Data Science Workshops

December 2019

Contents

Introduction

Table of Contents

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

1. Data Science Tools
2. R Introduction
3. R Regression Models
4. R Graphics
5. R Data Wrangling
6. Python Introduction
7. Python Web-Scraping
8. Stata Introduction
9. Stata Data Management
10. Stata Modeling & Graphing

These workshops are a work-in-progress, please provide feedback! Email: `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

General

Chapter 1

Data Science Tools

Topics

- Data science tool selection
- Data analysis pipelines
- Programming languages comparison
- Text editor and IDE comparison
- Tools for creating reports

1.1 Tools for working with data

Working with data effectively requires learning at least one programming or scripting language. You can get by without this, but it would be like trying to cook with only a butter knife; not recommended! Compared to using a menu-driven interface (e.g., SPSS or SAS) or a spreadsheet (e.g., Excel), using a programming language allows you to:

- reproduce results,
- correct errors and update output,
- reuse code,
- collaborate with others,
- automate repetitive tasks, and
- generate manuscripts, reports, and other documents from your code.

So, you need to learn a programming language for working with data, but which one should you learn? Since you'll be writing code you'll want to set up a comfortable environment for writing and editing that code. Which text editors are good for this? You'll probably also want to learn at least one markup language (e.g., LaTeX, Markdown) so that you can create reproducible manuscripts. What tools are good for this? These questions will guide our discussion, the goal of which is to help you decide which tools you should invest time in learning.

1.2 The puzzle pieces

As we've noted, working effectively with data requires using a number of tools.

1.2.1 Data analysis building blocks

The basic pieces are:

- a data storage and retrieval system,
- an editor for writing code,
- an interpreter or compiler for executing that code,
- a system for presenting results, and
- some "glue" to make all the pieces work together.

1.3 Examples

Before looking in detail at each of these building blocks we'll look at a few examples to get an intuitive feel for the basic elements.

1.3.1 Old-school example

In this example we're going to process data in a text file in a way that would be familiar to a statistician working forty years ago. Surprisingly, it's not much different from the way we would do it today. Programs come and go, but the basic ideas remain pretty much the same!

Specifically, we'll process the data in `1980_census.txt` by writing **fortran** code in the **vi** text editor and running it through the **fortran** compiler. Then we'll take the results and put them in to a **TeX** file, again using the **vi** editor to create the report. For "glue" we will use a terminal emulator running the bash shell. All of these tools were available in 1980, though some features have been added since that time.

OLD SCHOOL DEMO:

example	data storage	editor	program	report tool	glue
old school	ASCII text file	vi	fortran	TeX	Bourne (compatable) shell

1.3.2 Something old & something new

Next we're going to do the same basic process, this time using a modern text editor (**Atom**), a different programming language (**Python**), and a modern report generation system (**LaTeX** processed via **xelatex**). For the glue we're still going to use a shell.

OLD AND NEW DEMO:

example	data storage	editor	program	report tool	glue
old school	ASCII text file	vi	fortran	TeX	Bourne (compatable) shell
old and new	ASCII text file	Atom	python	LaTeX	Bash shell

1.3.3 A modern version

Finally, we'll produce the same report using modern tools. Remember, the process is basically the same: we're just using different tools.

MODERN DEMO:

example	data storage	editor	program	report tool	glue
old school	ASCII text file	vi	fortran	TeX	Bourne (compatable) shell
old and new	ASCII text file	Atom	python	LaTeX	Bash shell
modern	SQLite database	Rstudio	R	R Markdown	Rstudio

1.4 Data storage & retrieval

Data storage and retrieval is a fairly dry topic, so we won't spend too much time on it. There are roughly four types of technology for storing and retrieving data.

1.4.1 Text files

Storing data in text files (e.g., comma separated values, other delimited text formats) is simple and makes the data easy to access from just about any program. It is also good for archiving data since no specialized software is needed to read it. The main downsides are that retrieval is slow and often all-or-nothing, and the fact that storing metadata in plain text files is cumbersome.

1.4.2 Binary files

Many statistics packages and programming languages have a "native" binary data storage format. For example, Stata stores data in `.dta` files, and R stores data in `.rds` or `.Rdata` files. These storage formats usually more efficient than text files, and usually provide faster read/write access. They usually include a mechanism for storing metadata. The down side is that specialized software is required to read them (will Stata exist in 50 years? Are you sure?) and the ability to read them using other programs may be limited.

1.4.3 Databases

Storing data in a database requires more up-front planning and set up, but has several advantages. Databases provide fast selective retrieval and facilitate efficient storage and flexible retrieval.

1.4.4 Distributed file storage

Data that is too large to fit on a single hard drive may be stored and analyzed on a distributed file system or database such as the *Hadoop Distributed File System* or *Cassandra*. When working with data on this scale considerable infrastructure and specialized tools will be required.

1.5 Programming languages & statistics packages

There are tens of programs for statistics and data science available. Here we will focus only on the more popular programs that offer a wide range of features. Note that for specific applications a specialized program may be better, e.g., many people use Mplus for structural equation models and another program for everything else.

1.5.1 Programming language features

Things we want a statistics program to do include:

- read/write data from/to a variety of data storage systems,
- manipulate data,
- perform statistical methods,
- visualize data and results,
- export results in a variety of formats,
- be easy to use,
- be well documented,
- have a large user community.

Note that this list is deceptively simple; each item may include a diversity of complicated features. For example,"read/write data from/to a variety of data storage systems" may include reading from databases, image files, .pdf files, .html and .xml files from a website, and any number of proprietary data storage formats.

1.5.2 Program comparison

Program	Statistics	Visualization	Machine learning	Ease of use	Power/flexibility	Fun
Stata	Good	Servicable	Limited	Very easy	Low	Some
SPSS	OK	Servicable	Limited	Easy	Low	None
SAS	Good	Not great	Good	Moderate	Moderate	None
Matlab	Good	Good	Good	Moderate	Good	Some
R	Excellent	Excellent	Good	Moderate	Excellent	Yes
Python	Good	Good	Excellent	Moderate	Excellent	Yes

Program	Statistics	Visualization	Machine learning	Ease of use	Power/flexibility	Fun
Julia	OK	Excellent	Good	Hard	Excellent	Yes

1.5.3 Examples: Read data from a file & summarize

In this example we will compare the syntax for reading and summarizing data stored in a file.

- Stata

```
import delimited using "http://tutorials.iq.harvard.edu/R/Rgraphics/dataSets/EconomistData.csv"
sum
```

```
set more off
 "EconomistData.csv"
Picked up _JAVA_OPTIONS: -Dawt.useSystemAAFontSettings=gasp -Dswing.aatext=true -Dsun.java2d.oper
(6 vars, 173 obs)
sum

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
          v1 |        173          87    50.08493          1        173
     country |          0
     hdirank |        173    95.28324    55.00767          1        187
         hdi |        173    .6580867    .1755888       .286       .943
         cpi |        173    4.052023    2.116782        1.5        9.5
-------------+---------------------------------------------------------
      region |          0
```

- R

```
cpi <- read.csv("http://tutorials.iq.harvard.edu/R/Rgraphics/dataSets/EconomistData.csv")
summary(cpi)
```

```
      X              Country       HDI.Rank           HDI
Min.   :  1   Afghanistan:  1   Min.   :  1.00   Min.   :0.2860
1st Qu.: 44   Albania    :  1   1st Qu.: 47.00   1st Qu.:0.5090
Median : 87   Algeria    :  1   Median : 96.00   Median :0.6980
Mean   : 87   Angola     :  1   Mean   : 95.28   Mean   :0.6581
3rd Qu.:130   Argentina  :  1   3rd Qu.:143.00   3rd Qu.:0.7930
Max.   :173   Armenia    :  1   Max.   :187.00   Max.   :0.9430
              (Other)    :167
     CPI                      Region
Min.   :1.500   Americas         :31
1st Qu.:2.500   Asia Pacific     :30
Median :3.200   East EU Cemt Asia:18
Mean   :4.052   EU W. Europe     :30
3rd Qu.:5.100   MENA             :18
```

```
Max.   :9.500   SSA              :46
```

- Matlab

```
tmpfile = websave(tempname(), 'http://tutorials.iq.harvard.edu/R/Rgraphics/dataSets/EconomistData
cpi = readtable(tmpfile);
summary(cpi)
```

```
tmpfile = websave(tempname(), 'http://tutorials.iq.harvard.edu/R/Rgraphics/dataSets/EconomistData
cpi = readtable(tmpfile);
summary(cpi)

Variables:

    Var1: 173×1 cell array of character vectors

    Country: 173×1 cell array of character vectors

    HDI_Rank: 173×1 double

        Description:  Original column heading: 'HDI.Rank'
        Values:

            Min          1
            Median      96
            Max        187

    HDI: 173×1 double

        Values:

            Min        0.286
            Median     0.698
            Max        0.943

    CPI: 173×1 double

        Values:

            Min        1.5
            Median     3.2
            Max        9.5

    Region: 173×1 cell array of character vectors
'org_babel_eoe'

ans =

    'org_babel_eoe'
```

- Python

```
import pandas as pd
cpi = pd.read_csv('http://tutorials.iq.harvard.edu/R/Rgraphics/dataSets/EconomistData.csv')
cpi.describe(include = 'all')
```

```
Python 3.6.2 (default, Jul 20 2017, 03:52:27)
[GCC 7.1.1 20170630] on linux
Type "help", "copyright", "credits" or "license" for more information.
        Unnamed: 0 Country    HDI.Rank         HDI         CPI Region
count   173.000000     173  173.000000  173.000000  173.000000    173
unique         NaN     173         NaN         NaN         NaN      6
top            NaN    Oman         NaN         NaN         NaN    SSA
freq           NaN       1         NaN         NaN         NaN     46
mean     87.000000     NaN   95.283237    0.658087    4.052023    NaN
std      50.084928     NaN   55.007670    0.175589    2.116782    NaN
min       1.000000     NaN    1.000000    0.286000    1.500000    NaN
25%      44.000000     NaN   47.000000    0.509000    2.500000    NaN
50%      87.000000     NaN   96.000000    0.698000    3.200000    NaN
75%     130.000000     NaN  143.000000    0.793000    5.100000    NaN
max     173.000000     NaN  187.000000    0.943000    9.500000    NaN
```

1.5.4 Examples: Fit a linear regression

Fitting statistical models is pretty straight-forward in all popular programs.

- Stata

```
regress hdi cpi
```

```
regress hdi cpi

      Source |       SS           df       MS      Number of obs   =       173
-------------+----------------------------------   F(1, 171)       =    168.85
       Model |  2.63475703         1  2.63475703   Prob > F        =    0.0000
    Residual |   2.6682467       171  .015603782   R-squared       =    0.4968
-------------+----------------------------------   Adj R-squared   =    0.4939
       Total |  5.30300372       172  .030831417   Root MSE        =    .12492

------------------------------------------------------------------------------
         hdi |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
         cpi |   .0584696   .0044996    12.99   0.000     .0495876    .0673515
       _cons |   .4211666   .0205577    20.49   0.000     .3805871    .4617462
------------------------------------------------------------------------------
```

- R

```
summary(lm(HDI ~ CPI, data = cpi))
```

```
Call:
lm(formula = HDI ~ CPI, data = cpi)

Residuals:
     Min       1Q   Median       3Q      Max
-0.28452 -0.08380  0.01372  0.09157  0.24104

Coefficients:
            Estimate Std. Error t value Pr(>|t|)
(Intercept)  0.42117    0.02056   20.49   <2e-16 ***
CPI          0.05847    0.00450   12.99   <2e-16 ***
---
Signif. codes:  0 ‘***’ 0.001 ‘**’ 0.01 ‘*’ 0.05 ‘.’ 0.1 ‘ ’ 1

Residual standard error: 0.1249 on 171 degrees of freedom
Multiple R-squared:  0.4968,    Adjusted R-squared:  0.4939
F-statistic: 168.9 on 1 and 171 DF,  p-value: < 2.2e-16
```

- Matlab

```
fitlm(cpi, 'HDI~CPI')
```

```
fitlm(cpi, 'HDI~CPI')

ans =


Linear regression model:
    HDI ~ 1 + CPI

Estimated Coefficients:
                   Estimate       SE        tStat       pValue
                   --------    ---------    ------    ----------

    (Intercept)    0.42117      0.020558    20.487    6.7008e-48
    CPI            0.05847     0.0044996    12.994    2.6908e-27


Number of observations: 173, Error degrees of freedom: 171
Root Mean Squared Error: 0.125
R-squared: 0.497,  Adjusted R-Squared 0.494
F-statistic vs. constant model: 169, p-value = 2.69e-27
'org_babel_eoe'

ans =

    'org_babel_eoe'
```

- Python

```
import statsmodels.formula.api as model
X = cpi[['CPI']]
Y = cpi[['HDI']]
model.OLS(Y, X).fit().summary()
```

```
<class 'statsmodels.iolib.summary.Summary'>
"""
                            OLS Regression Results
==============================================================================
Dep. Variable:                    HDI   R-squared:                       0.885
Model:                            OLS   Adj. R-squared:                  0.884
Method:                 Least Squares   F-statistic:                     1325.
Date:                Thu, 31 Aug 2017   Prob (F-statistic):           9.89e-83
Time:                        23:16:45   Log-Likelihood:                 8.1584
No. Observations:                 173   AIC:                            -14.32
Df Residuals:                     172   BIC:                            -11.16
Df Model:                           1
Covariance Type:            nonrobust
==============================================================================
                 coef    std err          t      P>|t|      [0.025      0.975]
------------------------------------------------------------------------------
CPI            0.1402      0.004     36.401      0.000       0.133       0.148
==============================================================================
Omnibus:                       10.423   Durbin-Watson:                   1.616
Prob(Omnibus):                  0.005   Jarque-Bera (JB):               11.099
Skew:                          -0.599   Prob(JB):                      0.00389
Kurtosis:                       2.674   Cond. No.                         1.00
==============================================================================

Warnings:
[1] Standard Errors assume that the covariance matrix of the errors is correctly specified.
"""
```

1.5.5 Examples: Extract links for .html file

Retrieving data from a website is a common task. Here we parse a simple web page containing links to files we wish to download.

- Stata

```
disp "Ha ha ha! No, you do not want to use Stata for this!"
```

```
disp "Ha ha ha! No, you do not want to use Stata for this!"
Ha ha ha! No, you do not want to use Stata for this!
```

- R

```
library(xml2)
index_page <- read_html("http://tutorials.iq.harvard.edu/example_data/baby_names/EW/")
```

```
all_anchors <- xml_find_all(index_page, "//a")
all_hrefs <- xml_attr(all_anchors, "href")
data_hrefs <- grep("\\.csv$", all_hrefs, value = TRUE)
data_links <- paste0("http://tutorials.iq.harvard.edu/example_data/baby_names/EW/", data_hrefs)
data_links
```

```
 [1] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1996.csv"
 [2] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1997.csv"
 [3] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1998.csv"
 [4] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1999.csv"
 [5] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2000.csv"
 [6] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2001.csv"
 [7] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2002.csv"
 [8] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2003.csv"
 [9] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2004.csv"
[10] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2005.csv"
[11] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2006.csv"
[12] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2007.csv"
[13] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2008.csv"
[14] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2009.csv"
[15] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2010.csv"
[16] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2011.csv"
[17] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2012.csv"
[18] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2013.csv"
[19] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2014.csv"
[20] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2015.csv"
[21] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1996.csv"
[22] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1997.csv"
[23] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1998.csv"
[24] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1999.csv"
[25] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2000.csv"
[26] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2001.csv"
[27] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2002.csv"
[28] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2003.csv"
[29] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2004.csv"
[30] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2005.csv"
[31] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2006.csv"
[32] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2007.csv"
[33] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2008.csv"
[34] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2009.csv"
[35] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2010.csv"
[36] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2011.csv"
[37] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2012.csv"
[38] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2013.csv"
[39] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2014.csv"
[40] "http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2015.csv"
```

- Matlab

```
index_page = urlread('http://tutorials.iq.harvard.edu/example_data/baby_names/EW/');
all_hrefs = regexp(index_page, '<a href="([^"]*\.csv)">', 'tokens')';
all_hrefs = [all_hrefs{:}]';
all_links = strcat('http://tutorials.iq.harvard.edu/example_data/baby_names/EW/', all_hrefs)
```

```
index_page = urlread('http://tutorials.iq.harvard.edu/example_data/baby_names/EW/');
all_hrefs = regexp(index_page, '<a href="([^"]*\.csv)">', 'tokens')';
all_hrefs = [all_hrefs{:}]';
all_links = strcat('http://tutorials.iq.harvard.edu/example_data/baby_names/EW/', all_hrefs)

all_links =

  40×1 cell array

    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1996.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1997.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1998.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1999.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2000.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2001.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2002.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2003.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2004.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2005.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2006.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2007.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2008.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2009.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2010.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2011.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2012.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2013.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2014.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2015.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1996.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1997.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1998.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1999.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2000.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2001.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2002.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2003.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2004.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2005.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2006.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2007.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2008.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2009.csv'
```

```
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2010.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2011.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2012.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2013.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2014.csv'
    'http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2015.csv'
'org_babel_eoe'

ans =

    'org_babel_eoe'
```

- Python

```
from lxml import etree
import requests

index_text = requests.get('http://tutorials.iq.harvard.edu/example_data/baby_names/EW/').text
index_page = etree.HTML(index_text)
all_hrefs = [a.values() for a in index_page.findall(".//a")]
data_links = ['http://tutorials.iq.harvard.edu/example_data/baby_names/EW/' +
              href[0] for href in all_hrefs if 'csv' in href[0]]
for link in data_links:
    print(link)
```

```
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1996.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1997.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1998.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_1999.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2000.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2001.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2002.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2003.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2004.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2005.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2006.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2007.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2008.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2009.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2010.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2011.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2012.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2013.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2014.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/boys_2015.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1996.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1997.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1998.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_1999.csv
```

```
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2000.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2001.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2002.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2003.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2004.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2005.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2006.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2007.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2008.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2009.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2010.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2011.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2012.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2013.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2014.csv
http://tutorials.iq.harvard.edu/example_data/baby_names/EW/girls_2015.csv
```

1.6 Creating reports

Once you've analyzed your data you'll most likely want to communicate your results. For short informal projects this might take the form of a blog post or an email to your colleagues. For larger more formal projects you'll likely want to prepare a substantial report or manuscript for disseminating your findings via a journal publication or other means. Other common means of reporting research findings include posters or slides for a conference talk.

Regardless of the type of report, you may choose to use either a *markup language* or a WYSIWYG application like Microsoft Word/Powerpoint of a desktop publishing application such as Adobe InDesign.

1.6.1 Markup languages

A markup language is a system for producing a formatted document from a text file using information by the markup. A major advantage of markup languages is that the formatting instructions can be easily generated by the program you use for analyzing your data.

Markup languages include *HTML*, *LaTeX*, *Markdown* and many others. *LaTeX* and *Markdown* are currently popular among data scientists, although others are used as well.

Markdown is easy to write and designed to be human-readable. It is newer and somewhat less feature-full compared to LaTeX. It's main advantage is simplicity. *LaTeX* is more verbose but provides for just about any feature you'll ever need.

MARKDOWN DEMO LATEX DEMO

1.6.2 Word processors

Modern word processors are largely just graphical user interfaces that write a markup language (usually XML) for you. They are commonly used for creating reports, but care must be taken when doing so.

If you use a word processor to produce your reports you should

- use the structured outline feature,
- link rather than embed external resources (figures, tables, etc.),
- use cross-referencing features, and
- use a bibliography management system.

WORD PROCESSOR DEMO

1.7 Text editors & Integrated Development Environments

A text editor edits text obviously. But that is not all! At a minimum, a text editor will also have a mechanism for reading and writing text files. Most text editors do much more than this.

An IDE provides tools for working with code, such as syntax highlighting, code completion, jump-to-definition, execute/compile, package management, refactoring, etc. Of course an IDE includes a text editor.

Editors and IDE's are not really separate categories; as you add features to a text editor it becomes more like an IDE, and a simple IDE may provide little more than a text editor. For example, Emacs is commonly referred to as a text editor, but it provides nearly every feature you would expect an IDE to have.

A more useful distinction is between language-specific editors/IDEs and general purpose editors/IDEs. The former are typically easier to set up since they come pre-configured for use with a specific language. General purpose editors/IDEs typically provide language support via *plugins* and may require extensive configuration for each language.

1.7.1 Language specific editors & IDEs

Editor	Features	Ease of use	Language
RStudio	Excellent	Easy	R
Spyder	Excellent	Easy	Python
Stata do file editor	OK	Easy	Stata
SPSS syntax editor	OK	Easy	SPSS

LANGUAGE SPECIFIC IDE DEMO

1.7.2 General purpose editors & IDEs

Editor	Features	Ease of use	Language support
Vim	Excellent	Hard	Good
Emacs	Excellent	Hard	Excellent
VS code	Excellent	Easy	Very good
Atom	Good	Moderate	Good
Eclipse	Excellent	Easy	Good
Sublime Text	Good	Easy	Good
Notepad++	OK	Easy	OK
Textmate	Good	Moderate	Good
Kate	OK	Easy	Good

GENERAL PURPOSE EDITOR DEMO

1.8 Literate programming & notebooks

In one of the Early demos we say an example of embedding R code in a markdown document. A closely related approach is to create a *notebook* that includes the prose of the report, the code used for the analysis, and the results produced by that code.

1.8.1 Literate programming

Literate programming is the practice of embedding computer code in a natural language document. For example, using *RMarkdown* we can embed R code in a report authored using Markdown. Python and Stata have their own versions of literate programming using Markdown.

1.8.2 Notebooks

Notebooks go one step farther, and include the output produced by the original program directly in the notebook. Examples include *Jupyter*, *Appache Zeppelin*, and *Emacs Org Mode*.

NOTEBOOKS DEMO

1.9 Big data, annoying data, & computationally intensive methods

Thus far we've discussed popular programming languages, data storage and retrieval options, text editors, and reporting technology. These are the basic building blocks I recommend using just about any time you find yourself working with data. There are times however when more is needed. For example, you may wish to use distributed computing for large or resource intensive computations.

1.9.1 Computing clusters at Harvard

Harvard provides a number of computing clusters, including Odyssey and the Research Computing Environment. Using these systems will be much easier if you know the basic tools well. After all, you're still going to need data storage/retrieval, you'll still need a text editor write code, and a programming language to write it in. My advice is to master these basics, and learn the rest as you need it.

1.10 Wrap up

1.10.1 Feedback

These workshops are a work-in-progress, please provide any feedback to: `help@iq.harvard.edu`

1.10.2 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 II

R

Chapter 2

R Introduction

Topics

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

2.1 Setup

2.1.1 Class Structure

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

2.1.2 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/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.

2.1.3 Prerequisites

- None — assumes no prior knowledge of R
- Relatively slow-paced

2.1.4 Learning Outcomes

- Basic R syntax
- R package ecosystem
- Reading / writing data and finding help
- Data cleaning and manipulation
- Conditional operations

2.1.5 What is R?

R is a free language and environment for statistical computing and graphics. R has existed for over 25 years and is now the most popular software for data analysis. It has an extensive ecosystem of about 15,000 add on packages covering all aspects of statistics.

2.1.6 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) with many features.

There are also several different formats for writing code for R. Two of the most popular are:

1. **R scripts** — a type of plain text file that allows you to write R code and basic comments about the code:
2. **Rmarkdown** — a type of text file that allows you to include plain text with R code and easily convert the contents into HTML (for a webpage), MS Word, or PDF (via LaTeX). Many people write their journal papers, dissertations, and statistics/math class notes in Rmarkdown, since it is easy to use and to convert into other formats later.

Here are some resources for learning more about `rmarkdown`: `https://cran.r-project.org/web/packages/rmarkdown/vignettes/rmarkdown.html`

The following RStudio and `rmarkdown` cheatsheets will also provide a useful reference: `https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets_2019.pdf`

2.2 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.

2.3 R basics

2.3.1 Syntax

- R is case sensitive
- R ignores white space
- Object names should start with a letter

2.3.2 Asking for help

1. 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")
```

2. If you know the name of the package you want to use, then Googling "R *package-name*" will often get you to the documentation. Packages are hosted on several different repositories, including:

- CRAN: `https://cran.r-project.org/web/packages/available_packages_by_name.html`
- Bioconductor: `https://www.bioconductor.org/packages/release/bioc/`
- Github: `http://rpkg.gepuro.net/`
- R-Forge: `https://r-forge.r-project.org/R/?group_id=1326`

3. If you know the type of analysis you want to perform, you can Google "CRAN Task Views", where there are curated lists of packages `https://cran.r-project.org/web/views/`. If you want to know which packages are popular, you can look at `https://r-pkg.org`.

2.3.3 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!
```

2.3.4 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.

2.4 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.

2.4.1 Installing & using R packages

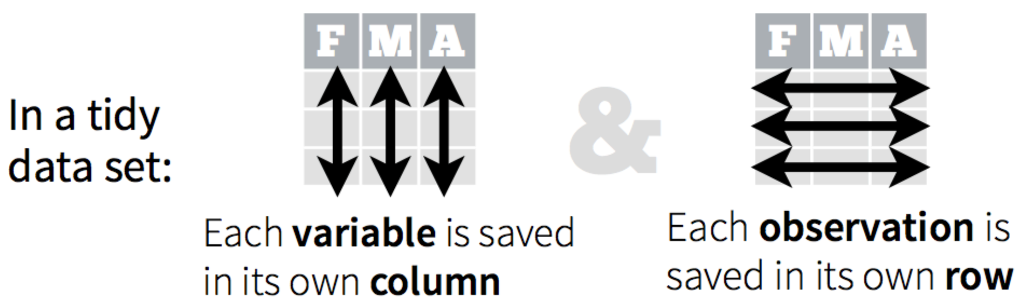
R is a modular environment that is extended by the use of **packages**. Packages are collections of functions or commands that are designed to perform specific tasks (e.g., fit a type of regression model). A large number of contributed packages are available (> 15,000).

Using an R package is a **two step process**:

1. Install the package onto your computer using the `install.packages()` function. This only needs to be done the **first time** you use the package.

2. Load the package into your R session's search path using the `library()` function. This needs to be done **each time** you use the package.

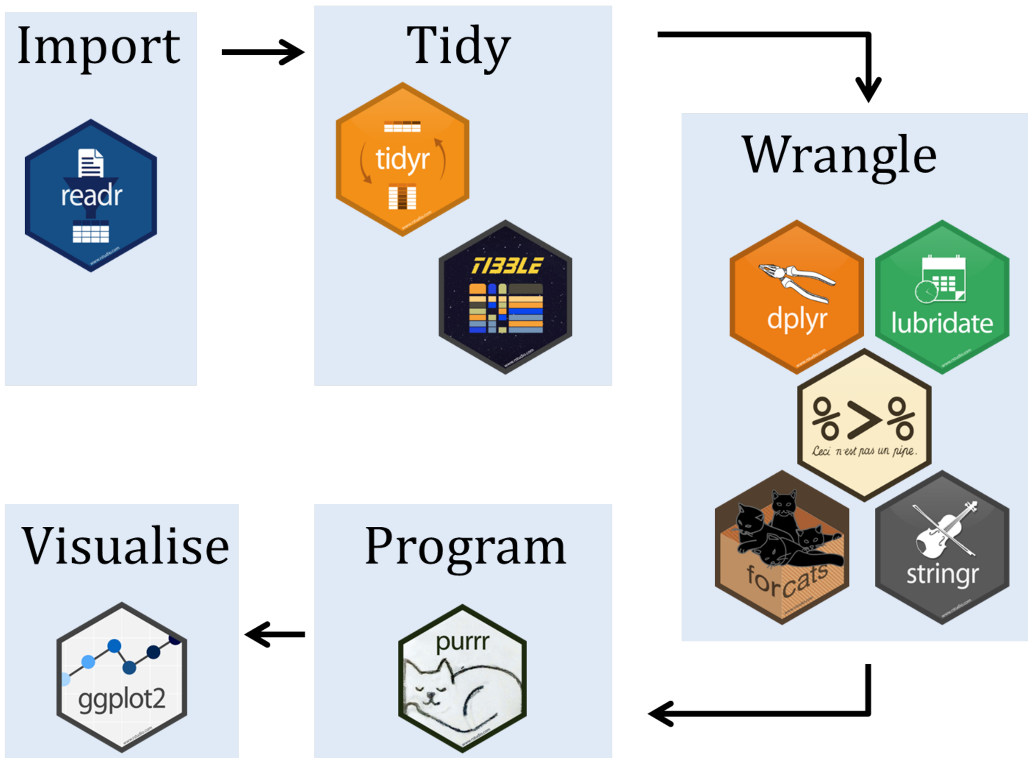
While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called `tidyverse` have been designed specifically for data science. All packages included in `tidyverse` share an underlying design philosophy, grammar, and data structures. This philosopy is rooted in the idea of "tidy data":



We will use `tidyverse` packages throughout the workshop, so let's install them now:

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

# when you install tidyverse for the first time you will be asked
# a question in the Console - please answer by typing "no" in the Console.

library(tidyverse)
```

A typical workflow for using `tidyverse` packages looks like this:



We can also install the `rmarkdown` package, which will allow us to combine our text and code into a formatted document at the end of the workshop:

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

2.4.2 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	`read_csv()`
tab separated	`read_delim()`
other delimited formats	`read_table()`
fixed width	`read_fwf()`

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.

2.4.3 Baby names data

As an example project we will analyze the popularity of baby names in the US from 1960 through 2017. The data were retrieved from `https://catalog.data.gov/dataset/baby-names-from-social-security-card-applications-national-level-data`.

Here are the questions we will use R to answer:

1. In which year did your name (or another name) occur most frequently by **count**?
2. Which names have the highest popularity by **proportion** for each sex and year?
3. How does the percentage of babies given one of the top 10 names of the year change over time?

2.5 Exercise 1

Reading the baby names data

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

1. Open the `read_csv` help page to determine how to use it to read in data.

```
##
```

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

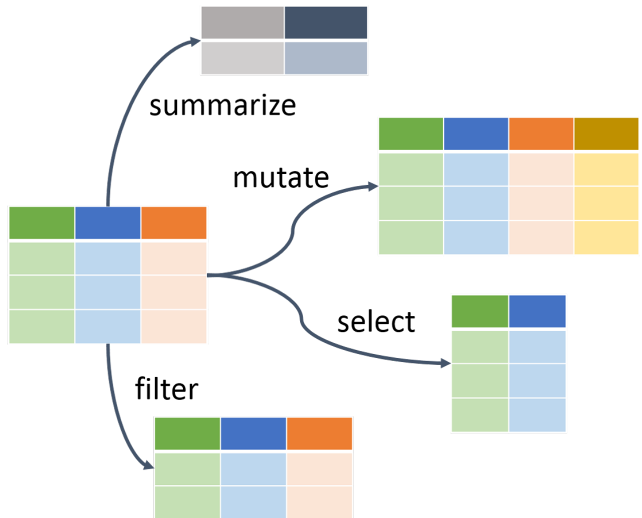
```
##
```

2.6 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`. Think of a `data.frame` as a spreadsheet.

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, as well as create new variables.



2.6.1 Filtering, selecting, & 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".

```
## Read in the baby names data if you haven't already
baby_names <- read_csv("babyNames.csv")
```

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

print(baby_names_alexmark) # explicit printing
baby_names_alexmark # implicit printing
```

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. If we would just like to glance at the first few lines we can use the `head()` function:

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

```
head(baby_names_subset)
head(baby_names_subset, n = 6) # default is n = 6
```

2.6.2 Other logical operators

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

Operator	Meaning
`==`	equal to
`!=`	not equal to
`>`	greater than
`>=`	greater than or equal to
`<`	less than
`<=`	less than or equal to
`%in%`	contained in

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

```
x <- 1:10 # a vector
x

x > 7 # a simple condition
x > 7 | x < 3 # two conditions combined

# x %in% vector
# elements of x matched in vector
x %in% c(1, 5, 10)
```

2.7 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`. 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.

```
##
```

2.8 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".

```
# unpiped version
baby_names_alexmark <- filter(baby_names, Year == 1992 & (Name == "Alex" | Name == "Mark"))
arrange(baby_names_alexmark, desc(Count))
```

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

Hint: try pronouncing "then" whenever you see `%>%`.

```
# pseudocode

# unpiped version
filter(dataset, condition)

# piped version
dataset %>% filter(condition)

# what the pipe is doing
output_of_thing_on_left %>% becomes_input_of_thing_on_right
```

Advantages of using the pipe:

1. We can avoid creating intermediate variables, such as `baby_names_alexmark`
2. Less to type
3. Easier to read and follow the logic (especially avoiding using nested functions)

2.9 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`.

```
##
```

2.10 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 a *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)
```

2.11 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 is shows boys and girls in different colors.

```
##
```

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

```
##
```

2.12 Finding the most popular names

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

2.12.1 Computing better measures of popularity

So far we've used `Count` as a measure of popularity. A better approach is to use proportion 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)
head(baby_names)
```

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

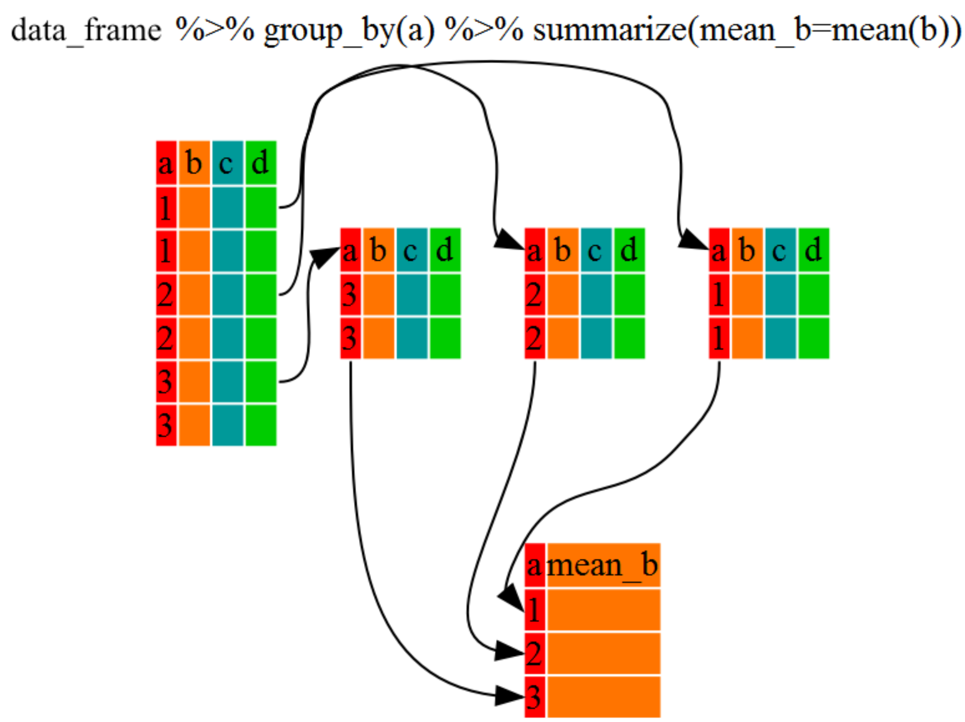
```
baby_names %>%
  select(Name, Sex, Year, Count_1k) %>%
  head()
```

2.12.2 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()`.



2.13 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(desc(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: `Year`, `Name`, `Sex`, and `Proportion`.

```
##
```

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?

```
##
```

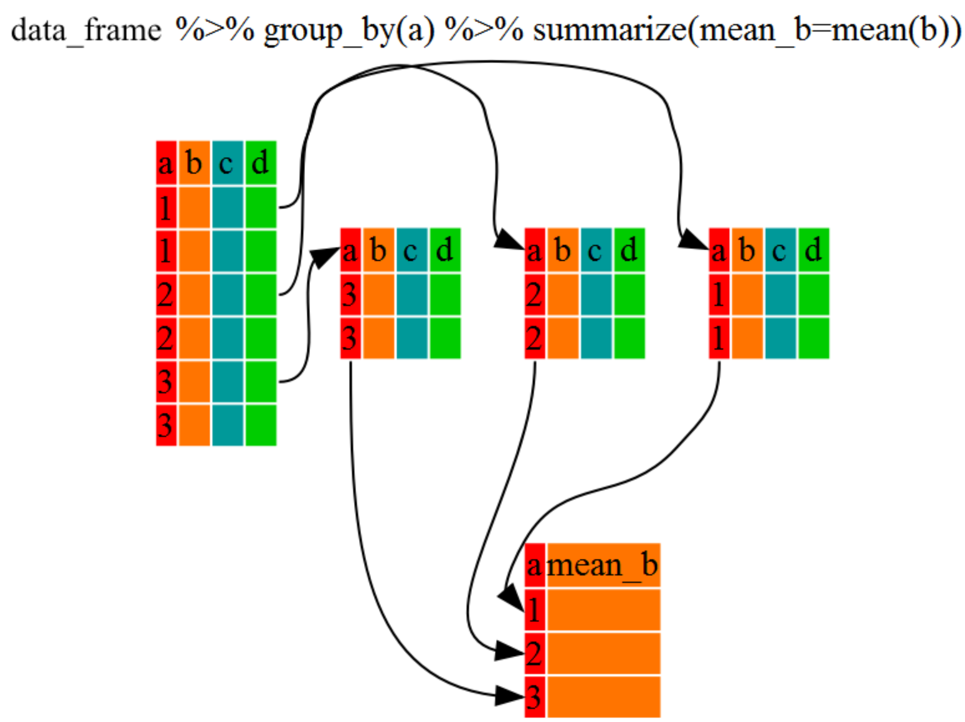
2.14 Percent choosing one of the top 10 names

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)) %>%
  ungroup()

head(bn_by_year)
```

2.15 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.

```
##
```

2.16 Saving our Work

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

2.16.1 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")
```

2.16.2 Saving multiple datasets

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

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

2.16.3 Saving & loading R workspaces

In addition to importing individual datasets, R can save and load entire "workspaces". The workspace is your current R working environment and includes any user-defined objects. At the end of a session, you can save an "image" of the current workspace, which allows you to automatically reload the objects you previously created.

```
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
```

2.17 Exercise solutions

2.17.1 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".
```

```
##
```

2.17.2 Ex 1: prototype

```
## read ?read_csv
```

```
baby_names <- read_csv("babyNames.csv")
```

2.17.3 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))
```

2.17.4 Ex 2.2: prototype

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

2.17.5 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 is 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")
```

2.17.6 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(desc(Count))` for each `Year X Sex` group.
```

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

2.17.7 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")
```

2.18 Wrap-up

2.18.1 Feedback

These workshops are a work-in-progress, please provide any feedback to: `help@iq.harvard.edu`

2.18.2 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/`
- Cheatsheets
  - `https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets_2019.pdf`
- Online tutorials
  - `http://www.codeschool.com/courses/try-r`
  - `http://www.datacamp.org`
  - `https://rmarkdown.rstudio.com/lesson-1.html`
  - `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/~hiebeler/comp/matlabR.pdf`
  - Python: `http://mathesaurus.sourceforge.net/matlab-python-xref.pdf`

Chapter 3

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

3.1 Setup

3.1.1 Class Structure

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

3.1.2 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/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.

3.1.3 Installing & using R packages

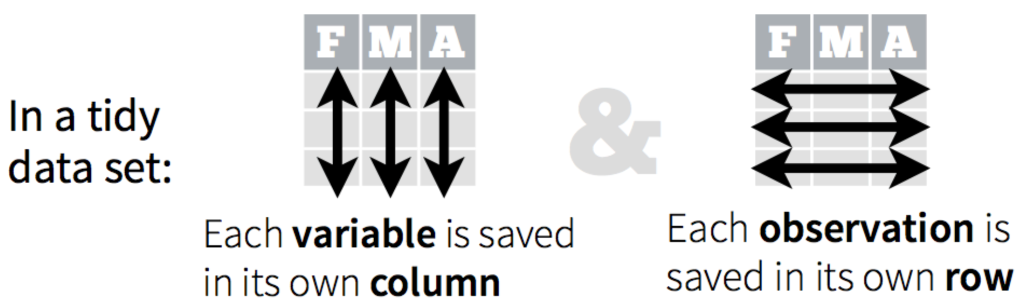
R is a modular environment that is extended by the use of **packages**. Packages are collections of functions or commands that are designed to perform specific tasks (e.g., fit a type of regression model). A large number of contributed packages are available (> 15,000).

Using an R package is a **two step process**:

1. Install the package onto your computer using the `install.packages()` function. This only needs to be done the **first time** you use the package.

2. Load the package into your R session's search path using the `library()` function. This needs to be done **each time** you use the package.

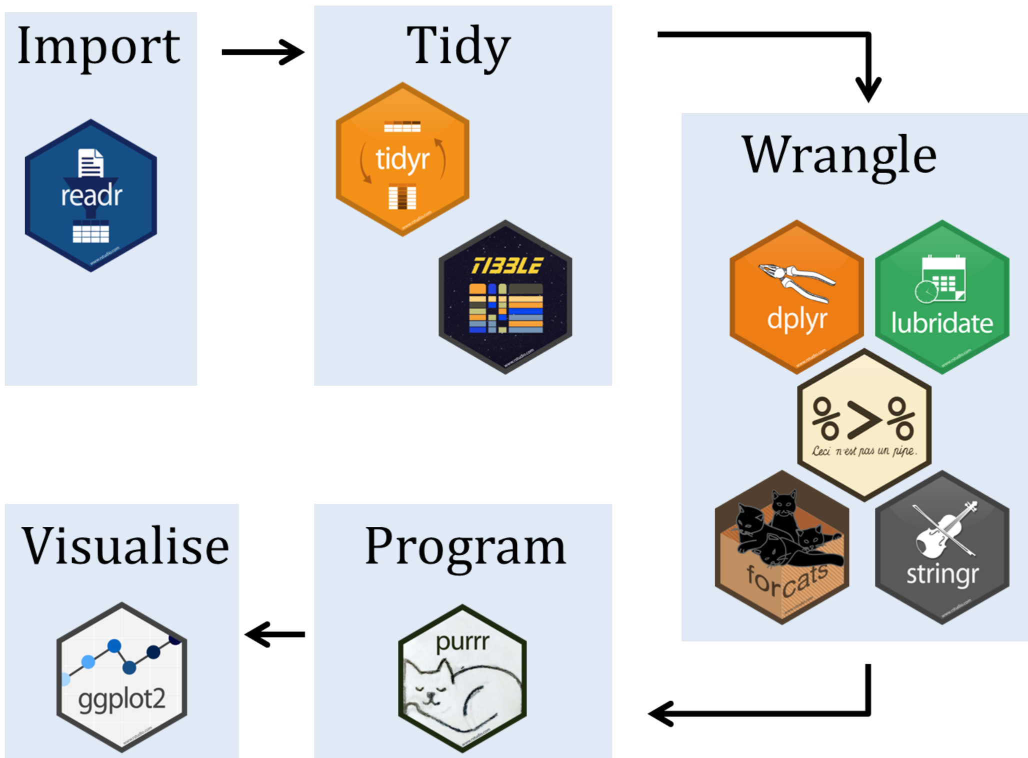
While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called tidyverse have been designed specifically for data science. All packages included in `tidyverse` share an underlying design philosophy, grammar, and data structures. This philosopy is rooted in the idea of "tidy data":



We will use `tidyverse` packages throughout the workshop, so let's install them now:

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

# when you install tidyverse for the first time you will be asked
# a question in the Console - please answer by typing "no" in the Console.

library(tidyverse)
```

A typical workflow for using `tidyverse` packages looks like this:



We can also install the `rmarkdown` package, which will allow us to combine our text and code into a formatted document at the end of the workshop:

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

The following RStudio, `tidyverse`, and `rmarkdown` cheatsheets will provide a useful reference: `https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets_2019.pdf`

Finally, lets install some packages that will help with modeling:

```
# install.packages("lme4")
library(lme4)   # for mixed models

# install.packages("emmeans")
library(emmeans)   # for marginal effects

# install.packages("effects")
library(effects)   # for predicted marginal means
```

3.1.4 Prerequisites

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.

3.1.5 Learning Outcomes

- R formula interface
- Fitting linear and non-linear models
- Understanding class methods
- Post-estimation tools
- Plotting model diagnostics

3.1.6 Workshop Outline

1. Preliminary steps before modeling
2. Modeling continuous outcomes
3. Modeling binary outcomes
4. Modeling clustered data

3.2 Before fitting a model

3.2.1 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")

  # look at the last few rows
  tail(states_data)
```

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

Variable	Description
metro	Metropolitan area population, %
waste	Per capita solid waste, tons

3.2.2 Examine the data

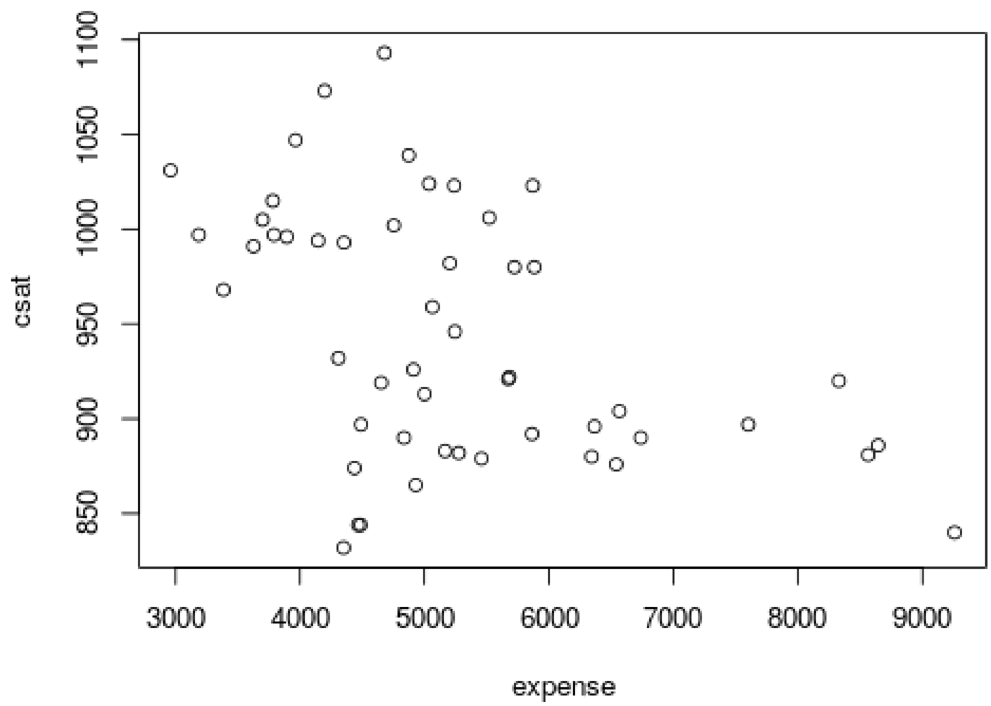
Start by examining the data to check for problems.

```
# 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)
```

3.2.3 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)
```



3.3 Models 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-pupal expenditures:

```
# R regression formula
outcome ~ pred1 + pred2 + pred3

# NOTE the ~ is a tilde

  # Fit our regression model
  sat_mod <- lm(csat ~ 1 + expense, # regression formula
                data=states_data) # data

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

3.3.1 Why is the association between expense & 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 ~ 1 + expense + percent, data = states_data) %>%
summary()
```

3.3.2 The `lm` class & 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)
```

Selected **post-estimation** tools:

Function	Package	Output
`summary()`	`stats` base R	standard errors, test statistics, p-values, GOF stats
`confint()`	`stats` base R	confidence intervals
`anova()`	`stats` base R	anova table (one model), model comparison (> one model)
`coef()`	`stats` base R	point estimates
`drop1()`	`stats` base R	model comparison
`predict()`	`stats` base R	predicted response values
`fitted()`	`stats` base R	predicted response values (for observed data)
`residuals()`	`stats` base R	residuals
`fixef()`	`lme4`	fixed effect point estimates (mixed models only)
`ranef()`	`lme4`	random effect point estimates (mixed models only)
`allEffects()`	`effects`	predicted marginal means
`emmeans()`	`emmeans`	predicted marginal means & marginal effects

3.3.3 OLS regression assumptions

OLS regression relies on several assumptions, including:

1. The model includes all relevant variables (i.e., no omitted variable bias).
2. The model is linear in the parameters (i.e., the coefficients and error term).
3. The error term has an expected value of zero.
4. All right-hand-side variables are uncorrelated with the error term.
5. No right-hand-side variables are a perfect linear function of other RHS variables.
6. Observations of the error term are uncorrelated with each other.
7. The error term has constant variance (i.e., homoscedasticity).
8. (Optional - only needed for inference). The error term is normally distributed.

Investigate assumptions #7 and #8 visually by plotting your model:

```
par(mfrow = c(2, 2)) # splits the plotting window into 4 panels
plot(sat_mod)
```

3.3.4 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 ~ 1 + expense + house + senate,
                     data = na.omit(states_data))

summary(sat_voting_mod) %>% coef()
```

```
# what does na.omit() do?

dat <- data.frame(
  x = 1:5,
  y = c(3, 2, 1, NA, 5),
  z = c(6, NA, 2, 7, 3))
dat

na.omit(dat) # listwise deletion of observations

# also see
?complete.cases
dat[with(dat, complete.cases(expense, house, senate)), ]
```

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

# compare using an F-test with the anova() function
anova(sat_mod, sat_voting_mod)
```

3.4 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?

3.5 Interactions & factors

3.5.1 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 ~ 1 + expense + income + expense : income, data=states_data)
sat_expense_by_percent <- lm(csat ~ 1 + expense * income, data=states_data) # same as above, b

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

3.5.2 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)

# arguments to the factor() function
# factor(x, levels, labels)

levels(states_data$region)
```

```
# Add region to the model
sat_region <- lm(csat ~ 1 + 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!**

3.5.3 Setting factor reference groups & contrasts

The default contrasts in R are "treatment" contrasts, or "dummy coding", with the first level as the reference category. We can also easily change the reference group or get all sets of pairwise contrasts.

```
# change the reference group
states_data$region <- relevel(states_data$region, ref = "Midwest")
m1 <- lm(csat ~ 1 + region, data=states_data)
summary(m1) %>% coef()

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

3.6 Exercise 1

Interactions & 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?

```
##
```

3.7 Models with binary outcomes

3.7.1 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.

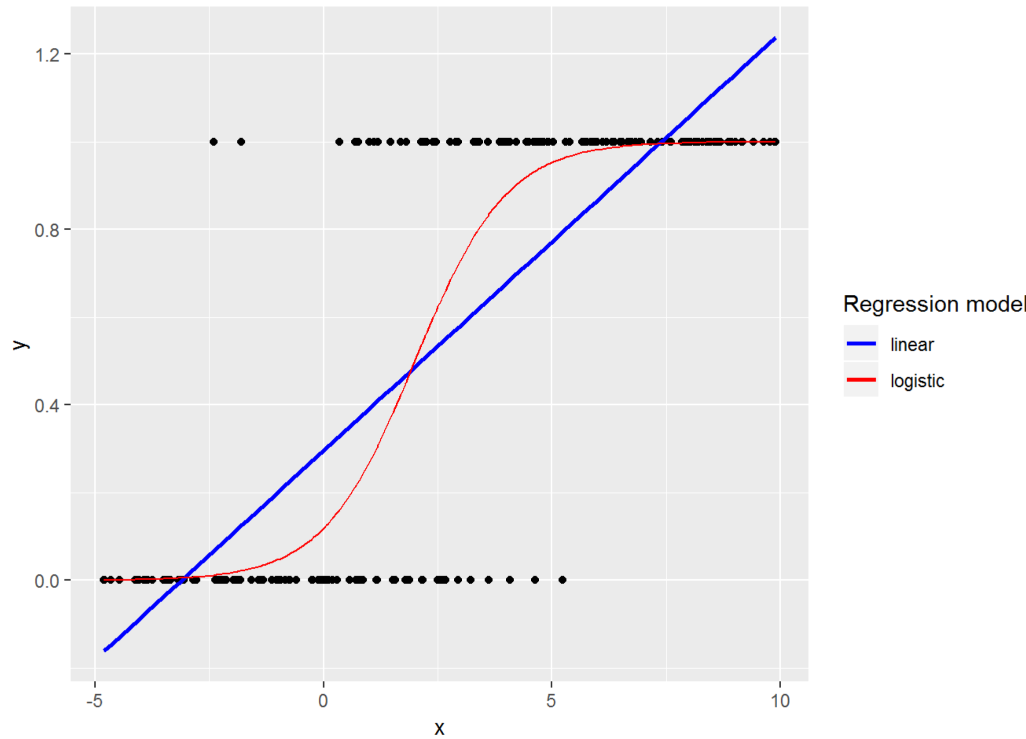
Load the National Health Interview Survey data:

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

3.7.2 Logistic regression example

Motivation for a logistic regression model; With a binary response:

1. Errors will not be normally distributed
2. Variance will not be homoskedastic
3. Predictions should be constrained to be on the interval [0, 1]



Anatomy of a glm:

```
# OLS model using lm()
# lm(outcome ~ 1 + pred1 + pred1, data = mydata)

# OLS model using glm()
# glm(outcome ~ 1 + pred1 + pred1, data = mydata, family = gaussian(link = "identity"))

# logistic model using glm()
# glm(outcome ~ 1 + pred1 + pred1, data = mydata, family = binomial(link = "logit"))
```

The `family` argument sets the error distribution for the model, while the `link` function argument relates the predictors to the expected value of the outcome.

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

```
str(NH11$hypev) # check stucture of hypev
levels(NH11$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 ~ 1 + age_p + sex + sleep + bmi,
```

```
                data = NH11, family = binomial(link = "logit"))
summary(hyp_out) %>% coef()
```

3.7.3 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!

IMAGE HERE FOR SCALES

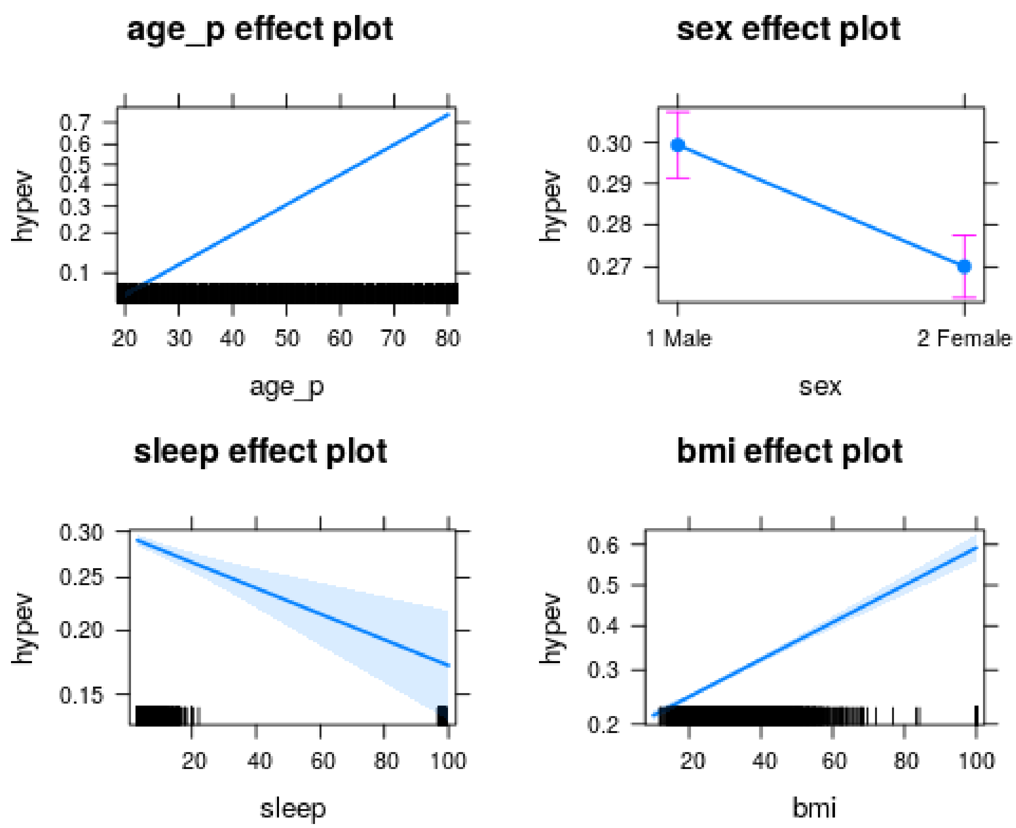
One solution is to transform the coefficients to make them easier to interpret. Here we transform into odds ratios:

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

3.7.4 Packages for computing & graphing predicted values

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

```
eff <- allEffects(hyp_out)
plot(eff, type = "response") # "response" refers to the probability scale

# generate a sequence at which to get predictions of the outcome
seq(20, 80, by = 5)

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



3.8 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`). Make sure you only keep the following two levels for `everwrk` (`1 Yes` and `2 No`). Hint: use the `factor()` function. Also, make sure to drop any `r_maritl` levels that do not contain observations. Hint: see `?droplevels`.

```
##
```

2. Predict the probability of working for each level of marital status. Hint: use `allEffects()`

```
##
```

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

3.9 Multilevel modeling

3.9.1 Multilevel modeling overview

- Multi-level (AKA hierarchical) models are a type of **mixed-effects** models
- Used to model data that are clustered
- Mixed-effecs 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 linear mixed models

3.9.2 The Exam data

The Exam data set contans 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")
```

3.9.3 The null model & 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).

```
# anatomy of lmer() function
# lmer(outcome ~ 1 + pred1 + pred2 + (1 | grouping_variable), data=mydata, REML = FALSE)


# 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 .161/(.161 + .852) = .159 = 16% 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).

3.9.4 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)
```

3.9.5 Multiple degree of freedom comparisons

As with `lm()` and `glm()` models, you can compare the two `lmer()` models using a likelihood ratio test with the `anova()` function.

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

3.9.6 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)
```

3.9.7 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 using a likelihood ratio test:

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

3.10 Exercise 3

Multilevel modeling

Use the `bh1996` dataset:

```
## install.packages("multilevel")
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.

```
##
```

3.11 Exercise solutions

3.11.1 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 ~ 1 + metro + pop + waste, data = states)
summary(mod_en_met_pop_waste)
anova(mod_en_met, mod_en_met_pop_waste)
```

3.11.2 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 ~ 1 + 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 ~ 1 + metro * waste + region, data = states)
anova(mod_en_region)
```

3.11.3 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`). Make sure you only keep the following two levels for `everwrk` (`1 Yes` and `2 No`). Hint: use the `factor()` function. Also, make sure to drop any `r_maritl` levels that do not contain observations. Hint: see `?droplevels`.

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

mod_wk_age_mar <- glm(everwrk ~ 1 + 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. Hint: use `allEffects()`

```
eff <- allEffects(mod_wk_age_mar)
as.data.frame(eff[["r_maritl"]])
```

3.11.4 Ex 3: prototype

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`)

```
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 ~ 1 + 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 ~ 1 + HRS + LEAD + (1 + HRS | GRP), data = bh1996)
anova(mod_grp1, mod_grp2)
```

3.12 Wrap-up

3.12.1 Feedback

These workshops are a work in progress, please provide any feedback to: `help@iq.harvard.edu`

3.12.2 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 4

R Graphics

Topics

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

4.1 Setup

4.1.1 Class Structure

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

4.1.2 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/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.

4.1.3 Installing & using R packages

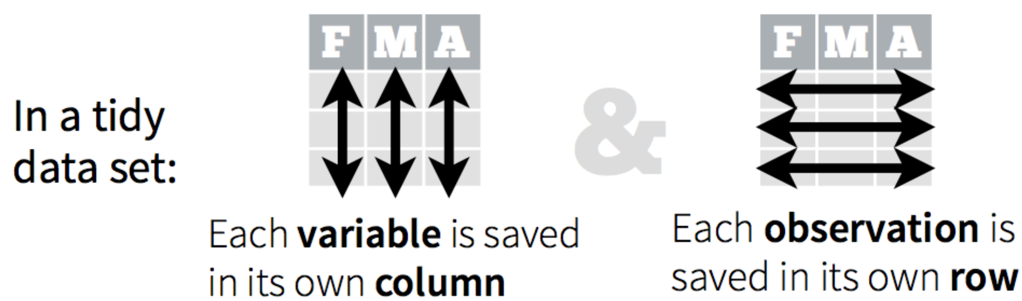
R is a modular environment that is extended by the use of **packages**. Packages are collections of functions or commands that are designed to perform specific tasks (e.g., fit a type of regression model). A large number of contributed packages are available (> 15,000).

Using an R package is a **two step process**:

1. Install the package onto your computer using the `install.packages()` function. This only needs to be done the **first time** you use the package.

2. Load the package into your R session's search path using the `library()` function. This needs to be done **each time** you use the package.

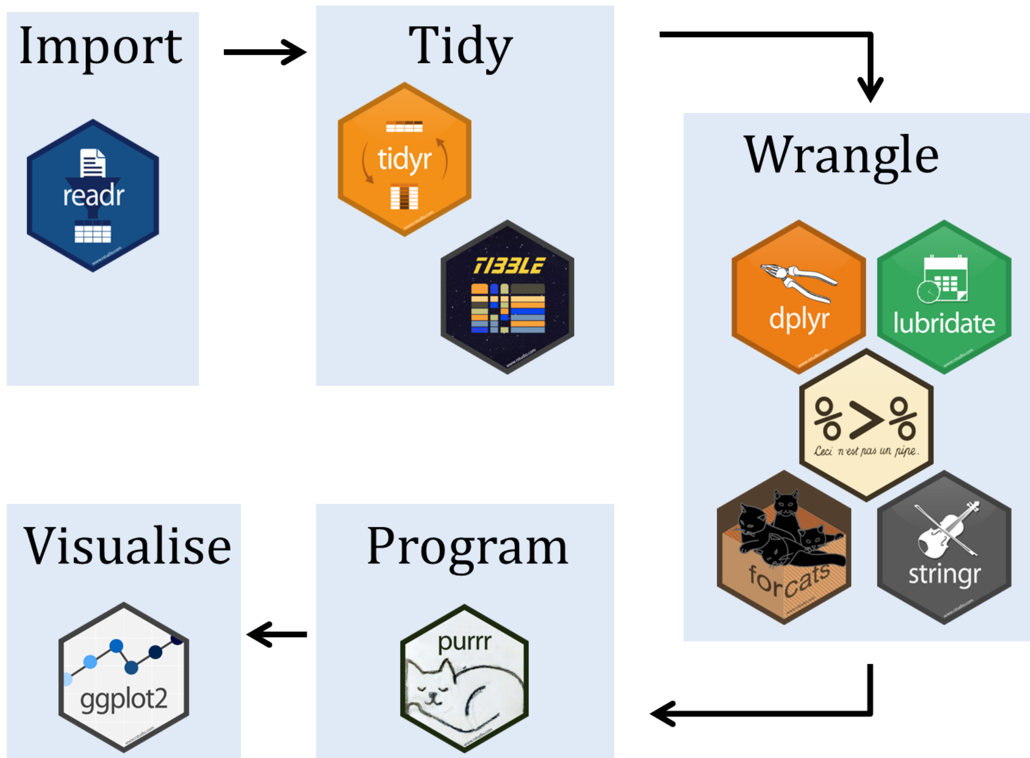
While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called tidyverse have been designed specifically for data science. All packages included in `tidyverse` share an underlying design philosophy, grammar, and data structures. This philosopy is rooted in the idea of "tidy data":



We will use `tidyverse` packages throughout the workshop, so let's install them now:

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

# when you install tidyverse for the first time you will be asked
# a question in the Console - please answer by typing "no" in the Console.

library(tidyverse)
```

A typical workflow for using `tidyverse` packages looks like this:



The ggplot2 package is contained within tidyverse, but we also want to install two additional packages, scales and ggrepel, which provide additional functionality.

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

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

We can also install the rmarkdown package, which will allow us to combine our text and code into a formatted document at the end of the workshop:

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

The following RStudio, tidyverse, ‘ggplot2, and rmarkdown cheatsheets will provide a useful reference: https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets_2019.pdf

4.1.4 Prerequisites

This is an intermediate R course:

- Assumes working knowledge of R

- Relatively fast-paced

4.1.5 Learning Outcomes

- R `ggplot2` graphics syntax
- Grammar of graphics
  - Aesthetics
  - Geometric objects
  - Scales
  - Faceting
- Combining all the above together in a single plot

4.1.6 Workshop Outline

1. Basic plots, **aesthetic mapping and inheritance**
2. Tailoring **statistical transformations** to particular plots
3. **Modifying scales** to change axes and add labels
4. **Faceting** to create many small plots
5. Changing plot **themes**

4.2 Why `ggplot2`?

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

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

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)

4.2.1 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

4.2.2 `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`)
- has sensible defaults for generating legends

4.3 Geometric objects & aesthetics

4.3.1 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.

4.3.2 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
`geom_point()`	Scatter plot	**`x`**,**`y`**,`alpha`,`color`,`fill`,`group`,`shape`,`size`,`stroke`

geom_	Usage	Aesthetics
`geom_line()`	Line plot	`x`,`y`,`alpha`,`color`,`linetype`,`size`
`geom_bar()`	Bar chart	`x`,`y`,`alpha`,`color`,`fill`,`group`,`linetype`,`size`
`geom_boxplot()`	Boxplot	`x`,**`lower`**,**`upper`**,**`middle`**,**`ymin`**,**`ymax`**,`alpha`,`color`,`fill`
`geom_density()`	Density plot	`x`,`y`,`alpha`,`color`,`fill`,`group`,`linetype`,`size`,`weight`
`geom_smooth()`	Conditional means	`x`,`y`,`alpha`,`color`,`fill`,`group`,`linetype`,`size`,`weight`
`geom_label()`	Text	`x`,`y`,**`label`**,`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_`.

4.3.2.1 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.

Example data: housing prices

Let's look at housing prices.

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

# create a subset for 1st quarter 2001
hp2001Q1 <- filter(housing, Date == 2001.25)
```

Step 1: create a blank canvas by specifying data:

```
ggplot(data = hp2001Q1)
```

Step 2: specify aesthetic mappings (how you want to map variables to visual aspects):

```
# here we map "Land_Value" and "Structure_Cost" to the x- and y-axes.
ggplot(data = hp2001Q1, mapping = aes(x = Land_Value, y = Structure_Cost))
```

Step 3: add new layers of geometric objects that will show up on the plot:

```
# here we use geom_point() to add a layer with point (dot) elements
# as the geometric shapes to represent the data.
ggplot(data = hp2001Q1, mapping = aes(x = Land_Value, y = Structure_Cost)) +
  geom_point()
```

4.3.2.2 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 — this is referred to as **aesthetic inheritance**. Our plot could use a regression line:

```
# get predicted values from a linear regression
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)) + # values for x and y are inherited from the ggplot() c
  geom_line(aes(y = pred_SC)) # add predicted values to the plot overriding the y values from the
```

4.3.2.3 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()
```

4.3.2.4 Text (label points)

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

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

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

4.3.3 Aesthetic mapping VS assignment

1. Variables are **mapped** to aesthetics within the `aes()` function

```
p1 +
  geom_point(aes(size = Home_Value))
```

2. Constants are **fixed** to aesthetics outside the `aes()` call

```
p1 +
  geom_point(size = 2)
```

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
```

4.3.4 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))
```

4.4 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`

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 in the previous plot blue.

```
##
```

3. Map the color of the the points to `Region`.

```
##
```

4. Keeping color mapped to `Region`, make the points bigger by setting size to 2

```
##
```

5. Keeping color mapped to `Region`, map the size of the points to `HDI_Rank`

```
##
```

4.5 Statistical transformations

4.5.1 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/`

4.5.2 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)
```

4.5.3 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()

head(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")
```

4.6 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 `?stat_smooth` and the `span` argument.

```
##
```

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

```
##
```

4.7 Scales

4.7.1 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 = y)` doesn't say *what* colors should be used. Also, `aes(size = z)` doesn't say *what* sizes 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.

4.7.2 Common scale arguments

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

- **name:** 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.

4.7.3 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")
```

4.7.4 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 = "blue",
                        high = "red",
                        mid = "gray60",
                        midpoint = 1994)
```

4.7.5 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/`

4.8 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`. NOTE: you can specify color by name (e.g., "blue") or by "Hex value" — see `https://www.color-hex.com/`.

```
##
```

4.9 Faceting

4.9.1 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

4.9.2 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.

4.9.3 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)
p5
```

4.10 Themes

4.10.1 What are themes?

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

- Axis label properties (e.g., font, size, color, etc.)

- 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/`

4.10.2 Overriding theme defaults

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

```
# theme(thing_to_modify = modifying_function(arg1, arg2))

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

All theme options are documented in `?theme`. We can also see the existing default values using:

```
theme_get()
```

4.10.3 Creating & 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, 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
```

4.11 Saving plots

We can save a plot to either a vector (e.g., pdf, eps, ps, svg) or raster (e.g., jpg, png, tiff, bmp, wmf) graphics file using the `ggsave()` function:

```
ggsave(filename = "myplot.pdf", plot = p5, device = "pdf", height = 6, width = 6, units = "in")
```

4.12 The #1 FAQ

4.12.1 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

Fixing, rather than mapping, the color aesthetic:

1. Produces verbose code when using many colors
2. Results in no legend being produced
3. Means you cannot change color scales

```
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

To avoid these pitfalls, we need to **map** our data to the color aesthetic. We can do this by **reshaping** our data from **wide format** to **long format**:

IMAGE HERE

```
home_land_byyear <- gather(housing_byyear,
                           value = "value",
                           key = "type",
                           Home_Value_Mean, Land_Value_Mean)

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

4.13 Exercise 3

For this exercise, we're going to use the built-in `midwest` dataset:

```
data("midwest", package = "ggplot2")
head(midwest)
```

1. Create a scatter plot with `area` on the x axis and the log of `poptotal` on the y axis.

```
##
```

2. Within the `geom_point()` call, map color to `state`, map size to the log of `popdensity`, and fix transparency (`alpha`) to 0.3.

```
##
```

3. Add a smoother and turn off plotting the confidence interval. Hint: see the `se` argument to `geom_smooth()`.

```
##
```

4. Facet the plot by `state`. Set the `scales` argument to `facet_wrap()` to allow separate ranges for the x-axis.

```
##
```

5. Change the default color scale to use the discrete `RColorBrewer` palette called `Set1`. Hint: see `?scale_color_brewer`.

```
##
```

6. BONUS: Change the default theme to `theme_bw()` and modify it so that the axis text and facet label background are blue. Hint: see `?theme` and especially `axis.text` and `strip.background`.

```
##
```

4.14 Exercise solutions

4.14.1 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. Map the color of the the points to `Region`.

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

4. Keeping color mapped to `Region`, make the points bigger by setting size to 2

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

5. Keeping color mapped to `Region`, map the size of the points to `HDI_Rank`

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

4.14.2 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 smoothing line on top of the scatter plot using the default *loess* method for `geom_smooth()`, but make it less smooth. Hint: see `?loess`.

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

5. BONUS: 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")
```

4.14.3 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`. NOTE: you can specify color by name (e.g., "blue") or by "Hex value" — see `https://www.color-hex.com/`.

```
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("red", "green", "blue", "orange", "grey", "brown"))
```

4.14.4 Ex 3: prototype

1. Create a scatter plot with `area` on the x axis and the log of `poptotal` on the y axis.

```
p6 <- ggplot(midwest, aes(x=area, y=log(poptotal)))
p6 + geom_point()
```

2. Within the `geom_point()` call, map color to `state`, map size to the log of `popdensity`, and fix transparency (`alpha`) to 0.3.

```
p6 <- p6 + geom_point(aes(color=state, size=log(popdensity)), alpha = 0.3)
```

3. Add a smoother and turn off plotting the confidence interval. Hint: see the `se` argument to `geom_smooth()`.

```
p6 <- p6 + geom_smooth(method="loess", se=FALSE)
```

4. Facet the plot by `state`. Set the `scales` argument to `facet_wrap()` to allow separate ranges for the x-axis.

```
p6 <- p6 + facet_wrap(~ state, scales = "free_x")
```

5. Change the default color scale to use the discrete `RColorBrewer` palette called `Set1`. Hint: see `?scale_color_brewer`.

```
p6 <- p6 + scale_color_brewer(palette = "Set1")
```

6. BONUS: Change the default theme to `theme_bw()` and modify it so that the axis text and facet label background are blue. Hint: see `?theme` and especially `axis.text` and `strip.background`.

```
p6 <- p6 + theme_bw() +
    theme(axis.title = element_text(color = "blue"),
          strip.background = element_rect(fill = "blue"))
```

4.15 Wrap-up

4.15.1 Feedback

These workshops are a work in progress, please provide any feedback to: `help@iq.harvard.edu`

4.15.2 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`
  - Examples: `http://r-statistics.co/Top50-Ggplot2-Visualizations-MasterList-R-Code.html`
  - Tutorial: `https://uc-r.github.io/ggplot_intro`
  - 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 5

R Data Wrangling

Topics

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

5.1 Setup

5.1.1 Class Structure

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

5.1.2 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/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.

5.1.3 Installing & using R packages

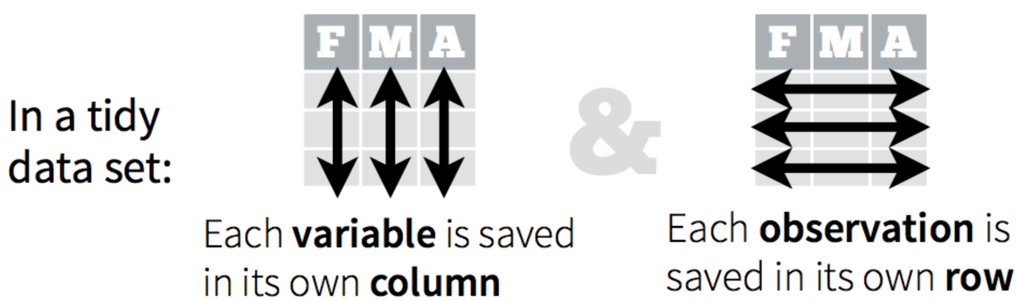
R is a modular environment that is extended by the use of **packages**. Packages are collections of functions or commands that are designed to perform specific tasks (e.g., fit a type of regression model). A large number of contributed packages are available (> 15,000).

Using an R package is a **two step process**:

1. Install the package onto your computer using the `install.packages()` function. This only needs to be done the **first time** you use the package.

2. Load the package into your R session's search path using the `library()` function. This needs to be done **each time** you use the package.

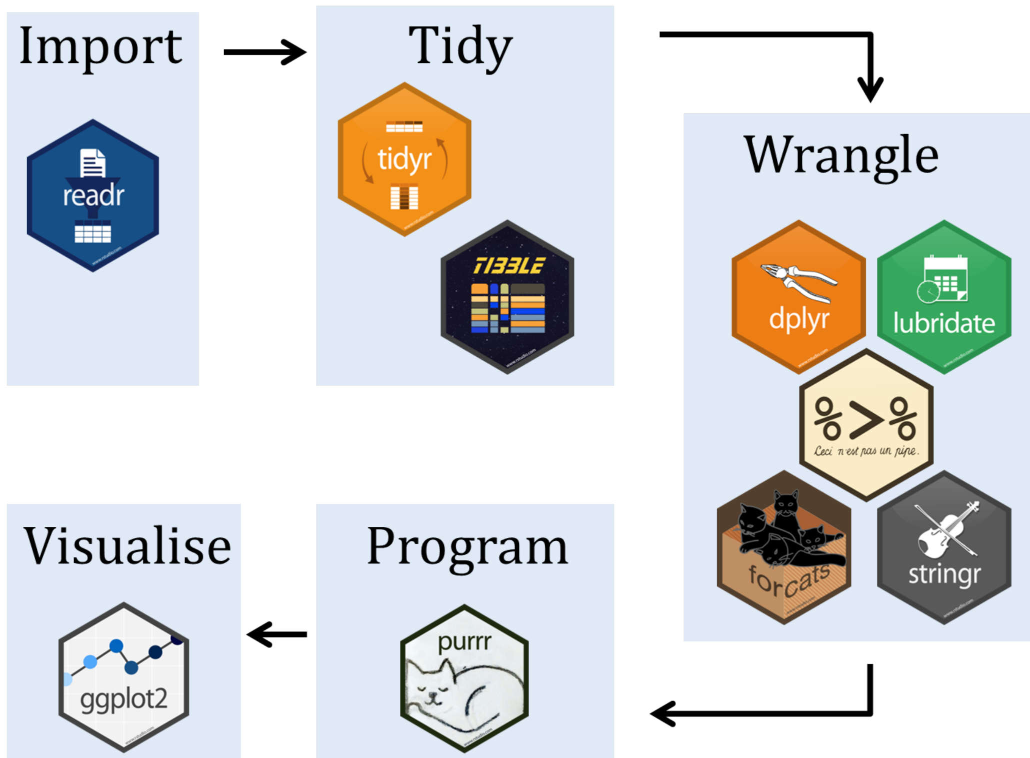
While R's built-in packages are powerful, in recent years there has been a big surge in well-designed *contributed packages* for R. In particular, a collection of R packages called tidyverse have been designed specifically for data science. All packages included in `tidyverse` share an underlying design philosophy, grammar, and data structures. This philosopy is rooted in the idea of "tidy data":



We will use `tidyverse` packages throughout the workshop, so let's install them now:

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

# when you install tidyverse for the first time you will be asked
# a question in the Console - please answer by typing "no" in the Console.

library(tidyverse)

library(readxl) # installed with tidyverse, but not loaded into R session
```

A typical workflow for using `tidyverse` packages looks like this:



We can also install the `rmarkdown` package, which will allow us to combine our text and code into a formatted document at the end of the workshop:

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

The following RStudio, `tidyverse`, and `rmarkdown` cheatsheets will provide a useful reference: `https://rstudio.com/wp-content/uploads/2019/01/Cheatsheets_2019.pdf`

5.1.4 Prerequisites

This is an intermediate / advanced R course:

- Assumes intermediate knowledge of R
- Relatively fast-paced

5.1.5 Learning Outcomes

- Understanding R data structures
- Iteration over data structures
- Writing functions
- Regular expressions (regex)
- Reshaping data

5.1.6 Workshop Outline

Example data

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

Overall Goal

Our mission is to extract and graph the **top 100** boys names in England and Wales for every year since 1996.

Exercise 0: Problems with the data

There are several things that make our goal challenging. Let's take a look at the data:

1. Locate the files named `1996boys_tcm77-254026.xlsx` and `2015boysnamesfinal.xlsx` and open them separately in a spreadsheet program.

   (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 difficult?

   In what ways is the format different between the two files?

Steps to accomplish this goal:

1. **Explore example data to highlight problems (see Exercise 0)**
2. **Reading data from multiple Excel worksheets into R data frames**
   - list Excel file names in a character vector
   - read Excel sheetnames into a list of character vectors
   - read Excel data for "Table 1" only into a list of data frames
3. **Clean up data within each R data frame**
   - sort and merge columns within each data frame inside the list
   - drop missing values from each data frame
   - reshape format from wide to long
4. **Organize the data into one large data frame and store it**
   - create a year column within each data frame in the list
   - append all the data frames in the list into one large data frame

NOTE: please make sure you close the Excel files before continuing with the workshop, otherwise you may encounter issues with file paths when reading the data into R.

5.2 Working with Excel worksheets

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

Name	Count	Year
JACK	10779	1996
DANIEL	10338	1996
THOMAS	9603	1996
JAMES	9385	1996
JOSHUA	7887	1996
MATTHEW	7426	1996
RYAN	6496	1996
JOSEPH	6193	1996
SAMUEL	6161	1996
LIAM	5802	1996
JORDAN	5750	1996
LUKE	5664	1996
CONNOR	5009	1996
ALEXANDER	4840	1996
BENJAMIN	4805	1996
ADAM	4538	1996
HARRY	4434	1996
JAKE	4331	1996
GEORGE	4287	1996
CALLUM	4281	1996
WILLIAM	4269	1996
MICHAEL	4187	1996
OLIVER	3655	1996
LEWIS	3569	1996
CHRISTOPHER	3483	1996

Figure 5.1: goal

cleaning and organizing it.

Each Excel file contains a worksheet with the boy 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 character 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 within `tidyverse` to list the worksheet names from this file.

```
excel_sheets(boy_file_names[1])
```

5.2.1 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 within `tidyverse` gives us a more consistent alternative, so we'll use that.

```
# map(object to iterate over, function that does task within each iteration)

map(boy_file_names, excel_sheets)
```

5.2.2 Filtering strings using regular expressions

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 within `tidyverse` because it is more user-friendly. `stringr` provides functions to:

1. detect
2. locate
3. extract
4. match
5. replace
6. combine
7. split

strings. 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:

1. The first argument to `str_subset()` is character vector we want to search in.
2. The second argument is a *regular expression* matching the pattern we want to retain.

If you are not familiar with regular expressions (regex), `http://www.regexr.com/` is a good place to start. Regex is essentially just a programmatic way of doing operations like "find" or "find and replace" in MS Word or Excel.

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,

```
# str_subset(character_vector, regex_pattern)

# nesting functions
str_subset(excel_sheets(boy_file_names[1]), pattern = "Table 1")

# piping functions
excel_sheets(boy_file_names[1]) %>% str_subset(pattern = "Table 1")
```

5.2.3 Writing your own functions

The next step is to retrieve worksheet names and subset them.

The `map*` functions are useful when you want to apply a function to a list or vector of inputs and obtain the return values for each input. 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 character vector containing file names.

However, there is no function that both:

1. Retrieves worksheet names, and
2. Subsets the names

So, we will have to write one. Fortunately, writing functions in R is easy. Functions require 3 elements:

1. A **name**
2. One or more **arguments**
3. A **body** containing computations

```
# Anatomy of a function
```

```
# function_name <- function(arg1, arg2, ....) {
#
#   # body of function - where stuff happens #
#
#   return( results )
# }

myfun <- function(x) {
  x^2
}

myfun(1:10)

myfun2 <- function(x, y) {
  z <- x^2 + y
  return(z)
}

myfun(x=1:10, y=42)
```

```
get_data_sheet_name <- function(file, term){
  excel_sheets(file) %>% str_subset(pattern = term)
}

# the goal is generalization
get_data_sheet_name(boy_file_names[1], term = "Table 1")
get_data_sheet_name(boy_file_names[1], term = "Table 2")
```

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

```
# map(object to iterate over,
#     function that does task within each iteration,
#     arguments to previous function)

map(boy_file_names,      # list object
    get_data_sheet_name, # function
    term = "Table 1")    # argument to previous function
```

5.3 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. We can use the `glimpse()` function from the `dplyr` package within `tidyverse` to view the output.

```r
tmp <- read_excel(
  path = boy_file_names[1],
  sheet = get_data_sheet_name(boy_file_names[1], term = "Table 1"),
  skip = 6
)

glimpse(tmp)
```

Note that R has added a suffix to each column name `...1`, `...2`, `...3`, etc. because duplicate names are not allowed, so the suffix serves to disambiguate. The trailing number represents the index of the column.

5.4 Exercise 1

1. Write a function called `read_boys_names` 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.

```r
##
```

2. Test your function by using it to read *one* of the boys names Excel files.

```r
##
```

3. Use the `map()` function to create a list of data frames called `boysNames` from all the Excel files, using the function you wrote in step 1.

```r
##
```

5.5 Data cleanup

Now that we've read in the data, we can see that there are some problems we need to fix. 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.

```r
# Rank 1:50 --- Names / Counts are in columns 2 and 3
# Rank 51:100 --- Names / Counts are in columns 6 and 7
glimpse(boysNames[[1]])

# Rank 1:50 --- Names / Counts are in columns 2 and 3
# Rank 51:100 --- Names / Counts are in columns 7 and 8
glimpse(boysNames[[10]])

# Rank 1:50 --- Names / Counts are in columns 2 and 3
```

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*	+105*
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	ELIJAH	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 5.2: messy

```
# Rank 51:100 --- Names / Counts are in columns 8 and 9
glimpse(boysNames[[20]])
```

In short, we want to go from this:

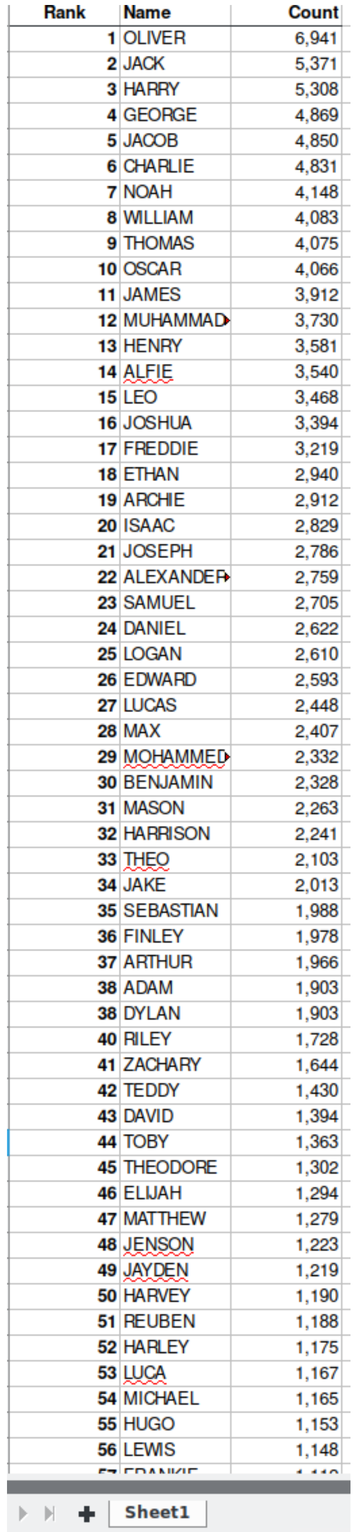
to this:

There are many ways to do this kind of data manipulation in R. We're going to use the `dplyr` and `tidyr` packages from within `tidyverse` to make our lives easier.

5.5.1 Selecting columns

Next we want to retain just the `Name...2`, `Name...6`, `Count...3` and `Count...7` columns. We can do that using the `select()` function:

```
boysNames[[1]]

boysNames[[1]] <- select(boysNames[[1]], Name...2, Name...6, Count...3, Count...7)
boysNames[[1]]
```



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

Figure 5.3: tidy

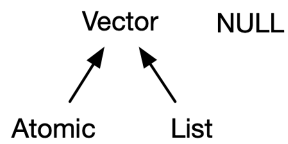


Figure 5.4: vector tree

5.5.2 R data types and structures

We've now encountered several different data types and data structures. Let's take a step back and survey the options available in R.

Data structures:

In R, the most foundational data structure is the **vector**, which comes in two basic forms:

1. **atomic**: only hold elements of the same type — **homogeneous**
2. **lists**: can hold elements of different types — **heterogeneous**

`NULL` is closely related to vectors and often serves the role of a generic zero length vector.

From these two forms, the following six structures are derived:

Type	Elements	Description
atomic vector	homogeneous	contains elements of the same **type**, one of: character, integer, double, logical, or complex
array	homogeneous	an atomic vector with attributes giving dimensions (1, 2, or >2)
matrix	homogeneous	an array with 2 dimesions
factor	homogeneous	an atomic integer vector containing only predefined values, storing categorical data
list	heterogeneous	a container whose elements are not restricted to a single mode and can encompass any mixture of data types
data.frame	heterogeneous	a rectangular list with elements (columns) containing atomic vectors of equal length

Each vector can have **attributes**, which is a named list of arbitrary metadata that can include the vector's **dimensions** and its **class**. The latter is a property assigned to an object that determines how generic functions operate with it, and thus which **methods** are available for it. The class of an object can be queried using the `class()` function.

You can learn more details about R data structures here: `https://adv-r.hadley.nz/vectors-chap.html`

Data types:

There are four primary types of atomic vectors. Collectively, integer and double vectors are known as numeric vectors. You can query the **type** of an object using the `typeof()` function.

Type	Description
character	"a", "swc"
integer	2L (the L tells R to store this as an integer)
double (floating point)	2, 15.5
logical	TRUE, FALSE

Coercion:

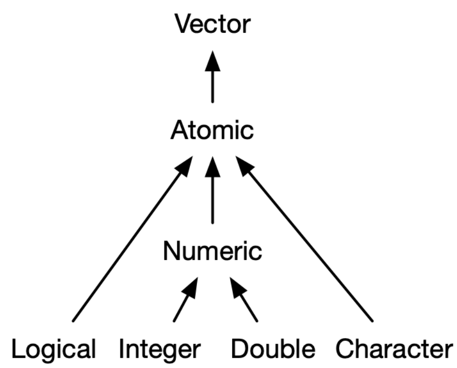


Figure 5.5: vector tree

If heterogeneous elements are stored in an atomic vector, R will **coerce** the vector to the simplest type required to store all the information. The order of coercion is roughly: logical -> integer -> numeric -> complex -> character -> list. For example:

```
x <- c(1, 2, 3)
typeof(x)

x <- c(1, 2, 3, "a")
typeof(x)
```

5.5.3 List indexing

Now that we know about data structures more generally, let's focus on the list structure we created for `boysNames`. Why are we using **double brackets** `[[` to index this list object, instead of the single brackets `[` we used to index atomic vectors?

```
# various data structures
numbers <- 1:10
letters <- LETTERS[1:4]
dat <- head(mtcars)
x <- 237

# combine in a list
mylist <- list(numbers, letters, dat, x)

# indexing the list
mylist[2]
class(mylist[2]) # a list

mylist[[2]]
class(mylist[[2]]) # a character vector
```

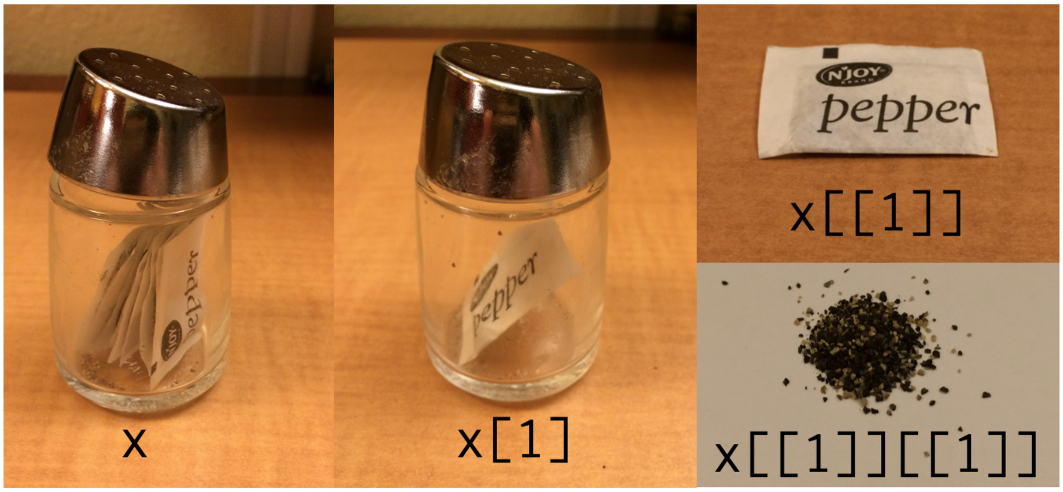


Figure 5.6: list indexing

5.5.4 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()` from the `tidyr` package within `tidyverse` to remove rows with missing values.

```
boysNames[[1]]

boysNames[[1]] <- boysNames[[1]] %>% drop_na()

boysNames[[1]]
```

5.6 Exercise 2

1. Write a function called `namecount` that takes a data frame as an argument and returns a modified version, which keeps only columns that include the strings `Name` and `Count` in the column names. HINT: see the `?matches` function.

```
##
```

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

```
##
```

3. Use the `map()` function to each `data.frame` in the list of boys names data and save it to the list called `boysNames`.

```
##
```

5.6.1 Re-arranging into a single table

Our final task is to re-arrange the 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]]

first_columns <- select(boysNames[[1]], Name = Name...2, Count = Count...3)
second_columns <- select(boysNames[[1]], Name = Name...6, Count = Count...7)

bind_rows(first_columns, second_columns)
```

5.7 Exercise 3

Cleanup all the data

In the previous examples we learned how to drop empty rows with `drop_na()`, 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.

1. Create a new function called `cleanupNamesData` that:

```
# 1) subsets data to include only those columns that include the term `Name` and `Count` and app

# 2) subset two separate data frames, with first and second set of `Name` and `Count` columns

# 3) append the two datasets
```

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

NOTE: some Excel files include extra blank columns between the first and second set of `Name` and `Count` columns, resulting in different numeric suffixes for the second set of columns. You will need to use a regular expression to match each of these different column names. HINT: see the `?matches` function.

```
##
```

5.8 Data organization & storage

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

5.8.1 One `data.frame` for each year

Right now we have a list of data frames, 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 to.

```
head(boysNames) %>% glimpse()

head(boy_file_names)

# use regex to extract years from filenames
Years <- str_extract(boy_file_names, pattern = "[0-9]{4}")
Years

names(boysNames) # returns NULL - no names in the list

# assign years to list names
names(boysNames) <- Years

names(boysNames) # returns the years as list names

head(boysNames) %>% glimpse()
```

5.8.2 One big `data.frame`

While storing the data in separate data.frames by year makes some sense, many operations will be easier if the data is simply stored in one big data.frame. 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 data.frames into one will result in losing the year information! Fortunately it is not too much trouble to add the year information to each data.frame before flattening.

```
# apply name of the list element (.y) as a new column in the data.frame (.x)
boysNames <- imap(boysNames, ~ mutate(.x, Year = as.integer(.y)))

boysNames[1]
```

5.9 Exercise 4

Make one big data.frame

1. Turn the list of boys names `data.frames` into a single `data.frame`. HINT: see `?bind_rows`.

```
##
```

2. Create a new directory called `all` within `dataSets` and write the data to a `.csv` file. HINT: see the `?dir.create` and `?write_csv` functions.

```
##
```

3. What were the five most popular names in 2013?

```
##
```

4. How has the popularity of the name "ANDREW" changed over time?

```
##
```

5.10 Exercise solutions

5.10.1 Ex 0: prototype

```
Locate the files named `1996boys_tcm77-254026.xlsx` and
`2015boysnamesfinal.xlsx` and open them separately in a
 spreadsheet program.

(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 difficult?

In what ways is the format different between the two files?
```

1. Multiple Excel sheets in each file, each with a different name, but each file contains a `Table 1`.
2. The data does not start on row one. Headers are on row 7, followed by a blank line, followed by the actual data.
3. The data is stored in an inconvenient way, with ranks 1-50 in the first set of columns and ranks 51-100 in a second set of columns.
4. The second worksheet `2015boysnamesfinal.xlsx` contains extra columns between the data of interest, resulting in the second set of columns (ranks 51-100) being placed in a different position.
5. The year from which the data comes is only reported in the Excel file name, not within the data itself.
6. There are notes below the data.

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.

5.10.2 Ex 1: prototype

```r
  ## 1. Write a function that takes a file name as an argument and reads
  ##    the worksheet containing "Table 1" from that file.

read_boys_names <- function(file, sheet_name) {
  read_excel(
    path = file,
    sheet = get_data_sheet_name(file, term = sheet_name),
    skip = 6
  )
}

  ## 2. Test your function by using it to read *one* of the boys names
  ##    Excel files.

read_boys_names(boy_file_names[1], sheet_name = "Table 1") %>% glimpse()


  ## 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_boys_names, sheet_name = "Table 1")
```

5.10.3 Ex 2: prototype

```r
  ## 1. Write a function that takes a `data.frame` as an argument and
  ##   returns a modified version, which keeps only columns that
  ##   include the strings `Name` and `Count` in the column names.
  ##   HINT: see the `?matches` function.

  namecount <- function(data) {
      select(data, matches("Name|Count"))
  }

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

  namecount(boysNames[[1]])

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

  boysNames <- map(boysNames, namecount)
```

5.10.4 Ex 3: prototype

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

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

## NOTE: some Excel files include extra blank columns between the first and second
## set of `Name` and `Count` columns, resulting in different numeric suffixes
## for the second set of columns. You will need to use a regular expression
## to match each of these different column names. HINT: see the `?matches` function.

cleanupNamesData <- function(file){

  # subset data to include only those columns that include the term `Name` and `Count`
  subsetted_file <- file %>%
    select(matches("Name|Count")) %>%
    drop_na()

  # subset two separate data frames, with first and second set of `Name` and `Count` columns
  first_columns <- select(subsetted_file, Name = Name...2, Count = Count...3)

  second_columns <- select(subsetted_file, Name = matches("Name...6|Name...7|Name...8"),
                                           Count = matches("Count...7|Count...8|Count...9"))

  # append the two datasets
  bind_rows(first_columns, second_columns)
}


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

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

5.10.5 Ex 4: prototype

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

```
## 1.  Turn the list of boys names `data.frames` into a single `data.frame`.

boysNames <- bind_rows(boysNames)
glimpse(boysNames)


## 2.  Create a new directory called `all` within `dataSets` and write the data to a `.csv` file
```

```r
##     HINT: see the `?dir.create` and `?write_csv` functions.

dir.create("dataSets/all")

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


## 3.  What were the five most popular names in 2013?

boysNames %>%
  filter(Year == 2013) %>%
  arrange(desc(Count)) %>%
  head()


## 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")
```

5.11 Wrap-up

5.11.1 Feedback

These workshops are a work in progress, please provide any feedback to: `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`
- 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.jstatso`|.org/v31/i01/paper

Part III

Python

Chapter 6

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

6.1 Setup

6.1.1 Class Structure

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

6.1.2 Software & Materials

6.1.2.1 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.

6.1.2.2 Download materials

- Download class materials at `https://github.com/IQSS/dss-workshops/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!

6.1.3 Prerequisites

This is an introductory Python course:

- Assumes no knowledge of Python
- Relatively slow-paced

6.1.4 Learning Outcomes

- Python language basics and common idioms
- Reading files and manipulating data in Python
- Iterating over data structures
- Python package and application ecosystem

6.1.5 Workshop Outline

As an example project we will analyze the text of Lewis Carroll's *Alice's Adventures in Wonderland*. We will use Python to read data from this text file and then, working with nested data structures, ask the following questions:

1. How many total and unique words are there?
2. How many chapters and paragraphs?
3. How many words are in each chapter, and what is the average words per chapter?
4. How many times is each main character mentioned?

6.2 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.

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.

6.3 Python basics

6.3.1 Launch Jupyter Notebook

1. Start the `Anaconda Navigator` program
2. Click the `Launch` button under `Jupyter Notebook`
3. Click `upload`, then in the pop-up window select the `PythonIntro` folder on the desktop
4. Open and upload 3 files (note, this is TWO steps): `PythonIntro.ipnb`, `Alice_in_wonderland.txt`, and `Characters.txt`

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.

6.3.2 Reading data 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*.

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

6.3.3 Python functions

In Python functions perform tasks, and take the form:

```
# function_name(arg1, arg2, arg3, ... argn)
```

where `arg1` etc. are arguments to the function.

The `open()` function

We can use the `open()` function to create a file **object** that makes a **connection** to the file. This means that the `alice_file` object name 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.

6.3.4 Assignment

In Python we can assign a result to an name using the `=` operator.

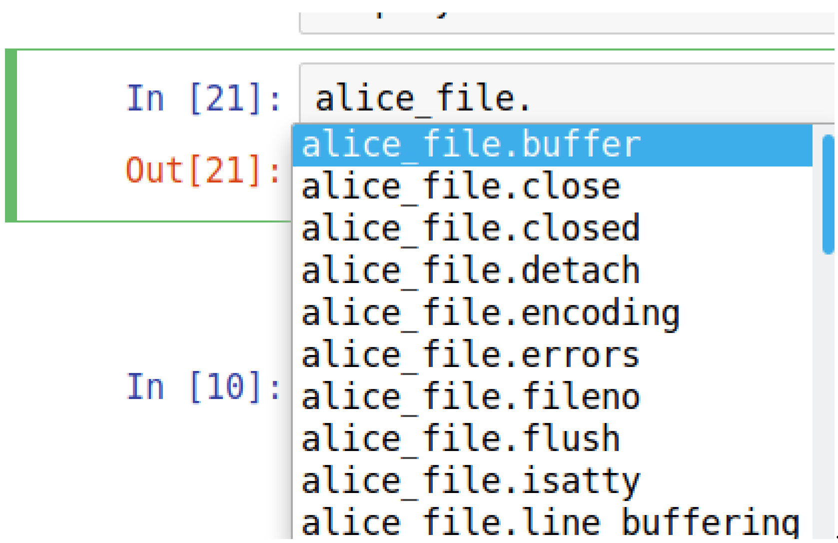
```
# objectName = thing_to_assign
x = 10
```

The name on the left of the equals sign (`alice_file`) is one that we chose. When choosing names, they must:

1. start with a *letter*
2. use only *letters*, *numbers* and *underscores*

6.3.5 Object methods

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 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.

6.4 Counting chapters, lines, & words

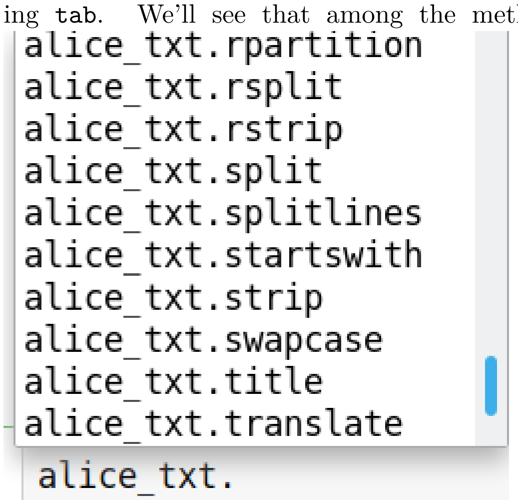
Now that we have the text, we can start answering some questions about it. To begin with, how many words does it contain? To answer this question, we can split the text up so there is one element per word, and then count the number of words.

6.4.1 Splitting a string into a list of words

How do we figure out how to split strings in Python? We can ask 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.



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() # returns a list (we talk about lists below)
type(alice_words)
len(alice_words) # counts elements in a data structure
```

6.4.2 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))

mySet = {1, 5, 9}
len(mySet)
```

There are 5295 unique words in the text.

6.5 Exercise 0

Reading text from a file & 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`.

6.5.1 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 data structure you're working with in Python. As you gain experience, you won't have to look these 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`.

6.5.2 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.

```
# syntax
# object[ start : end : by ]

# default
# object[ 0 : end : 1 ]
```

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:

```
# create a list
y = [1, "b", 3, "D", 5, 6]

y[0] # returns first element - the number 1 (yes, we count from zero!)
y[1] # returns second element - the letter "b"
y[ :3] # returns a list with only the first 3 elements, but index is of length 4 (0 to 3) becaus
y[2:5] # returns a list with elements 3, "D", 5
y[-1] # returns last element - the number 6
y[-4: ] # returns a list with last 4 elements
```

```
alice_words[11:20] # returns a list with words 11 through 20
alice_words[-10: ] # returns a list with the last 10 words
```

6.5.3 Counting chapters & 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`. Paragraphs are indicated by two new lines, and so our basic paragraph separator is `\n\n`. We can see this separator by looking at the content.

```
print(alice_txt[:500]) # explicit printing --- formats text nicely

alice_txt[:500] # returns content without printing it
```

```
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==========")
```

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

```
len(alice_paragraphs)
```

6.6 Exercise 1

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.

6.7 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*.

6.7.1 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.

The for loop syntax is:

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

Notice that the body of the for-loop is indented. This is important, because it is this indentation that defines the *body* of the loop — the place where things are done. White space matters in Python! A simple example:

```
for i in range(10):
    print(i)
print('DONE.')
```

Notice that "DONE." is only printed once, since `print('DONE.')` is not indented and is therefore outside of the body of the loop. An example using the Alice text:

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

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))
```

6.7.2 Organizing results in dictionaries

Our code for calculating the number 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.

The dictionary structure looks like:

```
mydict = {key1:value1, key2:value2, key3:value3}
```

A simple example:

```
mydict = {"apple":5, "pear":6, "grape":10}
print(mydict)

# compare to a list
mylist =[5, 6, 10]
print(mylist)
```

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

```
container = [] # a list

for i in range(10):
    container.append(i) # append elements to the list

print(container)
```

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

print(chapter_names)
```

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

```
# create counts for each chapter
chapter_Alice = []
for chapter in alice_chapters[1:]:
    chapter_Alice.append(chapter.count("Alice"))

# combine names and counts
mydict = dict(zip(chapter_names,
                  chapter_Alice))

print(mydict)

help(zip)
```

6.8 Exercise 2

Iterating & counting things

Now that we know how to iterate using for-loops, 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 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. 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.

6.9 Importing numpy & 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.

To use `numpy` or other packages, you must first import them.

```
import <package-name>
```

We can import `numpy` as follows:

```
import numpy
```

To use functions from a package, we can prefix the function with the package name, separated by a period:

```
<package-name>.<function_name>()
```

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))
```

6.10 Exercise solutions

6.10.1 Ex 0: prototype

```
# 1. Open the Characters.txt file and read its contents.

characters_file = open("Characters.txt")
characters_txt = characters_file.read()

# 2. Split text on newlines to produce a list with one element per line.
# Store the result as "alice_characters".

alice_characters = characters_txt.splitlines()
alice_characters
```

6.10.2 Ex 1: prototype

```
# 1. Count the number of main characters in the story (i.e., get the length of the list you crea

len(alice_characters)

# 2. Extract and print just the first character from the list you created in the previous exerci

print(alice_characters[0])

# 3. (BONUS, optional): Sort the list you created in step 2 alphabetically,
# and then extract the last element.

alice_characters.sort()
alice_characters[-1]
```

6.10.3 Ex 2: prototype

```
# 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

characters_txt = open("Characters.txt").read()
alice_txt = open("Alice_in_wonderland.txt", encoding="utf-8-sig").read()

# 2. Which chapter has the most words?
# Split the text into chaptes (i.e., split on "CHAPTER ") and use a for-loop to iterate over the
# For each chapter, split it into words and calculate the length.

words_per_chapter = []
```

```python
for chapter in alice_chapters:
    words_per_chapter.append(len(chapter.split()))
words_per_chapter

# 3. How many times is each character mentioned in the text?
# Iterate over the list of characters using a for-loop.
# For each character, call the count method with that character as the argument.

num_per_character = []
for character in characters_txt.splitlines():
    num_per_character.append(alice_txt.count(character))
num_per_character

# 4. (BONUS, optional): Put the character counts computed above in a
# dictionary with character names as the keys and counts as the values.

characters = characters_txt.splitlines()
dict(zip(characters, num_per_character))
```

6.11 Wrap-up

6.11.1 Feedback

These workshops are a work in progress, please provide any feedback to: `help@iq.harvard.edu`

6.11.2 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 7

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`

7.1 Setup

7.1.1 Class Structure

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

7.1.2 Software & Materials

7.1.2.1 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.

7.1.2.2 Download materials

- Download class materials at `https://github.com/IQSS/dss-workshops/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!

7.1.3 Prerequisites

This is an intermediate Python course:

- Assumes knowledge of Python, including:
  - lists
  - dictionaries
  - logical indexing
  - iteration with for-loops
- Assumes basic knowledge of web page structure
- Relatively fast-paced

If you need an introduction to Python or a refresher, we recommend our Python Introduction.

7.1.4 Learning Outcomes

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

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.

7.1.5 Workshop Outline

This workshop is organized into two parts:

1. Retrive information in JSON format
2. Parse HTML file

7.2 Preliminary questions

7.2.1 What is web scraping?

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

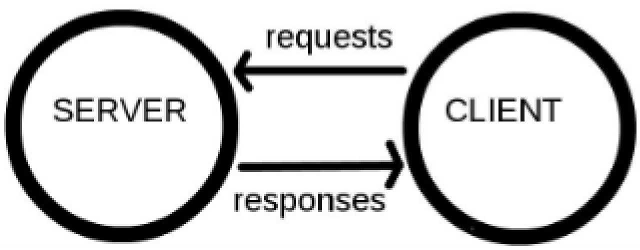
7.2.2 Is web scraping legal? Is it ethical?

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.

7.3 How Does the Web Work?

7.3.1 Components

Computers connected to the web are called clients and servers. A simplified diagram of how they interact might look like this:



- Clients are the typical web user's internet-connected devices (for example, your computer connected to your Wi-Fi) and web-accessing software available on those devices (usually a web brwoser like Firefox or Chrome).
- Servers are computers that store webpages, sites, or apps. When a client device wants to access a webpage, a copy of the webpage is downloaded from the server onto the client machine to be displayed in the user's web browser.
- Your internet connection allows you to send and receive data on the web.
- TCP/IP are the communication protocols that define how data should travel across the web.
- Domain Name Servers are like an address book for websites.
- HTTP is an application protocol that defines a language for clients and servers to speak to each other.

7.3.2 So What Happens, Exactly?

When you type a web address into your browser:

1. The browser goes to the DNS server, and finds the real address of the server that the website lives on.
2. The browser sends an HTTP request message to the server, asking it to send a copy of the website to the client. This message, and all other data sent between the client and the server, is sent across your internet connection using TCP/IP.

3. If the server approves the client's request, the server sends the client a "200 OK" message, and then starts sending the website's files to the browser as a series of small chunks called data packets.
4. The browser assembles the small chunks into a complete website and displays it to you.

7.4 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.

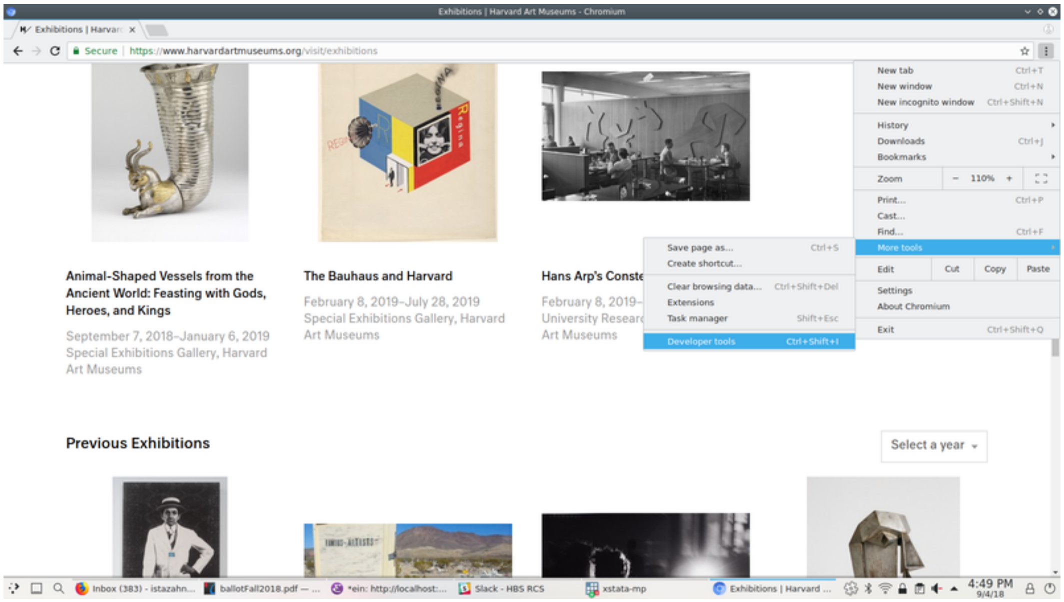
The basic strategy is pretty much the same for most scraping projects. We will use our web browser (Chrome or Firefox recommended) to examine the page you wish to retrieve data from, and copy/paste information from your web browser into your scraping program.

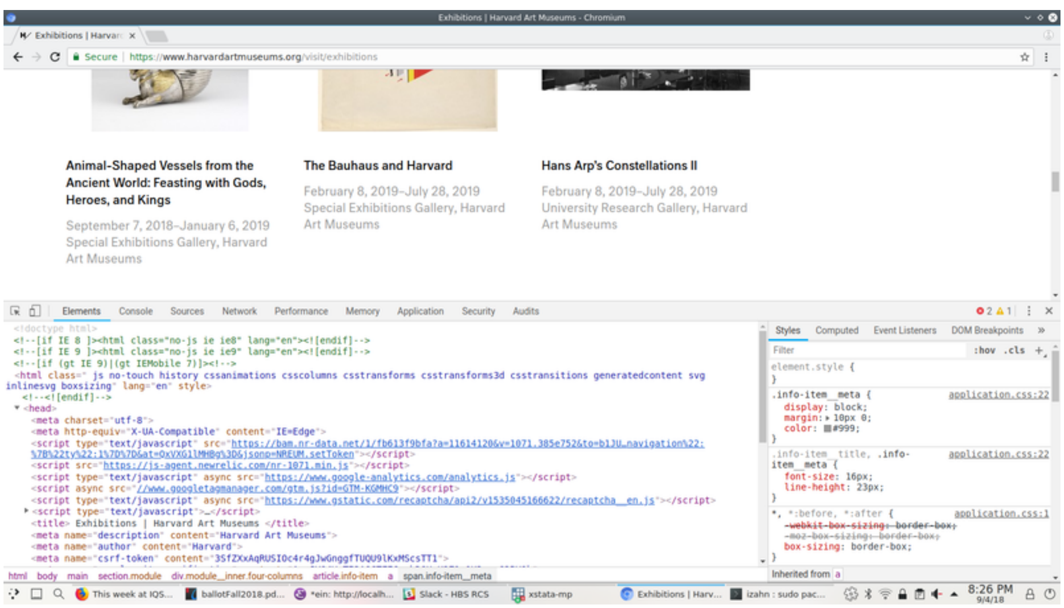
7.5 Take shortcuts if you can

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.

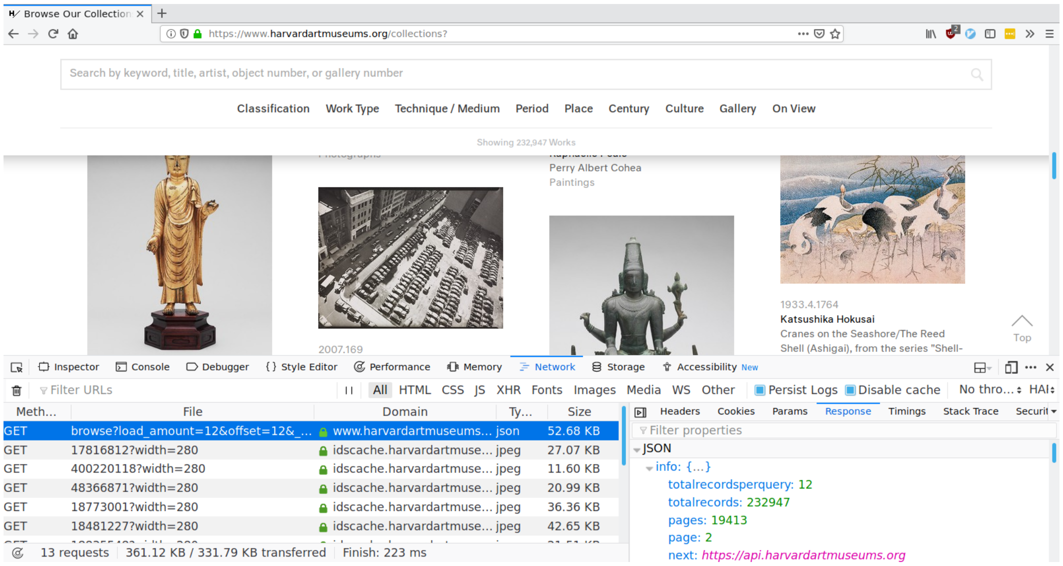
7.5.1 Examining the structure of our target web service

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.

7.5.2 Launch Jupyter Notebook

1. Start the `Anaconda Navigator` program
2. Click the `Launch` button under `Jupyter Notebook`
3. Click `upload`, then in the pop-up window select the `PythonWebScrape` folder on the desktop
4. Open and upload the `PythonWebScrape.ipynb` file (note, this is TWO steps)

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.

7.5.3 Making requests

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:

```python
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.

```python
import requests

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

```python
# ### 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!

```python
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 lean. But you have to admit that was pretty easy. If you can identify a service that returns the data you want in structured from, 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.

7.5.4 Organizing & 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.

```python
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")
```

7.5.5 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)
```

7.6 Exercise 0

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`

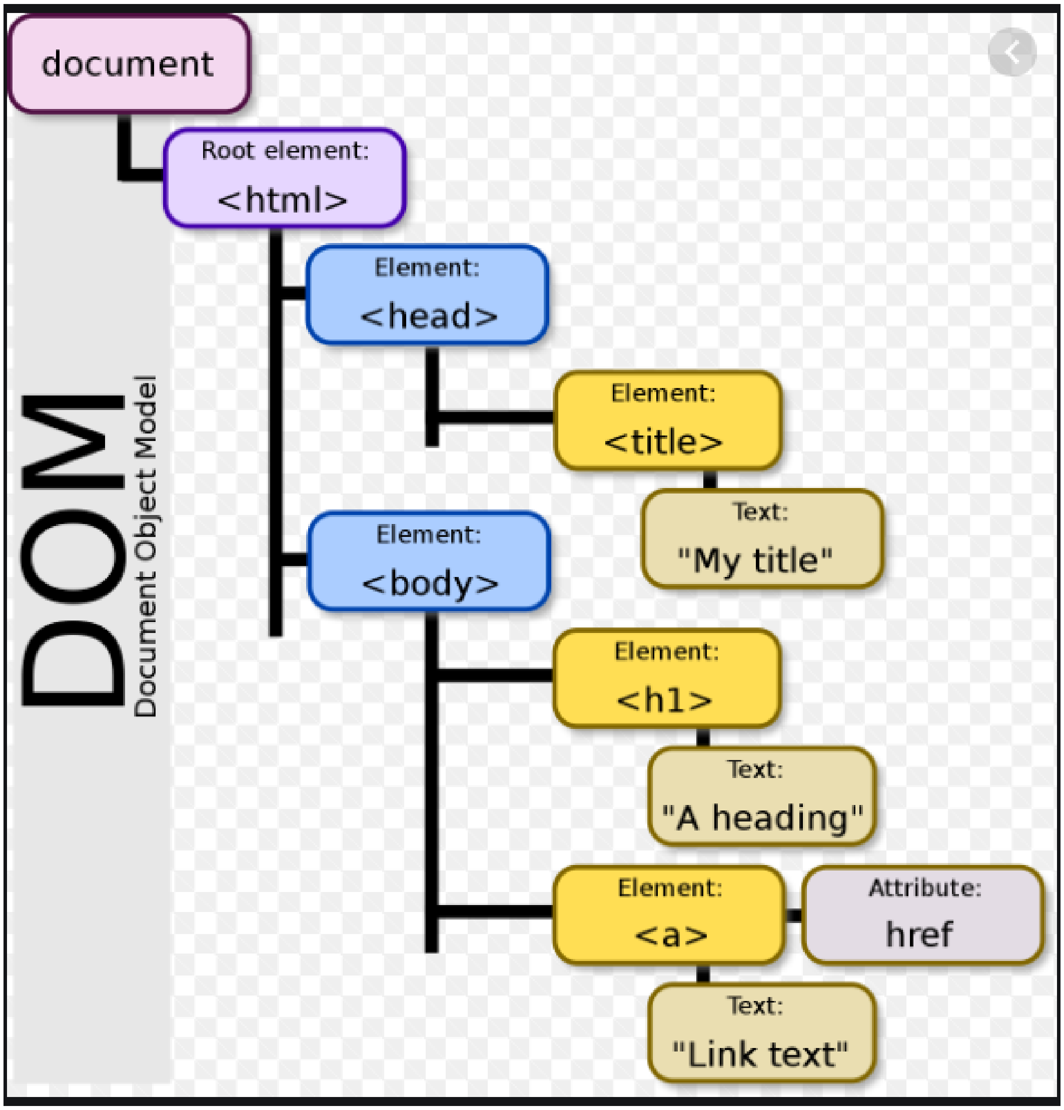
1. Using a web browser (Firefox or Chrome recommended) inspect the page at `https://www.harvardartmuseums.org/visit/exhibitions`. Examine the network traffic as you interact with the page. Try to find where the data displayed on that page comes from.
2. Make a `get` request in Python to retrieve the 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.

7.7 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.

7.7.1 Document Object Model (DOM)

To parse HTML, we need to have a nice tree structure that contains the whole HTML file through which we can locate the information. This tree-like structure is the Document Object Model (DOM). DOM is a cross-platform and language-independent interface that treats an XML or HTML document as a tree structure wherein each node is an object representing a part of the document. The DOM represents a document with a logical tree. Each branch of the tree ends in a node, and each node contains objects. DOM methods allow programmatic access to the tree; with them one can change the structure, style or content of a document. The following is an example of DOM hierarchy in an HTML document:



7.7.2 Retrieving HTML

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.

The first step is the same as before: we make at `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']
```

7.7.3 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 `BeautyfulSoup`.

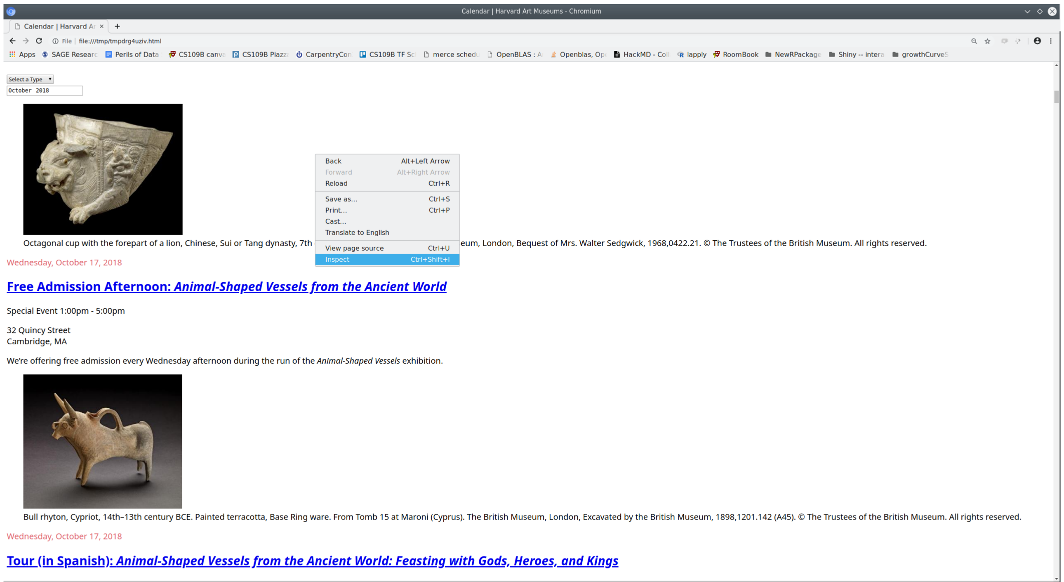
```
from lxml import html

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

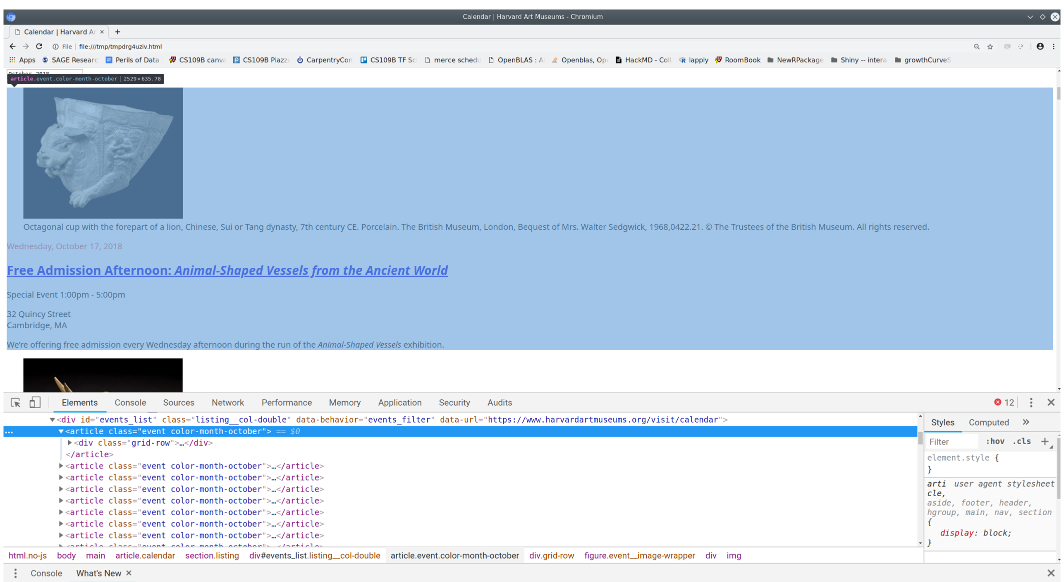
7.7.4 Using xpath to extract content from HTML

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

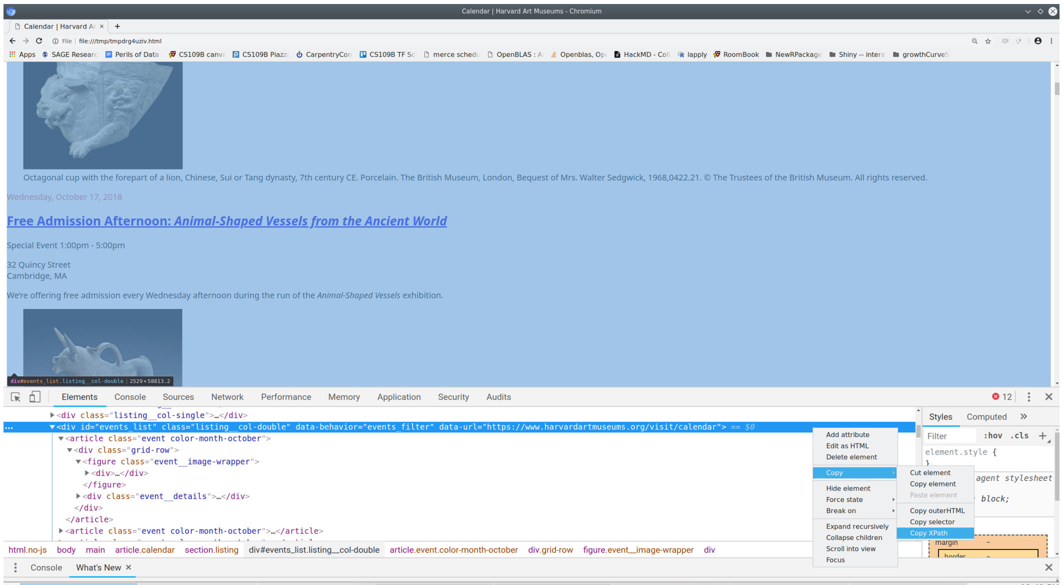
We can open the html document we retrieved and inspect it using our web browser.

```
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.



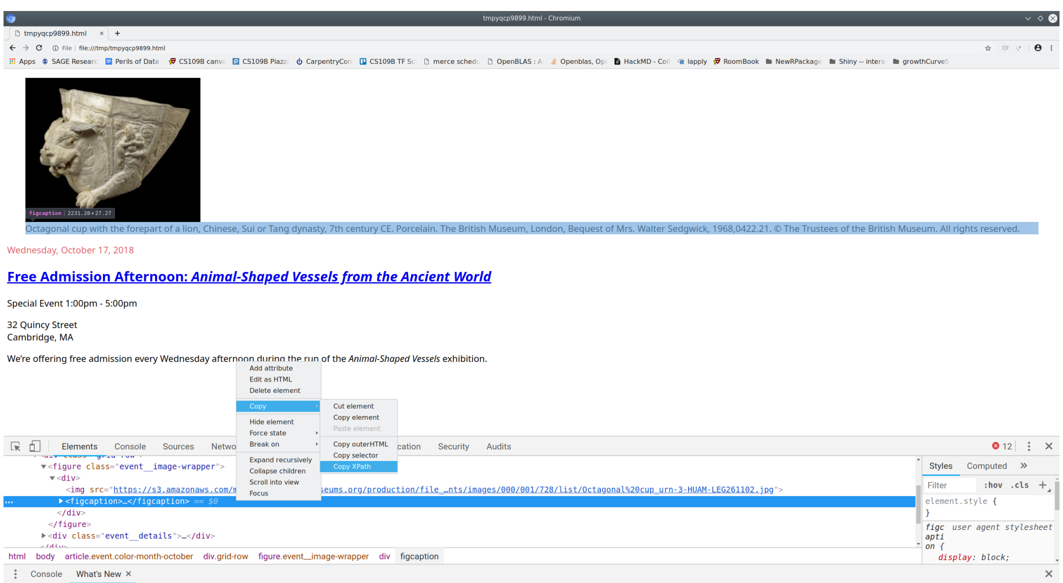
Next we can use python to extract the element of interest:

```
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.

```
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',
                    'localtion1': '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()

print(first_event_values)
```

7.7.5 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)
```

7.8 Exercise 1

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 if 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 `XPath`s to the facilities housed on level one. Use Python to extract the text from those `Xpath`s.
4. Bonus (optional): Write a *loop* or *list comprehension* in Python to retrieve data for all the levels.

7.9 `Scrapy`: for large / complex 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`.

7.10 Browser drivers: 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`.

7.11 Exercise solutions

7.11.1 Ex 0: prototype

7.11.2 Ex 1: prototype

7.12 Wrap-up

7.12.1 Feedback

These workshops are a work in progress, please provide any feedback to: `help@iq.harvard.edu`

7.12.2 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`

Part IV

Stata

Chapter 8

Stata Introduction

Topics

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

8.1 Setup

8.1.1 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/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!

8.1.2 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

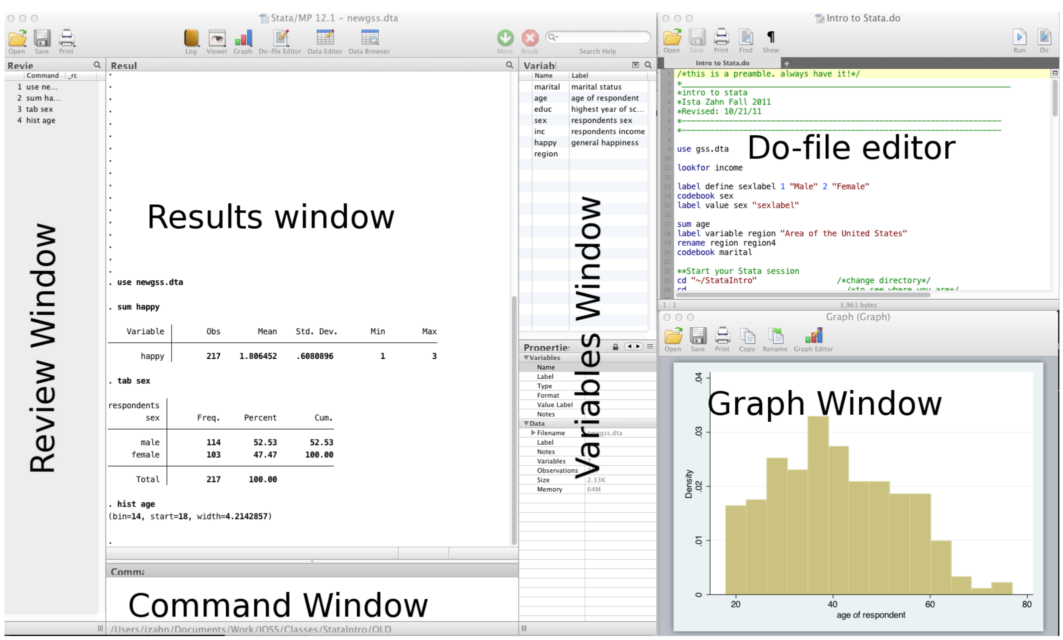
8.1.3 Goals

- 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

8.2 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

8.2.1 Stata interface



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

8.2.2 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

8.2.3 Stata help

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

8.2.4 General Stata command syntax

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

8.2.5 Commenting & 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!"
```

8.2.6 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"
```

8.3 Getting data into Stata

8.3.1 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
```

8.3.2 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

8.3.3 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

8.3.4 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
```

8.3.5 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

8.4 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`

8.5 Statistics & graphs

8.5.1 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 actuall 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

8.5.2 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
```

8.5.3 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 eudcation by marital status
```

8.6 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

8.7 Basic data management

8.7.1 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

8.7.2 Variable & 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
```

8.8 Exercise 2

Variable labels & 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

Var	Rename to	Label with
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"

8.9 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
|	or

- Note the double equals signs for testing equality

8.10 Generating & 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
```

8.11 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

8.12 Exercise solutions

8.12.1 Ex 0: prototype

8.12.2 Ex 1: prototype

8.12.3 Ex 2: prototype

8.12.4 Ex 3: prototype

8.13 Wrap-up

8.13.1 Feedback

These workshops are a work in progress, please provide any feedback to: `help@iq.harvard.edu`

8.13.2 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`
- Stata
  - UCLA website: `http://www.ats.ucla.edu/stat/Stata/`
  - Stata website: `http://www.stata.com/help.cgi?contents`
  - Email list: `http://www.stata.com/statalist/`

Chapter 9

Stata Data Management

Topics

- Generating and replacing variables
- Processing with `by` statements
- Missing values
- Variable types and conversion
- Merging, appending, and joining
- Creating summarized data sets

9.1 Setup

9.1.1 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/raw/master/Stata/StataDatMan.zip`
- Extract materials from the zipped directory `StataDatMan.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

9.1.2 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

9.1.3 Goals

- This is an introduction to data management in Stata
- Assumes basic knowledge of Stata
- Not appropriate for people already familiar with Stata
- If you are catching on before the rest of the class, experiment with command features described in help files
- Learning Objectives:
  - Basic data manipulation commands
  - Processing with `by` statements
  - Dealing with missing values
  - Variable types and conversion
  - Merging and appending datasets

9.2 Opening Files in Stata

- Look at bottom left hand corner of Stata screen
  - This is the directory Stata is currently reading from
- Files are located in the StataDatMan folder in your home directory
- Start by telling Stata where to look for these

```
// change directory
cd "~/Desktop/Stata/StataDatMan"

// Use dir to see what is in the directory:
dir
dir dataSets

// use the gss data set
use dataSets/gss.dta
```

```
set more off

cd "~/Desktop/Stata/StataDatMan"
/nfs/www/edu-harvard-iq-tutorials/Stata/StataDatMan


dir

total 100
drwxrwsr-x. 2 apache tutorwww  4096 Oct  9 08:44 dataSets/
-rwxrwxr-x. 1 izahn  tutorwww  1302 Oct  9 08:44 Exercises.do*
drwxrwsr-x. 2 apache tutorwww  4096 Oct  9 08:44 images/
drwxrwsr-x. 4 apache tutorwww  4096 Oct  9 08:44 StataDatMan/
-rwxrwxr-x. 1 izahn  tutorwww 17446 Oct  9 08:44 StataDatMan.do*
-rwxrwxr-x. 1 izahn  tutorwww 38153 Oct  9 08:44 StataDatMan.html*
-rwxrwxr-x. 1 izahn  tutorwww 20463 Oct  9 08:44 StataDatMan.org*
```

```
dir dataSets
```

```
total 2644
-rwxrwxr-x. 1 izahn tutorwww 275705 Oct  9 08:44 gss1.dta*
-rwxrwxr-x. 1 izahn tutorwww 263324 Oct  9 08:44 gss2.dta*
-rwxrwxr-x. 1 izahn tutorwww 532880 Oct  9 08:44 gssAddObserve.dta*
-rwxrwxr-x. 1 izahn tutorwww 527005 Oct  9 08:44 gssAppend.dta*
-rwxrwxr-x. 1 izahn tutorwww 527005 Oct  9 08:44 gsscompare1.dta*
-rwxrwxr-x. 1 izahn tutorwww 538755 Oct  9 08:44 gss.dta*
-rwxrwxr-x. 1 izahn tutorwww   1139 Oct  9 08:44 marital.dta*
```

```
use dataSets/gss.dta
```

9.3 Generating & replacing variables

9.3.1 Data Manipulation Commands

Basic commands you'll use for generating new variables or recoding existing variables:

- gen
- egen
- replace
- recode

Many different means of accomplishing the same thing in Stata – find what is comfortable (and easy) for you!

9.3.2 Generate & Replace

The `replace` command is often used with logic statements. Available logical operators include the following:

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

For example:

```
//create "hapnew" variable
gen hapnew = .
//set to 0 if happy equals 1
replace hapnew=0 if happy==1
//set to 1 if happy both and hapmar are greater than 3
replace hapnew=1 if happy>3 & hapmar>3
// tabulate the new
tab hapnew
```

```
gen hapnew = .
(1,419 missing values generated)

replace hapnew=0 if happy==1
(435 real changes made)

replace hapnew=1 if happy>3 & hapmar>3
(4 real changes made)

tab hapnew

     hapnew |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |        435       99.09       99.09
          1 |          4        0.91      100.00
------------+-----------------------------------
      Total |        439      100.00
```

9.3.3 Recode

The `recode` command is basically generate and replace combined. You can recode an existing variable OR use recode to create a new variable (via the `gen` option).

```
// recode the wrkstat variable
recode wrkstat (1=8) (2=7) (3=6) (4=5) (5=4) (6=3) (7=2) (8=1)
// recode wrkstat into a new variable named wrkstat2
recode wrkstat (1=8), gen(wrkstat2)
// tabulate workstat
tab wrkstat
```

```
recode wrkstat (1=8) (2=7) (3=6) (4=5) (5=4) (6=3) (7=2) (8=1)
(wrkstat: 1419 changes made)

recode wrkstat (1=8), gen(wrkstat2)
(32 differences between wrkstat and wrkstat2)

tab wrkstat

      LABOR FRCE |
```

```
           STATUS |      Freq.     Percent        Cum.
------------------+-----------------------------------
 WORKING FULLTIME |         32        2.26        2.26
 WORKING PARTTIME |        155       10.92       13.18
 TEMP NOT WORKING |         34        2.40       15.57
UNEMPL, LAID OFF |        214       15.08       30.66
          RETIRED |         29        2.04       32.70
           SCHOOL |         35        2.47       35.17
    KEEPING HOUSE |        146       10.29       45.45
            OTHER |        774       54.55      100.00
------------------+-----------------------------------
            Total |      1,419      100.00
```

The table below illustrates common forms of recoding

Rule	Example	Meaning
#=#	3=1	3 recoded to 1
##=#	`2. 9 | 2` `and . recoded to 9` `|` `| #/#` #	1/5=4
nonmissing=#	nonmiss=8	nonmissing recoded to 8
missing=#	miss=9	missing recoded to 9

9.3.4 egen

The `egen` command ("extensions" to the `gen` command) provides convenient methods for performing many common data manipulation tasks.

For example, we can use `egen` to create a new variable that counts the number of "yes" responses on computer, email and internet use:

```
// count number of yes on use comp email and net
egen compuser= anycount(usecomp usemail usenet), values(1)
tab compuser
```

```
egen compuser= anycount(usecomp usemail usenet), values(1)
tab compuser

    usecomp |
    usemail |
usenet == 1 |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |        623       43.90       43.90
          1 |        142       10.01       53.91
          2 |         78        5.50       59.41
          3 |        576       40.59      100.00
------------+-----------------------------------
```

```
     Total |      1,419      100.00
```

Here are some additional examples of `egen` in action:

```
// assess how much missing data each participant has:
egen countmiss = rowmiss(age-wifeft)
codebook countmiss
// compare values on multiple variables
egen ftdiff=diff(wkftwife wkfthusb)
codebook ftdiff
```

```
egen countmiss = rowmiss(age-wifeft)
codebook countmiss

-------------------------------------------------------------------------------
countmiss                                                           (unlabeled)
-------------------------------------------------------------------------------

                  type:  numeric (float)

                 range:  [0,7]                        units:  1
         unique values:  6                        missing .:  0/1,419

            tabulation:  Freq.  Value
                           296  0
                           215  1
                           113  2
                             7  3
                           782  6
                             6  7

egen ftdiff=diff(wkftwife wkfthusb)
codebook ftdiff

-------------------------------------------------------------------------------
ftdiff                                                   diff wkftwife wkfthusb
-------------------------------------------------------------------------------

                  type:  numeric (float)

                 range:  [0,1]                        units:  1
         unique values:  2                        missing .:  0/1,419

            tabulation:  Freq.  Value
                         1,169  0
                           250  1
```

You will need to refer to the documentation to discover what else `egen` can do: type “help egen” in Stata to get a complete list of functions.

9.4 Exercise 0

Generate, Replace, Recode & Egen

Open the gss.dta data.

1. Generate a new variable that represents the squared value of age.
2. Generate a new variable equal to "1" if income is greater than "19".
3. Create a new variable that counts the number of missing responses for each respondent. What is the maximum number of missing variables?

9.5 By processing

9.5.1 The bysort Command

Sometimes, you'd like to create variables based on different categories of a single variable. For example, say you want to look at happiness based on whether an individual is male or female. The "bysort" prefix does just this:

```
// tabulate happy separately for male and female
bysort sex: tab happy
// generate summary statistics using bysort
bysort state: egen stateincome = mean(income)
bysort degree: egen degreeincome = mean(income)
bysort marital: egen marincomesd = sd(income)
```

```
bysort sex: tab happy

-------------------------------------------------------------------------------
-> sex = Male

      GENERAL |
    HAPPINESS |      Freq.     Percent        Cum.
--------------+-----------------------------------
   VERY HAPPY |        189       30.39       30.39
 PRETTY HAPPY |        350       56.27       86.66
NOT TOO HAPPY |         73       11.74       98.39
           NA |         10        1.61      100.00
--------------+-----------------------------------
        Total |        622      100.00

-------------------------------------------------------------------------------
-> sex = Female

      GENERAL |
    HAPPINESS |      Freq.     Percent        Cum.
--------------+-----------------------------------
   VERY HAPPY |        246       30.87       30.87
```

```
 PRETTY HAPPY |        447       56.09       86.95
NOT TOO HAPPY |         84       10.54       97.49
           DK |          1        0.13       97.62
           NA |         19        2.38      100.00
--------------+-----------------------------------
        Total |        797      100.00

bysort state: egen stateincome = mean(income)
variable state not found
r(111);
bysort degree: egen degreeincome = mean(income)
bysort marital: egen marincomesd = sd(income)
```

9.5.2 by prefix vs. by options

Some commands won't work with by prefix, but instead have a `by` option:

```
// generate separate histograms for female and male
hist nethrs, by(sex)
```

9.6 Missing values

You always need to consider how missing values are coded when recoding variables.

- Stata's symbol for a missing value is `.`
- Stata interprets `.` as a large value
- Easy to make mistakes!

To identify highly educated women, we might use the command:

```
// generate and replace without considering missing values
gen hi_ed=0
replace hi_ed=1 if wifeduc>15
// What happens to our missing values?
tab hi_ed, mi nola
```

```
gen hi_ed=0
replace hi_ed=1 if wifeduc>15
(944 real changes made)

tab hi_ed, mi nola

      hi_ed |      Freq.     Percent        Cum.
------------+-----------------------------------
          0 |        475       33.47       33.47
          1 |        944       66.53      100.00
------------+-----------------------------------
      Total |      1,419      100.00
```

It looks like around 66% have higher education, but look closer:

```
// gen hi_ed2, but don't set a value if wifeduc is missing
gen hi_ed2 = 0 if wifeduc != .
// only replace non-missing
replace hi_ed2=1 if wifeduc >15 & wifeduc !=.
//check to see that missingness is preserved
tab hi_ed2, mi
```

```
gen hi_ed2 = 0 if wifeduc != .
(797 missing values generated)

replace hi_ed2=1 if wifeduc >15 & wifeduc !=.
(147 real changes made)

 |        797       56.17      100.00
------------+-----------------------------------
      Total |      1,419      100.00
```

The correct value is 10%. Moral of the story? Be careful with missing values and remember that Stata considers missing values to be large!

9.6.1 Bulk Conversion to Missing Values

Often the data collection/generating procedure will have used some other value besides `.` to represent missing values. The `mvdecode` command will convert all these values to missing. For example:

```
mvdecode _all, mv(999)
```

```
mvdecode _all, mv(999)
```

- The "_all" command tells Stata to do this to all variables
- Use this command carefully!
  - If you have any variables where "999" is a legitimate value, Stata is going to recode it to missing
  - As an alternative, you could list var names separately rather than using "_all"

9.7 Variable types

Stata uses two main types of variables: String and Numeric. To be able to perform any mathematical operations, your variables need to be in a numeric format. Stata can store numbers with differing levels of precision, as described in the table below.

type	Minimum	Maximum	being 0	bytes
byte	-127	100	+/-1	1
int	-32,767	32,740	+/-1	2
long	-2,147,483,647	2,147,483,620	+/-1	4

type	Minimum	Maximum	being 0	bytes
float	-1.70141173319*1038	1.70141173319*1038	+/-10-38	4
double	-8.9884656743*10307	8.9884656743*10307	+/-10-323	8

- Precision for float is 3.795x10-8.
- Precision for double is 1.414x10-16.

9.7.1 Converting to & from Strings

Stata provides several ways to convert to and from strings. You can use `tostring` and `destring` to convert from one type to the other:

```
// convert degree to a string
tostring degree, gen(degree_s)
// and back to a number
destring degree_s, gen(degree_n)
```

```
tostring degree, gen(degree_s)
degree_s generated as str1

destring degree_s, gen(degree_n)
degree_s has all characters numeric; degree_n generated as byte
```

Use `decode` and `encode` to convert to/from variable labels:

```
// convert degree to a descriptive string
decode degree, gen(degree_s2)
// and back to a number with labels
encode degree_s2, gen(degree_n2)
```

```
decode degree, gen(degree_s2)

encode degree_s2, gen(degree_n2)
```

9.7.2 Converting Strings to Date/Time

Often date/time variables start out as strings – You'll need to convert them to numbers using one of the conversion functions listed below.

Format	Meaning	String-to-numeric conversion function
%tc	milliseconds	clock(string, mask)
%td	days	date(string, mask)
%tw	weeks	weekly(string, mask)
%tm	months	monthly(string, mask)
%tq	quarters	quarterly(string, mask)
%ty	years	yearly(string, mask)

Date/time variables are stored as the number of units elapsed since 01jan1960 00:00:00.000. For example, the `date` function returns the number of days since that time, and the `clock` function returns the number of milliseconds since that time.

```
// create string variable and convert to date
gen date = "November 9 2020"
gen date1 = date(date, "MDY")
list date1 in 1/5
```

```
gen date = "November 9 2020"
gen date1 = date(date, "MDY")
list date1 in 1/5

     +-------+
     | date1 |
     |-------|
  1. | 22228 |
  2. | 22228 |
  3. | 22228 |
  4. | 22228 |
  5. | 22228 |
     +-------+
```

9.7.3 Formatting Numbers as Dates

Once you have converted the string to a number you can format it for display. You can simply accept the defaults used by your formatting string or provide details to customize it.

```
// format so humans can read the date
format date1 %d
list date1 in 1/5
// format with detail
format date1 %tdMonth_dd,_CCYY
list date1 in 1/5
```

```
format date1 %d
list date1 in 1/5

     +-----------+
     |     date1 |
     |-----------|
  1. | 09nov2020 |
  2. | 09nov2020 |
  3. | 09nov2020 |
  4. | 09nov2020 |
  5. | 09nov2020 |
     +-----------+

format date1 %tdMonth_dd,_CCYY
```

```
list date1 in 1/5

     +-------------------+
     |             date1 |
     |-------------------|
  1. | November 9, 2020 |
  2. | November 9, 2020 |
  3. | November 9, 2020 |
  4. | November 9, 2020 |
  5. | November 9, 2020 |
     +-------------------+
```

9.8 Exercise 1

Missing Values, String Conversion, & by Processing

1. Recode values "99" and "98" on the variable, "hrs1" as "missing."
2. Recode the marital variable into a "string" variable and then back into a numeric variable.
3. Create a new variable that associates each individual with the average number of hours worked among individuals with matching educational degrees (see the last "by" example for inspiration).

9.9 Merging, appending, & joining

9.9.1 Appending Datasets

Sometimes you have observations in two different datasets, or you'd like to add observations to an existing dataset. In this case you can use the `append` command to add observations to the end of the observations in the master dataset. For example:

```
clear
// from the append help file
webuse even
list
webuse odd
list
// Append even data to the end of the odd data
append using "http://www.stata-press.com/data/r14/even"
list
clear
```

```
clear

webuse even
(6th through 8th even numbers)
```

```
list

     +---------------+
     | number   even |
     |---------------|
  1. |      6     12 |
  2. |      7     14 |
  3. |      8     16 |
     +---------------+
webuse odd
(First five odd numbers)
list

     +--------------+
     | number   odd |
     |--------------|
  1. |      1     1 |
  2. |      2     3 |
  3. |      3     5 |
  4. |      4     7 |
  5. |      5     9 |
     +--------------+

append using "http://www.stata-press.com/data/r14/even"
list

     +---------------------+
     | number   odd   even |
     |---------------------|
  1. |      1     1      . |
  2. |      2     3      . |
  3. |      3     5      . |
  4. |      4     7      . |
  5. |      5     9      . |
     |---------------------|
  6. |      6     .     12 |
  7. |      7     .     14 |
  8. |      8     .     16 |
     +---------------------+
clear
```

To keep track of where observations came from, use the `generate` option as shown below:

```
webuse odd
append using "http://www.stata-press.com/data/r14/even", generate(observesource)
list
clear
```

```
webuse odd
```

```
(First five odd numbers)
 ce)
list

     +--------------------------------+
     | number   odd   observ~e   even |
     |--------------------------------|
  1. |      1     1          0      . |
  2. |      2     3          0      . |
  3. |      3     5          0      . |
  4. |      4     7          0      . |
  5. |      5     9          0      . |
     |--------------------------------|
  6. |      6     .          1     12 |
  7. |      7     .          1     14 |
  8. |      8     .          1     16 |
     +--------------------------------+
clear
```

There is a "force" option will allow for data type mismatches, but again this is not recommended.

Remember, `append` is for adding observations (i.e., rows) from a second data set.

9.9.2 Merging Datasets

You can `merge` variables from a second dataset to the dataset you're currently working with.

- Current active dataset = master dataset
- Dataset you'd like to merge with master = using dataset

There are different ways that you might be interested in merging data:

- Two datasets with same participant pool, one row per participant (1:1)
- A dataset with one participant per row with a dataset with multiple rows per participant (1:many or many:1)

Before you begin:

- Identify the "ID" that you will use to merge your two datasets
- Determine which variables you'd like to merge
- In Stata >= 11, data does NOT have to be sorted
- Variable types must match across datasets (there is a "force" option to get around this, but not recommended)

```
// Adapted from the merge help page
webuse autosize
list
webuse autoexpense
list
```

```
webuse autosize
merge 1:1 make using "http://www.stata-press.com/data/r14/autoexpense"
list
clear

// keep only the matches (AKA "inner join")
webuse autosize, clear
merge 1:1 make using "http://www.stata-press.com/data/r14/autoexpense", keep(match) nogen
list
clear
```

```
webuse autosize
(1978 Automobile Data)
list

     +------------------------------------+
     | make               weight   length |
     |------------------------------------|
  1. | Toyota Celica       2,410      174 |
  2. | BMW 320i            2,650      177 |
  3. | Cad. Seville        4,290      204 |
  4. | Pont. Grand Prix    3,210      201 |
  5. | Datsun 210          2,020      165 |
     |------------------------------------|
  6. | Plym. Arrow         3,260      170 |
     +------------------------------------+
webuse autoexpense
(1978 Automobile Data)
list

     +---------------------------------+
     | make                price   mpg |
     |---------------------------------|
  1. | Toyota Celica       5,899    18 |
  2. | BMW 320i            9,735    25 |
  3. | Cad. Seville       15,906    21 |
  4. | Pont. Grand Prix    5,222    19 |
  5. | Datsun 210          4,589    35 |
     +---------------------------------+

webuse autosize
(1978 Automobile Data)
merge 1:1 make using "http://www.stata-press.com/data/r14/autoexpense"

    Result                           # of obs.
    -----------------------------------------
    not matched                             1
        from master                         1  (_merge==1)
```

```
        from using                          0  (_merge==2)

    matched                                 5  (_merge==3)
    -----------------------------------------
list

     +----------------------------------------------------------------------+
     | make                weight   length    price   mpg            _merge |
     |----------------------------------------------------------------------|
  1. | BMW 320i             2,650      177    9,735    25       matched (3) |
  2. | Cad. Seville         4,290      204   15,906    21       matched (3) |
  3. | Datsun 210           2,020      165    4,589    35       matched (3) |
  4. | Plym. Arrow          3,260      170        .     .   master only (1) |
  5. | Pont. Grand Prix     3,210      201    5,222    19       matched (3) |
     |----------------------------------------------------------------------|
  6. | Toyota Celica        2,410      174    5,899    18       matched (3) |
     +----------------------------------------------------------------------+
clear


webuse autosize, clear
(1978 Automobile Data)
 match) nogen

    Result                           # of obs.
    -----------------------------------------
    not matched                             0
    matched                                 5
    -----------------------------------------
list

     +---------------------------------------------------+
     | make                weight   length    price   mpg |
     |---------------------------------------------------|
  1. | BMW 320i             2,650      177    9,735    25 |
  2. | Cad. Seville         4,290      204   15,906    21 |
  3. | Datsun 210           2,020      165    4,589    35 |
  4. | Pont. Grand Prix     3,210      201    5,222    19 |
  5. | Toyota Celica        2,410      174    5,899    18 |
     +---------------------------------------------------+
clear
```

Remember, `merge` is for adding variables (i.e., columns) from a second data set.

9.9.3 Merge Options

There are several options that provide more fine-grain control over what happens to non-id columns contained in both data sets. If you've carefully cleaned and prepared the data prior to merging this shouldn't be an issue, but here are some details about how stata handles this situation.

- In standard merge, the master dataset is the authority and WON'T CHANGE
- If your master dataset has missing data and some of those values are not missing in your using dataset, specify "update" – this will fill in missing data in master
- If you want data from your using dataset to overwrite that in your master, specify "replace update" – this will replace master data with using data UNLESS the value is missing in the using dataset

9.9.4 Many-to-many merges

Stata allows you to specify merges like `merge m:m id using newdata.dta`, but I have never seen this do anything useful. To quote the official Stata manual:

> `m:m` specifies a many-to-many merge and is a **bad idea**. In an `m:m` merge, observations are matched within equal values of the key variable(s), with the first observation being matched to the first; the second, to the second; and so on. If the master and using have an unequal number of observations within the group, then the last observation of the shorter group is used repeatedly to match with subsequent observations of the longer group. Thus **m:m merges are dependent on the current sort order—something which should never happen**. **Because m:m merges are such a bad idea, we are not going to show you an example**. If you think that you need an m:m merge, then you probably need to work with your data so that you can use a 1:m or m:1 merge. Tips for this are given in Troubleshooting m:m merges below

(emphasis added).

If you are thinking about using `merge m:m` chances are good that you actually need `joinby`. Here is a quick example, modified from the `joinby` help page.

```
clear
webuse parent
list
webuse children
list
// Complete and utter nonsense!
merge m:m family_id using http://www.stata-press.com/data/r14/parent
// You want joinby instead
clear
webuse children
joinby family_id using http://www.stata-press.com/data/r14/parent
```

Remember, `merge m:m` is old and broken; **do not use**. Anytime you think you might want `m:m` you should use `joinby` instead.

9.10 Creating summarized data sets

9.10.1 Collapse

Collapse will take master data and create a new dataset of summary statistics

- Useful in hierarchical linear modeling if you'd like to create aggregate, summary statistics
- Can generate group summary data for many descriptive stats
- Can also attach weights

Before you collapse:

- Save your master dataset and then save it again under a new name (this will prevent collapse from writing over your original data_
- Consider issues of missing data. Do you want Stata to use all possible observations? If not, the `cw` (casewise) option will make casewise deletions

```
// Adapted from the collapse help page
clear
webuse college
list
// mean and sd by hospital
collapse (mean) mean_gpa = gpa mean_hour = hour (sd) sd_gpa = gpa sd_hour = hour, by(year)
list
clear
```

```
clear
webuse college
list

     +----------------------------+
     | gpa   hour   year   number |
     |----------------------------|
  1. | 3.2     30      1        3 |
  2. | 3.5     34      1        2 |
  3. | 2.8     28      1        9 |
  4. | 2.1     30      1        4 |
  5. | 3.8     29      2        3 |
     |----------------------------|
  6. | 2.5     30      2        4 |
  7. | 2.9     35      2        5 |
  8. | 3.7     30      3        4 |
  9. | 2.2     35      3        2 |
 10. | 3.3     33      3        3 |
     |----------------------------|
 11. | 3.4     32      4        5 |
 12. | 2.9     31      4        2 |
     +----------------------------+
```

```
 our, by(year)
list

     +-------------------------------------------------+
     | year   mean_gpa   mean_h~r     sd_gpa    sd_hour |
     |-------------------------------------------------|
  1. |    1        2.9       30.5   .6055301   2.516612 |
  2. |    2   3.066667   31.33333   .6658328    3.21455 |
  3. |    3   3.066667   32.66667   .7767453   2.516612 |
  4. |    4       3.15       31.5   .3535534   .7071068 |
     +-------------------------------------------------+
clear
```

You could also generate different statistics for multiple variables

9.11 Exercise 2

Merge, Append, & Collapse

Open the gss2.dta dataset. This dataset contains only half of the variables that are in the complete gss dataset.

1. Merge dataset gss1.dta with dataset gss2.dta. The identification variable is "id."
2. Open the gss.dta dataset and merge in data from the "marital.dta" dataset, which includes income information grouped by individuals' marital status. The marital dataset contains collapsed data regarding average statistics of individuals based on their marital status.
3. Open the gssAppend.dta dataset and Create a new dataset that combines the observations in gssAppend.dta with those in gssAddObserve.dta.
4. Open the gss.dta dataset. Create a new dataset that summarizes mean and standard deviation of income based on individuals' degree status ("degree"). In the process of creating this new dataset, rename your three new variables.

9.12 Exercise Solutions

9.12.1 Ex 0: prototype

Open the gss.dta data.

1. Generate a new variable that represents the squared value of age.

```
use dataSets/gss.dta, clear
gen age2 = age^2
```

2. Generate a new variable equal to "1" if income is greater than "19".

```
    describe income
    label list income
    recode income (99=.) (98=.)
    gen highincome =0 if income != .
    replace highincome=1 if income>19
    sum highincome
```

3. Create a new variable that counts the number of missing responses for each respondent. What is the maximum number of missing variables?

```
egen nmissing = rowmiss(_all)
sum nmissing
```

9.12.2 Ex 1: prototype

1. Recode values "99" and "98" on the variable, "hrs1" as "missing."

```
use dataSets/gss.dta, clear
sum hrs1
recode hrs1 (99=.) (98=.)
sum hrs1
```

2. Recode the marital variable into a "string" variable and then back into a numeric variable.

```
tostring marital, gen(marstring)
destring marstring, gen(mardstring)
//compare with
decode marital, gen(marital_s)
encode marital_s, gen(marital_n)

describe marital marstring mardstring marital_s marital_n
sum marital marstring mardstring marital_s marital_n
```

3. Create a new variable that associates each individual with the average number of hours worked among individuals with matching educational degrees (see the last "by" example for inspiration).

```
bysort degree: egen hrsdegree = mean(hrs1)
tab hrsdegree
tab hrsdegree degree
```

9.12.3 Ex 2: prototype

Open the gss2.dta dataset. This dataset contains only half of the variables that are in the complete gss dataset.

1. Merge dataset gss1.dta with dataset gss2.dta. The identification variable is "id."

```
use dataSets/gss2.dta, clear
merge 1:1 id using dataSets/gss1.dta
save gss3.dta, replace
```

2. Open the gss.dta dataset and merge in data from the “marital.dta” dataset, which includes income information grouped by individuals’ marital status. The marital dataset contains collapsed data regarding average statistics of individuals based on their marital status.

```
use dataSets/gss.dta, clear
merge m:1 marital using dataSets/marital.dta, nogenerate replace update
save gss4.dta, replace
```

3. Open the gssAppend.dta dataset and Create a new dataset that combines the observations in gssAppend.dta with those in gssAddObserve.dta.

```
use dataSets/gssAppend.dta, clear
append using dataSets/gssAddObserve, generate(observe)
```

4. Open the gss.dta dataset. Create a new dataset that summarizes mean and standard deviation of income based on individuals’ degree status (“degree”). In the process of creating this new dataset, rename your three new variables.

```
use dataSets/gss.dta, clear
save collapse2.dta, replace
use collapse2.dta, clear
collapse (mean) meaninc=income (sd) sdinc=income, by(marital)
```

9.13 Wrap-up

9.13.1 Feedback

These workshops are a work-in-progress, please provide any feedback to: `help@iq.harvard.edu`

9.13.2 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`
- Stata

- UCLA website: `http://www.ats.ucla.edu/stat/Stata/`
- Stata website: `http://www.stata.com/help.cgi?contents`
- Email list: `http://www.stata.com/statalist/`

Chapter 10

Stata Modeling & Graphing

Topics

- Stata modeling
  - Simple regression
  - Multiple regression
  - Interactions
  - Exporting regression tables
  - Testing model assumptions
- Stata graphing
  - Univariate graphs
  - Bivariate graphs

10.1 Setup

10.1.1 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/raw/master/Stata/StataModGraph.zip`
- Extract materials from the zipped directory `StataModGraph.zip` (Right-click => Extract All on Windows, double-click on Mac) and move them to your desktop!

10.1.2 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

10.1.3 Goals

- This is an introduction to modeling and visualization in Stata
- Assumes basic knowledge of Stata
- Not appropriate for people already familiar with Stata
- If you are catching on before the rest of the class, experiment with command features described in help files
- Learning Objectives:
  - Fit models in Stata
  - Test modeling assumptions
  - Plot basic graphs in Stata
  - Plot two-way graphs

10.2 Fitting models in Stata

10.2.1 Today's Dataset

- We have data on a variety of variables for all 50 states
- Population, density, energy use, voting tendencies, graduation rates, income, etc.
- We're going to be predicting SAT scores
- Univariate Regression: SAT scores and Education Expenditures
- Does the amount of money spent on education affect the mean SAT score in a state?
- Dependent variable: csat
- Independent variable: expense

10.2.2 Opening Files in Stata

- Look at bottom left hand corner of Stata screen
  - This is the directory Stata is currently reading from
- Files are located in the StataStatistics folder on the Desktop
- Start by telling Stata where to look for these

```
// change directory
cd "~/Desktop/Stata/StataStatGraph"
```

```
set more off

cd "~/Desktop/Stata/StataStatGraph"
/nfs/www/edu-harvard-iq-tutorials/Stata/StataStatGraph
```

- Use dir to see what is in the directory:

```
dir
cd dataSets
```

```
dir
cd ..
```

```
dir

total 8
drwxr-sr-x. 2 izahn tutorwww 4096 Oct 22 21:59 dataSets/
drwxr-sr-x. 3 izahn tutorwww 4096 Oct 22 21:59 images/
cd dataSets
/nfs/www/edu-harvard-iq-tutorials/Stata/StataStatGraph/dataSets
dir

total 21008
-rwxr-xr-x. 1 izahn tutorwww 21103444 Oct 22 21:59 NatNeighCrimeStudy.dta*
-rwxr-xr-x. 1 izahn tutorwww     8977 Oct 22 21:59 states.dta*
-rwxr-xr-x. 1 izahn tutorwww   298191 Oct 22 21:59 TimePollPubSchools.dta*
cd ..
/nfs/www/edu-harvard-iq-tutorials/Stata/StataStatGraph
```

- Load the data

```
// use the states data set
use dataSets/states.dta
```

```
use dataSets/states.dta
(U.S. states data 1990-91)
```

10.3 Simple regression

10.3.1 Steps for Running Regression

1. Examine descriptive statistics
2. Look at relationship graphically and test correlation(s)
3. Run and interpret regression
4. Test regression assumptions

10.3.2 Preliminaries

- We want to predict csat scores from expense
- First, let's look at some descriptives

```
// generate summary statistics for csat and expense
sum csat expense
```

```
sum csat expense

    Variable |        Obs        Mean    Std. Dev.       Min        Max
```

```
-------------+---------------------------------------------------------
        csat |         51     944.098    66.93497        832       1093
     expense |         51    5235.961    1401.155       2960       9259
```

- We want to predict csat scores from expense
- First, let's look at some descriptives

```
// look at codebok
codebook csat expense
```

```
codebook csat expense

-------------------------------------------------------------------------------
csat                                                   Mean composite SAT score
-------------------------------------------------------------------------------

                  type:  numeric (int)

                 range:  [832,1093]                   units:  1
         unique values:  45                       missing .:  0/51

                  mean:   944.098
              std. dev:    66.935

           percentiles:        10%       25%       50%       75%       90%
                               874       886       926       997      1024

-------------------------------------------------------------------------------
expense                                        Per pupil expenditures prim&sec
-------------------------------------------------------------------------------

                  type:  numeric (int)

                 range:  [2960,9259]                  units:  1
         unique values:  51                       missing .:  0/51

                  mean:   5235.96
              std. dev:   1401.16

           percentiles:        10%       25%       50%       75%       90%
                              3782      4351      5000      5865      6738
```

- Next, view relationship graphically
- Scatterplots work well for univariate relationships

```
// graph expense by csat
twoway scatter expense csat
```

- Next look at the correlation matrix

```
// correlate csat and expense
pwcorr csat expense, star(.05)
```

```
pwcorr csat expense, star(.05)

             |     csat  expense
-------------+------------------
        csat |   1.0000 
     expense |  -0.4663*  1.0000
```

- Not very interesting with only one predictor

10.3.3 SAT scores & Education Expenditures

```
regress csat expense
```

```
regress csat expense

      Source |       SS           df       MS      Number of obs   =        51
-------------+----------------------------------   F(1, 49)        =     13.61
       Model |  48708.3001         1  48708.3001   Prob > F        =    0.0006
    Residual |   175306.21        49  3577.67775   R-squared       =    0.2174
-------------+----------------------------------   Adj R-squared   =    0.2015
       Total |   224014.51        50  4480.2902    Root MSE        =    59.814

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     expense |  -.0222756   .0060371    -3.69   0.001    -.0344077   -.0101436
       _cons |   1060.732    32.7009    32.44   0.000     995.0175    1126.447
------------------------------------------------------------------------------
```

10.3.4 OLS Assumptions

- Assumption 1: Specification is appropriate (i.e., no relevant omitted variables)
- Assumption 2: Homoscedasticity (The variance around the regression model is the same for all values of the predictor variable)
- Assumption 3: Errors are independent
- Assumption 4: Relationships are linear
- Assumption 5: Normal Distribution (only needed for inference)
  - The errors of regression equation are normally distributed

10.3.4.1 Specification

The model specification should be informed by theory - i.e., our substantive knowledge of the subject matter. It's important to include all relevant predictors in the model, otherwise our estimates will be biased.

- Goodness of fit

10.3.4.2 Homoscedasticity

```
rvfplot
```

```
rvfplot
```

10.3.4.3 Normality

- A simple histogram of the residuals can be informative

```
// graph the residual values of csat
predict resid, residual
histogram resid, normal
```

```
predict resid, residual
histogram resid, normal
(bin=7, start=-131.81111, width=38.329487)
```

10.4 Multiple Regression

- Just keep adding predictors
- Let's try adding some predictors to the model of SAT scores
- income :: % students taking SATs
- percent :: % adults with HS diploma (high)

10.4.1 Preliminaries

- As before, start with descriptive statistics and correlations

```
// descriptive statistics and correlations
sum income percent high
pwcorr csat expense income percent high
```

```
sum income percent high

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
      income |         51    33.95657    6.423134     23.465     48.618
     percent |         51    35.76471    26.19281          4         81
        high |         51    76.26078    5.588741       64.3       86.6
pwcorr csat expense income percent high
```

```
             |     csat  expense   income  percent     high
-------------+---------------------------------------------
        csat |   1.0000
     expense |  -0.4663   1.0000
      income |  -0.4713   0.6784   1.0000
     percent |  -0.8758   0.6509   0.6733   1.0000
        high |   0.0858   0.3133   0.5099   0.1413   1.0000
```

- regress csat on exense, income, percent, and high

```
regress csat expense income percent high
```

```
regress csat expense income percent high

      Source |       SS           df       MS      Number of obs   =        51
-------------+----------------------------------   F(4, 46)        =     51.86
       Model |  183354.603         4  45838.6508   Prob > F        =    0.0000
    Residual |  40659.9067        46  883.911016   R-squared       =    0.8185
-------------+----------------------------------   Adj R-squared   =    0.8027
       Total |   224014.51        50   4480.2902   Root MSE        =    29.731

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     expense |   .0045604    .004384     1.04   0.304    -.0042641     .013385
      income |   .4437858   1.138947     0.39   0.699    -1.848795    2.736367
     percent |  -2.533084   .2454477   -10.32   0.000    -3.027145   -2.039024
        high |   2.086599   .9246023     2.26   0.029     .2254712    3.947727
       _cons |   836.6197   58.33238    14.34   0.000     719.2027    954.0366
------------------------------------------------------------------------------
```

10.5 Exercise 0

Multiple Regression

Open the datafile, states.dta.

1. Select a few variables to use in a multiple regression of your own. Before running the regression, examine descriptive of the variables and generate a few scatterplots.
2. Run your regression
3. Examine the plausibility of the assumptions of normality and homogeneity

10.6 Interactions

- What if we wanted to test an interaction between percent & high?
- Option 1: generate product terms by hand

```
// generate product of percent and high
gen percenthigh = percent*high
regress csat expense income percent high percenthigh
```

```
gen percenthigh = percent*high
regress csat expense income percent high percenthigh

      Source |       SS           df       MS      Number of obs   =        51
-------------+----------------------------------   F(5, 45)        =     46.11
       Model |  187430.401         5  37486.0801   Prob > F        =    0.0000
    Residual |  36584.1091        45  812.980201   R-squared       =    0.8367
-------------+----------------------------------   Adj R-squared   =    0.8185
       Total |   224014.51        50  4480.2902   Root MSE        =    28.513

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     expense |   .0045575   .0042044     1.08   0.284    -.0039107    .0130256
      income |   .0887856    1.10374     0.08   0.936    -2.134261    2.311832
     percent |  -8.143002   2.516509    -3.24   0.002    -13.21151   -3.074493
        high |   .4240906   1.156545     0.37   0.716    -1.905311    2.753492
 percenthigh |   .0740926   .0330909     2.24   0.030     .0074441    .1407411
       _cons |    972.525    82.5457    11.78   0.000     806.2695    1138.781
------------------------------------------------------------------------------
```

- What if we wanted to test an interaction between percent & high?
- Option 2: Let Stata do your dirty work

```
// use the # sign to represent interactions
regress csat percent high c.percent#c.high
// same as . regress csat c.percent##high
```

```
regress csat percent high c.percent#c.high

      Source |       SS           df       MS      Number of obs   =        51
-------------+----------------------------------   F(3, 47)        =     77.39
       Model |  186302.091         3  62100.6971   Prob > F        =    0.0000
    Residual |  37712.4186        47  802.391885   R-squared       =    0.8317
-------------+----------------------------------   Adj R-squared   =    0.8209
       Total |   224014.51        50  4480.2902   Root MSE        =    28.327

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     percent |   -8.15717   2.488388    -3.28   0.002    -13.16316   -3.151179
        high |   .6674578   1.082615     0.62   0.541    -1.510482    2.845398
             |
   c.percent#|
      c.high |   .0764271   .0324919     2.35   0.023     .0110619    .1417924
```

```
             |
       _cons |   974.9354   81.98078    11.89   0.000     810.0113    1139.859
------------------------------------------------------------------------------
```

10.6.1 Categorical Predictors

- For categorical variables, we first need to dummy code
- Use region as example
  - Option 1: create dummy codes before fitting regression model

```
// create region dummy codes using tab
tab region, gen(region)

//regress csat on region
regress csat region1 region2 region3
```

```
tab region, gen(region)

Geographica |
   l region |      Freq.     Percent        Cum.
------------+-----------------------------------
       West |         13       26.00       26.00
    N. East |          9       18.00       44.00
      South |         16       32.00       76.00
    Midwest |         12       24.00      100.00
------------+-----------------------------------
      Total |         50      100.00


regress csat region1 region2 region3

      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(3, 46)        =      9.61
       Model |  82049.4719         3  27349.824   Prob > F        =    0.0000
    Residual |  130911.908        46  2845.91105   R-squared       =    0.3853
-------------+----------------------------------   Adj R-squared   =    0.3452
       Total |   212961.38        49  4346.15061   Root MSE        =    53.347

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     region1 |  -63.77564   21.35592    -2.99   0.005    -106.7629    -20.7884
     region2 |  -120.5278   23.52385    -5.12   0.000    -167.8788   -73.17672
     region3 |  -80.08333   20.37225    -3.93   0.000    -121.0906   -39.07611
       _cons |   1010.083   15.39998    65.59   0.000     979.0848    1041.082
------------------------------------------------------------------------------
```

- For categorical variables, we first need to dummy code

- Use region as example
    - Option 2: Let Stata do it for you

```
// regress csat on region using fvvarlist syntax
// see help fvvarlist for details
regress csat i.region
```

```
regress csat i.region

      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(3, 46)        =      9.61
       Model |  82049.4719         3  27349.824   Prob > F        =    0.0000
    Residual |  130911.908        46  2845.91105   R-squared       =    0.3853
-------------+----------------------------------   Adj R-squared   =    0.3452
       Total |   212961.38        49  4346.15061   Root MSE        =    53.347

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      region |
    N. East  |  -56.75214   23.13285    -2.45   0.018    -103.3161   -10.18813
      South  |  -16.30769   19.91948    -0.82   0.417    -56.40353    23.78814
    Midwest  |   63.77564   21.35592     2.99   0.005      20.7884    106.7629
             |
       _cons |   946.3077   14.79582    63.96   0.000     916.5253    976.0901
------------------------------------------------------------------------------
```

10.7 Exercise 1

Regression, Categorical Predictors, & Interactions

Open the datafile, states.dta.

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 a categorical variable to your regression (remember, it will need to be dummy coded). You could use region or generate a new categorical variable from one of the continuous variables in the dataset.

10.8 Exporting & saving results

10.8.1 Regression tables

- Usually when we're running regression, we'll be testing multiple models at a time
- Can be difficult to compare results

- Stata offers several user-friendly options for storing and viewing regression output from multiple models
- First, download the necessary packages:

```
// install outreg2 package
findit outreg2
```

10.8.2 Saving & replaying

- You can store regression model results in Stata

```
// fit two regression models and store the results
regress csat expense income percent high
estimates store Model1
regress csat expense income percent high i.region
estimates store Model2
```

```
regress csat expense income percent high

      Source |       SS           df       MS      Number of obs   =        51
-------------+----------------------------------   F(4, 46)        =     51.86
       Model |  183354.603         4  45838.6508   Prob > F        =    0.0000
    Residual |  40659.9067        46  883.911016   R-squared       =    0.8185
-------------+----------------------------------   Adj R-squared   =    0.8027
       Total |   224014.51        50   4480.2902   Root MSE        =    29.731

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     expense |   .0045604    .004384     1.04   0.304    -.0042641     .013385
      income |   .4437858   1.138947     0.39   0.699    -1.848795    2.736367
     percent |  -2.533084   .2454477   -10.32   0.000    -3.027145   -2.039024
        high |   2.086599   .9246023     2.26   0.029     .2254712    3.947727
       _cons |   836.6197   58.33238    14.34   0.000     719.2027    954.0366
------------------------------------------------------------------------------
estimates store Model1
regress csat expense income percent high i.region

      Source |       SS           df       MS      Number of obs   =        50
-------------+----------------------------------   F(7, 42)        =     51.07
       Model |  190570.293         7  27224.3275   Prob > F        =    0.0000
    Residual |  22391.0874        42  533.121128   R-squared       =    0.8949
-------------+----------------------------------   Adj R-squared   =    0.8773
       Total |   212961.38        49  4346.15061   Root MSE        =    23.089

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
```

```
     expense |   -.004375   .0044603    -0.98   0.332    -.0133763    .0046263
      income |   1.306164    .950279     1.37   0.177    -.6115765    3.223905
     percent |  -2.965514   .2496481   -11.88   0.000    -3.469325   -2.461704
        high |   3.544804   1.075863     3.29   0.002     1.373625    5.715983
             |
      region |
    N. East  |   80.81334    15.4341     5.24   0.000     49.66607    111.9606
      South  |   33.61225   13.94521     2.41   0.020     5.469676    61.75483
    Midwest  |   32.15421   10.20145     3.15   0.003     11.56686    52.74157
             |
       _cons |   724.8289   79.25065     9.15   0.000     564.8946    884.7631
------------------------------------------------------------------------------
estimates store Model2
```

- Stored models can be recalled

```
// Display Model1
estimates replay Model1
```

```
estimates replay Model1

-------------------------------------------------------------------------------
Model Model1
-------------------------------------------------------------------------------

      Source |       SS           df       MS      Number of obs   =        51
-------------+----------------------------------   F(4, 46)        =     51.86
       Model |  183354.603         4  45838.6508   Prob > F        =    0.0000
    Residual |  40659.9067        46  883.911016   R-squared       =    0.8185
-------------+----------------------------------   Adj R-squared   =    0.8027
       Total |   224014.51        50  4480.2902    Root MSE        =    29.731

------------------------------------------------------------------------------
        csat |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
     expense |   .0045604    .004384     1.04   0.304    -.0042641     .013385
      income |   .4437858   1.138947     0.39   0.699    -1.848795    2.736367
     percent |  -2.533084   .2454477   -10.32   0.000    -3.027145   -2.039024
        high |   2.086599   .9246023     2.26   0.029     .2254712    3.947727
       _cons |   836.6197   58.33238    14.34   0.000     719.2027    954.0366
------------------------------------------------------------------------------
```

- Stored models can be compared

```
// Compare Model1 and Model2 coefficients
estimates table Model1 Model2
```

```
estimates table Model1 Model2

----------------------------------------
```

```
    Variable |   Model1       Model2
-------------+--------------------------
     expense |  .00456044   -.00437502
      income |  .44378583    1.3061642
     percent | -2.5330843   -2.9655142
        high |  2.0865991    3.5448038
             |
      region |
    N. East  |                80.813342
      South  |                33.612251
    Midwest  |                32.154215
             |
       _cons |  836.61966    724.82886
----------------------------------------
```

10.8.3 Exporting to Excel

- Avoid human error when transferring coefficients into tables
- Excel can be used to format publication-ready tables

```
outreg2 [Model1 Model2] using csatprediction.xls, replace
```

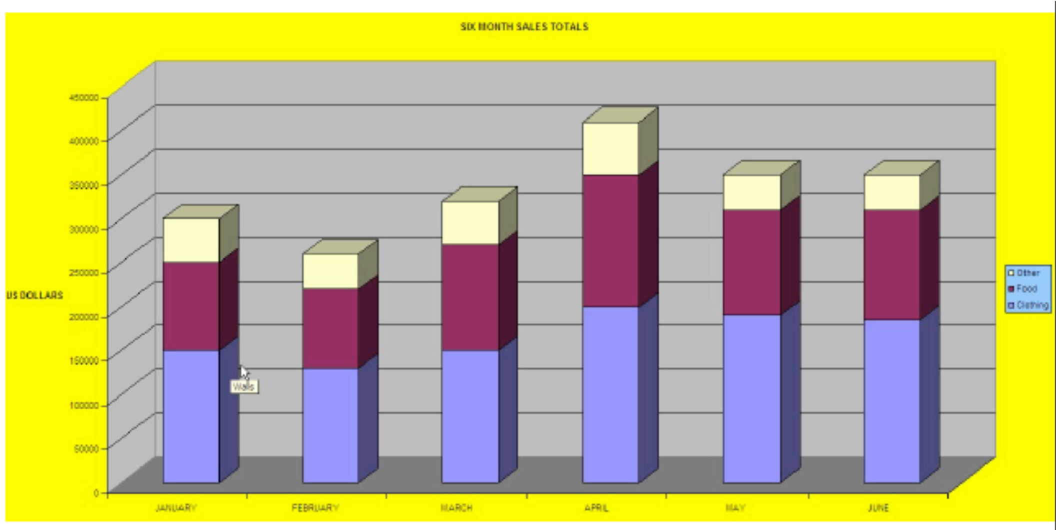
```
outreg2 [Model1 Model2] using csatprediction.xls, replace
~/ado/plus/o/outreg2.ado
csatprediction.xls
dir : seeout
```

10.9 Graphing in Stata

10.9.1 Graphing Strategies

- Keep it simple
- Labels, labels, labels!!
- Avoid cluttered graphs
- Every part of the graph should be meaningful
- Avoid:
  - Shading
  - Distracting colors
  - Decoration
- Always know what you're working with before you get started
  - Recognize scale of data
  - If you're using multiple variables – how do their scales align?
- Before any graphing procedure review variables with `codebook`, `sum`, `tab`, etc.
- HELPFUL STATA HINT: If you want your command to go on multiple lines use `///` at end of each line

10.9.2 Terrible Graph



10.9.3 Much Better Graph

10.10 Univariate Graphics

10.10.1 Our First Dataset

- Time Magazine Public School Poll
  - Based on survey of 1,000 adults in U.S.
  - Conducted in August 2010
  - Questions regarding feelings about parental involvement, teachers union, current potential for reform
- Open Stata and call up the datafile for today

```
// Step 1: tell Stata where to find data:
cd "~/StataGraphics/dataSets"
// Step 2: call up our dataset:
use TimePollPubSchools.dta
```

10.10.2 Single Continuous Variables

Example: Histograms

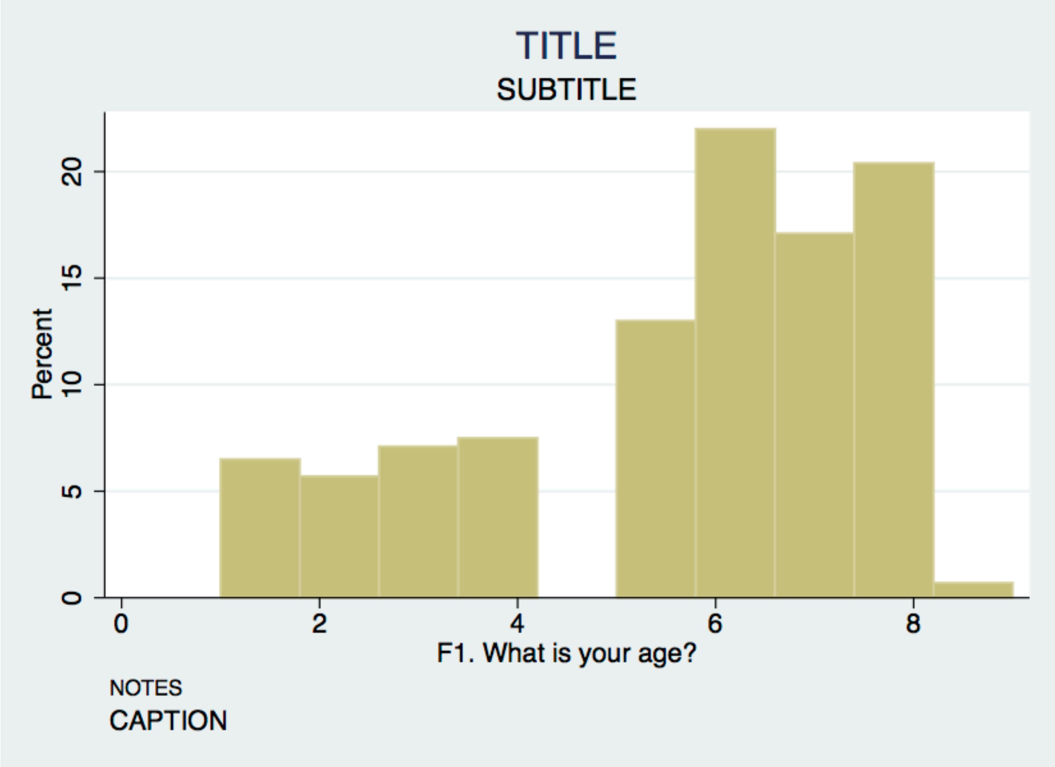
- Stata assumes you're working with continuous data
- Very simple syntax:
  - `hist varname`
- Put a comma after your varname and start adding options
  - `bin(#)` : change the number of bars that the graph displays
  - `normal` : overlay normal curve
  - `addlabels` : add actual values to bars

Histogram Options

- To change the numeric depiction of your data add these options after the comma
  - Choose one: density fraction frequency percent
- Be sure to properly describe your histogram:
  - `title(insert name of graph)`
  - `subtitle(insert subtitle of graph)`
  - `note(insert note to appear at bottom of graph)`
  - `caption(insert caption to appear below notes)`

Histogram Example

```
hist F1, bin(10) percent title(TITLE) ///
  subtitle(SUBTITLE) caption(CAPTION) note(NOTES)
```

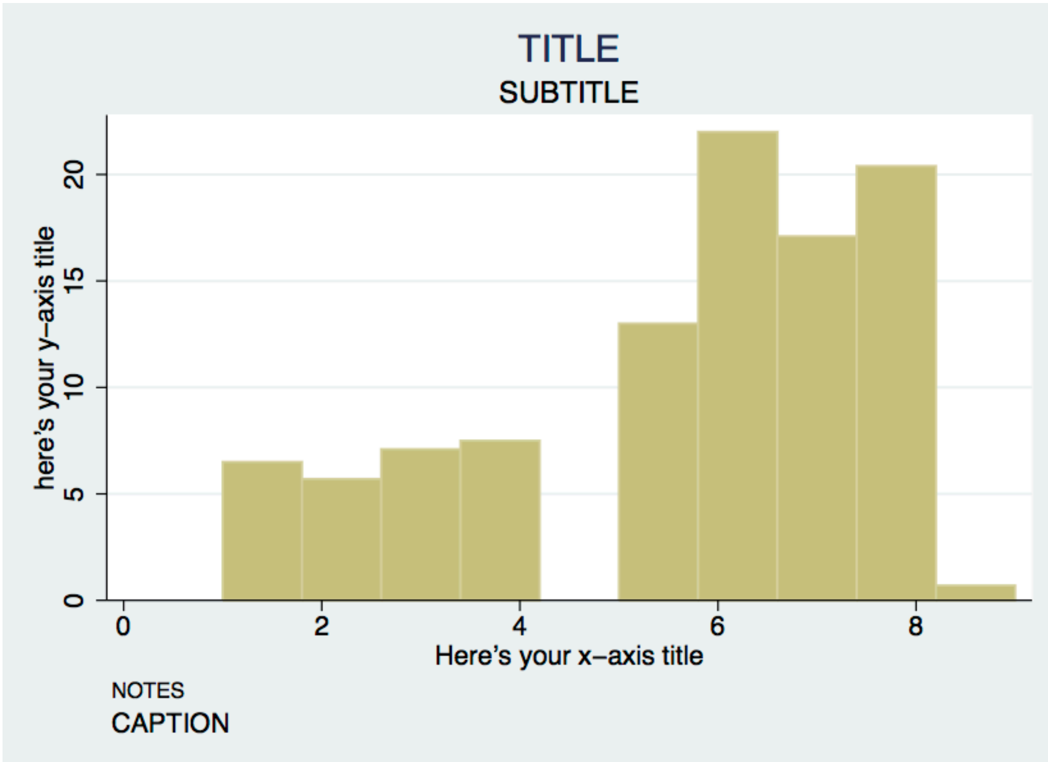


Axis Titles & Labels

- Axis title options (default is variable label):
  - `xtitle(insert x axis name)`
  - `ytitle(insert y axis name)`
- Don't want axis titles?
  - `xtitle("")`
  - `ytitle("")`
- Add labels to X or Y axis:
  - xlabel(insert x axis label)
  - ylabel(insert y axis label)
- Tell Stata how to scale each axis
  - xlabel(start#(increment)end#)
  - xlabel(0(5)100)
- This would label x-axis from 0-100 in increments of 5

Axis Labels Example

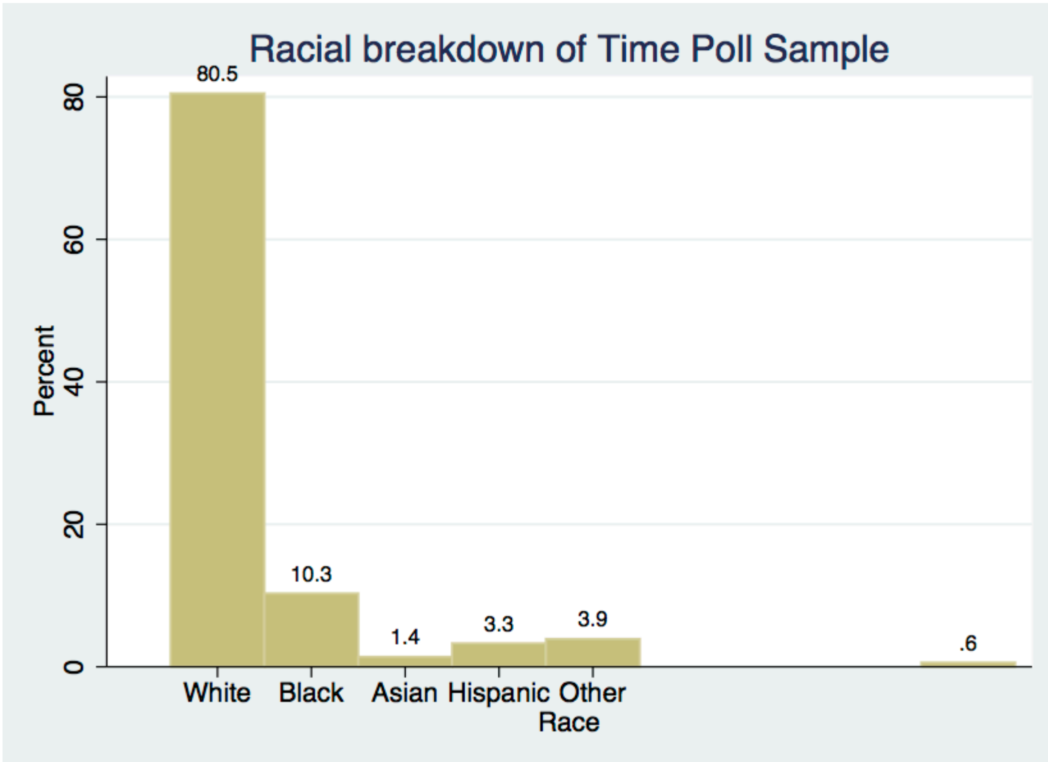
```
hist F1, bin(10) percent title(TITLE) subtitle(SUBTITLE) ///
    caption(CAPTION) note(NOTES) ///
    xtitle(Here's your x-axis title) ///
ytitle(here's your y-axis title)
```



10.10.3 Single Categorical Variables

- We can also use the `hist` command for bar graphs
  - Simply specify "discrete" with options
- Stata will produce one bar for each level (i.e. category) of variable
- Use `xlabel` command to insert names of individual categories

```
hist F4, title(Racial breakdown of Time Poll Sample) xtitle(Race) ///
ytitle(Percent) xlabel(1 "White" 2 "Black" 3 "Asian" 4 "Hispanic" ///
 5 "Other") discrete percent addlabels
```



10.11 Exercise 2

Histograms Bar Graphs

1. Open the datafile, NatNeighCrimeStudy.dta.
2. Create a histogram of the tract-level poverty rate (variable name: `T_POVRTY`).
3. Insert the normal curve over the histogram
4. Change the numeric representation on the Y-axis to "percent"
5. Add appropriate titles to the overall graph and the x axis and y axis. Also, add a note that states the source of this data.
6. Open the datafile, TimePollPubSchools.dta
7. Create a histogram of the question, "What grade would you give your child's school" (variable name: Q11). Be sure to tell Stata that this is a categorical variable.
8. Format this graph so that the axes have proper titles and labels. Also, add an appropriate title to the overall graph that goes onto two lines. Add a note stating the source of the data.

10.12 Bivariate Graphics

10.12.1 Next Dataset:

- National Neighborhood Crime Study (NNCS)
  - N=9,593 census tracts in 2000
  - Explore sources of variation in crime for communities in the United States
  - Tract-level data: crime, social disorganization, disadvantage, socioeconomic inequality
  - City-level data: labor market, socioeconomic inequality, population change

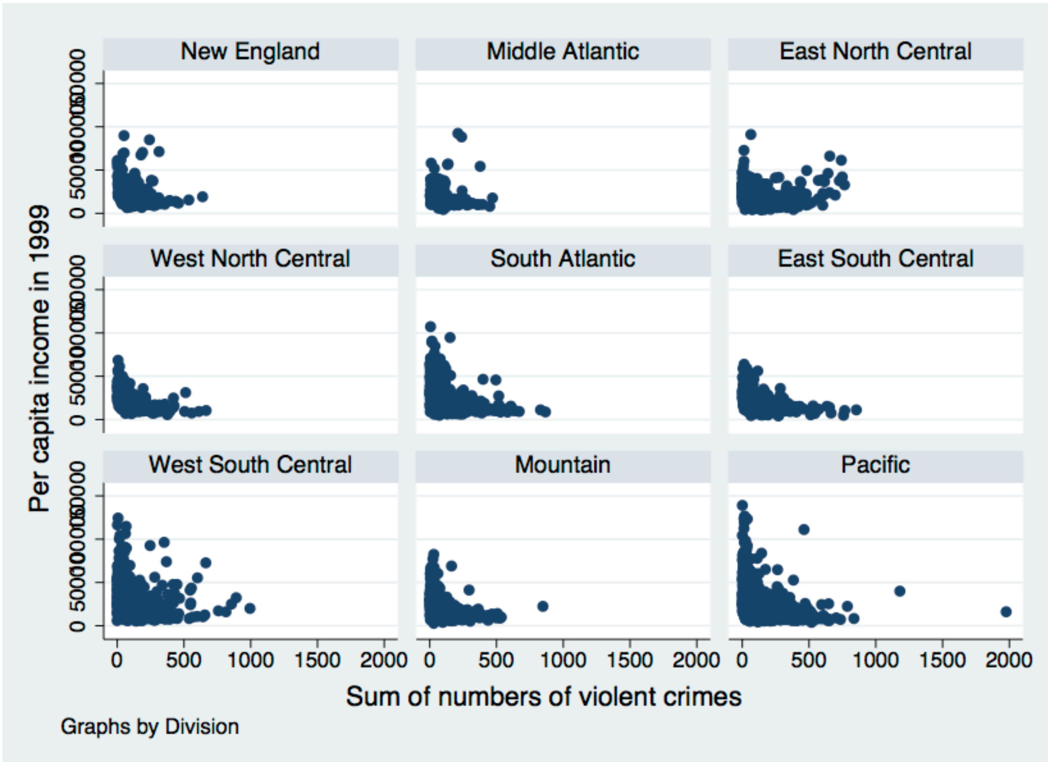
10.12.2 The Twoway Family

- `twoway` is basic Stata command for all twoway graphs
- Use `twoway` anytime you want to make comparisons among variables
- Can be used to combine graphs (i.e., overlay one graph with another
  - e.g., insert line of best fit over a scatter plot
- Some basic examples:

```
use NatNeighCrimeStudy.dta
twoway scatter T_PERCAP T_VIOLNT
twoway dropline T_PERCAP T_VIOLNT
twoway  lfitci T_PERCAP T_VIOLNT
```

Twoway & the `by` Statement

```
twoway scatter T_PERCAP T_VIOLNT, by(DIVISION)
```



Twoway Title Options

- Same title options as with histogram
  - title(insert name of graph)
  - subtitle(insert subtitle of graph)
  - note(insert note to appear at bottom of graph)
  - caption(insert caption to appear below notes)

Twoway Title Options Example

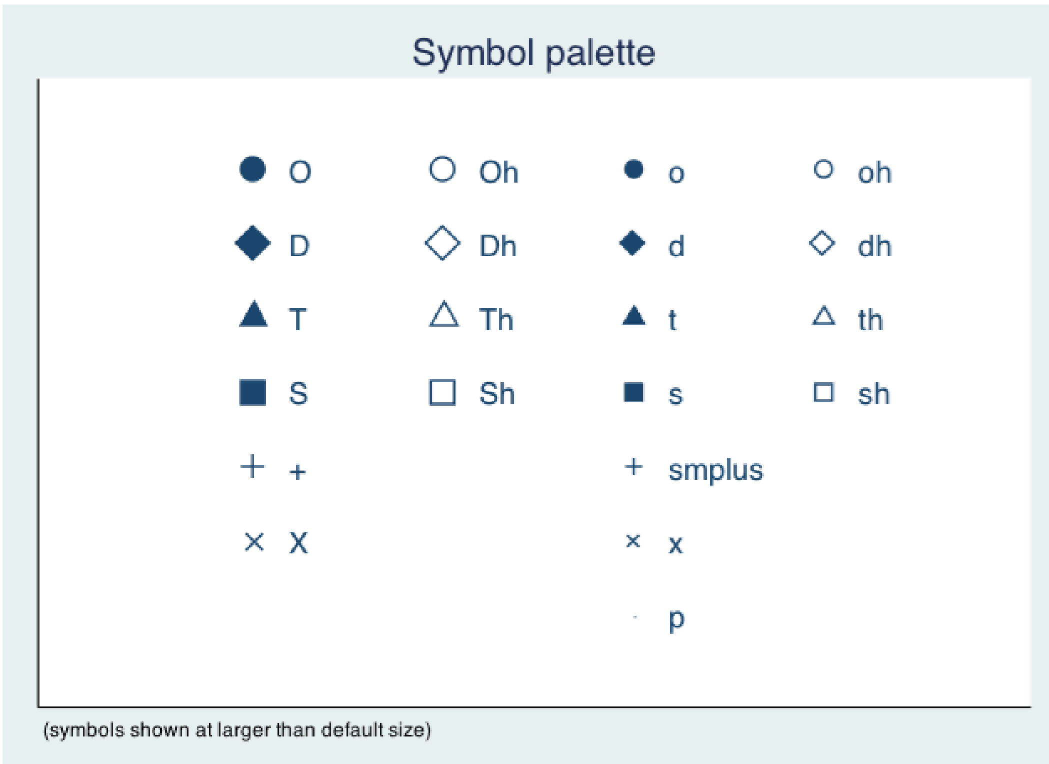
```
twoway scatter T_PERCAP T_VIOLNT, ///
    title(Comparison of Per Capita Income ///
          and Violent Crime Rate at Tract level) ///
xtitle(Violent Crime Rate) ytitle(Per Capita Income) ///
    note(Source: National Neighborhood Crime Study 2000)
```

- The title is a bit cramped–let's fix that:

```
twoway scatter T_PERCAP T_VIOLNT, ///
    title("Comparison of Per Capita Income" ///
"and Violent Crime Rate at Tract level") ///
xtitle(Violent Crime Rate) ytitle(Per Capita Income) ///
note(Source: National Neighborhood Crime Study 2000)
```

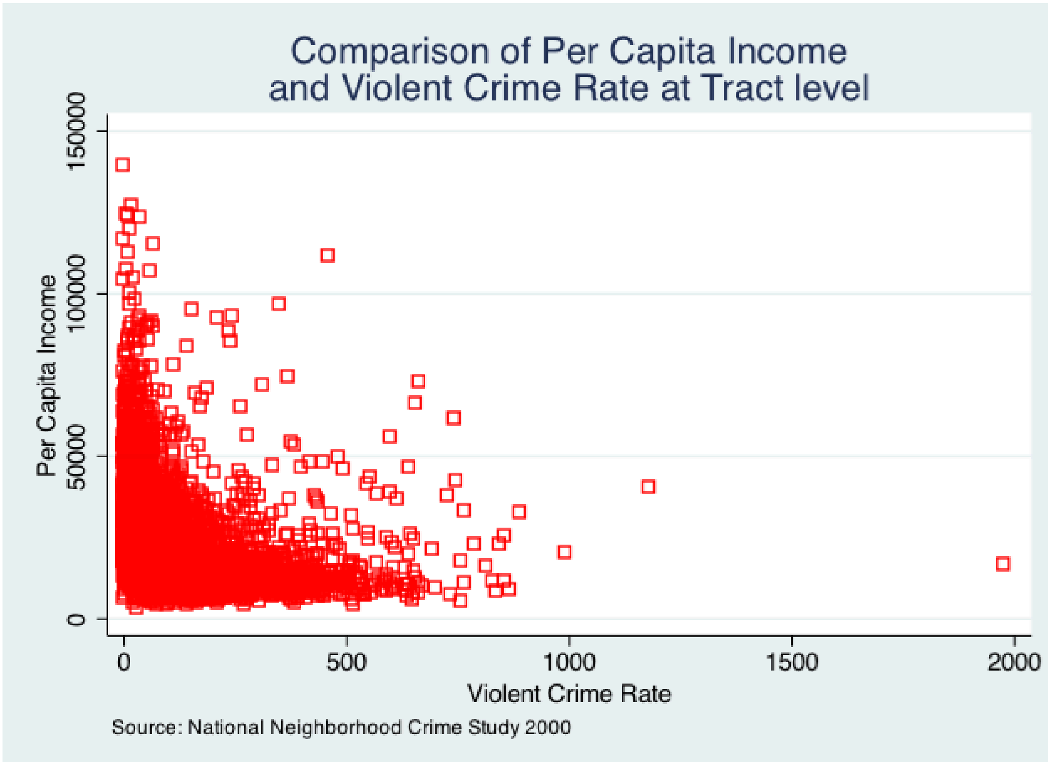
Twoway Symbol Options

- A variety of symbol shapes are available: use `palette symbolpalette` to seem them and `msymbol()` to set them



Twoway Symbol Options

```
twoway scatter T_PERCAP T_VIOLNT, ///
    title("Comparison of Per Capita Income" ///
"and Violent Crime Rate at Tract level") ///
xtitle(Violent Crime Rate) ytitle(Per Capita Income) ///
note(Source: National Neighborhood Crime Study 2000) ///
msymbol(Sh) mcolor("red")
```



10.12.3 Overlaying Twoway Graphs

- Very simple to combine multiple graphs...just put each graph command in parentheses
  - `twoway (scatter var1 var2) (lfit var1 var2)`
- Add individual options to each graph within the parentheses
- Add overall graph options as usual following the comma
  - `twoway (scatter var1 var2) (lfit var1 var2), options`

Overlaying Points & Lines

```
twoway (scatter T_PERCAP T_VIOLNT) ///
    (lfit T_PERCAP T_VIOLNT), ///
    title("Comparison of Per Capita Income" ///
          "and Violent Crime Rate at Tract level") ///
    xtitle(Violent Crime Rate) ytitle(Per Capita Income) ///
    note(Source: National  Neighborhood Crime Study 2000)
```

Overlaying Points & Labels

```
twoway (scatter T_PERCAP T_VIOLNT if T_VIOLNT==1976, ///
        mlabel(CITY)) (scatter T_PERCAP T_VIOLNT), ///
    title("Comparison of Per Capita Income" ///
          "and Violent Crime Rate at Tract level") ///
```

```
    xlabel(0(200)2400) note(Source: National Neighborhood ///
                            Crime Study 2000) legend(off)
```

10.13 Exercise 3

The TwoWay Family

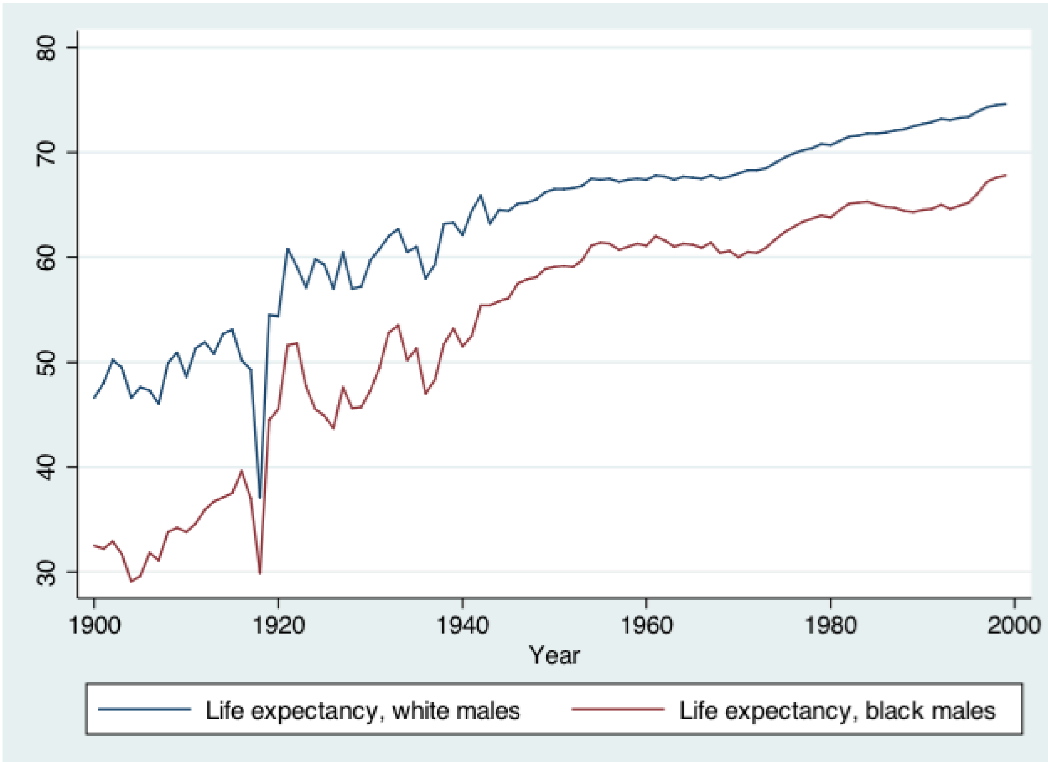
Open the datafile, NatNeighCrimeStudy.dta.

1. Create a basic twoway scatterplot that compares the city unemployment rate (`C_UNEMP`) to the percent secondary sector low-wage jobs (`C_SSLOW`)
2. Generate the same scatterplot, but this time, divide the plot by the dummy variable indicating whether the city is located in the south or not (`C_SOUTH`)
3. Change the color of the symbol that you use in this scatter plot
4. Change the type of symbol you use to a marker of your choice
5. Notice in your scatterplot that is broken down by `C_SOUTH` that there is an outlier in the upper right hand corner of the "Not South" graph. Add the city name label to this marker.
6. Review the options available under "help twowayoptions" and change one aspect of your graph using an option that we haven't already reviewed

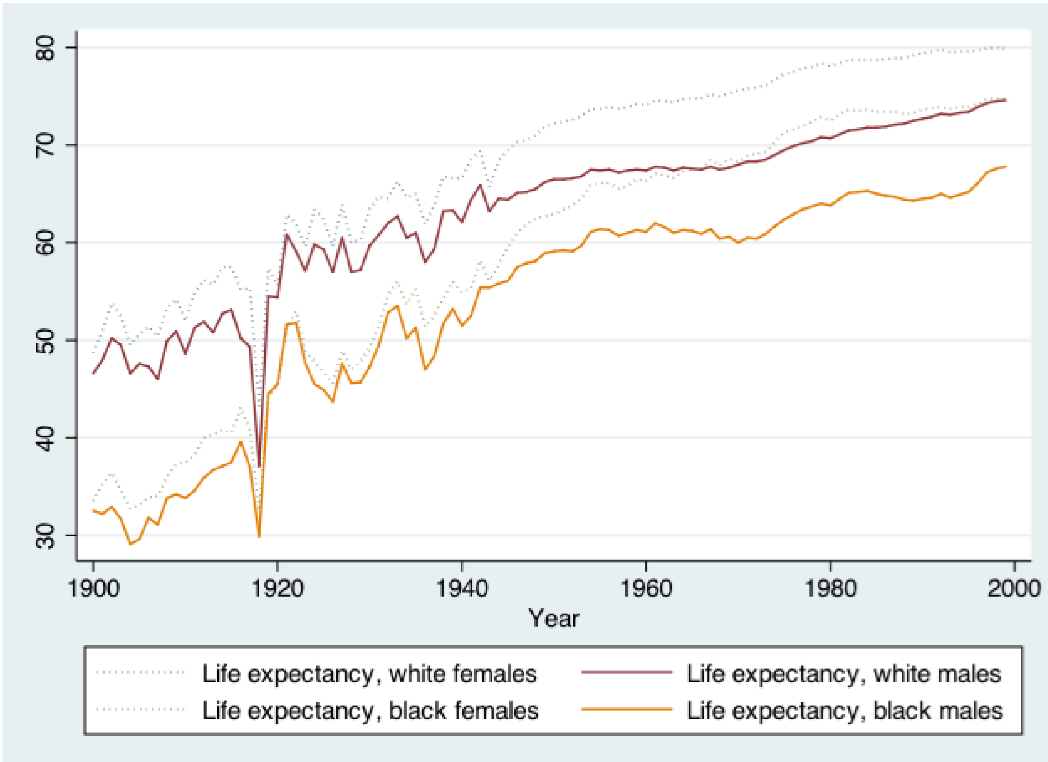
10.14 Twoway Line Graphs

- Line graphs helpful for a variety of data
  - Especially any type of time series data
- We'll use data on US life expectancy from 1900-1999
  - `webuse uslifeexp, clear`

```
webuse uslifeexp, clear
twoway (line le_wm year, mcolor("red")) ///
    (line le_bm year, mcolor("green"))
```

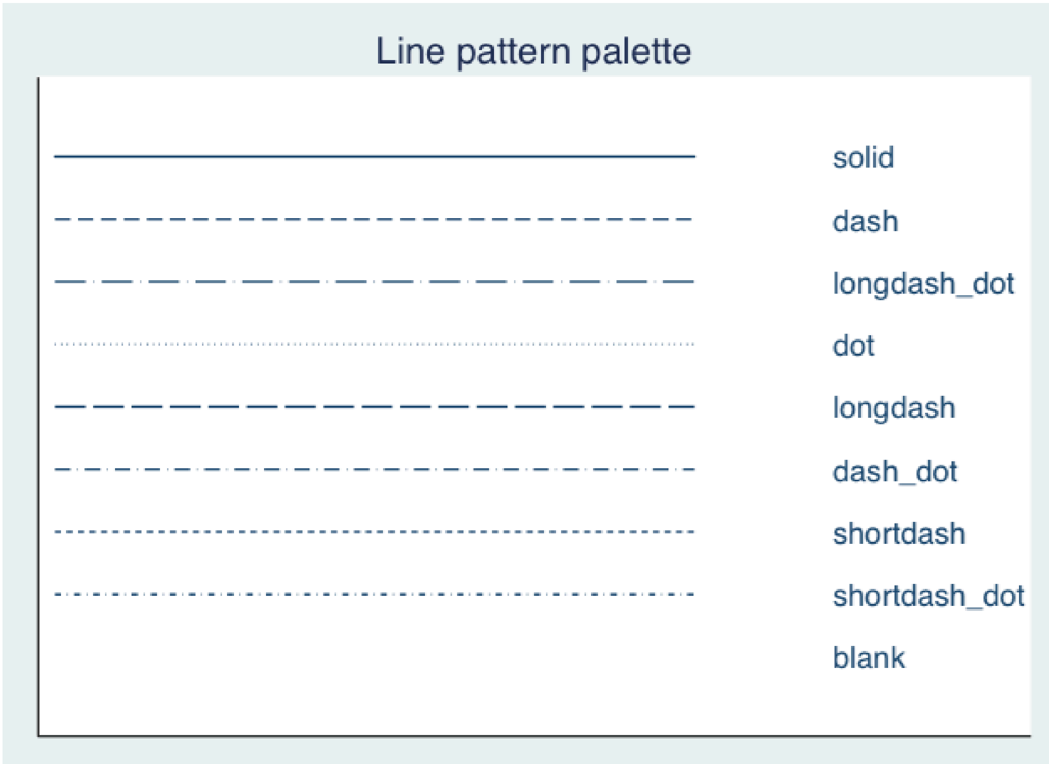



```
twoway (line (le_wfemale le_wmale le_bf le_bm) year, ///
    lpattern(dot solid dot solid))
```



Stata Graphing Lines

```
palette linepalette
```



10.15 Exporting Graphs

- From Stata, right click on image and select "save as" or try syntax:
  - `graph export myfig.esp, replace`
- In Microsoft Word: insert -> picture -> from file
  - Or, right click on graph in Stata and copy and paste into MS Word

10.16 Exercise Solutions

10.16.1 Ex 0: prototype

10.16.2 Ex 1: prototype

10.16.3 Ex 2: prototype

10.16.4 Ex 3: prototype

10.17 Wrap-up

10.17.1 Feedback

These workshops are a work in progress, please provide any feedback to: `help@iq.harvard.edu`

10.17.2 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`
- Stata
  - UCLA website: `http://www.ats.ucla.edu/stat/Stata/`
  - Stata website: `http://www.stata.com/help.cgi?contents`
  - Email list: `http://www.stata.com/statalist/`