  - Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>
- HBS

- Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>
- Software (all free!):
  - R and R package download: <http://cran.r-project.org>
  - Rstudio download: <http://rstudio.org>
  - ESS (emacs R package): <http://ess.r-project.org/>
- Online tutorials
  - <http://www.codeschool.com/courses/try-r>
  - <http://www.datacamp.org>
  - <http://swirlstats.com/>
  - <http://r4ds.had.co.nz/>
- Getting help:
  - Documentation and tutorials: <http://cran.r-project.org/other-docs.html>
  - Recommended R packages by topic: <http://cran.r-project.org/web/views/>
  - Mailing list: <https://stat.ethz.ch/mailman/listinfo/r-help>
  - StackOverflow: <http://stackoverflow.com/questions/tagged/r>
  - R-Bloggers: <https://www.r-bloggers.com/>
- Coming from ...
  - Stata: <http://www.princeton.edu/~otorres/RStata.pdf>
  - SAS/SPSS: <http://r4stats.com/books/free-version/>
  - Matlab: <http://www.math.umaine.edu/~hiebler/comp/matlabR.pdf>
  - Python: <http://mathesaurus.sourceforge.net/matlab-python-xref.pdf>

## Chapter 2

# R Regression Models

### Topics

- R formula interface
- Run and interpret variety of regression models in R
- Factor contrasts to test specific hypotheses
- Model comparisons
- Predicted marginal effects

## 2.1 Setup

### 2.1.1 Software & materials

You should have R and RStudio installed — if not:

- Download and install R: <http://cran.r-project.org>
- Download and install RStudio: <https://www.rstudio.com/products/rstudio/download/#download>

Download materials:

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/R/Rmodels.zip>
- Extract materials from the zipped directory `Rmodels.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

Start RStudio and create a new project:

- On Windows click the start button and search for RStudio. On Mac RStudio will be in your applications folder.

- In Rstudio go to File -> New Project.
- Choose Existing Directory and browse to the Rmodels directory.
- Choose File -> Open File and select the blank version of the .Rmd file.

While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called **tidyverse** have been designed specifically for data science. All packages included in **tidyverse** share an underlying design philosophy, grammar, and data structures. We will use **tidyverse** packages throughout the workshop, so let's install them now:

```
# install.packages("tidyverse")  
library(tidyverse)
```

### 2.1.2 Goals

Class Structure and Organization:

- Ask questions at any time. Really!
- Collaboration is encouraged - please spend a minute introducing yourself to your neighbors!

This is an intermediate R course:

- Assumes working knowledge of R
- Relatively fast-paced
- This is not a statistics course! We assume you know the theory behind the models

## 2.2 Before fitting a model

### 2.2.1 Load the data

List the data files we're going to work with:

```
list.files("dataSets")
```

We're going to use the **states** data first, which originally appeared in *Statistics with Stata* by Lawrence C. Hamilton.

```
# read the states data  
states_data <- read_rds("dataSets/states.rds")
```

Variable	Description
csat	Mean composite SAT score
expense	Per pupil expenditures
percent	% HS graduates taking SAT
income	Median household income, \$1,000
region	Geographic region: West, N. East, South, Midwest
house	House '91 environ. voting, %
senate	Senate '91 environ. voting, %
energy	Per capita energy consumed, Btu
metro	Metropolitan area population, %
waste	Per capita solid waste, tons

### 2.2.2 Examine the data

Start by examining the data to check for problems.

```
# summary of expense and csat columns, all rows
sts_ex_sat <- subset(states_data, select = c("expense", "csat"))
summary(sts_ex_sat)
# correlation between expense and csat
cor(sts_ex_sat)
```

### 2.2.3 Plot the data

Plot the data to look for multivariate outliers, non-linear relationships etc.

```
# scatter plot of expense vs csat
plot(sts_ex_sat)
```

## 2.3 Regression with continuous outcomes

- Ordinary least squares (OLS) regression models can be fit with the `lm()` function
- For example, we can use `lm` to predict SAT scores based on per-pupil expenditures:

```
# Fit our regression model
sat_mod <- lm(csat ~ expense, # regression formula
              data=states_data) # data

# Summarize and print the results
summary(sat_mod) # show regression coefficients table
```

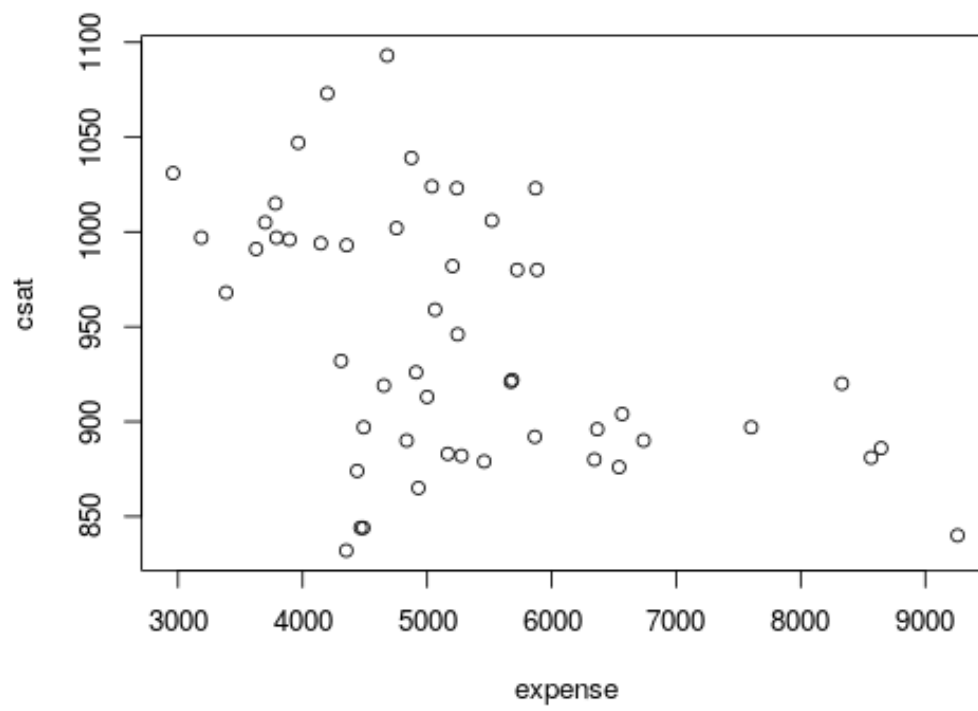


Figure 2.1



### 2.3.1 Why is the association between expense and SAT scores *negative*?

Many people find it surprising that the per-capita expenditure on students is negatively related to SAT scores. The beauty of multiple regression is that we can try to pull these apart. What would the association between expense and SAT scores be if there were no difference among the states in the percentage of students taking the SAT?

```
lm(csat ~ expense + percent, data = states_data) %>%
summary()
```

### 2.3.2 The `lm` class and methods

OK, we fit our model. Now what?

- Examine the model object:

```
class(sat_mod)
str(sat_mod)
names(sat_mod)
methods(class = class(sat_mod))
```

- Use function methods to get more information about the fit

```
summary(sat_mod)
summary(sat_mod) %>% coef()
methods("summary")
confint(sat_mod)
```

### 2.3.3 OLS regression assumptions

- OLS regression relies on several assumptions, including that the residuals are normally distributed and homoscedastic, the errors are independent and the relationships are linear.
- Investigate these assumptions visually by plotting your model:

```
par(mfrow = c(2, 2))
plot(sat_mod)
```

### 2.3.4 Comparing models

Do congressional voting patterns predict SAT scores over and above expense? Fit two models and compare them:

```

# fit another model, adding house and senate as predictors
sat_voting_mod <- lm(csat ~ expense + house + senate,
                     data = na.omit(states_data))

sat_mod <- update(sat_mod, data=na.omit(states_data))

# compare using the anova() function
anova(sat_mod, sat_voting_mod)
summary(sat_voting_mod) %>% coef()

```

## 2.4 Exercise 0

### Ordinary least squares regression

Use the *states.rds* data set. Fit a model predicting energy consumed per capita (energy) from the percentage of residents living in metropolitan areas (metro). Be sure to

1. Examine/plot the data before fitting the model
2. Print and interpret the model **summary**
3. plot the model to look for deviations from modeling assumptions

Select one or more additional predictors to add to your model and repeat steps 1-3. Is this model significantly better than the model with *metro* as the only predictor?

## 2.5 Interactions and factors

### 2.5.1 Modeling interactions

Interactions allow us assess the extent to which the association between one predictor and the outcome depends on a second predictor. For example: Does the association between expense and SAT scores depend on the median income in the state?

```

# Add the interaction to the model
sat_expense_by_percent <- lm(csat ~ expense + income + expense : income, data=states_data)
sat_expense_by_percent <- lm(csat ~ expense * income, data=states_data) # same as above, but s

# Show the regression coefficients table
summary(sat_expense_by_percent) %>% coef()

```

### 2.5.2 Regression with categorical predictors

Let's try to predict SAT scores from region, a categorical variable. Note that you must make sure R does not think your categorical variable is numeric.

```

# make sure R knows region is categorical
str(states_data$region)
states_data$region <- factor(states_data$region)

# Add region to the model
sat_region <- lm(csat ~ region, data=states_data)

# Show the results
summary(sat_region) %>% coef() # show the regression coefficients table
anova(sat_region) # show ANOVA table

```

Again, make sure to tell R which variables are categorical by converting them to factors!

### 2.5.3 Setting factor reference groups and contrasts

In the previous example we use the default contrasts for region. The default in R is treatment contrasts, with the first level as the reference. We can change the reference group or use another coding scheme using the `C` function.

```

# print default contrasts
contrasts(states_data$region)

# change the reference group
states_data$region <- relevel(states_data$region, ref = "Midwest")
m1 <- lm(csat ~ region, data=states_data)
summary(m1) %>% coef()

# get all pairwise contrasts between means
# install.packages("emmeans")
library(emmeans)
means <- emmeans(m1, specs = ~ region)
means
contrast(means, method = "pairwise")

# change the coding scheme
lm(csat ~ C(region, contr.helmert), data=states_data) %>%
summary() %>%
coef()

```

See `?contr.treatment` for other coding schemes and also `?contrasts` and `?relevel`.

## 2.6 Exercise 1

### Interactions and factors

Use the states data set.

1. Add on to the regression equation that you created in exercise 1 by generating an interaction term and testing the interaction.
2. Try adding region to the model. Are there significant differences across the four regions?

## 2.7 Regression with binary outcomes

### 2.7.1 Logistic regression

This far we have used the `lm` function to fit our regression models. `lm` is great, but limited—in particular it only fits models for continuous dependent variables. For categorical dependent variables we can use the `glm()` function.

For these models we will use a different dataset, drawn from the National Health Interview Survey. From the CDC website:

The National Health Interview Survey (NHIS) has monitored the health of the nation since 1957. NHIS data on a broad range of health topics are collected through personal household interviews. For over 50 years, the U.S. Census Bureau has been the data collection agent for the National Health Interview Survey. Survey results have been instrumental in providing data to track health status, health care access, and progress toward achieving national health objectives.

Load the National Health Interview Survey data:

```
NH11 <- read_rds("dataSets/NatHealth2011.rds")
```

### 2.7.2 Logistic regression example

Let's predict the probability of being diagnosed with hypertension based on age, sex, sleep, and bmi

```
str(NH11$hypev) # check structure of hypev
levels(NH11$hypev) # check levels of hypev

# collapse all missing values to NA
NH11$hypev <- factor(NH11$hypev, levels=c("2 No", "1 Yes"))

# run our regression model
hyp_out <- glm(hypev ~ age_p + sex + sleep + bmi,
               data = NH11, family = binomial(link = "logit"))
summary(hyp_out) %>% coef()
```

### 2.7.3 Logistic regression coefficients

Generalized linear models use link functions, so raw coefficients are difficult to interpret. For example, the age coefficient of .06 in the previous model tells us that for every one unit increase in age, the log odds of hypertension diagnosis increases by 0.06. Since most of us are not used to thinking in log odds this is not too helpful!

One solution is to transform the coefficients to make them easier to interpret

```
hyp_out_tab <- summary(hyp_out) %>% coef()
hyp_out_tab[, "Estimate"] <- coef(hyp_out) %>% exp()
hyp_out_tab
```

### 2.7.4 Packages for computing and graphing predicted values

Instead of doing all this ourselves, we can use the effects package to compute quantities of interest for us.

```
library(effects)
eff <- allEffects(hyp_out)
eff2 <- allEffects(hyp_out, xlevels = list(age_p, seq(20, 80, by = 5)))
plot(eff)
as.data.frame(eff) # confidence intervals
```

## 2.8 Exercise 2

### Logistic regression

Use the NH11 data set that we loaded earlier.

1. Use glm to conduct a logistic regression to predict ever worked (everwrk) using age (age\_p) and marital status (r\_maritl).
2. Predict the probability of working for each level of marital status.

Note that the data is not perfectly clean and ready to be modeled. You will need to clean up at least some of the variables before fitting the model.

## 2.9 Multilevel modeling

### 2.9.1 Multilevel modeling overview

- Multi-level (AKA hierarchical) models are a type of mixed-effects models

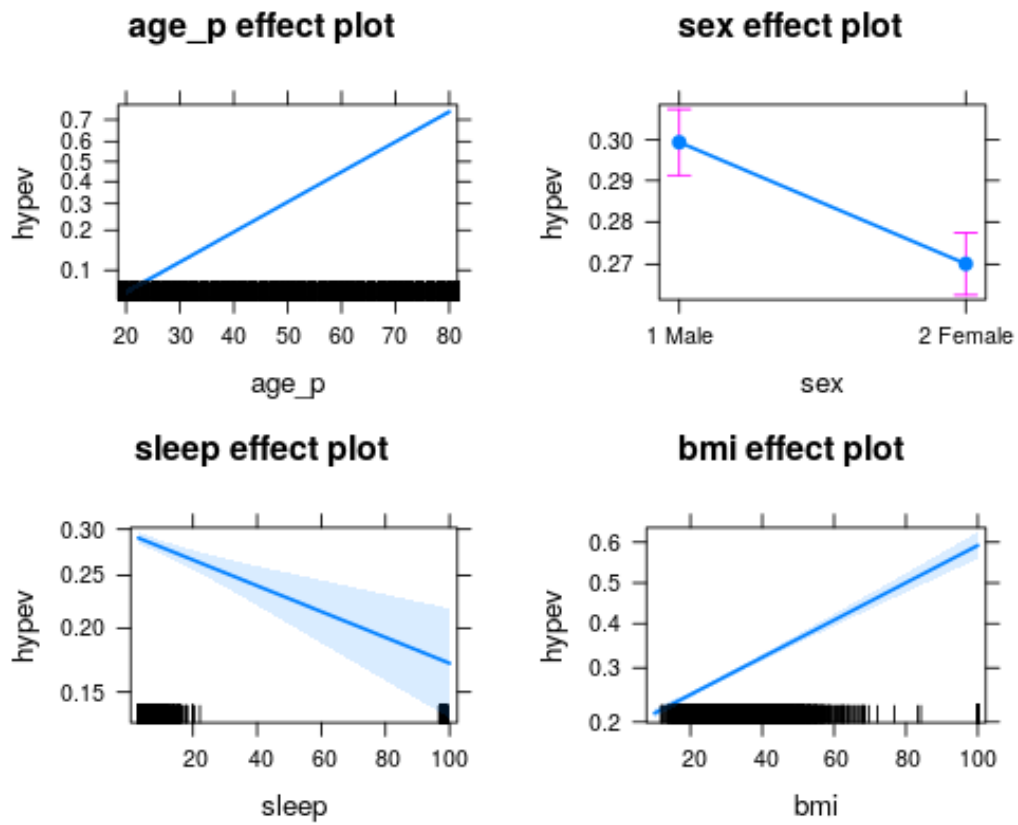


Figure 2.2

- Used to model variation due to group membership where the goal is to generalize to a population of groups
- Can model different intercepts and/or slopes for each group
- Mixed-effects models include two types of predictors: fixed-effects and random effects
- Fixed-effects – observed levels are of direct interest (.e.g, sex, political party...)
- Random-effects – observed levels not of direct interest: goal is to make inferences to a population represented by observed levels
- In R the lme4 package is the most popular for mixed effects models
- Use the `lmer` function for liner mixed models, `glmer` for generalized mixed models

```
library(lme4)
```

### 2.9.2 The Exam data

The Exam data set contains exam scores of 4,059 students from 65 schools in Inner London. The variable names are as follows:

Variable	Description
school	School ID - a factor.
normexam	Normalized exam score.
standLRT	Standardised LR test score.
student	Student id (within school) - a factor

```
Exam <- read_rds("dataSets/Exam.rds")
```

### 2.9.3 The null model and ICC

As a preliminary step it is often useful to partition the variance in the dependent variable into the various levels. This can be accomplished by running a null model (i.e., a model with a random effects grouping structure, but no fixed-effects predictors).

```
# null model, grouping by school but not fixed effects.
Norm1 <- lmer(normexam ~ 1 + (1 | school),
              data=na.omit(Exam), REML = FALSE)
summary(Norm1)
```

The is  $.169 / (.169 + .848) = .17$ : 17% of the variance is at the school level.

There is no consensus on how to calculate p-values for MLMs; hence why they are omitted from the `lme4` output. But, if you really need p-values, the `lmerTest` package will calculate p-values for you (using the Satterthwaite approximation) and you can use the same model syntax:

```
# install.packages("lmerTest")
Norm1_test <- lmerTest::lmer(normexam ~ 1 + (1 | school),
                             data=na.omit(Exam), REML = FALSE)
summary(Norm1_test)
```

### 2.9.4 Adding fixed-effects predictors

Predict exam scores from student's standardized tests scores

```
Norm2 <- lmer(normexam ~ 1 + standLRT + (1 | school),
              data=na.omit(Exam), REML = FALSE)
summary(Norm2)
```

### 2.9.5 Multiple degree of freedom comparisons

As with `lm` and `glm` models, you can compare the two `lmer` models using the `anova` function.

```
anova(Norm1, Norm2)
```

### 2.9.6 Random slopes

Add a random effect of students' standardized test scores as well. Now in addition to estimating the distribution of intercepts across schools, we also estimate the distribution of the slope of exam on standardized test.

```
Norm3 <- lmer(normexam ~ 1 + standLRT + (1 + standLRT | school),
              data = na.omit(Exam), REML = FALSE)
summary(Norm3)
```

### 2.9.7 Test the significance of the random slope

To test the significance of a random slope just compare models with and without the random slope term

```
anova(Norm2, Norm3)
```

## 2.10 Exercise 3

### Multilevel modeling

Use the dataset, `bh1996`:



```
data(bh1996, package="multilevel")
```

From the data documentation:

Variables are Leadership Climate (LEAD), Well-Being (WBEING), and Work Hours (HRS). The group identifier is named “GRP”.

1. Create a null model predicting wellbeing (“WBEING”)
2. Calculate the ICC for your null model
3. Run a second multi-level model that adds two individual-level predictors, average number of hours worked (“HRS”) and leadership skills (“LEAD”) to the model and interpret your output.
4. Now, add a random effect of average number of hours worked (“HRS”) to the model and interpret your output. Test the significance of this random term.

## 2.11 Exercise solutions

### 2.11.1 Ex 0: prototype

Use the *states.rds* data set.

```
states <- read_rds("dataSets/states.rds")
```

Fit a model predicting energy consumed per capita (energy) from the percentage of residents living in metropolitan areas (metro). Be sure to

1. Examine/plot the data before fitting the model

```
states_en_met <- subset(states, select = c("metro", "energy"))
summary(states_en_met)
plot(states_en_met)
cor(states_en_met, use="pairwise")
```

2. Print and interpret the model summary

```
mod_en_met <- lm(energy ~ metro, data = states)
summary(mod_en_met)
```

3. plot the model to look for deviations from modeling assumptions

```
plot(mod_en_met)
```

Select one or more additional predictors to add to your model and repeat steps 1-3. Is this model significantly better than the model with *metro* as the only predictor?

```
states_en_met_pop_wst <- subset(states, select = c("energy", "metro", "pop", "waste"))
summary(states_en_met_pop_wst)
plot(states_en_met_pop_wst)
cor(states_en_met_pop_wst, use = "pairwise")

mod_en_met_pop_waste <- lm(energy ~ metro + pop + waste, data = states)
summary(mod_en_met_pop_waste)
anova(mod_en_met, mod_en_met_pop_waste)
```

### 2.11.2 Ex 1: prototype

Use the states data set.

1. Add on to the regression equation that you created in exercise 1 by generating an interaction term and testing the interaction.

```
mod_en_metro_by_waste <- lm(energy ~ metro * waste, data = states)
```

2. Try adding a region to the model. Are there significant differences across the four regions?

```
mod_en_region <- lm(energy ~ metro * waste + region, data = states)
anova(mod_en_region)
```

### 2.11.3 Ex 2: prototype

Use the NH11 data set that we loaded earlier. Note that the data is not perfectly clean and ready to be modeled. You will need to clean up at least some of the variables before fitting the model.

1. Use glm to conduct a logistic regression to predict ever worked (*everwrk*) using age (*age\_p*) and marital status (*r\_maritl*).

```
nh11_wrk_age_mar <- subset(NH11, select = c("everwrk", "age_p", "r_maritl"))
summary(nh11_wrk_age_mar)

NH11 <- transform(NH11,
  everwrk = factor(everwrk, levels = c("1 Yes", "2 No")),
```

```

r_maritl = droplevels(r_maritl))

mod_wk_age_mar <- glm(everwrk ~ age_p + r_maritl, data = NH11,
                      family = binomial(link = "logit"))

summary(mod_wk_age_mar)

```

2. Predict the probability of working for each level of marital status.

```

library(effects)
data.frame(Effect("r_maritl", mod_wk_age_mar))

```

### 2.11.4 Ex 3: prototype

Use the dataset, bh1996:

```
data(bh1996, package="multilevel")
```

From the data documentation:

Variables are Cohesion (COHES), Leadership Climate (LEAD), Well-Being (WBEING) and Work Hours (HRS). The group identifier is named “GRP”.

1. Create a null model predicting wellbeing (“WBEING”)

```

library(lme4)
mod_grp0 <- lmer(WBEING ~ 1 + (1 | GRP), data = bh1996)
summary(mod_grp0)

```

3. Run a second multi-level model that adds two individual-level predictors, average number of hours worked (“HRS”) and leadership skills (“LEAD”) to the model and interpret your output.

```

mod_grp1 <- lmer(WBEING ~ HRS + LEAD + (1 | GRP), data = bh1996)
summary(mod_grp1)

```

3. Now, add a random effect of average number of hours worked (“HRS”) to the model and interpret your output. Test the significance of this random term.

```

mod_grp2 <- lmer(WBEING ~ HRS + LEAD + (1 + HRS | GRP), data = bh1996)
anova(mod_grp1, mod_grp2)

```

## 2.12 Wrap-up

### 2.12.1 Feedback

These workshops are a work in progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 2.12.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>
  - Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>
- HBS
  - Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>

## Chapter 3

# R Graphics

### Topics

- R `ggplot2` package
- Setup basic plots
- Add and modify scales and legends
- Manipulate plot labels
- Change and create plot themes

## 3.1 Setup

### 3.1.1 Software & materials

You should have R and RStudio installed — if not:

- Download and install R: <http://cran.r-project.org>
- Download and install RStudio: <https://www.rstudio.com/products/rstudio/download/#download>

Download materials:

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/R/Rgraphics.zip>
- Extract materials from the zipped directory `Rgraphics.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

Start RStudio and create a new project:

- On Windows click the start button and search for RStudio. On Mac RStudio will be in your applications folder.

- In Rstudio go to **File -> New Project**.
- Choose **Existing Directory** and browse to the **Rgraphics** directory.
- Choose **File -> Open File** and select the blank version of the **.Rmd** file.

While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called **tidyverse** have been designed specifically for data science. All packages included in **tidyverse** share an underlying design philosophy, grammar, and data structures. We will use **tidyverse** packages throughout the workshop, so let's install them now:

```
# install.packages("tidyverse")  
library(tidyverse)
```

### 3.1.2 Goals

Class Structure and Organization:

- Ask questions at any time. Really!
- Collaboration is encouraged - please spend a minute introducing yourself to your neighbors!

This is an intermediate R course:

- Assumes working knowledge of R
- Relatively fast-paced
- Focus is on **ggplot2** graphics; other packages will not be covered

### 3.1.3 Starting at the end

By the end of the workshop you will be able to reproduce this graphic from the Economist:

## 3.2 Why ggplot2?

**ggplot2** is a package within in the **tidyverse** suite of packages. Advantages of **ggplot2** include:

- consistent underlying **grammar of graphics** (Wilkinson, 2005)
- plot specification at a high level of abstraction
- very flexible
- theme system for polishing plot appearance
- mature and complete graphics system
- many users, active mailing list

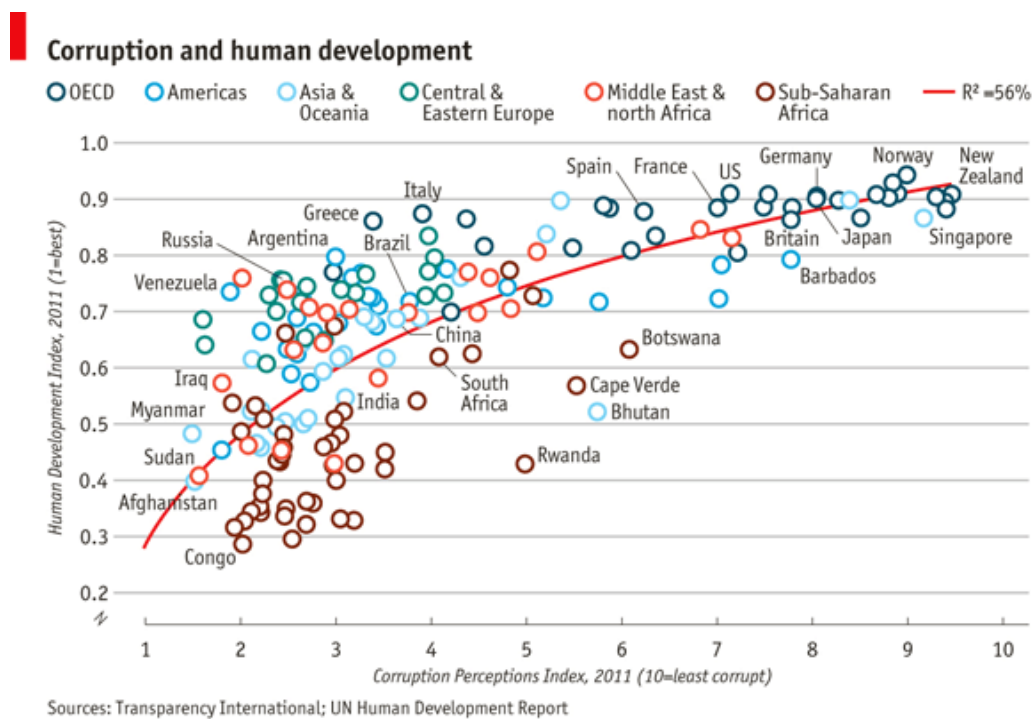


Figure 3.1: img

That said, there are some things you cannot (or should not) do with `ggplot2`:

- 3-dimensional graphics (see the `rgl` package)
- Graph-theory type graphs (nodes/edges layout; see the `igraph` package)
- Interactive graphics (see the `ggvis` package)

### 3.2.1 What is the Grammar Of Graphics?

The basic idea: independently specify plot building blocks and combine them to create just about any kind of graphical display you want. Building blocks of a graph include the following (**bold denotes essential elements**):

- **data**
- **aesthetic mapping**
- **geometric object**
- statistical transformations
- scales
- coordinate system
- position adjustments
- faceting
- themes

### 3.2.2 Example data: housing prices

Let's look at housing prices.

```
housing <- read_csv("dataSets/landdata-states.csv")
head(housing[1:5])
```

### 3.2.3 ggplot2 VS base graphics

Compared to base graphics, `ggplot2`

- is more verbose for simple / canned graphics
- is less verbose for complex / custom graphics
- does not have methods (data should always be in a `data.frame`)
- uses a different system for adding plot elements

### 3.2.4 For simple graphs

Base graphics histogram example:



```
hist(housing$Home_Value)
```

ggplot2 histogram example:

```
library(ggplot2)
ggplot(housing, aes(x = Home_Value)) +
  geom_histogram()
```

### 3.2.5 For more complex graphs

Base graphics colored scatter plot example:

```
plot(Home_Value ~ Date,
     col = factor(State),
     data = filter(housing, State %in% c("MA", "TX")))

legend("topleft",
      legend = c("MA", "TX"),
      col = c("black", "red"),
      pch = 1)
```

ggplot2 colored scatter plot example:

```
ggplot(filter(housing, State %in% c("MA", "TX")),
      aes(x=Date,
          y=Home_Value,
          color=State))+
  geom_point()
```

ggplot2 wins!

## 3.3 Geometric objects and aesthetics

### 3.3.1 Aesthetic mapping

In ggplot land *aesthetic* means “something you can see”. Examples include:

- position (i.e., on the x and y axes)
- color (“outside” color)
- fill (“inside” color)
- shape (of points)
- linetype
- size

Each type of geom accepts only a subset of all aesthetics; refer to the geom help pages to see what mappings each geom accepts. Aesthetic mappings are set with the `aes()` function.

### 3.3.2 Geometric objects (geom)

Geometric objects are the actual marks we put on a plot. Examples include:

- points (`geom_point()`, for scatter plots, dot plots, etc.)
- lines (`geom_line()`, for time series, trend lines, etc.)
- boxplot (`geom_boxplot()`, for boxplots!)

A plot **must have at least one geom**; there is no upper limit. You can add a geom to a plot using the `+` operator.

Each `geom_` has a particular set of aesthetic mappings associated with it. Some examples are provided below, with required aesthetics in **bold** and optional aesthetics in plain text:

geom_	Usage	Aesthetics
<code>geom_point()</code>	Scatter plot	<b>x,y</b> ,alpha,color,fill,group,shape,size,stroke
<code>geom_line()</code>	Line plot	<b>x,y</b> ,alpha,color,linetype,size
<code>geom_bar()</code>	Bar chart	<b>x,y</b> ,alpha,color,fill,group,linetype,size
<code>geom_boxplot()</code>	Boxplot	<b>x,lower,upper,middle,ymin,ymax</b> ,alpha,color,fill
<code>geom_density()</code>	Density plot	<b>x,y</b> ,alpha,color,fill,group,linetype,size,weight
<code>geom_smooth()</code>	Conditional means	<b>x,y</b> ,alpha,color,fill,group,linetype,size,weight
<code>geom_label()</code>	Text	<b>x,y,label</b> ,alpha,angle,color,family,fontface,size

You can get a list of all available geometric objects and their associated aesthetics at <https://ggplot2.tidyverse.org/reference/>

or simply type `geom_<tab>` in any good R IDE (such as Rstudio or ESS) to see a list of functions starting with `geom_`.

#### 3.3.2.1 Points (scatterplot)

Now that we know about geometric objects and aesthetic mapping, we can make a `ggplot()`. `geom_point()` requires mappings for `x` and `y`, all others are optional.

```
hp2001Q1 <- filter(housing, Date == 2001.25)
ggplot(hp2001Q1,
  aes(y = Structure.Cost, x = Land_Value)) +
  geom_point()
```

```
ggplot(hp2001Q1,
  aes(y = Structure.Cost, x = log(Land_Value))) +
  geom_point()
```

### 3.3.2.2 Lines (prediction line)

A plot constructed with `ggplot()` can have more than one geom. In that case the mappings established in the `ggplot()` call are plot defaults that can be added to or overridden. Our plot could use a regression line:

```
hp2001Q1$pred_SC <- lm(Structure_Cost ~ log(Land_Value), data = hp2001Q1) %>%
  predict()

p1 <- ggplot(hp2001Q1, aes(x = log(Land_Value), y = Structure_Cost))

p1 + geom_point(aes(color = Home_Value)) +
  geom_line(aes(y = pred_SC))
```

### 3.3.2.3 Smoothers

Not all geometric objects are simple shapes; the smooth geom includes a line and a ribbon.

```
p1 +
  geom_point(aes(color = Home_Value)) +
  geom_smooth()
```

### 3.3.2.4 Text (label points)

Each geom accepts a particular set of mappings; for example `geom_text()` accepts a `labels` mapping.

```
p1 +
  geom_text(aes(label=State), size = 3)

## install.packages("ggrepel")
library(ggrepel)

p1 +
  geom_point() +
  geom_text_repel(aes(label=State), size = 3)
```

## 3.3.3 Aesthetic mapping VS assignment

Note that variables are mapped to aesthetics with the `aes()` function, while fixed aesthetics are set outside the `aes()` call. This sometimes leads to confusion, as in this example:

```
p1 +
  geom_point(aes(size = 2), # incorrect! 2 is not a variable
            color="red") # this is fine -- all points red
```

### 3.3.4 Mapping variables to other aesthetics

Other aesthetics are mapped in the same way as x and y in the previous example.

```
p1 +  
  geom_point(aes(color = Home_Value, shape = region))
```

## 3.4 Exercise 0

The data for the exercises is available in the `dataSets/EconomistData.csv` file. Read it in with

```
dat <- read_csv("dataSets/EconomistData.csv")
```

Original sources for these data are <http://www.transparency.org/content/download/64476/1031428> [http://hdrstats.undp.org/en/indicators/display\\_cf\\_xls\\_indicator.cfm?indicator\\_id=103106&lang=en](http://hdrstats.undp.org/en/indicators/display_cf_xls_indicator.cfm?indicator_id=103106&lang=en)

These data consist of *Human Development Index* and *Corruption Perception Index* scores for several countries.

1. Create a scatter plot with CPI on the x axis and HDI on the y axis.
2. Color the points blue.
3. Map the color of the the points to Region.
4. Make the points bigger by setting size to 2
5. Map the size of the points to HDI\_Rank

## 3.5 Statistical transformations

### 3.5.1 Why transform data?

Some plot types (such as scatterplots) do not require transformations; each point is plotted at x and y coordinates equal to the original value. Other plots, such as boxplots, histograms, prediction lines etc. require statistical transformations:

- for a boxplot the y values must be transformed to the median and 1.5(IQR)
- for a smoother the y values must be transformed into predicted values

Each geom has a default statistic, but these can be changed. For example, the default statistic for `geom_histogram()` is `stat_bin()`:

```
args(geom_histogram)  
args(stat_bin)
```

Here is a list of geoms and their default statistics <https://ggplot2.tidyverse.org/reference/>

### 3.5.2 Setting arguments

Arguments to `stat_` functions can be passed through `geom_` functions. This can be slightly annoying because in order to change it you have to first determine which stat the geom uses, then determine the arguments to that stat.

For example, here is the default histogram of `Home.Value`:

```
p2 <- ggplot(housing, aes(x = Home_Value))
p2 + geom_histogram()
```

can change it by passing the `binwidth` argument to the `stat_bin()` function:

```
p2 + geom_histogram(stat = "bin", binwidth=4000)
```

### 3.5.3 Changing the transformation

Sometimes the default statistical transformation is not what you need. This is often the case with pre-summarized data:

```
housing_sum <-
  housing %>%
  group_by(State) %>%
  summarize(Home_Value_Mean = mean(Home_Value)) %>%
  ungroup()

rbind(head(housing_sum), tail(housing_sum))

ggplot(housing_sum, aes(x=State, y=Home_Value_Mean)) +
  geom_bar()
```

What is the problem with the previous plot? Basically we take binned and summarized data and ask ggplot to bin and summarize it again (remember, `geom_bar()` defaults to `stat = stat_count`; obviously this will not work. We can fix it by telling `geom_bar()` to use a different statistical transformation function:

```
ggplot(housing_sum, aes(x=State, y=Home_Value_Mean)) +
  geom_bar(stat="identity")
```

## 3.6 Exercise 1

1. Re-create a scatter plot with CPI on the x axis and HDI on the y axis (as you did in the previous exercise).
2. Overlay a smoothing line on top of the scatter plot using `geom_smooth()`.

3. Overlay a smoothing line on top of the scatter plot using `geom_smooth()`, but use a linear model for the predictions. Hint: see `?stat_smooth`.
4. Overlay a smoothing line on top of the scatter plot using the default *loess* method for `geom_smooth()`, but make it less smooth. Hint: see `?loess`.
5. BONUS: Overlay a smoothing line on top of the scatter plot using `geom_line()`. Hint: change the statistical transformation.

## 3.7 Scales

### 3.7.1 Controlling aesthetic mapping

Aesthetic mapping (i.e., with `aes()`) only says that a variable should be mapped to an aesthetic. It doesn't say *how* that should happen. For example, when mapping a variable to *shape* with `aes(shape = x)` you don't say *what* shapes should be used. Similarly, `aes(color = z)` doesn't say *what* colors should be used. Describing what colors/shapes/sizes etc. to use is done by modifying the corresponding *scale*. In `ggplot2` scales include

- position
- color and fill
- size
- shape
- line type

Scales are modified with a series of functions using a `scale_<aesthetic>_<type>` naming scheme. Try typing `scale_<tab>` to see a list of scale modification functions.

### 3.7.2 Common scale arguments

The following arguments are common to most scales in `ggplot2`:

- **name:** the first argument gives the axis or legend title
- **limits:** the minimum and maximum of the scale
- **breaks:** the points along the scale where labels should appear
- **labels:** the labels that appear at each break

Specific scale functions may have additional arguments; for example, the `scale_color_continuous()` function has arguments `low` and `high` for setting the colors at the low and high end of the scale.

### 3.7.3 Scale modification examples

Start by constructing a dotplot showing the distribution of home values by Date and State.

```
p4 <- ggplot(housing, aes(x = State, y = Home_Price_Index)) +
  geom_point(aes(color = Date), alpha = 0.5, size = 1.5,
    position = position_jitter(width = 0.25, height = 0)))
```

Now modify the breaks for the color scales

```
p4 +
  scale_color_continuous(name="",
    breaks = c(1976, 1994, 2013),
    labels = c("'76", "'94", "'13"))
```

Next change the low and high values to blue and red:

```
p4 +
  scale_color_continuous(name="",
    breaks = c(1976, 1994, 2013),
    labels = c("'76", "'94", "'13"),
    low = "blue", high = "red")
```

Now mute the colors:

```
library(scales)

p4 +
  scale_color_continuous(name="",
    breaks = c(1976, 1994, 2013),
    labels = c("'76", "'94", "'13"),
    low = muted("blue"), high = muted("red"))
```

### 3.7.4 Using different color scales

ggplot2 has a wide variety of color scales; here is an example using `scale_color_gradient2()` to interpolate between three different colors.

```
p4 +
  scale_color_gradient2(name="",
    breaks = c(1976, 1994, 2013),
    labels = c("'76", "'94", "'13"),
    low = muted("blue"),
    high = muted("red"),
    mid = "gray60",
    midpoint = 1994)
```

### 3.7.5 Available scales

- Partial combination matrix of available scales



scale_	Types	Examples
scale_color_	identity	scale_fill_continuous()
scale_fill_	manual	scale_color_discrete()
scale_size_	continuous	scale_size_manual()
	discrete	scale_size_discrete()
scale_shape_	discrete	scale_shape_discrete()
scale_linetype_	identity	scale_shape_manual()
	manual	scale_linetype_discrete()
scale_x_	continuous	scale_x_continuous()
scale_y_	discrete	scale_y_discrete()
	reverse	scale_x_log()
	log	scale_y_reverse()
	date	scale_x_date()
	datetime	scale_y_datetime()

Note that in RStudio you can type `scale_` followed by `tab` to get the whole list of available scales. For a complete list of available scales see <https://ggplot2.tidyverse.org/reference/>

## 3.8 Exercise 2

1. Create a scatter plot with CPI on the x axis and HDI on the y axis. Color the points to indicate region.
2. Modify the x, y, and color scales so that they have more easily-understood names (e.g., spell out “Human development Index” instead of “HDI”). Hint: see `?scale_x_discrete`.
3. Modify the color scale to use specific values of your choosing. Hint: see `?scale_color_manual`.

## 3.9 Faceting

### 3.9.1 What is faceting?

- Faceting is `ggplot2` parlance for **small multiples**
- The idea is to create separate graphs for subsets of data
- `ggplot2` offers two functions for creating small multiples:
  1. `facet_wrap()`: define subsets as the levels of a single grouping variable
  2. `facet_grid()`: define subsets as the crossing of two grouping variables
- Facilitates comparison among plots, not just of geoms within a plot

### 3.9.2 What is the trend in housing prices in each state?

- Start by using a technique we already know; map State to color:

```
p5 <- ggplot(housing, aes(x = Date, y = Home_Value))  
p5 + geom_line(aes(color = State))
```

There are two problems here; there are too many states to distinguish each one by color, and the lines obscure one another.

### 3.9.3 Faceting to the rescue

We can remedy the deficiencies of the previous plot by faceting by state rather than mapping state to color.

```
(p5 <- p5 + geom_line() +  
  facet_wrap(~ State, ncol = 10))
```

There is also a `facet_grid()` function for faceting in two dimensions.

## 3.10 Themes

### 3.10.1 What are themes?

The `ggplot2` theme system handles non-data plot elements such as

- Axis labels
- Plot background
- Facet label background
- Legend appearance

Built-in themes include:

- `theme_gray()` (default)
- `theme_bw()`
- `theme_classic()`

```
p5 + theme_linedraw()
```

```
p5 + theme_light()
```

You can see a list of available built-in themes here <https://ggplot2.tidyverse.org/reference/>

### 3.10.2 Overriding theme defaults

Specific theme elements can be overridden using `theme()`. For example:

```
p5 + theme_minimal() +
  theme(text = element_text(color = "turquoise"))
```

All theme options are documented in `?theme`.

### 3.10.3 Creating and saving new themes

You can create new themes, as in the following example:

```
theme_new <- theme_bw() +
  theme(plot.background = element_rect(size = 1, color = "blue", fill = "black"),
        text=element_text(size = 12, family = "Serif", color = "ivory"),
        axis.text.y = element_text(colour = "purple"),
        axis.text.x = element_text(colour = "red"),
        panel.background = element_rect(fill = "pink"),
        strip.background = element_rect(fill = muted("orange")))

p5 + theme_new
```

You can see all the plot elements that can be changed by `theme()` using:

```
names(theme_get())

# see all arguments for each plot element
theme_get()
```

## 3.11 The #1 FAQ

### 3.11.1 Map aesthetic to different columns

The most frequently asked question goes something like this: *I have two variables in my data.frame, and I'd like to plot them as separate points, with different color depending on which variable it is. How do I do that?*

**Wrong**

```
housing_byyear <-
  housing %>%
  group_by(Date) %>%
  summarize(Home_Value_Mean = mean(Home_Value),
```

```

      Land_Value_Mean = mean(Land_Value)) %>%
  ungroup()

ggplot(housing_byyear, aes(x=Date)) +
  geom_line(aes(y=Home_Value_Mean), color="red") +
  geom_line(aes(y=Land_Value_Mean), color="blue")

```

### Right

```

home_land_byyear <- gather(housing_byyear,
                           value = "value",
                           key = "type",
                           Home_Value, Land_Value)

ggplot(home_land_byyear, aes(x=Date, y=value, color=type)) +
  geom_line()

```

## 3.12 Putting it all together

### 3.12.1 Challenge: recreate this Economist graph

Graph source: <http://www.economist.com/node/21541178>

Building off of the graphics you created in the previous exercises, put the finishing touches to make it as close as possible to the original economist graph.

### 3.12.2 Challenge solution:

Lets start by creating the basic scatter plot, then we can make a list of things that need to be added or changed. The basic plot looks like this:

```

dat <- read_csv("dataSets/EconomistData.csv")

pc1 <- ggplot(dat, aes(x = CPI, y = HDI, color = Region))
pc1 + geom_point()

```

To complete this graph we need to:

- [ ] add a trend line
- [ ] change the point shape to open circle
- [ ] change the order and labels of Region
- [ ] label select points
- [ ] fix up the tick marks and labels
- [ ] move color legend to the top

- [ ] title, label axes, remove legend title
- [ ] theme the graph with no vertical guides
- [ ] add model R2 (hard)
- [ ] add sources note (hard)
- [ ] final touches to make it perfect (use image editor for this)

### 3.12.2.1 Adding the trend line

Adding the trend line is not too difficult, though we need to guess at the model being displayed on the graph. A little bit of trial and error leads to

```
pc2 <- pc1 +
  geom_smooth(mapping = aes(linetype = "r2"), # "r2" is a placeholder for where we'll later put
    method = "lm",
    formula = y ~ x + log(x), se = FALSE,
    color = "red")
pc2 + geom_point()
```

Notice that we put the `geom_line` layer first so that it will be plotted underneath the points, as was done on the original graph.

### 3.12.2.2 Use open points

This one is a little tricky. We know that we can change the shape with the `shape` argument, what value do we set `shape` to? The example shown in `?shape` can help us:

```
## A look at all 25 symbols
df2 <- data.frame(x = 1:5, y = 1:25, z = 1:25)

s <- ggplot(df2, aes(x = x, y = y))
s + geom_point(aes(shape = z), size = 4) + scale_shape_identity()
```

This shows us that *shape 1* is an open circle, so

```
pc2 +
  geom_point(shape = 1, size = 4)
```

That is better, but unfortunately the size of the line around the points is much narrower than on the original.

```
(pc3 <- pc2 + geom_point(shape = 1, size = 2.5, stroke = 1.25))
```

### 3.12.2.3 Labelling points

This one is tricky in a couple of ways. First, there is no attribute in the data that separates points that should be labelled from points that should not be. So the first step is to identify those points.

```
pointsToLabel <- c("Russia", "Venezuela", "Iraq", "Myanmar", "Sudan",
  "Afghanistan", "Congo", "Greece", "Argentina", "Brazil",
  "India", "Italy", "China", "South Africa", "Spain",
  "Botswana", "Cape Verde", "Bhutan", "Rwanda", "France",
  "United States", "Germany", "Britain", "Barbados", "Norway", "Japan",
  "New Zealand", "Singapore")
```

Now we can label these points using `geom_text`, like this:

```
(pc4 <- pc3 +
  geom_text(aes(label = Country),
    color = "gray20",
    data = filter(dat, Country %in% pointsToLabel)))
```

This more or less gets the information across, but the labels overlap in a most unpleasing fashion. We can use the `ggrepel` package to make things better, but if you want perfection you will probably have to do some hand-adjustment.

```
library(ggrepel)

(pc4 <- pc3 +
  geom_text_repel(aes(label = Country),
    color = "gray20",
    data = filter(dat, Country %in% pointsToLabel),
    force = 10))
```

### 3.12.2.4 Change the region labels and order

Things are starting to come together. There are just a couple more things we need to add, and then all that will be left are theme changes.

Comparing our graph to the original we notice that the labels and order of the Regions in the color legend differ. To correct this we need to change both the labels and order of the Region variable. We can do this with the `factor` function.

```
dat$Region <- factor(dat$Region,
  levels = c("EU W. Europe",
    "Americas",
    "Asia Pacific",
    "East EU Cemt Asia",
```

```

      "MENA",
      "SSA"),
  labels = c("OECD",
            "Americas",
            "Asia &\nOceania",
            "Central &\nEastern Europe",
            "Middle East &\nnorth Africa",
            "Sub-Saharan\nAfrica"))

```

Now when we construct the plot using these data the order should appear as it does in the original.

```

pc4$data <- dat
pc4

```

### 3.12.2.5 Add title and format axes

The next step is to add the title and format the axes. We do that using the `scales` system in `ggplot2`.

```

library(grid)

(pc5 <- pc4 +
  scale_x_continuous(name = "Corruption Perceptions Index, 2011 (10=least corrupt)",
                    limits = c(.9, 10.5),
                    breaks = 1:10) +
  scale_y_continuous(name = "Human Development Index, 2011 (1=Best)",
                    limits = c(0.2, 1.0),
                    breaks = seq(0.2, 1.0, by = 0.1)) +
  scale_color_manual(name = "",
                    values = c("#24576D",
                              "#099DD7",
                              "#28AADC",
                              "#248E84",
                              "#F2583F",
                              "#96503F")) +
  ggtitle("Corruption and Human development"))

```

### 3.12.2.6 Theme tweaks

Our graph is almost there. To finish up, we need to adjust some of the theme elements, and label the axes and legends. This part usually involves some trial and error as you figure out where things need to be positioned. To see what these various theme settings do you can change them and observe the results.

```
library(grid) # for the `unit()` function

(pc6 <- pc5 +
  theme_minimal() + # start with a minimal theme and add what we need
  theme(text = element_text(color = "gray20"),
        legend.position = c("top"), # position the legend in the upper left
        legend.direction = "horizontal",
        legend.justification = 0.1, # anchor point for legend.position.
        legend.text = element_text(size = 11, color = "gray10"),
        axis.text = element_text(face = "italic"),
        axis.title.x = element_text(vjust = -1), # move title away from axis
        axis.title.y = element_text(vjust = 2), # move away for axis
        axis.ticks.y = element_blank(), # element_blank() is how we remove elements
        axis.line = element_line(color = "gray40", size = 0.5),
        axis.line.y = element_blank(),
        panel.grid.major = element_line(color = "gray50", size = 0.5),
        panel.grid.major.x = element_blank()
  ))
```

### 3.12.2.7 Add model R2 and source note

The last bit of information that we want to have on the graph is the variance explained by the model represented by the trend line. Lets fit that model and pull out the R2 first, then think about how to get it onto the graph.

```
mR2 <- summary(lm(HDI ~ CPI + log(CPI), data = dat))$r.squared
mR2 <- paste0(format(mR2, digits = 2), "%")
```

OK, now that we've calculated the values, let's think about how to get them on the graph. ggplot2 has an `annotate()` function, but this is not convenient for adding elements outside the plot area. The `grid` package has nice functions for doing this, so we'll use those.

And here it is, our final version!

```
png(file = "R/Rgraphics/images/econScatter10.png", width = 700, height = 500)
p <- ggplot(dat,
  mapping = aes(x = CPI, y = HDI)) +
  geom_smooth(mapping = aes(linetype = "r2"),
    method = "lm",
    formula = y ~ x + log(x), se = FALSE,
    color = "red") +
  geom_point(mapping = aes(color = Region),
    shape = 1,
    size = 4,
    stroke = 1.5) +
  geom_text_repel(mapping = aes(label = Country, alpha = labels),
    color = "gray20",
```



```

data = transform(dat,
  labels = Country %in% c("Russia",
                          "Venezuela",
                          "Iraq",
                          "Mayanmar",
                          "Sudan",
                          "Afghanistan",
                          "Congo",
                          "Greece",
                          "Argentina",
                          "Italy",
                          "Brazil",
                          "India",
                          "China",
                          "South Africa",
                          "Spain",
                          "Cape Verde",
                          "Bhutan",
                          "Rwanda",
                          "France",
                          "Botswana",
                          "France",
                          "US",
                          "Germany",
                          "Britain",
                          "Barbados",
                          "Japan",
                          "Norway",
                          "New Zealand",
                          "Singapore"))) +
scale_x_continuous(name = "Corruption Perception Index, 2011 (10=least corrupt)",
  limits = c(1.0, 10.0),
  breaks = 1:10) +
scale_y_continuous(name = "Human Development Index, 2011 (1=best)",
  limits = c(0.2, 1.0),
  breaks = seq(0.2, 1.0, by = 0.1)) +
scale_color_manual(name = "",
  values = c("#24576D",
             "#099DD7",
             "#28AADC",
             "#248E84",
             "#F2583F",
             "#96503F"),
  guide = guide_legend(nrow = 1, order=1)) +
scale_alpha_discrete(range = c(0, 1),
  guide = FALSE) +
scale_linetype(name = "",

```

```

        breaks = "r2",
        labels = list(bquote(R^2==.(mR2))),
        guide = guide_legend(override.aes = list(linetype = 1, size = 2, color = "red"))
ggtitle("Corruption and human development") +
labs(caption="Sources: Transparency International; UN Human Development Report") +
theme_bw() +
theme(panel.border = element_blank(),
      panel.grid = element_blank(),
      panel.grid.major.y = element_line(color = "gray"),
      text = element_text(color = "gray20"),
      axis.title.x = element_text(face="italic"),
      axis.title.y = element_text(face="italic"),
      legend.position = "top",
      legend.direction = "horizontal",
      legend.box = "horizontal",
      legend.text = element_text(size = 12),
      plot.caption = element_text(hjust=0),
      plot.title = element_text(size = 16, face = "bold"))
p
dev.off()

```

Comparing it to the original suggests that we've got most of the important elements.

## 3.13 Exercise solutions

### 3.13.1 Ex 0: prototype

1. Create a scatter plot with CPI on the x axis and HDI on the y axis.

```

ggplot(dat, aes(x = CPI, y = HDI)) +
  geom_point()

```

2. Color the points in the previous plot blue.

```

ggplot(dat, aes(x = CPI, y = HDI)) +
  geom_point(color = "blue")

```

3. Color the points in the previous plot according to *Region*.

```

ggplot(dat, aes(x = CPI, y = HDI)) +
  geom_point(aes(color = Region))

```

4. Make the points bigger by setting size to 2

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point(aes(color = Region), size = 2)
```

5. Map the size of the points to HDI.Rank

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point(aes(color = Region, size = HDI_Rank))
```

### 3.13.2 Ex 1: prototype

1. Re-create a scatter plot with CPI on the x axis and HDI on the y axis (as you did in the previous exercise).

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point()
```

2. Overlay a smoothing line on top of the scatter plot using `geom_smooth`

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point() +  
  geom_smooth()
```

3. Overlay a smoothing line on top of the scatter plot using `geom_smooth`, but use a linear model for the predictions. Hint: see `?stat_smooth`.

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point() +  
  geom_smooth(method = "lm")
```

4. Overlay a loess (method = "loess") smoothing line on top of the scatter plot using `geom_line`. Hint: change the statistical transformation.

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point() +  
  geom_line(stat = "smooth", method = "loess")
```

4. BONUS: Overlay a smoothing line on top of the scatter plot using the *loess* method, but make it less smooth. Hint: see `?loess`.

```
ggplot(dat, aes(x = CPI, y = HDI)) +  
  geom_point() +  
  geom_smooth(span = .4)
```

### 3.13.3 Ex 2: prototype

1. Create a scatter plot with CPI on the x axis and HDI on the y axis. Color the points to indicate region.

```
ggplot(dat, aes(x = CPI, y = HDI, color = Region)) +  
  geom_point()
```

2. Modify the x, y, and color scales so that they have more easily-understood names (e.g., spell out “Human development Index” instead of “HDI”).

```
ggplot(dat, aes(x = CPI, y = HDI, color = Region)) +  
  geom_point() +  
  scale_x_continuous(name = "Corruption Perception Index") +  
  scale_y_continuous(name = "Human Development Index") +  
  scale_color_discrete(name = "Region of the world")
```

3. Modify the color scale to use specific values of your choosing. Hint: see `?scale_color_manual`.

```
ggplot(dat, aes(x = CPI, y = HDI, color = Region)) +  
  geom_point() +  
  scale_x_continuous(name = "Corruption Perception Index") +  
  scale_y_continuous(name = "Human Development Index") +  
  scale_color_manual(name = "Region of the world",  
                     values = c("#24576D",  
                                "#099DD7",  
                                "#28AADC",  
                                "#248E84",  
                                "#F2583F",  
                                "#96503F"))
```

## 3.14 Wrap-up

### 3.14.1 Feedback

These workshops are a work in progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 3.14.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>

- Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>
- HBS
  - Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>
- ggplot2
  - Reference: <https://ggplot2.tidyverse.org/reference/>
  - Cheatsheets: [https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets\\_2019.pdf](https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets_2019.pdf)
  - Mailing list: <http://groups.google.com/group/ggplot2>
  - Wiki: <https://github.com/hadley/ggplot2/wiki>
  - Website: <http://had.co.nz/ggplot2/>
  - StackOverflow: <http://stackoverflow.com/questions/tagged/ggplot>



## Chapter 4

# R Data Wrangling

### Topics

- Iterating over files
- Filtering with regular expressions (regex)
- Writing your own functions
- Reshaping data
- Loading Excel worksheets

## 4.1 Setup

### 4.1.1 Software & materials

You should have R and RStudio installed — if not:

- Download and install R: <http://cran.r-project.org>
- Download and install RStudio: <https://www.rstudio.com/products/rstudio/download/#download>

Download materials:

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/R/RDataWrangling.zip>
- Extract materials from the zipped directory `RDataWrangling.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

Start RStudio and create a new project:

- On Windows click the start button and search for RStudio. On Mac RStudio will be in your applications folder.

- In Rstudio go to **File -> New Project**.
- Choose **Existing Directory** and browse to the `RDataWrangling` directory.
- Choose **File -> Open File** and select the blank version of the `.Rmd` file.

While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called `tidyverse` have been designed specifically for data science. All packages included in `tidyverse` share an underlying design philosophy, grammar, and data structures. We will use `tidyverse` packages throughout the workshop, so let's install them now:

```
# install.packages("tidyverse")  
library(tidyverse)
```

### 4.1.2 Goals

Class Structure and Organization:

- Ask questions at any time. Really!
- Collaboration is encouraged - please spend a minute introducing yourself to your neighbors!

This is an intermediate R course:

- Assumes working knowledge of R
- Relatively fast-paced
- Data scientists are known and celebrated for modeling and visually displaying information, but down in the data science engine room there is a lot of less glamorous work to be done. Before data can be used effectively it must often be cleaned, corrected, and reformatted. This workshop introduces the basic tools needed to make your data behave, including data reshaping, regular expressions and other text manipulation tools.

## 4.2 Example project

It is common for data to be made available on a website somewhere, either by a government agency, research group, or other organizations and entities. Often the data you want is spread over many files, and retrieving it all one file at a time is tedious and time consuming. Such is the case with the baby names data we will be using today.

The UK Office for National Statistics provides yearly data on the most popular baby names going back to 1996. The data is provided separately for boys and girls and is stored in Excel spreadsheets.

I have downloaded all the excel files containing boys names data from <https://www.ons.gov.uk/peoplepopulationandcommunity/birthsdeathsandmarriages/livebirths/>



`datasets/babynamesenglandandwalesbabynamesstatisticsboys` and made them available at <http://tutorials.iq.harvard.edu/R/RDataManagement/data/boysNames.zip>.

Our mission is to extract and graph the **top 100** boys names in England and Wales for every year since 1996. There are several things that make this challenging.

### 4.2.1 Problems with the data

While it was good of the UK Office for National Statistics to provide baby name data, they were not very diligent about arranging it in a convenient or consistent format.

## 4.3 Exercise 0

Our mission is to extract and graph the **top 100** boys names in England and Wales for every year since 1996. There are several things that make this challenging.

1. Locate the file named `1996boys_tcm77-254026.xlsx` and open it in a spreadsheet. (If you don't have a spreadsheet program installed on your computer you can download one from <https://www.libreoffice.org/download/download/>). What issues can you identify that might make working with these data more difficult?
2. Locate the file named `2015boysnamesfinal.xlsx` and open it in a spreadsheet. In what ways is the format different than the format of `1996boys_tcm77-254026.xlsx`? How might these differences make it more difficult to work with these data?

## 4.4 Useful data manipulation packages

As you can see, the data is in quite a messy state. Note that this is not a contrived example; this is exactly the way the data came to us from the UK government website! Let's start cleaning and organizing it. The `tidyverse` suite of packages provides many modern conveniences that will make this job easier.

```
library(tidyverse)
```

## 4.5 Working with Excel worksheets

Each Excel file contains a worksheet with the baby names data we want. Each file also contains additional supplemental worksheets that we are not currently interested in. As noted above, the worksheet of interest differs from year to year, but always has "Table 1" in the sheet name.

The first step is to get a vector of file names.

```
boy.file.names <- list.files("dataSets/boys", full.names = TRUE)
```

Now that we’ve told R the names of the data files we can start working with them. For example, the first file is

```
boy.file.names[[1]]
```

and we can use the `excel_sheets` function from the *readxl* package to list the worksheet names from this file.

```
library(readxl)

excel_sheets(boy.file.names[[1]])
```

#### 4.5.1 Iterating over file names with `map`

Now that we know how to retrieve the names of the worksheets in an Excel file we could start writing code to extract the sheet names from each file, e.g.,

```
excel_sheets(boy.file.names[[1]])
excel_sheets(boy.file.names[[2]])
## ...
excel_sheets(boy.file.names[[20]])
```

This is not a terrible idea for a small number of files, but it is more convenient to let R do the iteration for us. We could use a `for` loop, or `sapply`, but the `map` family of functions from the *purrr* package gives us a more consistent alternative, so we’ll use that.

```
library(purrr)
map(boy.file.names, excel_sheets)
```

#### 4.5.2 Filtering strings using regular expressions

In order to extract the correct worksheet names we need a way to extract strings containing “Table 1”. Base R provides some string manipulation capabilities (see `?regex`, `?sub` and `?grep`), but we will use the *stringr* package because it is more user-friendly.

The *stringr* package provides functions to *detect*, *locate*, *extract*, *match*, *replace*, *combine* and *split* strings (among other things).

Here we want to detect the pattern “Table 1”, and only return elements with this pattern. We can do that using the `str_subset` function. The first argument to `str_subset` is character vector we want to search in. The second argument is a *regular expression* matching the pattern we want to retain.

If you are not familiar with regular expressions, <http://www.regexr.com/> is a good place to start.

Now that we know how to filter character vectors using `str_subset` we can identify the correct sheet in a particular Excel file. For example,

```
library(stringr)
str_subset(excel_sheets(boy.file.names[[1]]), "Table 1")
```

### 4.5.3 Writing your own functions

The `map*` functions are useful when you want to apply a function to a list or vector of inputs and obtain the return values. This is very convenient when a function already exists that does exactly what you want. In the examples above we mapped the `excel_sheets` function to the elements of a vector containing file names. But now there is no function that both retrieves worksheet names and subsets them. Fortunately, writing functions in R is easy.

```
get.data.sheet.name <- function(file, pattern) {
  str_subset(excel_sheets(file), pattern)
}
```

Now we can map this new function over our vector of file names.

```
map(boy.file.names,
    get.data.sheet.name,
    pattern = "Table 1")
```

## 4.6 Reading Excel data files

Now that we know the correct worksheet from each file we can actually read those data into R. We can do that using the `read_excel` function.

We'll start by reading the data from the first file, just to check that it works. Recall that the actual data starts on row 7, so we want to skip the first 6 rows.

```
tmp <- read_excel(
  boy.file.names[1],
  sheet = get.data.sheet.name(boy.file.names[1],
                              pattern = "Table 1"),
  skip = 6)

library(dplyr, quietly=TRUE)
glimpse(tmp)
```

## 4.7 Exercise 1

1. Write a function that takes a file name as an argument and reads the worksheet containing “Table 1” from that file. Don’t forget to skip the first 6 rows.
2. Test your function by using it to read *one* of the boys names Excel files.
3. Use the `map` function to read data from all the Excel files, using the function you wrote in step 1.

## 4.8 Data cleanup

Now that we’ve read in the data we still have some cleanup to do. Specifically, we need to:

1. fix column names
2. get rid of blank row and the top and the notes at the bottom
3. get rid of extraneous “changes in rank” columns if they exist
4. transform the side-by-side tables layout to a single table.

In short, we want to go from this:

to this:

There are many ways to do this kind of data manipulation in R. We’re going to use the *dplyr* and *tidyr* packages to make our lives easier. (Both packages were installed as dependencies of the *tidyverse* package.)

### 4.8.1 Selecting columns

Next we want to retain just the `Name`, `Name__1` and `Count`, `Count__1` columns. We can do that using the `select` function:

```
boysNames[[1]]  
  
boysNames[[1]] <- select(boysNames[[1]], Name, Name__1, Count, Count__1)  
boysNames[[1]]
```

### 4.8.2 Dropping missing values

Next we want to remove blank rows and rows used for notes. An easy way to do that is to use `drop_na` to remove rows with missing values.

```
boysNames[[1]]  
  
boysNames[[1]] <- drop_na(boysNames[[1]])  
boysNames[[1]]
```

Rank	Name	Count	since 2014	since 2005	Rank	Name	Count	since 2014	since 2005
6	CHARLIE	4,831	-1	+6	56	LEWIS	1,148	-10	-37
7	NOAH	4,148	+4	+44	57	FRANKIE	1,112	+7	+93*
8	WILLIAM	4,083	+2		58	LUKE	1,095	-14	-45
9	THOMAS	4,075	-3	-6	59	STANLEY	1,078	+1	+85*
10	OSCAR	4,066	-2	+45	60	TOMMY	1,075	-5	+63*
11	JAMES	3,912	-2	-7	61	JUDE	1,040	+4	+42*
12	MUHAMMAD	3,730	+2	+40	62	BLAKE	1,024	-5	+79*
13	HENRY	3,581	+2	+31	63	LOUIE	1,002	+4	+44*
14	ALFIE	3,540	-2	+9	64	NATHAN	997	-2	-29
15	LEO	3,468	+1	+22	65	GABRIEL	989	+13	+31
16	JOSHUA	3,394	-3	-14	66	CHARLES	985	-3	-17
17	FREDDIE	3,219	+3	+62	67	BOBBY	983	+4	+45*
18	ETHAN	2,940			68	MOHAMMAD	976	-12	
19	ARCHIE	2,912	-2	+19	69	RYAN	955		-44
20	ISAAC	2,829	+5	+33	70	TYLER	948	-23	-41
21	JOSEPH	2,786	-2	-11	71	ELLIOTT	938	+1	+54*
22	ALEXANDER	2,759			72	ALBERT	933	+12	+142*
23	SAMUEL	2,705	-2	-16	73	ELLIOT	926	+10	+9
24	DANIEL	2,622		-18	74	RORY	912	+13	+68*
25	LOGAN	2,610	-2	+52	75	ALEX	900		-22
26	EDWARD	2,593	+5	+20	76	FREDERICK	875	+5	+22
27	LUCAS	2,448	+3	+31	77	OLLIE	873	-3	+152*
28	MAX	2,407	-2	+3	78	LOUIS	854	-10	-35
29	MOHAMMED	2,332	-2	-9	79	DEXTER	850	-6	+216*
30	BENJAMIN	2,328	-2	-19	80	JAXON	837	+35*	+1055*
31	MASON	2,263	-2	+29	81	LIAM	836	-5	-53
32	HARRISON	2,241		+7	82	JACKSON	818	+18	+154*
33	THEO	2,103	+4	+85*	83	CALLUM	798	-1	-69
34	JAKE	2,013	-1	-18	83	RONNIE	798	+3	+77*
35	SEBASTIAN	1,988	+3	+54	85	LEON	795		-10
36	FINLEY	1,978		+28	86	KAI	775	-9	-20
37	ARTHUR	1,966	+4	+98*	87	AARON	773	-7	-42
38	ADAM	1,903	+1	-12	88	ROMAN	763	+22*	+109*
38	DYLAN	1,903	-4	-14	89	AUSTIN	751		+171*
40	RILEY	1,728	-5	+34	90	ELLIS	721	+4	-3
41	ZACHARY	1,644	-1	+54	91	JAMIE	708	-3	-58
42	TEDDY	1,430	+24	+226*	91	REGGIE	708	+18*	+201*
43	DAVID	1,394	-7	+18	93	SETH	703	-3	+88*
44	TOBY	1,363	-2	+4	94	CARTER	689	+24*	+182*
45	THEODORE	1,302	+14	+113*	95	FELIX	680	+3	+48*
46	ELUAH	1,294	+7	+109*	96	IBRAHIM	674	-5	+32*
47	MATTHEW	1,279	+2	-32	97	SONNY	670	-2	+17*
48	JENSON	1,223	+13	+127*	98	KIAN	665	-44	-35
49	JAYDEN	1,219	-6	+35	99	CALEB	659	-6	+32*
50	HARVEY	1,190	-2	-23	100	CONNOR	642	-21	-70

Notes:

These rankings have been produced using the exact spelling of the name given at birth registration. Similar names with different spellings have been counted separately.

Births where the name was not stated have been excluded from these figures. Of the 358,136 baby boys in the 2015 dataset, 14 were excluded for this reason.

The sum of the counts for individual names appearing in Table 2 and Table 3 may not equal the count in Table 1. This is because births where the usual residence of mother was not stated at the time of registration have been excluded from the counts in Table 2 and Table 3.

\* denotes new entry to top 100

Figure 4.1: messy

Rank	Name	Count
1	OLIVER	6,941
2	JACK	5,371
3	HARRY	5,308
4	GEORGE	4,869
5	JACOB	4,850
6	CHARLIE	4,831
7	NOAH	4,148
8	WILLIAM	4,083
9	THOMAS	4,075
10	OSCAR	4,066
11	JAMES	3,912
12	MUHAMMAD	3,730
13	HENRY	3,581
14	ALFIE	3,540
15	LEO	3,468
16	JOSHUA	3,394
17	FREDDIE	3,219
18	ETHAN	2,940
19	ARCHIE	2,912
20	ISAAC	2,829
21	JOSEPH	2,786
22	ALEXANDER	2,759
23	SAMUEL	2,705
24	DANIEL	2,622
25	LOGAN	2,610
26	EDWARD	2,593
27	LUCAS	2,448
28	MAX	2,407
29	MOHAMMED	2,332
30	BENJAMIN	2,328
31	MASON	2,263
32	HARRISON	2,241
33	THEO	2,103
34	JAKE	2,013
35	SEBASTIAN	1,988
36	FINLEY	1,978
37	ARTHUR	1,966
38	ADAM	1,903
38	DYLAN	1,903
40	RILEY	1,728
41	ZACHARY	1,644
42	TEDDY	1,430
43	DAVID	1,394
44	TOBY	1,363
45	THEODORE	1,302
46	ELIJAH	1,294
47	MATTHEW	1,279
48	JENSON	1,223
49	JAYDEN	1,219
50	HARVEY	1,190
51	REUBEN	1,188
52	HARLEY	1,175
53	LUCA	1,167
54	MICHAEL	1,165
55	HUGO	1,153
56	LEWIS	1,148
57	FRANKIE	1,110

Figure 4.2: tidy

Finally, we will want to filter out missing do this for all the elements in `boysNames`, a task I leave to you.

## 4.9 Exercise 2

1. Write a function that takes a `data.frame` as an argument and returns a modified version including only columns named `Name`, `Name__1`, `Count`, or `Count__1`.
2. Test your function on the first `data.frame` in the list of baby names data.
3. Use the `map` function to each `data.frame` in the list of baby names data.

### 4.9.1 Re-arranging into a single table

Our final task is to re-arrange to data so that it is all in a single table instead of in two side-by-side tables. For many similar tasks the `gather` function in the *tidyr* package is useful, but in this case we will be better off using a combination of `select` and `bind_rows`.

```
boysNames[[1]]
bind_rows(select(boysNames[[1]], Name, Count),
          select(boysNames[[1]], Name = Name__1, Count = Count__1))
```

## 4.10 Exercise 3

### Cleanup all the data

In the previous examples we learned how to drop empty rows with `filter`, select only relevant columns with `select`, and re-arrange our data with `select` and `bind_rows`. In each case we applied the changes only to the first element of our `boysNames` list.

Your task now is to use the `map` function to apply each of these transformations to all the elements in `boysNames`.

## 4.11 Data organization and storage

Now that we have the data cleaned up and augmented, we can turn our attention to organizing and storing the data.

### 4.11.1 One table for each year

Right now we have a list of tables, one for each year. This is not a bad way to go. It has the advantage of making it easy to work with individual years; it has the disadvantage of making it more difficult to examine questions that require data from multiple years. To

make the arrangement of the data clearer it helps to name each element of the list with the year it corresponds too.

```
glimpse(head(boysNames))

years <- str_extract(boy.file.names, "[0-9]{4}")
boysNames <- setNames(boysNames, years)
glimpse(head(boysNames))
```

### 4.11.2 One big table

While storing the data in separate tables by year makes some sense, many operations will be easier if the data is simply stored in one big table. We've already seen how to turn a list of data.frames into a single data.frame using `bind_rows`, but there is a problem; The year information is stored in the names of the list elements, and so flattening the tables into one will result in losing the year information! Fortunately it is not too much trouble to add the year information to each table before flattening.

```
boysNames <- imap(boysNames,
  function(data, name) {
    mutate(data, Year = as.integer(name))
  })
boysNames <- bind_rows(boysNames)

glimpse(boysNames)
```

## 4.12 Exercise 4

### Make one big table

Turn the list of boys names data.frames into a single table.

Create a directory under `data/all` and write the data to a `.csv` file.

Finally, repeat the previous exercise, this time working with the data in one big table.

## 4.13 Exercise solutions

### 4.13.1 Ex 0: prototype

1. Locate the file named `1996boys_tcm77-254026.xlsx` and open it in a spreadsheet. (If you don't have a spreadsheet program installed on your computer you can download one from <https://www.libreoffice.org/download/download/>). What issues can you identify that might make working with these data more difficult?



The data does not start on row one. Headers are on row 7, followed by a blank line, followed by the actual data.

The data is stored in an inconvenient way, with ranks 1-50 in the first set of columns and ranks 51-100 in a separate set of columns.

There are notes below the data.

3. Locate the file named `2015boysnamesfinal.xlsx` and open it in a spreadsheet. In what ways is the format different than the format of `1996boys_tcm77-254026.xlsx`? How might these differences make it more difficult to work with these data?

The worksheet containing the data of interest is in different positions and has different names from one year to the next. However, it always includes “Table 1” in the worksheet name.

Some years include columns for “changes in rank”, others do not.

These differences will make it more difficult to automate re-arranging the data since we have to write code that can handle different input formats.

### 4.13.2 Ex 1: prototype

```
## 1. Write a function that takes a file name as an argument and reads
##    the worksheet containing "Table 1" from that file.
read.baby.names <- function(file) {
  sheet.name <- str_subset(excel_sheets(file), "Table 1")
  read_excel(file, sheet = sheet.name, skip = 6)
}

## 2. Test your function by using it to read *one* of the boys names
##    Excel files.
glimpse(read.baby.names(boy.file.names[1]))

## 3. Use the `map` function to read data from all the Excel files,
##    using the function you wrote in step 1.
boysNames <- map(boy.file.names, read.baby.names)
```

### 4.13.3 Ex 2: prototype

```
## 1. Write a function that takes a `data.frame` as an argument and
##    returns a modified version including only columns named `Name`,
##    `Name__1`, `Count`, or `Count__1`.
```

```

namecount <- function(data) {
  select(data, Name, Name__1, Count, Count__1)
}

## 2. Test your function on the first `data.frame` in the list of baby
##    names data.

namecount(boysNames[[1]])

## 3. Use the `map` function to each `data.frame` in the list of baby
##    names data.

babyNames <- map(boysNames, namecount)

```

#### 4.13.4 Ex 3: prototype

There are different ways you can go about it. Here is one:

```

## write a function that does all the cleanup
cleanupNamesData <- function(x) {
  filtered <- filter(x, !is.na(Name)) # drop rows with no Name value
  selected <- select(filtered, Name, Count, Name__1, Count__1) # select just Name and Count columns
  bind_rows(select(selected, Name, Count), # re-arrange into two columns
            select(selected, Name = Name__1, Count = Count__1))
}

## test it out on the second data.frame in the list
glimpse(boysNames[[2]]) # before cleanup
glimpse(cleanupNamesData(boysNames[[2]])) # after cleanup

## apply the cleanup function to all the data.frames in the list
boysNames <- map(boysNames, cleanupNamesData)

```

#### 4.13.5 Ex 4: prototype

Working with the data in one big table is often easier.

```

boysNames <- bind_rows(boysNames)

dir.create("data/all")

write_csv(boysNames, "data/all/boys_names.csv")

## What were the five most popular names in 2013?
slice(arrange(filter(boysNames, Year == 2013),

```

```

      desc(Count)),
    1:5)

## How has the popularity of the name "ANDREW" changed over time?
andrew <- filter(boysNames, Name == "ANDREW")

ggplot(andrew, aes(x = Year, y = Count)) +
  geom_line() +
  ggtitle("Popularity of \"Andrew\", over time")

```

## 4.14 Wrap-up

### 4.14.1 Feedback

These workshops are a work in progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 4.14.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>
  - Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>
- HBS
  - Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>
- R
  - Learn from the best: <http://adv-r.had.co.nz/>; <http://r4ds.had.co.nz/>
  - R documentation: <http://cran.r-project.org/manuals.html>
  - Collection of R tutorials: <http://cran.r-project.org/other-docs.html>
  - R for Programmers (by Norman Matloff, UC-Davis) <http://heather.cs.ucdavis.edu/~matloff/R/RProg.pdf>
  - Calling C and Fortran from R (by Charles Geyer, UMinn) <http://www.stat.umn.edu/~charlie/rc/>
  - State of the Art in Parallel Computing with R (Schmidberger et al.) <http://www.jstatsoft.org/v31/i01/paper>



# Part II

# Python



## Chapter 5

# Python Introduction

### Topics

- Reading data
- Basic functions
- Finding help
- Indexing data objects
- Working with text data
- Conditional operations
- Iterating over data structures
- Lists and dictionaries

## 5.1 Setup

### 5.1.1 Software & Materials

#### 5.1.1.1 Install the Anaconda Python distribution

If using your own computer please install the Anaconda Python distribution from <https://www.anaconda.com/download/>. (Note that Python version  $\leq 3.0$  differs considerably from more recent releases. For this workshop you will need version  $\geq 3.6.x$ )

Accepting the defaults proposed by the Anaconda installer is generally recommended.

#### 5.1.1.2 Download workshop materials

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/Python/PythonIntro.zip>
- Extract materials from the zipped directory `PythonIntro.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

### 5.1.1.3 Launch Jupyter Notebook

Start the **Anaconda Navigator** program in the usual way. Click the or **Launch** button under **Jupyter Notebook**.

## 5.1.2 Goals

In this workshop you will \* learn about the python package and application ecosystem, \* learn python language basics and common idioms, and, \* practice reading files and manipulating data in python.

A more general goal is to get you comfortable with Python so that it seems less scary and mystifying than it perhaps does now. Note that this is by no means a complete or thorough introduction to Python! It's just enough to get by.

This workshop is relatively *informal*, *example-oriented*, and *hands-on*. We won't spend much time examining language features in detail. Instead we will work through an example, and learn some things about the language along the way.

As an example project we will analyze the text of Lewis Carroll's *Alice's Adventures in Wonderland*. Among the questions we will use Python to answer are: \* How many total and unique words are there? \* How many chapters and paragraphs? \* How many words are in each chapter, and what is the average words per chapter? \* How many times is each main character mentioned?

## 5.2 What is Python?

Python is a relatively easy to learn general purpose programming language. People use Python to manipulate, analyze, and visualize data, make web sites, write games, and much more. Youtube, DropBox, and BitTorrent are among the things people used python to make.

Like most popular open source programming languages, Python can be thought of as a *platform* that runs a huge number and variety of packages. The language itself is mostly valuable because it makes it easy to create and use a large number of useful packages.

## 5.3 How can I interact with Python?

A number of interfaces designed to make it easy to interact with Python are available. The Anaconda distribution that we installed earlier includes both a web-based *Jupyter Notebook* and a more conventional Integrated Development Environment called *Spyder*. For this workshop I encourage you to use *Jupyter Notebook*. In real life you should experiment and choose the interface that you find most comfortable.

To get started, start the *Jupyter Notebook* application, and navigate to the *PythonIntro* directory you downloaded and extracted earlier. Start a new notebook by clicking **New => Python 3** as shown below.



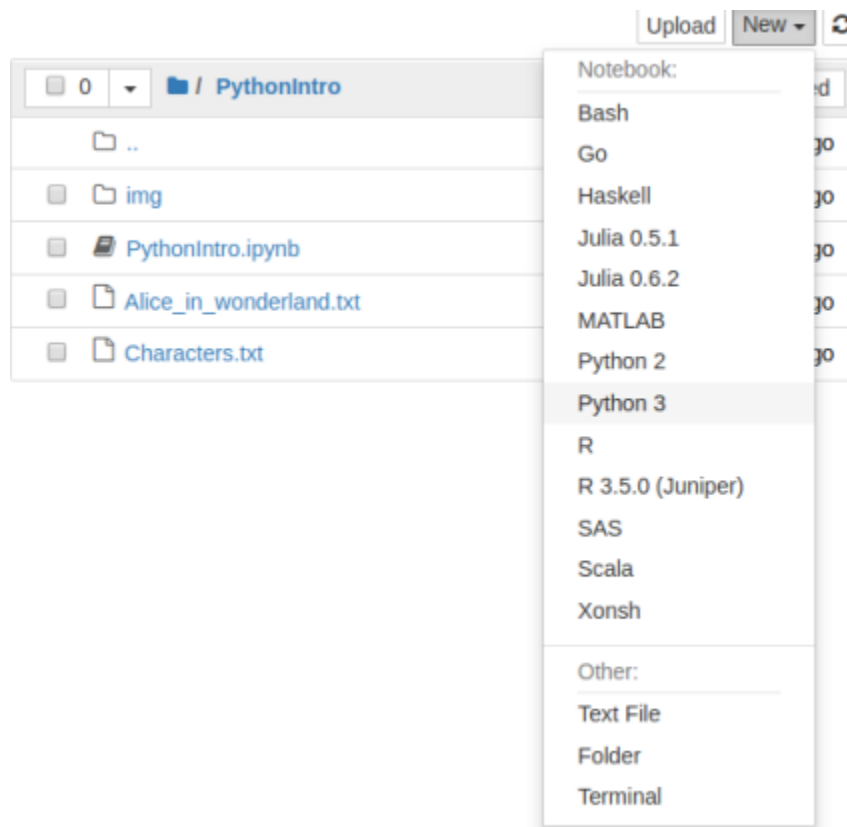


Figure 5.1: notebook\_new.png

A Jupyter Notebook contains one or more *cells* containing notes or code. To insert a new cell click the + button in the upper left. To execute a cell, select it and press **Control+Enter** or click the **Run** button at the top.

## 5.4 Reading the text of Alice in Wonderland from a file

Reading information from a file is the first step in many projects, so we'll start there. The workshop materials you downloaded earlier include a file named `Alice_in_wonderland.txt` which contains the text of Lewis Carroll's *Alice's Adventures in Wonderland*.

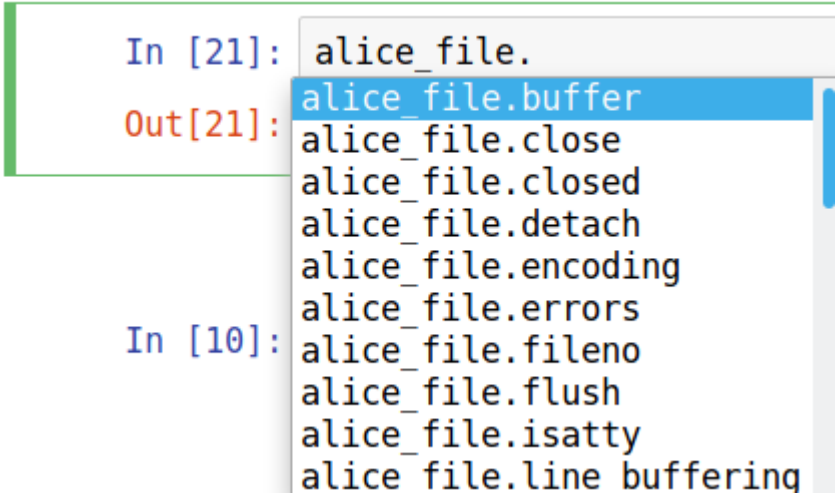
We can open a connection to a file using the `open` function, and store the result using the `=` operator.

```
alice_file = open("Alice_in_wonderland.txt")
```

The name on the left of the equals sign (`alice_file`) is one that we chose. When choosing names, *start with a letter*, and use only *letters, numbers and underscores*.

The `alice_file` object we just created does *not* contain the contents of `Alice_in_wonderland.txt`. It is a representation in Python of the *file itself* rather than the *contents* of the file.

The `alice_file` object provides *methods* that we can use to do things with it. Methods are invoked using syntax that looks like `ObjectName.method()`. You can see the methods available for acting on an object by typing the object's name followed by a `.` and pressing the `tab` key. For example, typing `alice_file.` and pressing `tab` will display a list of methods



The screenshot shows a Jupyter Notebook interface. On the left, there are three input prompts: `In [21]:`, `Out[21]:`, and `In [10]:`. The `In [21]:` prompt is followed by `alice_file.`. The `Out[21]:` prompt is followed by a list of methods: `alice_file.buffer`, `alice_file.close`, `alice_file.closed`, `alice_file.detach`, `alice_file.encoding`, `alice_file.errors`, `alice_file.fileno`, `alice_file.flush`, `alice_file.isatty`, and `alice file.line buffering`. The `In [10]:` prompt is followed by `alice_file.`. The list of methods is displayed in a blue box, and the `alice_file.buffer` method is highlighted in blue.

as shown below.

Among the methods we have for doing things with our `alice_file` object is one named `read`. We can use the `help` function to learn more about it.

```
help(alice_file.read)
```

Since `alice_file.read` looks promising, we will invoke this method and see what it does.

```
alice_txt = alice_file.read()
print(alice_txt[:500]) # the [:500] gets the first 500 character -- more on this later.
```

That's all there is to it! We've read the contents of `Alice_in_wonderland.txt` and stored this text in a Python object we named `alice_txt`. Now let's start to explore this object, and learn some more things about Python along the way.

## 5.5 Counting chapters, lines, and words

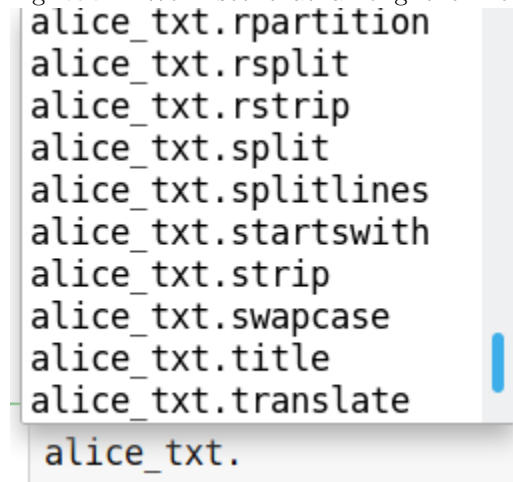
Now that we have the text we can start answering some questions about it. To begin with, how many words does it contain? To answer this question we can split the text up so there is one element per word, and then count the number of words.

### 5.5.1 Splitting a string into a list of words

How do we figure out how to split strings in Python? By asking Python what our `alice_txt` object is and what methods it provides. We can ask Python what things are using the `type` function, like this:

```
type(alice_txt)
```

Python tells us that `alice_txt` is of type `str` (i.e., it is a string). We can find out what methods are available for working strings by typing `alice_txt.` and pressing tab. We'll see that among the methods is one named `split`, as shown below.



```
alice_txt.rpartition
alice_txt.rsplitleft
alice_txt.rstrip
alice_txt.split
alice_txt.splitlines
alice_txt.startswith
alice_txt.strip
alice_txt.swapcase
alice_txt.title
alice_txt.translate
alice_txt.
```

To learn how to use this method we can check the documentation.

```
help(alice_txt.split)
```

Since the default is to split on whitespace (spaces, newlines, tabs) we can get a reasonable word count simply by calling the `split` method and counting the number of elements in the result.

```
alice_words = alice_txt.split()
len(alice_words)
```

### 5.5.2 Using sets to calculate the number of unique words

According to our computation above, there are about 26 thousand total words in *Alice's Adventures in Wonderland*. But how many *unique* words are there? Python has a special data structure called a *set* that makes it easy to find out. A *set* drops all duplicates, giving a collection of the unique elements.

```
len(set(alice_words))
```

There are 5295 unique words in the text.

## 5.6 Exercise: Reading text from a file and splitting

*Alice's Adventures in Wonderland* is full of memorable characters. The main characters from the story are listed, one-per-line, in the file named `Characters.txt`.

NOTE: we will not always explicitly demonstrate everything you need to know in order to complete an exercise. Instead we focus on teaching you how to discover available methods and how use the `help` function to learn how to use them. It is expected that you will spend some time during the exercises looking for appropriate methods and perhaps reading documentation.

1. Open the `Characters.txt` file and read its contents.
2. Split text on newlines to produce a list with one element per line. Store the result as `"alice_characters"`.

““

### 5.6.1 Working with lists

The `split` methods we used to break up the text of *Alice in Wonderland* into words produced a *list*. A lot of the techniques we'll use later to analyze this text also produce lists, so its worth taking a minute to learn more about them.

It is always a good idea to know what type of things you're working with in Python. As you gain experience, you won't have to look this things up as often, but even experienced Python programmers use the `type` function to learn about the objects they are working with.

```
type(alice_words)
```

A *list* in Python is used to store a collection of items. As with other types in Python, you can get a list of methods by typing the name of the object followed by a `.` and pressing `tab`.

#### 5.6.1.1 Extracting subsets from lists

Among the things you can do with a list is extract subsets using bracket indexing notation. This is useful in many situations, including the current one where we want to inspect a long list without printing out the whole thing.

The examples below show how indexing works in Python.

```
alice_words[0] # first word (yes, we count from zero!)
```

```
alice_words[1] # second word
```

```
alice_words[:10] # first 10 words
```

```
alice_words[10:20] # words 11 through 20
```

```
alice_words[-1] # the last word
```

```
alice_words[-10:] # the last 10 words
```

Note that the displayed representation of lists and other data structures in python often closely matches the syntax used to create them. For example, we can create a list using square brackets, just as we see when we print a list:

```
['her',  
 'own',  
 'child-life',  
 'and',  
 'the',  
 'happy',  
 'summer',  
 'days.',  
 'THE',  
 'END']
```

### 5.6.1.2 Sorting and other in-place methods

There are many other things we can do with lists besides extracting subsets using bracket indexing. For example, there are methods to append and remove elements from a list. When using a list method that you are unfamiliar with, it is always a good idea to read the documentation.

Note that many methods modify the object *in place*. For example, if we wanted to sort the last 10 words in `alice_words` we would do it like this:

```
last_10 = alice_words[-10:]
print(last_10)
last_10.sort()
print(last_10)
```

## 5.6.2 Counting chapters and paragraphs

Now that we know how to split a string and how to work with the resulting list, we can split on chapter markers to count the number of chapters. All we need to do is specify the string to split on. Since each chapter is marked with the string 'CHAPTER ' followed by the chapter number, we can split the text up into chapters using this as the separator.

```
alice_chapters = alice_txt.split("CHAPTER ")
len(alice_chapters)
```

Since the first element contains the material *before* the first chapter, this tells us there are twelve chapters in the book.

We can count paragraphs in a similar way. Paragraphs are indicated by a blank line, i.e., two newlines in a row. When working with strings we can represent newlines with `\n`, so our basic paragraph separator is `\n\n`.

```
alice_paragraphs = alice_txt.split("\n\n")
```

Before counting the number of paragraphs, I want to inspect the result to see if it looks correct:

```
print(alice_paragraphs[0], "\n=====")
print(alice_paragraphs[1], "\n=====")
print(alice_paragraphs[2], "\n=====")
print(alice_paragraphs[3], "\n=====")
print(alice_paragraphs[4], "\n=====")
print(alice_paragraphs[5], "\n=====")
```

We're counting the title, author, and chapter lines as paragraphs, but this will do for a rough count.

```
len(alice_paragraphs)
```

## 5.7 Exercise: count the number of main characters

So far we've learned that there are 12 chapters, around 830 paragraphs, and about 26 thousand words in *Alice's Adventures in Wonderland*. Along the way we've also learned how to open a file and read its contents, split strings, calculate the length of objects, discover methods for string and list objects, and index/subset lists in Python. Now it is time for you to put these skills to use to learn something about the main characters in the story.

1. Count the number of main characters in the story (i.e., get the length of the list you created in previous exercise).
2. Extract and print just the first character from the list you created in the previous exercise.
3. (BONUS, optional): Sort the list you created in step 2 alphabetically, and then extract the last element.

## 5.8 Working with nested structures: words within paragraphs within chapters

This far our analysis has treated the text as a “flat” data structure. For example, when we counted words we just counted words in the whole document, rather than counting the number of words in each chapter. If we want to treat our document as a nested structure, with words forming sentences, sentences forming paragraphs, paragraphs forming chapters, and chapters forming the book, we need to learn some additional tools. Specifically, we need to learn how to iterate over lists (or other collections) and do things with each element in a collection.

There are several ways to iterate in Python, of which we will focus on *for loops* and *list comprehensions*.

### 5.8.1 Iterating over paragraphs using for-loops

A *for loop* is a way of cycling through the elements of a collection and doing something with each one. As a simple example, we can cycle through the first 6 paragraphs and print each one. Cycling through with a loop makes it easy to insert a separator between the paragraphs, making it much easier to read the output.

```
for paragraph in alice_paragraphs[:6]:
    print(paragraph)
    print('=====')
print('DONE.')
```

Notice that the syntax of a for-loop is

```
for <thing> in <collection>:  
    do stuff with <thing>
```

Notice also that the body of the for-loop is indented. This is important, because it is this indentation that defines the body of the loop. Notice that “DONE.” is only printed once, since `print('DONE.')` is not indented and is therefore outside of the body of the loop.

Loops in Python are great because the syntax is relatively simple, and because they are very powerful. Inside of the body of a loop you can use all the tools you use elsewhere in python.

Here is one more example of a loop, this time iterating over all the chapters and calculating the number of paragraphs in each chapter.

```
for chapter in alice_chapters[1:]:  
    paragraphs = chapter.split("\n\n")  
    print(len(paragraphs))
```

### 5.8.2 Iterating and collecting paragraphs per chapter using list comprehension

We could use for-loops to fill in lists of values, but there is a special syntax in Python that is often better for this use case. This special syntax is called a *list comprehension* and it looks like this:

```
paragraphs_per_chapter = [len(chapter.split("\n\n"))  
                           for chapter in alice_chapters[1:]]  
print(paragraphs_per_chapter)
```

Notice that *list comprehension* is very similar to a *for loop*, though the order is different. In a *for-loop* the `for` part comes first and the expressions that make up the body come second and are indented. In a *list comprehension* the expression comes first and the `for` part comes afterward. Notice also the square brackets surrounding the whole thing – these brackets are what tells Python that you want a list.

Here is another list comprehension that counts the number of times the name “Alice” appears in each chapter.

```
alices_per_chapter = [chapter.count("Alice") for chapter in alice_chapters]  
print(alices_per_chapter)
```

### 5.8.3 Organizing results in dictionaries

Our code for calculating the number of of times “Alice” was mentioned per chapter worked, but with a little effort we can make it much easier to interpret by associating each count



with the chapter it corresponds to. In Python we can use a `dict` (i.e., “dictionary”) to store key-value pairs.

First, we can iterate over each chapter and grab just the first line (that is, the chapter titles). These will become our keys.

```
chapter_names = [chapter.splitlines()[0] for chapter in alice_chapters[1:]]
print(chapter_names)
```

Finally we can combine the chapter titles and counts and convert them to a dictionary.

```
dict(zip(chapter_names,
        [chapter.count("Alice")
         for chapter in alice_chapters]))
```

## 5.9 Exercise: Iterating and counting things

Now that we know how to iterate using for-loops and list comprehensions the possibilities really start to open up. For example, we can use these techniques to count the number of times each character appears in the story.

1. Make sure you have both the text and the list of characters.

Open and read both “`Alice_in_wonderland.txt`” and “`Characters.txt`” if you have not already done so.

2. Which chapter has the most words?

Split the text into chapters (i.e., split on “`CHAPTER`”) and use a for-loop or list comprehension to iterate over the chapters. For each chapter, split it into words and calculate the length.

3. How many times is each character mentioned in the text?

Iterate over the list of characters using a for-loop or list comprehension. For each character, call the `count` method with that character as the argument.

4. (BONUS, optional): Put the character counts computed above in a dictionary with character names as the keys and counts as the values.
5. (BONUS, optional): Use a nested list comprehension to calculate the number of times each character is mentioned in each chapter.

## 5.10 Importing numpy and calculating simple statistics

Now that we know how to iterate over lists and calculate numbers for each element, we may wish to do some simple math using these numbers. For example, we may want to calculate the mean and standard deviation of the distribution of the number of paragraphs in each chapter. Python has a handful of math functions built-in (e.g., `min` and `max`) but built-in math support is pretty limited.

When you find that something isn't available in Python itself, its time to look for a package that does it. Although it is somewhat overkill for simply calculating a mean we're going to use a popular package called *numpy* for this. The *numpy* package is included in the Anaconda Python distribution we are using, so we don't need to install it separately.

In order to use *numpy* or other packages, you must first import them. We can import numpy as follows:

```
import numpy
```

The *numpy* package is very popular and includes a lot of useful functions. For example, we can use it to calculate means and standard deviations:

```
print(numpy.mean(paragraphs_per_chapter))
print(numpy.std(paragraphs_per_chapter))
```

and compute correlations:

```
words_per_chapter = [len(chapter.split()) for chapter in alice_chapters]
alices_per_chapter = [chapter.count("Alice") for chapter in alice_chapters]

print(numpy.corrcoef(words_per_chapter, alices_per_chapter))
```

## 5.11 Wrap-up

### 5.11.1 Feedback

These workshops are a work in progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 5.11.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>
  - Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>

- HBS
  - Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>
- Graphics
  - matplotlib: <https://matplotlib.org/>
  - seaborn: <https://seaborn.pydata.org/>
  - plotly: <https://plot.ly/python/>
- Quantitative Data Analysis
  - numpy: <http://www.numpy.org/>
  - scipy: <https://www.scipy.org/>
  - pandas: <https://pandas.pydata.org/>
  - scikit-learn: <http://scikit-learn.org/stable/>
  - statsmodels: <http://www.statsmodels.org/stable/>
- Text analysis
  - textblob: <https://textblob.readthedocs.io/en/dev/>
  - nltk: <http://www.nltk.org/>
  - Gensim: <https://radimrehurek.com/gensim/>
- Webscraping
  - scrapy: <https://scrapy.org/>
  - requests: <http://docs.python-requests.org/en/master/>
  - lxml: <https://lxml.de/>
  - BeautifulSoup: <https://www.crummy.com/software/BeautifulSoup/>
- Social Network Analysis
  - networkx: <https://networkx.github.io/>
  - graph-tool: <https://graph-tool.skewed.de/>



## Chapter 6

# Python Web-Scraping

### Topics

- Web basics
- Making web requests
- Inspecting web sites
- Retrieving web data
- Using Xpaths to retrieve `html` content
- Parsing `html` content
- Cleaning and storing text from `html`

## 6.1 Setup

### 6.1.1 Software & Materials

#### 6.1.1.1 Install the Anaconda Python distribution

If using your own computer please install the Anaconda Python distribution from <https://www.anaconda.com/download/>. (Note that Python version  $\leq 3.0$  differs considerably from more recent releases. For this workshop you will need version  $\geq 3.6.x$ )

Accepting the defaults proposed by the Anaconda installer is generally recommended.

#### 6.1.1.2 Workshop notes

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/Python/PythonWebScrape.zip>
- Extract materials from the zipped directory `PythonWebScrape.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

Start the Jupyter Notebook application and open the `PythonWebScrape.ipynb` file in the `PythonWebScrape` folder you downloaded previously.

### 6.1.2 Goals

In this workshop you will

- learn basic web scraping principles and techniques,
- learn how to use the `requests` package in Python,
- practice making requests and manipulating responses from the server.

This workshop is relatively *informal*, *example-oriented*, and *hands-on*. We will learn by working through an example web scraping project.

Note that this is **not** an introductory workshop. Familiarity with Python, including but not limited to knowledge of lists and dictionaries, indexing, and loops and / or comprehensions is assumed. If you need an introduction to Python or a refresher, we recommend the IQSS Introduction to Python.

Note also that this workshop will not teach you everything you need to know in order to retrieve data from any web service you might wish to scrape. You can expect to learn just enough to be dangerous.

## 6.2 Preliminary questions

### 6.2.1 What is web scraping?

Web scraping is the activity of automating retrieval of information from a web service designed for human interaction.

### 6.2.2 Is web scraping legal? Is it ethical?

It depends. If you have legal questions seek legal counsel. You can mitigate some ethical issues by building delays and restrictions into your web scraping program so as to avoid impacting the availability of the web service for other users or the cost of hosting the service for the service provider.

## 6.3 Example project

In this workshop I will demonstrate web scraping techniques using the Collections page at <https://www.harvardartmuseums.org/collections> and let you use the skills you'll learn to retrieve information from other parts of the Harvard Art Museums website.

The basic strategy is pretty much the same for most scraping projects. We will use our web browser (Chrome or Firefox recommended) to examine the page you wish to retrieve data from, and copy/paste information from your web browser into your scraping program.

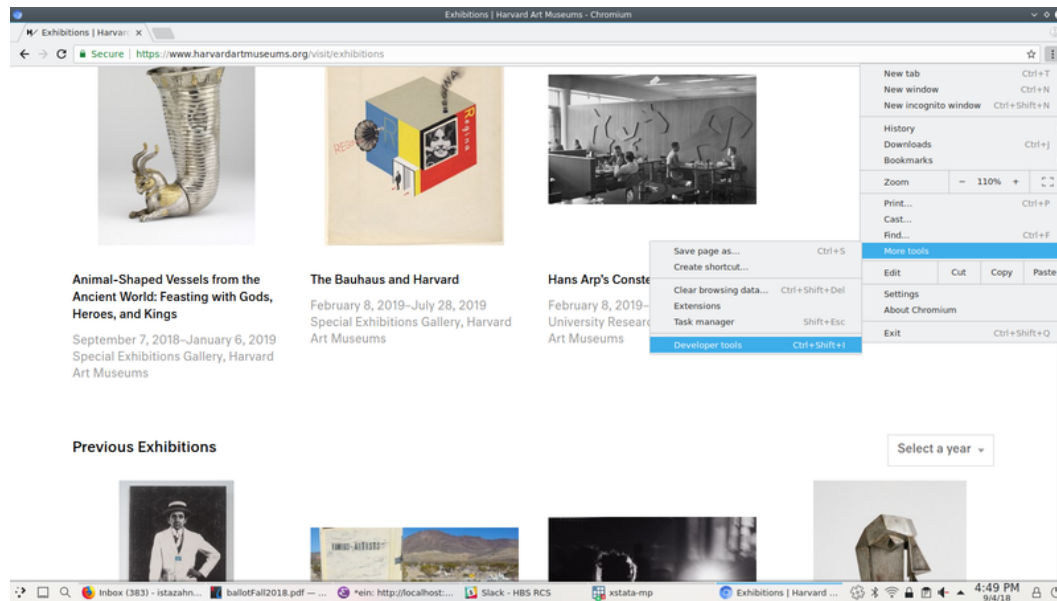


Figure 6.1

## 6.4 Take shortcuts if you can

We wish to extract information from <https://www.harvardartmuseums.org/collections>. Like most modern web pages, a lot goes on behind the scenes to produce the page we see in our browser. Our goal is to pull back the curtain to see what the website does when we interact with it. Once we see how the website works we can start retrieving data from it. If we are lucky we'll find a resource that returns the data we're looking for in a structured format like JSON or XML.

### 6.4.1 Examining the structure of our target web service

We start by opening the collections web page in a web browser and inspecting it.

If we scroll down to the bottom of the Collections page, we'll see a button that says "Load More". Let's see what happens when we click on that button. To do so, click on "Network" in the developer tools window, then click the "Load More Collections" button. You should see a list of requests that were made as a result of clicking that button, as shown below.

If we look at that second request, the one to a script named `browse`, we'll see that it returns all the information we need, in a convenient format called JSON. All we need to retrieve collection data is call make GET requests to <https://www.harvardartmuseums.org/browse> with the correct parameters.

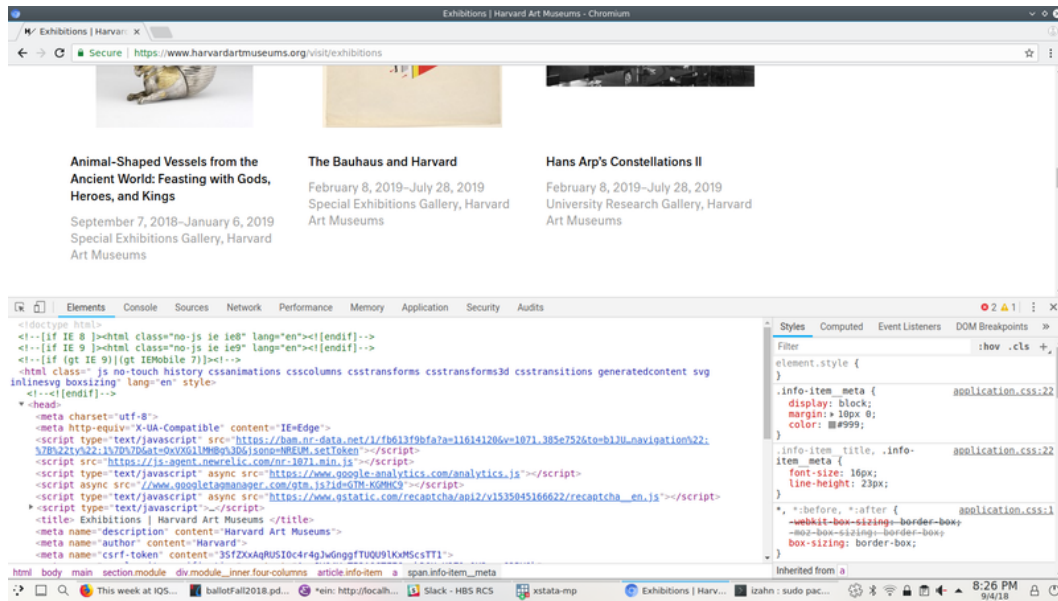


Figure 6.2

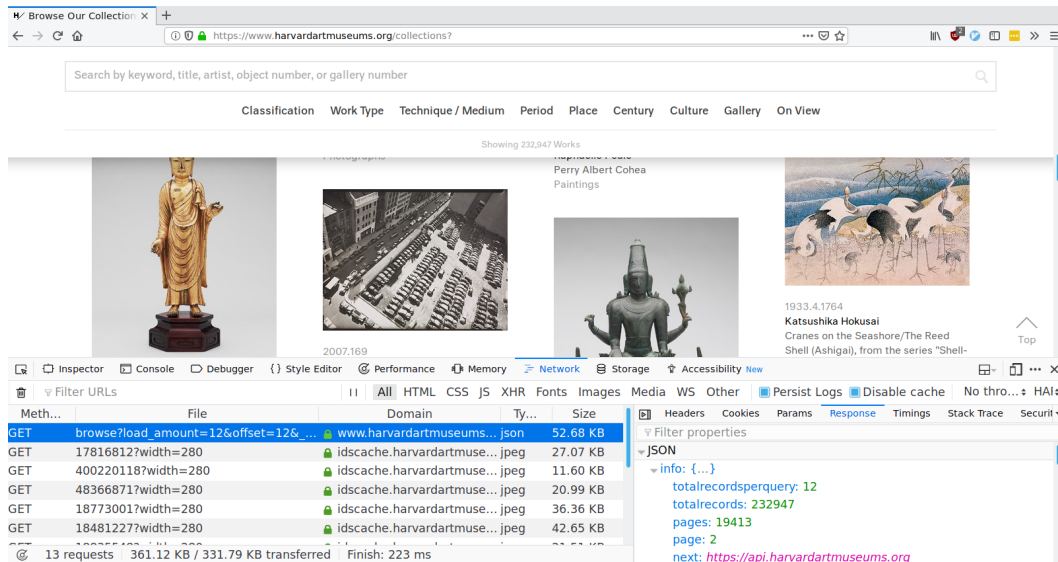


Figure 6.3



### 6.4.2 Making requests using python

The URL we want to retrieve data from has the following structure

scheme	domain	path	parameters
https	www.harvardartmuseums.org	browse	load_amount=10&offset=0

It is often convenient to create variables containing the domain(s) and path(s) you'll be working with, as this allows you to swap out paths and parameters as needed. Note that the path is separated from the domain with / and the parameters are separated from the path with ?. If there are multiple parameters they are separated from each other with a &.

For example, we can define the domain and path of the collections URL as follows:

```
museum_domain = 'https://www.harvardartmuseums.org'
collection_path = 'browse'

collection_url = (museum_domain
                  + "/"
                  + collection_path)

print(collection_url)
```

Note that we omit the parameters here because it is usually easier to pass them as a `dict` when using the `requests` library in Python. This will become clearer shortly.

Now that we've constructed the URL we wish interact with we're ready to make our first request in Python.

```
import requests

collections1 = requests.get(
    collection_url,
    params = {'load_amount': 10,
              'offset': 0}
)

# ### Parsing JSON data
# We already know from inspecting network traffic in our web
# browser that this URL returns JSON, but we can use Python to verify
# this assumption.
collections1.headers['Content-Type']
```

Since JSON is a structured data format, parsing it into python data structures is easy. In fact, there's a method for that!

```
collections1 = collections1.json()
print(collections1)
```

That's it. Really, we are done here. Everyone go home!

OK not really, there is still more we can learn. But you have to admit that was pretty easy. If you can identify a service that returns the data you want in structured form, web scraping becomes a pretty trivial enterprise. We'll discuss several other scenarios and topics, but for some web scraping tasks this is really all you need to know.

### 6.4.3 Organizing and saving the data

The records we retrieved from <https://www.harvardartmuseums.org/browse> are arranged as a list of dictionaries. We can easily select the fields of arrange these data into a pandas `DataFrame` to facilitate subsequent analysis.

```
import pandas as pd
```

```
records1 = pd.DataFrame.from_records(collections1['records'])
```

```
print(records1)
```

and write the data to a file.

```
records1.to_csv("records1.csv")
```

### 6.4.4 Iterating to retrieve all the data

Of course we don't want just the first page of collections. How can we retrieve all of them?

Now that we know the web service works, and how to make requests in Python, we can iterate in the usual way.

```
records = []
for offset in range(0, 50, 10):
    param_values = {'load_amount': 10, 'offset': offset}
    current_request = requests.get(collection_url, params = param_values)
    records += current_request.json()['records']
```

```
## convert list of dicts to a `DataFrame`
records_final = pd.DataFrame.from_records(records)
```

```
# write the data to a file.
records_final.to_csv("records_final.csv")

print(records_final)
```

### 6.4.5 Exercise: Retrieve exhibits data

In this exercise you will retrieve information about the art exhibitions at Harvard Art Museums from <https://www.harvardartmuseums.org/visit/exhibitions>

1. Using a web browser (Firefox or Chrome recommended) inspect the page at <https://www.harvardartmuseums.org/visit/exhibitions>. Examine the network traffic as you interact with the page. Try to find where the data displayed on that page comes from.
2. Make a `get` request in Python to retrieve the data from the URL identified in step1.
3. Write a *loop* or *list comprehension* in Python to retrieve data for the first 5 pages of exhibitions data.
4. Bonus (optional): Convert the data you retrieved into a pandas `DataFrame` and save it to a `.csv` file.

## 6.5 Parse html if you have to

As we've seen, you can often inspect network traffic or other sources to locate the source of the data you are interested in and the API used to retrieve it. You should always start by looking for these shortcuts and using them where possible. If you are really lucky, you'll find a shortcut that returns the data as JSON or XML. If you are not quite so lucky, you will have to parse HTML to retrieve the information you need.

For example, when I inspected the network traffic while interacting with <https://www.harvardartmuseums.org/visit/calendar> I didn't see any requests that returned JSON data. The best we can do appears to be <https://www.harvardartmuseums.org/visit/calendar?date=>, which unfortunately returns HTML.

### 6.5.1 Retrieving HTML

The first step is the same as before: we make a `GET` request.

```
calendar_path = 'visit/calendar'

calendar_url = (museum_domain # recall that we defined museum_domain earlier
               + "/"
               + calendar_path)

print(calendar_url)
```

```
events0 = requests.get(calendar_url, params = {'date': '2018-11'})
```

As before we can check the headers to see what type of content we received in response to our request.

```
events0.headers['Content-Type']
```

### 6.5.2 Parsing HTML using the lxml library

Like JSON, HTML is structured; unlike JSON it is designed to be rendered into a human-readable page rather than simply to store and exchange data in a computer-readable format. Consequently, parsing HTML and extracting information from it is somewhat more difficult than parsing JSON.

While JSON parsing is built into the Python `requests` library, parsing HTML requires a separate library. I recommend using the HTML parser from the `lxml` library; others prefer an alternative called `BeautifulSoup`.

```
from lxml import html

events_html = html.fromstring(events0.text)
```

### 6.5.3 Using xpath to extract content from HTML

`XPath` is a tool for identifying particular elements within a HTML document. The developer tools built into modern web browsers make it easy to generate `XPaths` that can be used to identify the elements of a web page that we wish to extract.

We can open the HTML document we retrieved and inspect it using our web browser.

```
html.open_in_browser(events_html, encoding = 'UTF-8')
```

Once we identify the element containing the information of interest we can use our web browser to copy the `XPath` that uniquely identifies that element.

Next we can use python to extract the element of interest:

```
events_list_html = events_html.xpath('//*[@id="events_list"]')[0]
```

Once again we can use a web browser to inspect the HTML we're currently working with, and to figure out what we want to extract from it. Let's look at the first element in our events list.

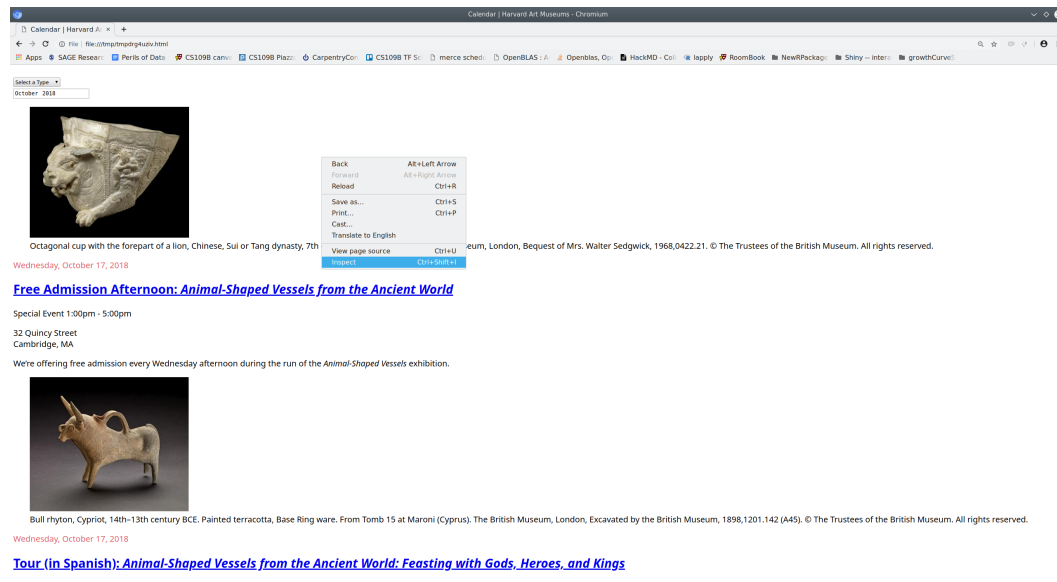


Figure 6.4

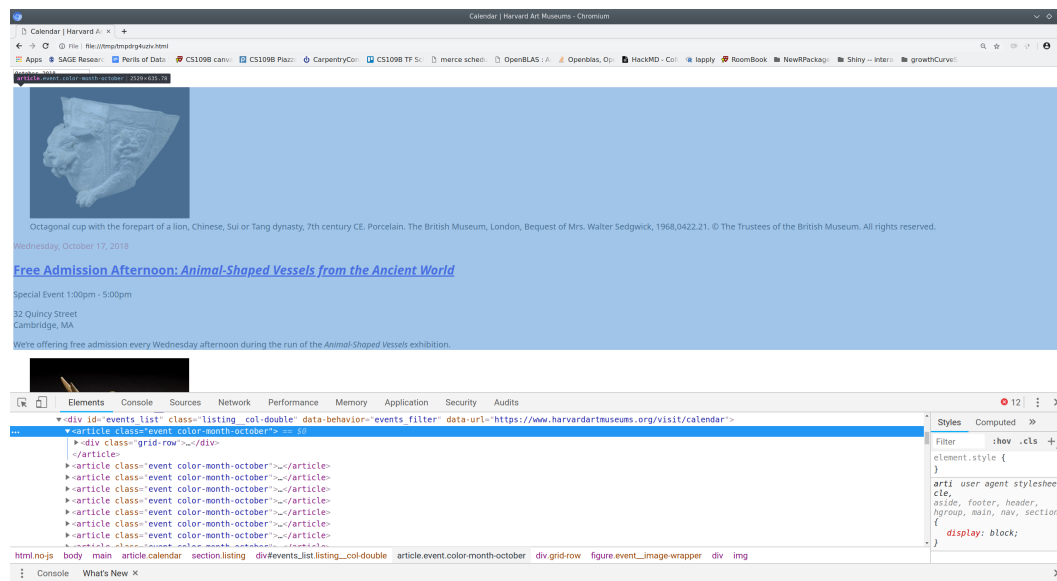


Figure 6.5

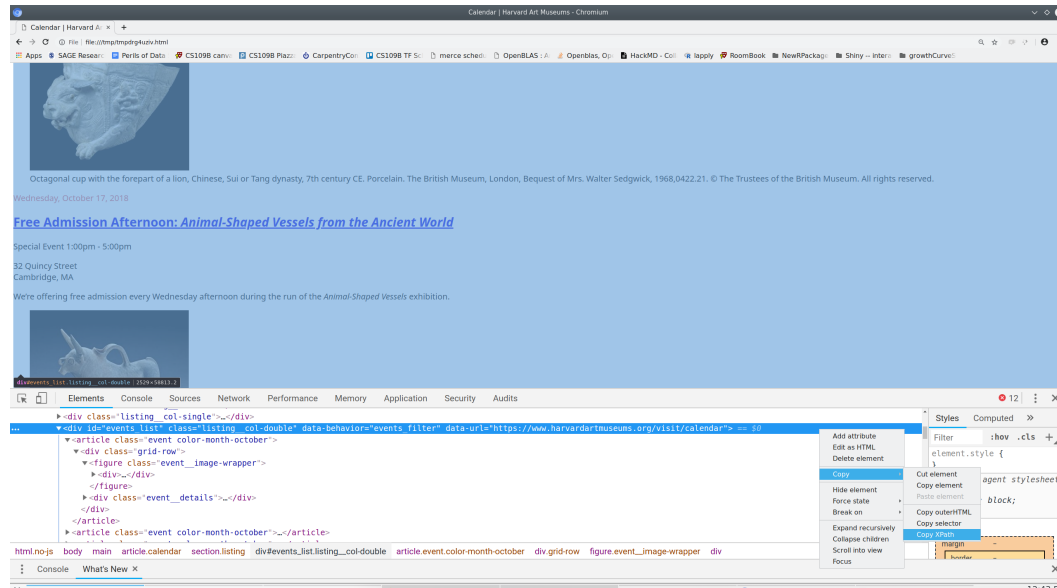


Figure 6.6

```
first_event_html = events_list_html[0]
html.open_in_browser(first_event_html, encoding = 'UTF-8')
```

As before we can use our browser to find the xpath of the elements we want.

(Note that the `html.open_in_browser` function adds enclosing `html` and `body` tags in order to create a complete web page for viewing. This requires that we adjust the `xpath` accordingly.)

By repeating this process for each element we want, we can build a list of the `xpaths` to those elements.

```
elements_we_want = {'figcaption': 'div/figure/div/figcaption',
                    'date': 'div/div/header/time',
                    'title': 'div/div/header/h2/a',
                    'time': 'div/div/div/p[1]/time',
                    'location1': 'div/div/div/p[2]/span/span[1]',
                    'location2': 'div/div/div/p[2]/span/span[2]'}
}
```

Finally, we can iterate over the elements we want and extract them.

```
first_event_values = {}
for key in elements_we_want.keys():
    element = first_event_html.xpath(elements_we_want[key])[0]
    first_event_values[key] = element.text_content().strip()
```

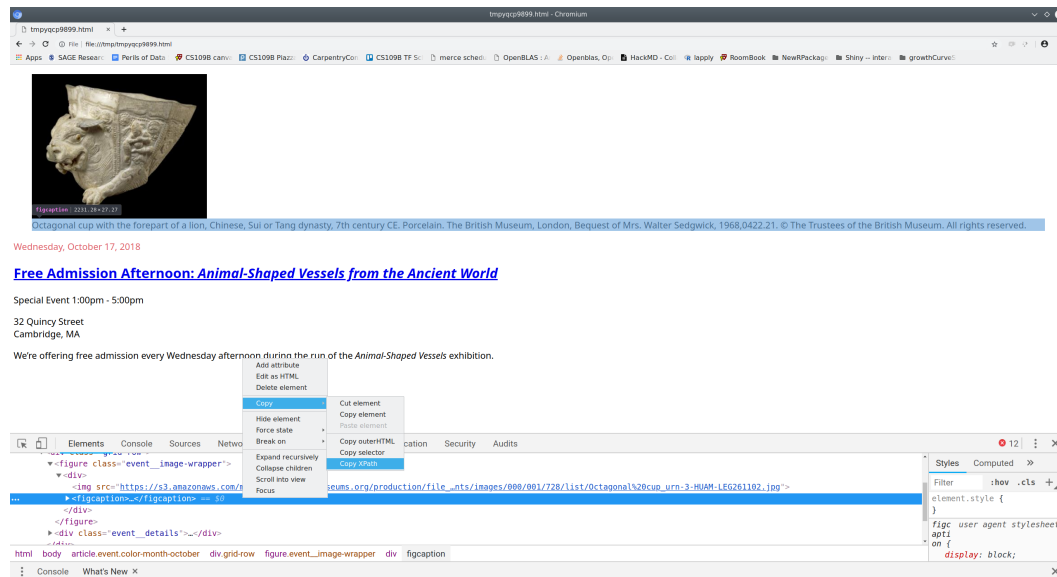


Figure 6.7

```
print(first_event_values)
```

#### 6.5.4 Iterating to retrieve content from a list of HTML elements

So far we've retrieved information only for the first event. To retrieve data for all the events listed on the page we need to iterate over the events. If we are very lucky, each event will have exactly the same information structured in exactly the same way and we can simply extend the code we wrote above to iterate over the events list.

Unfortunately not all these elements are available for every event, so we need to take care to handle the case where one or more of these elements is not available. We can do that by defining a function that tries to retrieve a value and returns an empty string if it fails.

```
def get_event_info(event, path):
    try:
        info = event.xpath(path)[0].text.strip()
    except:
        info = ''
    return info
```

Armed with this function we can iterate over the list of events and extract the available information for each one.

```

all_event_values = {}
for key in elements_we_want.keys():
    key_values = []
    for event in events_list_html:
        key_values.append(get_event_info(event, elements_we_want[key]))
    all_event_values[key] = key_values

```

For convenience we can arrange these values in a pandas `DataFrame` and save them as `.csv` files, just as we did with our exhibitions data earlier.

```

all_event_values = pd.DataFrame.from_dict(all_event_values)

all_event_values.to_csv("all_event_values.csv")

print(all_event_values)

```

### 6.5.5 Exercise: parsing HTML

In this exercise you will retrieve information about the physical layout of the Harvard Art Museums. The web page at <https://www.harvardartmuseums.org/visit/floor-plan> contains this information in HTML from.

1. Using a web browser (Firefox or Chrome recommended) inspect the page at <https://www.harvardartmuseums.org/visit/floor-plan>. Copy the XPath to the element containing the list of level information. (HINT: the element of interest is a `ul`, i.e., `unordered list`.)
2. Make a `get` request in Python to retrieve the web page at <https://www.harvardartmuseums.org/visit/floor-plan>. Extract the content from your request object and parse it using `html.fromstring` from the `lxml` library.
3. Use your web browser to find the XPaths to the facilities housed on level one. Use Python to extract the text from those Xpaths.
4. Bonus (optional): Write a *loop* or *list comprehension* in Python to retrieve data for all the levels.

## 6.6 Use Scrapy for large or complicated projects

Scraping websites using the `requests` library to make GET and POST requests, and the `lxml` library to process HTML is a good way to learn basic web scraping techniques. It is a good choice for small to medium size projects. For very large or complicated scraping tasks the `scrapy` library offers a number of conveniences, including asynchronously retrieval, session management, convenient methods for extracting and storing values, and more. More information about `scrapy` can be found at <https://doc.scrapy.org>.



## 6.7 Use a browser driver as a last resort

It is sometimes necessary (or sometimes just easier) to use a web browser as an intermediary rather than communicating directly with a web service. This method has the advantage of being about to use the javascript engine and session management features of a web browser; the main disadvantage is that it is slower and tends to be more fragile than using `requests` or `scrapy` to make requests directly from python. For small scraping projects involving complicated sites with CAPTHAs or lots of complicated javascript using a browser driver can be a good option. More information is available at [https://www.seleniumhq.org/docs/03\\_webdriver.jsp](https://www.seleniumhq.org/docs/03_webdriver.jsp).

## 6.8 Wrap-up

### 6.8.1 Feedback

These workshops are a work in progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 6.8.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>
  - Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>
- HBS
  - Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>



## Part III

# Stata



## Chapter 7

# Stata Introduction

### Topics

- Stata interface and Do-files
- Finding help
- Reading and writing data
- Basic summary statistics
- Basic graphs
- Basic data management

## 7.1 Setup

### 7.1.1 Software & Materials

Laptop users: you will need a copy of Stata installed on your machine. Harvard FAS affiliates can install a licensed version from <http://downloads.fas.harvard.edu/download>

- Download class materials at <https://github.com/IQSS/dss-workshops-redux/raw/master/Stata/StataIntro.zip>
- Extract materials from the zipped directory **StataIntro.zip** (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

### 7.1.2 Organization

- Please feel free to ask questions at any point if they are relevant to the current topic (or if you are lost!)
- Collaboration is encouraged - please introduce yourself to your neighbors!
- If you are using a laptop, you will need to adjust file paths accordingly
- Make comments in your Do-file - save on flash drive or email to yourself

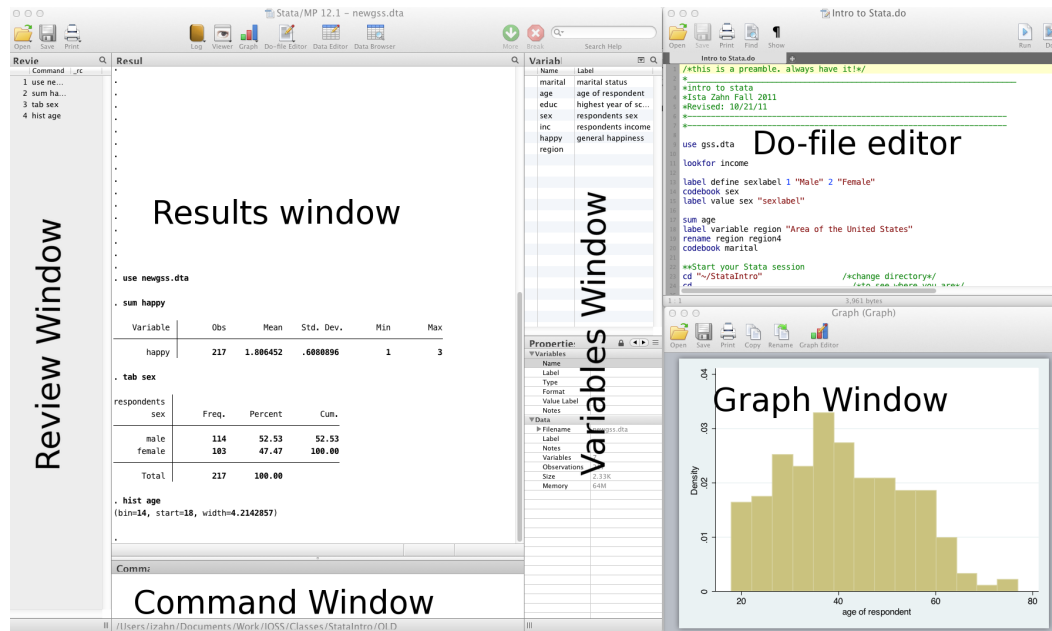


Figure 7.1

### 7.1.3 Goals

- This is an **introduction** to Stata
- Assumes no/very little knowledge of Stata
- Not appropriate for people already familiar with Stata
- Learning Objectives:
  - Familiarize yourself with the Stata interface
  - Get data in and out of Stata
  - Compute statistics and construct graphical displays
  - Compute new variables and transformations

## 7.2 Why stata?

- Used in a variety of disciplines
- User-friendly
- Great guides available on web
- Excellent modeling capabilities
- Student and other discount packages available at reasonable cost

### 7.2.1 Stata interface

- Review and Variable windows can be closed (user preference)
- Command window can be shortened (recommended)

### 7.2.2 Do-files

- You can type all the same commands into the Do-file that you would type into the command window
- BUT...the Do-file allows you to **save** your commands
- Your Do-file should contain ALL commands you executed – at least all the “correct” commands!
- I recommend never using the command window or menus to make CHANGES to data
- Saving commands in Do-file allows you to keep a written record of everything you have done to your data
  - Allows easy replication
  - Allows you to go back and re-run commands, analyses and make modifications

### 7.2.3 Stata help

To get help in Stata type **help** followed by topic or command, e.g., **help codebook**.

### 7.2.4 General Stata command syntax

Most Stata commands follow the same basic syntax: **Command varlist, options**.

### 7.2.5 Commenting and formatting syntax

Start with comment describing your Do-file and use comments throughout

\* Use '\*' to comment a line and '//' for in-line comments

\* Make Stata say hello:

```
disp "Hello " "World!" // 'disp' is short for 'display'
```

- Use /// to break varlists over multiple lines:

```
disp "Hello" ///
    " World!"
```

### 7.2.6 Let's get started

- Launch the Stata program (MP or SE, does not matter unless doing computationally intensive work)
  - Open up a new Do-file
  - Run our first Stata code!

\* change directory

```
// cd "C://Users/dataclass/Desktop/StataIntro"
```

## 7.3 Getting data into Stata

### 7.3.1 Data file commands

- Next, we want to open our data file
- Open/save data sets with “use” and “save”:

```
cd dataSets

// open the gss.dta data set
use gss.dta, clear

// save data file:
save newgss.dta, replace // "replace" option means OK to overwrite existing file
```

### 7.3.2 A note about path names

- If your path has no spaces in the name (that means all directories, folders, file names, etc. can have no spaces), you can write the path as is
- If there are spaces, you need to put your pathname in quotes
- Best to get in the habit of quoting paths

### 7.3.3 Where’s my data?

- Data editor (**browse**)
- Data editor (**edit**)
  - Using the data editor is discouraged (why?)
- Always keep any changes to your data in your Do-file
- Avoid temptation of making manual changes by viewing data via the browser rather than editor

### 7.3.4 What if my data is not a Stata file?

- Import delimited text files

```
* import data from a .csv file
import delimited gss.csv, clear
```

```
* save data to a .csv file
export delimited gss_new.csv, replace
```

- Import data from SAS



```
* import/export SAS xport files
clear
import sasxport gss.xpt
export sasxport gss_new, replace
```

- Import data from Excel

```
* import/export Excel files
clear
import excel gss.xlsx
export excel gss_new, replace
```

### 7.3.5 What if my data is from another statistical software program?

- SPSS/PASW will allow you to save your data as a Stata file
  - Go to: file -> save as -> Stata (use most recent version available)
  - Then you can just go into Stata and open it
- Another option is **StatTransfer**, a program that converts data from/to many common formats, including SAS, SPSS, Stata, and many more

### 7.3.6 Exercise 0: Importing data

1. Save any work you've done so far. Close down Stata and open a new session.
2. Start Stata and open your `.do` file.
3. Change directory (`cd`) to the `dataSets` folder.
4. Try opening the following files:
  - A comma separated value file: `gss.csv`
  - An Excel file: `gss.xlsx`

## 7.4 Statistics and graphs

### 7.4.1 Frequently used commands

- Commands for reviewing and inspecting data:
  - `describe` // labels, storage type etc.
  - `sum` // statistical summary (mean, sd, min/max etc.)
  - `codebook` // storage type, unique values, labels
  - `list` // print actual values
  - `tab` // (cross) tabulate variables
  - `browse` // view the data in a spreadsheet-like window

First, let's ask Stata for help about these commands:

```
help sum
```

```
use gss.dta, clear
```

```
sum educ // statistical summary of education
```

```
codebook region // information about how region is coded
```

```
tab sex // numbers of male and female participants
```

- If you run these commands without specifying variables, Stata will produce output for every variable

### 7.4.2 Basic graphing commands

- Univariate distribution(s) using **hist**

```
/* Histograms */
hist educ
```

```
// histogram with normal curve; see "help hist" for other options
hist age, normal
```

- View bivariate distributions with scatterplots

```
/* scatterplots */
twoway (scatter educ age)
```

```
graph matrix educ age inc
```

### 7.4.3 The by command

- Sometimes, you'd like to generate output based on different categories of a grouping variable
- The “by” command does just this

```
* By Processing
```

```
bysort sex: tab happy // tabulate happy separately for men and women
```

```
bysort marital: sum educ // summarize education by marital status
```

#### 7.4.4 Exercise 1: Descriptive statistics

1. Use the dataset, `gss.dta`
2. Examine a few selected variables using the `describe`, `sum` and `codebook` commands
3. Tabulate the variable, “marital,” with and without labels
4. Summarize the variable, “income” by marital status
5. Cross-tabulate marital with region
6. Summarize the variable `happy` for married individuals only

## 7.5 Basic data management

### 7.5.1 Labels

- You never know why and when your data may be reviewed
- ALWAYS label every variable no matter how insignificant it may seem
- Stata uses two sets of labels: **variable labels** and **value labels**
- Variable labels are very easy to use – value labels are a little more complicated

### 7.5.2 Variable and value labels

- Variable labels

```
/* Labelling and renaming */
// Label variable inc "household income"
label var inc "household income"

// change the name 'educ' to 'education'
rename educ education

// you can search names and labels with 'lookfor'
lookfor household
```

- Value labels are a two step process: define a value label, then assign defined label to variable(s)

```
/*define a value label for sex */
label define mySexLabel 1 "Male" 2 "Female"

/* assign our label set to the sex variable*/
label val sex mySexLabel
```

## 7.6 Exercise 2: Variable labels and value labels

1. Open the data set `gss.csv`

2. Familiarize yourself with the data using describe, sum, etc.
3. Rename and label variables using the following codebook:

Var	Rename to	Label with
v1	marital	marital status
v2	age	age of respondent
v3	educ	education
v4	sex	respondent's sex
v5	inc	household income
v6	happy	general happiness
v7	region	region of interview

1. Add value labels to your `marital` variable using this codebook:

Value	Label
1	"married"
2	"widowed"
3	"divorced"
4	"separated"
5	"never married"

### 7.6.1 Working on subsets

- It is often useful to select just those rows of your data where some condition holds—for example select only rows where sex is 1 (male)
- The following operators allow you to do this:

Operator	Meaning
==	equal to
!=	not equal to
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
&	and

- Note the double equals signs for testing equality

### 7.6.2 Generating and replacing variables

- Create new variables using `gen`

```
// create a new variable named mc_inc
// equal to inc minus the mean of inc
gen mc_inc = inc - 15.37
```

- Sometimes useful to start with blank values and fill them in based on values of existing variables

```
/* the 'generate and replace' strategy */
// generate a column of missings
gen age_wealth = .

// Next, start adding your qualifications
replace age_wealth=1 if age<30 & inc < 10
replace age_wealth=2 if age<30 & inc > 10
replace age_wealth=3 if age>30 & inc < 10
replace age_wealth=4 if age>30 & inc > 10

// conditions can also be combined with "or"
gen young=0
replace young=1 if age_wealth==1 | age_wealth==2
```

## 7.7 Exercise 3: Manipulating variables

1. Use the dataset, `gss.dta`
2. Generate a new variable, `age2` equal to `age` squared
3. Generate a new `high_income` variable that will take on a value of “1” if a person has an income value greater than “15” and “0” otherwise
4. Generate a new `divorced_separated` dummy variable that will take on a value of “1” if a person is either divorced or separated and “0” otherwise

## 7.8 Wrap-up

### 7.8.1 Feedback

These workshops are a work in progress, please provide any feedback to: [help@iq.harvard.edu](mailto:help@iq.harvard.edu)

### 7.8.2 Resources

- IQSS
  - Workshops: <https://dss.iq.harvard.edu/workshop-materials>
  - Data Science Services: <https://dss.iq.harvard.edu/>
  - Research Computing Environment: <https://iqss.github.io/dss-rce/>
- HBS
  - Research Computing Services workshops: <https://training.rcs.hbs.org/workshops>
  - Other HBS RCS resources: <https://training.rcs.hbs.org/workshop-materials>
  - RCS consulting email: <mailto:research@hbs.edu>