Importing, Scraping, and Exporting Data with R

*"What we have is a data glut."* - Vernon Vinge

Data are being generated by everything around us at all times. Every digital process and social media exchange produces it. Systems, sensors and mobile devices transmit it. Countless databases collect it. Data are arriving from multiple sources at an alarming rate and analysts and organizations are seeking ways to leverage these new sources of information. Consequently, analysts need to understand how to *get* data from these data sources. Furthermore, since analysis is often a collaborative effort analysts also need to know how to share their data.

This section covers the process of [importing](#import), [scraping](#scrape), and [exporting](#export) data. First, I cover the basics of importing tabular and spreadsheet data. Second, since modern day data wrangling often includes scraping data from the flood of web-based data becoming available to organizations and analysts, I cover the fundamentals of web-scraping with R. This includes importing spreadsheet data files stored online, scraping HTML text and data tables, and leveraging APIs. Third, although getting data into R is essential, I also cover the equally important process of getting data out of R. Consequently, this section will give you a strong foundation for the different ways to get your data into and out of R.

# Importing Data

The first step to any data analysis process is to *get* the data. Data can come from many sources but two of the most common include text and Excel files. This chapter covers how to import data into R by reading data from common [text files](#csv) and [Excel spreadsheets](#excel). In addition, I cover how to load data from [saved R object files](#robject) for holding or transferring data that has been processed in R. In addition to the the commonly used base R functions to perform data importing, I will also cover functions from the popular [readr](https://cran.rstudio.com/web/packages/readr/), [xlsx](https://cran.rstudio.com/web/packages/xlsx/), and [readxl](https://cran.rstudio.com/web/packages/readxl/) packages.

## Reading data from text files

Text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Text file formats use delimiters to separate the different elements in a line, and each line of data is in its own line in the text file. Therefore, importing different kinds of text files can follow a fairly consistent process once you've identified the delimiter.

There are two main groups of functions that we can use to read in text files:

* [Base R functions](#base_text_import)
* [readr package functions](#readr_text_import)

### Base R functions

read.table() is a multipurpose work-horse function in base R for importing data. The functions read.csv() and read.delim() are special cases of read.table() in which the defaults have been adjusted for efficiency. To illustrate these functions let's work with a CSV file that is saved in our working directory which looks like:

variable 1,variable 2,variable 3  
10,beer,TRUE  
25,wine,TRUE  
8,cheese,FALSE

To read in the CSV file we can use read.csv(). Note that when we assess the structure of the data set that we read in, variable.2 is automatically coerced to a factor variable and variable.3 is automatically coerced to a logical variable. Furthermore, any whitespace in the column names are replaced with a ".".

mydata = read.csv("mydata.csv")  
mydata  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
str(mydata)  
## 'data.frame': 3 obs. of 3 variables:  
## $ variable.1: int 10 25 8  
## $ variable.2: Factor w/ 3 levels "beer","cheese",..: 1 3 2  
## $ variable.3: logi TRUE TRUE FALSE

However, we may want to read in variable.2 as a character variable rather then a factor. We can take care of this by changing the stringsAsFactors argument. The default has stringsAsFactors = TRUE; however, setting it equal to FALSE will read in the variable as a character variable.

mydata\_2 = read.csv("mydata.csv", stringsAsFactors = FALSE)  
mydata\_2  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
str(mydata\_2)  
## 'data.frame': 3 obs. of 3 variables:  
## $ variable.1: int 10 25 8  
## $ variable.2: chr "beer" "wine" "cheese"  
## $ variable.3: logi TRUE TRUE FALSE

As previously stated read.csv is just a wrapper for read.table but with adjusted default arguments. Therefore, we can use read.table to read in this same data. The two arguments we need to be aware of are the field separator (sep) and the argument indicating whether the file contains the names of the variables as its first line (header). In read.table the defaults are sep = "" and header = FALSE whereas in read.csv the defaults are sep = "," and header = TRUE. There are multiple other arguments we can use for certain situations which we illustrate below:

# provides same results as read.csv above  
read.table("mydata.csv", sep=",", header = TRUE, stringsAsFactors = FALSE)  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
# set column and row names  
read.table("mydata.csv", sep=",", header = TRUE, stringsAsFactors = FALSE,  
 col.names = c("Var 1", "Var 2", "Var 3"),  
 row.names = c("Row 1", "Row 2", "Row 3"))  
## Var.1 Var.2 Var.3  
## Row 1 10 beer TRUE  
## Row 2 25 wine TRUE  
## Row 3 8 cheese FALSE  
  
# manually set the classes of the columns   
set\_classes <- read.table("mydata.csv", sep=",", header = TRUE,  
 colClasses = c("numeric", "character", "character"))  
str(set\_classes)  
## 'data.frame': 3 obs. of 3 variables:  
## $ variable.1: num 10 25 8  
## $ variable.2: chr "beer" "wine" "cheese"  
## $ variable.3: chr "TRUE" "TRUE" "FALSE"  
  
# limit the number of rows to read in  
read.table("mydata.csv", sep=",", header = TRUE, nrows = 2)  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE

In addition to CSV files, there are other text files that read.table works with. The primary difference is what separates the elements. For example, tab delimited text files typically end with the .txt extension. You can also use the read.delim() function as, similiar to read.csv(), read.delim() is a wrapper of read.table() with defaults set specifically for tab delimited files.

# reading in tab delimited text files  
read.delim("mydata.txt")  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
# provides same results as read.delim  
read.table("mydata.txt", sep="\t", header = TRUE)  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE

### readr package

Compared to the equivalent base functions, [readr](https://cran.rstudio.com/web/packages/readr/) functions are around 10x faster. They bring consistency to importing functions, they produce data frames in a data.table format which are easier to view for large data sets, the default settings removes the "hassels" of stringsAsFactors, and they have a more flexible column specification.

To illustrate, we can use read\_csv() which is equivalent to base R's read.csv() function. However, note that read\_csv() maintains the full variable name (whereas read.csv eliminates any spaces in variable names and fills it with '.'). Also, read\_csv() automatically sets stringsAsFactors = FALSE, which can be a [controversial topic](http://simplystatistics.org/2015/07/24/stringsasfactors-an-unauthorized-biography/).

library(readr)  
mydata\_3 = read\_csv("mydata.csv")  
mydata\_3  
## variable 1 variable 2 variable 3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
str(mydata\_3)  
## Classes 'tbl\_df', 'tbl' and 'data.frame': 3 obs. of 3 variables:  
## $ variable 1: int 10 25 8  
## $ variable 2: chr "beer" "wine" "cheese"  
## $ variable 3: logi TRUE TRUE FALSE

read\_csv also offers many additional arguments for making adjustments to your data as you read it in:

# specify the column class using col\_types  
read\_csv("mydata.csv", col\_types = list(col\_double(),   
 col\_character(),   
 col\_character()))  
## variable 1 variable 2 variable 3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
# we can also specify column classes with a string  
# in this example d = double, \_ skips column, c = character  
read\_csv("mydata.csv", col\_types = "d\_c")  
## variable 1 variable 3  
## 1 10 TRUE  
## 2 25 TRUE  
## 3 8 FALSE  
  
# set column names  
read\_csv("mydata.csv", col\_names = c("Var 1", "Var 2", "Var 3"), skip = 1)  
## Var 1 Var 2 Var 3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
# set the maximum number of lines to read in  
read\_csv("mydata.csv", n\_max = 2)  
## variable 1 variable 2 variable 3  
## 1 10 beer TRUE  
## 2 25 wine TRUE

Similar to base R, readr also offers functions to import .txt files (read\_delim()), fixed-width files (read\_fwf()), general text files (read\_table()), and more.

These examples provide the basics for reading in text files. However, sometimes even text files can offer unanticipated difficulties with their formatting. Both the base R and readr functions offer many arguments to deal with different formatting issues and I suggest you take time to look at the help files for these functions to learn more (i.e. ?read.table). Also, you will find [more resources at the end of this chapter](#importing_resources) for importing files.

## Reading data from Excel files

With Excel still being the spreadsheet software of choice its important to be able to efficiently import and export data from these files. Often, R users will simply resort to exporting the Excel file as a CSV file and then import into R using read.csv; however, this is far from efficient. This section will teach you how to eliminate the CSV step and to import data directly from Excel using two different packages:

* [xlsx package](#xlsx_import)
* [readxl package](#readxl_import)

Note that there are several packages available to connect R with Excel (i.e. gdata, RODBC, XLConnect, RExcel, etc.); however, I am only going to cover the two main packages that I use which provide all the fundamental requirements I've needed for dealing with Excel.

### xlsx package

The [xlsx](https://cran.rstudio.com/web/packages/xlsx/) package provides tools neccessary to interact with Excel 2007 (and older) files from R. Many of the benefits of the xlsx come from being able to *export* and *format* Excel files from R. Some of these capabilities will be covered in the [Exporting Data](#export) chapter; however, in this section we will simply cover *importing* data from Excel with the xlsx package.

To illustrate, we'll use similar data from the [previous section](#base_text_import); however, saved as an .xlsx file in our working director. To import the Excel data we simply use the read.xlsx() function:

library(xlsx)  
  
# read in first worksheet using a sheet index or name  
read.xlsx("mydata.xlsx", sheetName = "Sheet1")  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
read.xlsx("mydata.xlsx", sheetIndex = 1)  
## variable.1 variable.2 variable.3  
## 1 10 beer TRUE  
## 2 25 wine TRUE  
## 3 8 cheese FALSE  
  
# read in second worksheet  
read.xlsx("mydata.xlsx", sheetName = "Sheet2")  
## variable.4 variable.5  
## 1 Dayton johnny  
## 2 Columbus amber  
## 3 Cleveland tony  
## 4 Cincinnati alice

Since Excel is such a flexible spreadsheet software, people often make notes, comments, headers, etc. at the beginning or end of the files which we may not want to include. If we want to read in data that starts further down in the Excel worksheet we can include the startRow argument. If we have a specific range of rows (or columns) to include we can use the rowIndex (or colIndex) argument.

# a worksheet with comments in the first two lines  
read.xlsx("mydata.xlsx", sheetName = "Sheet3")  
## HEADER..COMPANY.A NA.  
## 1 What if we want to disregard header text in Excel file? <NA>  
## 2 variable 6 variable 7  
## 3 200 Male  
## 4 225 Female  
## 5 400 Female  
## 6 310 Male  
  
# read in all data below the second line  
read.xlsx("mydata.xlsx", sheetName = "Sheet3", startRow = 3)  
## variable.6 variable.7  
## 1 200 Male  
## 2 225 Female  
## 3 400 Female  
## 4 310 Male  
  
# read in a range of rows  
read.xlsx("mydata.xlsx", sheetName = "Sheet3", rowIndex = 3:5)  
## variable.6 variable.7  
## 1 200 Male  
## 2 225 Female

We can also change the class type of the columns when we read them in:

# read in data without changing class type  
mydata\_sheet1.1 <- read.xlsx("mydata.xlsx", sheetName = "Sheet1")  
  
str(mydata\_sheet1.1)  
## 'data.frame': 3 obs. of 3 variables:  
## $ variable.1: num 10 25 8  
## $ variable.2: Factor w/ 3 levels "beer","cheese",..: 1 3 2  
## $ variable.3: logi TRUE TRUE FALSE  
  
# read in data and change class type  
mydata\_sheet1.2 <- read.xlsx("mydata.xlsx", sheetName = "Sheet1",  
 stringsAsFactors = FALSE,  
 colClasses = c("double", "character", "logical"))  
  
str(mydata\_sheet1.2)  
## 'data.frame': 3 obs. of 3 variables:  
## $ variable.1: num 10 25 8  
## $ variable.2: chr "beer" "wine" "cheese"  
## $ variable.3: logi TRUE TRUE FALSE

Another useful argument is keepFormulas which allows you to see the text of any formulas in the Excel spreadsheet:

# by default keepFormula is set to FALSE so only  
# the formula output will be read in  
read.xlsx("mydata.xlsx", sheetName = "Sheet4")  
## Future.Value Rate Periods Present.Value  
## 1 500 0.065 10 266.3630  
## 2 600 0.085 6 367.7671  
## 3 750 0.080 11 321.6621  
## 4 1000 0.070 16 338.7346  
  
# changing the keepFormula to TRUE will display the equations  
read.xlsx("mydata.xlsx", sheetName = "Sheet4", keepFormulas = TRUE)  
## Future.Value Rate Periods Present.Value  
## 1 500 0.065 10 A2/(1+B2)^C2  
## 2 600 0.085 6 A3/(1+B3)^C3  
## 3 750 0.080 11 A4/(1+B4)^C4  
## 4 1000 0.070 16 A5/(1+B5)^C5

### readxl package

[readxl](https://cran.rstudio.com/web/packages/readxl/) is one of the newest packages for accessing Excel data with R and was developed by [Hadley Wickham](https://twitter.com/hadleywickham) and the [RStudio](https://www.rstudio.com/) team who also developed the readr package. This package works with both legacy .xls formats and the modern xml-based .xlsx format. Similar to readr the readxl functions are based on a C++ library so they are extremely fast. Unlike most other packages that deal with Excel, readxl has no external dependencies, so you can use it to read Excel data on just about any platform. Additional benefits readxl provides includes the ability to load dates and times as POSIXct formatted dates, automatically drops blank columns, and returns outputs as data.table formatted which provides easier viewing for large data sets.

To read in Excel data with readxl you use the read\_excel() function which has very similar operations and arguments as xlsx. A few important differences you will see below include: readxl will automatically convert date and date-time variables to POSIXct formatted variables, character variables will not be coerced to factors, and logical variables will be read in as integers.

library(readxl)  
  
mydata <- read\_excel("mydata.xlsx", sheet = "Sheet5")  
mydata  
## variable 1 variable 2 variable 3 variable 4 variable 5  
## 1 10 beer 1 2015-11-20 2015-11-20 13:30:00  
## 2 25 wine 1 <NA> 2015-11-21 16:30:00  
## 3 8 <NA> 0 2015-11-22 2015-11-22 14:45:00  
  
str(mydata)  
## Classes 'tbl\_df', 'tbl' and 'data.frame': 3 obs. of 5 variables:  
## $ variable 1: num 10 25 8  
## $ variable 2: chr "beer" "wine" NA  
## $ variable 3: num 1 1 0  
## $ variable 4: POSIXct, format: "2015-11-20" NA ...  
## $ variable 5: POSIXct, format: "2015-11-20 13:30:00" "2015-11-21 16:30:00" ...

The available arguments allow you to change the data as you import it. Some examples are provided:

# change variable names by skipping the first row  
# and using col\_names to set the new names  
read\_excel("mydata.xlsx", sheet = "Sheet5", skip = 1,   
 col\_names = paste("Var", 1:5))  
## Var 1 Var 2 Var 3 Var 4 Var 5  
## 1 10 beer 1 42328 2015-11-20 13:30:00  
## 2 25 wine 1 NA 2015-11-21 16:30:00  
## 3 8 <NA> 0 42330 2015-11-22 14:45:00  
  
# sometimes missing values are set as a sentinel value  
# rather than just left blank - (i.e. "999")  
read\_excel("mydata.xlsx", sheet = "Sheet6")  
## variable 1 variable 2 variable 3 variable 4  
## 1 10 beer 1 42328  
## 2 25 wine 1 999  
## 3 8 999 0 42330  
  
# we can change these to missing values with na argument  
read\_excel("mydata.xlsx", sheet = "Sheet6", na = "999")  
## variable 1 variable 2 variable 3 variable 4  
## 1 10 beer 1 42328  
## 2 25 wine 1 NA  
## 3 8 <NA> 0 42330

One unique difference between readxl and xlsx is how to deal with column types. Whereas read.xlsx() allows you to change the column types to integer, double, numeric, character, or logical; read\_excel() restricts you to changing column types to blank, numeric, date, or text. The "blank" option allows you to skip columns; however, to change variable 3 to a logical TRUE/FALSE variable requires a second step.

mydata\_ex <- read\_excel("mydata.xlsx", sheet = "Sheet5",  
 col\_types = c("numeric", "blank", "numeric",   
 "date", "blank"))  
mydata\_ex  
## variable 1 variable 3 variable 4  
## 1 10 1 2015-11-20  
## 2 25 1 <NA>  
## 3 8 0 2015-11-22  
  
# change variable 3 to a logical variable  
mydata\_ex$`variable 3` <- as.logical(mydata\_ex$`variable 3`)  
mydata\_ex  
## variable 1 variable 3 variable 4  
## 1 10 TRUE 2015-11-20  
## 2 25 TRUE <NA>  
## 3 8 FALSE 2015-11-22

## Load data from saved R object file

Sometimes you may need to save data or other R objects outside of your workspace. You may want to share R data/objects with co-workers, transfer between projects or computers, or simply archive them. There are three primary ways that people tend to save R data/objects: as .RData, .rda, or as .rds files. The differences behind when you use each will be covered in the [Saving data as an R object file](#save_object) section. This section will simply shows how to load these data/object forms.

load("mydata.RData")  
  
load(file = "mydata.rda")  
  
name <- readRDS("mydata.rds")

## Additional resources

In addition to text and Excel files, there are multiple other ways that data are stored and exchanged. Commercial statistical software such as SPSS, SAS, Stata, and Minitab often have the option to store data in a specific format for that software. In addition, analysts commonly use databases to store large quantities of data. R has good support to work with these additional options which we did not cover here. The following provides a list of additional resources to learn about data importing for these specific cases:

* [R data import/export manual](https://cran.r-project.org/doc/manuals/R-data.html)
* [Working with databases](https://cran.r-project.org/doc/manuals/R-data.html#Relational-databases)
  + [MySQL](https://cran.r-project.org/web/packages/RMySQL/index.html)
  + [Oracle](https://cran.r-project.org/web/packages/ROracle/index.html)
  + [PostgreSQL](https://cran.r-project.org/web/packages/RPostgreSQL/index.html)
  + [SQLite](https://cran.r-project.org/web/packages/RSQLite/index.html)
  + [Open Database Connectivity databases](https://cran.rstudio.com/web/packages/RODBC/)
* [Importing data from commercial software](https://cran.r-project.org/doc/manuals/R-data.html#Importing-from-other-statistical-systems)
  + The [foreign](http://www.rdocumentation.org/packages/foreign) package provides functions that help you load data files from other programs such as [SPSS](http://www.r-bloggers.com/how-to-open-an-spss-file-into-r/), [SAS](http://rconvert.com/sas-vs-r-code-compare/5-ways-to-convert-sas-data-to-r/), [Stata](http://www.r-bloggers.com/how-to-read-and-write-stata-data-dta-files-into-r/), and others into R.

# Scraping Data

Rapid growth of the World Wide Web has significantly changed the way we share, collect, and publish data. Vast amount of information is being stored online, both in structured and unstructured forms. Regarding certain questions or research topics, this has resulted in a new problem - no longer is the concern of data scarcity and inaccessibility but, rather, one of overcoming the tangled masses of online data.

Collecting data from the web is not an easy process as there are many technologies used to distribute web content (i.e. [HTML](https://en.wikipedia.org/wiki/HTML), [XML](https://en.wikipedia.org/wiki/XML), [JSON](https://en.wikipedia.org/wiki/JSON)). Therefore, dealing with more advanced web scraping requires familiarity in accessing data stored in these technologies via R. Through this chapter I will provide an introduction to some of the fundamental tools required to perform basic web scraping. This includes [importing spreadsheet data files stored online](#importing_spreadsheet_data), [scraping HTML text](#scraping_HTML_text), [scraping HTML table data](#scraping_HTML_tables), and [leveraging APIs to scrape data](#scraping_api).

My purpose in the following sections is to discuss these topics at a level meant to get you started in web scraping; however, this area is vast and complex and this chapter will far from provide you expertise level insight. To advance your knowledge I highly recommend getting copies of [*XML and Web Technologies for Data Sciences with R*](http://www.amazon.com/XML-Web-Technologies-Data-Sciences/dp/1461478995) and [*Automated Data Collection with R*](http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX).

## Importing tabular and Excel files stored online

The most basic form of getting data from online is to import tabular (i.e. .txt, .csv) or Excel files that are being hosted online. This is often not considered *web scraping*[[1]](#footnote-55); however, I think its a good place to start introducing the user to interacting with the web for obtaining data. Importing tabular data is especially common for the many types of government data available online. A quick perusal of [Data.gov](https://www.data.gov/) illustrates nearly 188,510 examples. In fact, we can provide our first example of importing online tabular data by downloading the Data.gov CSV file that lists all the federal agencies that supply data to Data.gov.

# the url for the online CSV  
url <- "https://www.data.gov/media/federal-agency-participation.csv"  
  
# use read.csv to import  
data\_gov <- read.csv(url, stringsAsFactors = FALSE)  
  
# for brevity I only display first 6 rows  
data\_gov[1:6, c(1,3:4)]  
## Agency.Name Datasets Last.Entry  
## 1 Commodity Futures Trading Commission 3 01/12/2014  
## 2 Consumer Financial Protection Bureau 2 09/26/2015  
## 3 Consumer Financial Protection Bureau 2 09/26/2015  
## 4 Corporation for National and Community Service 3 01/12/2014  
## 5 Court Services and Offender Supervision Agency 1 01/12/2014  
## 6 Department of Agriculture 698 12/01/2015

Downloading Excel spreadsheets hosted online can be performed just as easily. Recall that there is not a base R function for importing Excel data; however, several packages exist to handle this capability. One package that works smoothly with pulling Excel data from urls is [gdata](https://cran.r-project.org/web/packages/gdata/index.html). With gdata we can use read.xls() to download this [Fair Market Rents for Section 8 Housing](http://catalog.data.gov/dataset/fair-market-rents-for-the-section-8-housing-assistance-payments-program) Excel file from the given url.

library(gdata)  
  
# the url for the online Excel file  
url <- "http://www.huduser.org/portal/datasets/fmr/fmr2015f/FY2015F\_4050\_Final.xls"  
  
# use read.xls to import  
rents <- read.xls(url)  
  
rents[1:6, 1:10]  
## fips2000 fips2010 fmr2 fmr0 fmr1 fmr3 fmr4 county State CouSub  
## 1 100199999 100199999 788 628 663 1084 1288 1 1 99999  
## 2 100399999 100399999 762 494 643 1123 1318 3 1 99999  
## 3 100599999 100599999 670 492 495 834 895 5 1 99999  
## 4 100799999 100799999 773 545 652 1015 1142 7 1 99999  
## 5 100999999 100999999 773 545 652 1015 1142 9 1 99999  
## 6 101199999 101199999 599 481 505 791 1061 11 1 99999

Note that many of the arguments covered in the [Importing Data chapter](#excel) (i.e. specifying sheets to read from, skipping lines) also apply to read.xls(). In addition, gdata provides some useful functions (sheetCount() and sheetNames()) for identifying if multiple sheets exist prior to downloading.

Another common form of file storage is using zip files. For instance, the [Bureau of Labor Statistics](http://www.bls.gov/home.htm) (BLS) stores their [public-use microdata](http://www.bls.gov/cex/pumdhome.htm) for the [Consumer Expenditure Survey](http://www.bls.gov/cex/home.htm) in .zip files. We can use download.file() to download the file to your working directory and then work with this data as desired.

url <- "http://www.bls.gov/cex/pumd/data/comma/diary14.zip"  
  
# download .zip file and unzip contents  
download.file(url, dest="dataset.zip", mode="wb")   
unzip ("dataset.zip", exdir = "./")  
  
# assess the files contained in the .zip file which  
# unzips as a folder named "diary14"  
list.files("diary14")  
## [1] "dtbd141.csv" "dtbd142.csv" "dtbd143.csv" "dtbd144.csv" "dtid141.csv"  
## [6] "dtid142.csv" "dtid143.csv" "dtid144.csv" "expd141.csv" "expd142.csv"  
## [11] "expd143.csv" "expd144.csv" "fmld141.csv" "fmld142.csv" "fmld143.csv"  
## [16] "fmld144.csv" "memd141.csv" "memd142.csv" "memd143.csv" "memd144.csv"  
  
# alternatively, if we know the file we want prior to unzipping  
# we can extract the file without unzipping using unz():  
zip\_data <- read.csv(unz("dataset.zip", "diary14/expd141.csv"))  
zip\_data[1:5, 1:10]  
## NEWID ALLOC COST GIFT PUB\_FLAG UCC EXPNSQDY EXPN\_QDY EXPNWKDY EXPN\_KDY  
## 1 2825371 0 6.26 2 2 190112 1 D 3 D  
## 2 2825371 0 1.20 2 2 190322 1 D 3 D  
## 3 2825381 0 0.98 2 2 20510 3 D 2 D  
## 4 2825381 0 0.98 2 2 20510 3 D 2 D  
## 5 2825381 0 2.50 2 2 20510 3 D 2 D

The .zip archive file format is meant to compress files and are typically used on files of significant size. For instance, the Consumer Expenditure Survey data we downloaded in the previous example is over 10MB. Obviously there may be times in which we want to get specific data in the .zip file to analyze but not always permanently store the entire .zip file contents. In these instances we can use the following [process](http://stackoverflow.com/questions/3053833/using-r-to-download-zipped-data-file-extract-and-import-data) proposed by [Dirk Eddelbuettel](https://twitter.com/eddelbuettel) to temporarily download the .zip file, extract the desired data, and then discard the .zip file.

# Create a temp. file name  
temp <- tempfile()  
  
# Use download.file() to fetch the file into the temp. file  
download.file("http://www.bls.gov/cex/pumd/data/comma/diary14.zip",temp)  
  
# Use unz() to extract the target file from temp. file  
zip\_data2 <- read.csv(unz(temp, "diary14/expd141.csv"))  
  
# Remove the temp file via unlink()  
unlink(temp)  
  
zip\_data2[1:5, 1:10]  
## NEWID ALLOC COST GIFT PUB\_FLAG UCC EXPNSQDY EXPN\_QDY EXPNWKDY EXPN\_KDY  
## 1 2825371 0 6.26 2 2 190112 1 D 3 D  
## 2 2825371 0 1.20 2 2 190322 1 D 3 D  
## 3 2825381 0 0.98 2 2 20510 3 D 2 D  
## 4 2825381 0 0.98 2 2 20510 3 D 2 D  
## 5 2825381 0 2.50 2 2 20510 3 D 2 D

One last common scenario I'll cover when importing spreadsheet data from online is when we identify multiple data sets that we'd like to download but are not centrally stored in a .zip format or the like. As a simple example lets look at the [average consumer price data](http://www.bls.gov/data/#prices) from the BLS. The BLS holds multiple data sets for different types of commodities within one [url](http://download.bls.gov/pub/time.series/ap/); however, there are separate links for each individual data set. More complicated cases of this will have the links to tabular data sets scattered throughout a webpage[[2]](#footnote-67). The [XML](https://cran.r-project.org/web/packages/XML/index.html) package provides the useful getHTMLLinks() function to identify these links.

library(XML)  
  
# url hosting multiple links to data sets  
url <- "http://download.bls.gov/pub/time.series/ap/"  
  
# identify the links available  
links <- getHTMLLinks(url)  
  
links  
## [1] "/pub/time.series/"   
## [2] "/pub/time.series/ap/ap.area"   
## [3] "/pub/time.series/ap/ap.contacts"   
## [4] "/pub/time.series/ap/ap.data.0.Current"   
## [5] "/pub/time.series/ap/ap.data.1.HouseholdFuels"  
## [6] "/pub/time.series/ap/ap.data.2.Gasoline"   
## [7] "/pub/time.series/ap/ap.data.3.Food"   
## [8] "/pub/time.series/ap/ap.footnote"   
## [9] "/pub/time.series/ap/ap.item"   
## [10] "/pub/time.series/ap/ap.period"   
## [11] "/pub/time.series/ap/ap.series"   
## [12] "/pub/time.series/ap/ap.txt"

This allows us to assess which files exist that may be of interest. In this case the links that we are primarily interested in are the ones that contain "data" in their name (links 4-7 listed above). We can use the [stringr](https://cran.r-project.org/web/packages/stringr/index.html) package to extract these desired links which we will use to download the data.

library(stringr)  
  
# extract names for desired links and paste to url  
links\_data <- links[str\_detect(links, "data")]  
  
# paste url to data links to have full url for data sets  
# use str\_sub and regexpr to paste links at appropriate   
# starting point  
filenames <- paste0(url, str\_sub(links\_data,   
 start = regexpr("ap.data", links\_data)))  
  
filenames  
## [1] "http://download.bls.gov/pub/time.series/ap/ap.data.0.Current"   
## [2] "http://download.bls.gov/pub/time.series/ap/ap.data.1.HouseholdFuels"  
## [3] "http://download.bls.gov/pub/time.series/ap/ap.data.2.Gasoline"   
## [4] "http://download.bls.gov/pub/time.series/ap/ap.data.3.Food"

We can now proceed to develop a simple for loop function (which you will learn about in the [loop control statements chapter](#control_structures)) to download each data set. We store the results in a list which contains 4 items, one item for each data set. Each list item contains the url in which the data was extracted from and the dataframe containing the downloaded data. We're now ready to analyze these data sets as necessary.

# create empty list to dump data into  
data\_ls <- list()  
  
for(i in 1:length(filenames)){  
 url <- filenames[i]  
 data <- read.delim(url)  
 data\_ls[[length(data\_ls) + 1]] <- list(url = filenames[i], data = data)  
}  
  
str(data\_ls)  
## List of 4  
## $ :List of 2  
## ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.0.Current"  
## ..$ data:'data.frame': 144712 obs. of 5 variables:  
## .. ..$ series\_id : Factor w/ 878 levels "APU0000701111 ",..: 1 1 ...  
## .. ..$ year : int [1:144712] 1995 1995 1995 1995 1995 1995 ...  
## .. ..$ period : Factor w/ 12 levels "M01","M02","M03",..: 1 2 3 4 ...  
## .. ..$ value : num [1:144712] 0.238 0.242 0.242 0.236 0.244 ...  
## .. ..$ footnote\_codes: logi [1:144712] NA NA NA NA NA NA ...  
## $ :List of 2  
## ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.1.Hou..."  
## ..$ data:'data.frame': 90339 obs. of 5 variables:  
## .. ..$ series\_id : Factor w/ 343 levels "APU000072511 ",..: 1 1 ...  
## .. ..$ year : int [1:90339] 1978 1978 1979 1979 1979 1979 1979 ...  
## .. ..$ period : Factor w/ 12 levels "M01","M02","M03",..: 11 12 ...  
## .. ..$ value : num [1:90339] 0.533 0.545 0.555 0.577 0.605 0.627 ...  
## .. ..$ footnote\_codes: logi [1:90339] NA NA NA NA NA NA ...  
## $ :List of 2  
## ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.2.Gas..."  
## ..$ data:'data.frame': 69357 obs. of 5 variables:  
## .. ..$ series\_id : Factor w/ 341 levels "APU000074712 ",..: 1 1 ...  
## .. ..$ year : int [1:69357] 1973 1973 1973 1974 1974 1974 1974 ...  
## .. ..$ period : Factor w/ 12 levels "M01","M02","M03",..: 10 11 ...  
## .. ..$ value : num [1:69357] 0.402 0.418 0.437 0.465 0.491 0.528 ...  
## .. ..$ footnote\_codes: logi [1:69357] NA NA NA NA NA NA ...  
## $ :List of 2  
## ..$ url : chr "http://download.bls.gov/pub/time.series/ap/ap.data.3.Food"  
## ..$ data:'data.frame': 122302 obs. of 5 variables:  
## .. ..$ series\_id : Factor w/ 648 levels "APU0000701111 ",..: 1 1 ...  
## .. ..$ year : int [1:122302] 1980 1980 1980 1980 1980 1980 1980 ...  
## .. ..$ period : Factor w/ 12 levels "M01","M02","M03",..: 1 2 3 4 ...  
## .. ..$ value : num [1:122302] 0.203 0.205 0.211 0.206 0.207 0.21 ...  
## .. ..$ footnote\_codes: logi [1:122302] NA NA NA NA NA NA ...

These examples provide the basics required for downloading most tabular and Excel files from online. However, this is just the beginning of importing/scraping data from the web. Next, we'll start exploring the more conventional forms scraping text and data stored in HTML webpages.

## Scraping HTML text

Vast amount of information exists across the interminable webpages that exist online. Much of this information are "unstructured" text that may be useful in our analyses. This section covers the basics of scraping these texts from online sources. Throughout this section I will illustrate how to extract different text components of webpages by dissecting the [Wikipedia page on web scraping](https://en.wikipedia.org/wiki/Web_scraping). However, its important to first cover one of the basic components of HTML elements as we will leverage this information to pull desired information. I offer only enough insight required to begin scraping; I highly recommend [*XML and Web Technologies for Data Sciences with R*](http://www.amazon.com/XML-Web-Technologies-Data-Sciences/dp/1461478995) and [*Automated Data Collection with R*](http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX) to learn more about HTML and XML element structures.

HTML elements are written with a start tag, an end tag, and with the content in between: <tagname>content</tagname>. The tags which typically contain the textual content we wish to scrape, and the tags we will leverage in the next two sections, include:

* <h1>, <h2>,...,<h6>: Largest heading, second largest heading, etc.
* <p>: Paragraph elements
* <ul>: Unordered bulleted list
* <ol>: Ordered list
* <li>: Individual List item
* <div>: Division or section
* <table>: Table

For example, text in paragraph form that you see online is wrapped with the HTML paragraph tag <p> as in:

<p>  
This paragraph represents  
a typical text paragraph  
in HTML form  
</p>

It is through these tags that we can start to extract textual components (also referred to as nodes) of HTML webpages.

### Scraping HTML Nodes

To scrape online text we'll make use of the relatively newer [rvest](https://cran.r-project.org/web/packages/rvest/index.html) package. rvest was created by the RStudio team inspired by libraries such as [beautiful soup](http://www.crummy.com/software/BeautifulSoup/) which has greatly simplified web scraping. rvest provides multiple functionalities; however, in this section we will focus only on extracting HTML text with rvest. Its important to note that rvest makes use of of the pipe operator (%>%) developed through the [magrittr package](https://cran.r-project.org/web/packages/magrittr/index.html). If you are not familiar with the functionality of %>% I recommend you jump to the chapter on [Simplifying Your Code with %>%](#pipe) so that you have a better understanding of what's going on with the code.

To extract text from a webpage of interest, we specify what HTML elements we want to select by using html\_nodes(). For instance, if we want to scrape the primary heading for the [Web Scraping Wikipedia webpage](https://en.wikipedia.org/wiki/Web_scraping) we simply identify the <h1> node as the node we want to select. html\_nodes() will identify all <h1> nodes on the webpage and return the HTML element. In our example we see there is only one <h1> node on this webpage.

library(rvest)  
  
scraping\_wiki <- read\_html("https://en.wikipedia.org/wiki/Web\_scraping")  
  
scraping\_wiki %>%  
 html\_nodes("h1")  
## {xml\_nodeset (1)}  
## [1] <h1 id="firstHeading" class="firstHeading" lang="en">Web scraping</h1>

To extract only the heading text for this <h1> node, and not include all the HTML syntax we use html\_text() which returns the heading text we see at the top of the [Web Scraping Wikipedia page](https://en.wikipedia.org/wiki/Web_scraping).

scraping\_wiki %>%  
 html\_nodes("h1") %>%  
 html\_text()  
## [1] "Web scraping"

If we want to identify all the second level headings on the webpage we follow the same process but instead select the <h2> nodes. In this example we see there are 10 second level headings on the [Web Scraping Wikipedia page](https://en.wikipedia.org/wiki/Web_scraping).

scraping\_wiki %>%  
 html\_nodes("h2") %>%  
 html\_text()  
## [1] "Contents"   
## [2] "Techniques[edit]"   
## [3] "Legal issues[edit]"   
## [4] "Notable tools[edit]"   
## [5] "See also[edit]"   
## [6] "Technical measures to stop bots[edit]"  
## [7] "Articles[edit]"   
## [8] "References[edit]"   
## [9] "See also[edit]"   
## [10] "Navigation menu"

Next, we can move on to extracting much of the text on this webpage which is in paragraph form. We can follow the same process illustrated above but instead we'll select all <p> nodes. This selects the 17 paragraph elements from the web page; which we can examine by subsetting the list p\_nodes to see the first line of each paragraph along with the HTML syntax. Just as before, to extract the text from these nodes and coerce them to a character string we simply apply html\_text().

p\_nodes <- scraping\_wiki %>%   
 html\_nodes("p")  
  
length(p\_nodes)  
## [1] 17  
  
p\_nodes[1:6]  
## {xml\_nodeset (6)}  
## [1] <p><b>Web scraping</b> (<b>web harvesting</b> or <b>web data extract ...  
## [2] <p>Web scraping is closely related to <a href="/wiki/Web\_indexing" t ...  
## [3] <p/>  
## [4] <p/>  
## [5] <p>Web scraping is the process of automatically collecting informati ...  
## [6] <p>Web scraping may be against the <a href="/wiki/Terms\_of\_use" titl ...  
  
  
p\_text <- scraping\_wiki %>%  
 html\_nodes("p") %>%  
 html\_text()  
  
p\_text[1]  
## [1] "Web scraping (web harvesting or web data extraction) is a computer software technique of extracting information from websites. Usually, such software programs simulate human exploration of the World Wide Web by either implementing low-level Hypertext Transfer Protocol (HTTP), or embedding a fully-fledged web browser, such as Mozilla Firefox."

Not too bad; however, we may not have captured all the text that we were hoping for. Since we extracted text for all <p> nodes, we collected all identified paragraph text; however, this does not capture the text in the bulleted lists. For example, when you look at the [Web Scraping Wikipedia page](https://en.wikipedia.org/wiki/Web_scraping) you will notice a significant amount of text in bulleted list format following the third paragraph under the [**Techniques**](https://en.wikipedia.org/wiki/Web_scraping#Techniques) heading. If we look at our data we'll see that that the text in this list format are not capture between the two paragraphs:

p\_text[5]  
## [1] "Web scraping is the process of automatically collecting information from the World Wide Web. It is a field with active developments sharing a common goal with the semantic web vision, an ambitious initiative that still requires breakthroughs in text processing, semantic understanding, artificial intelligence and human-computer interactions. Current web scraping solutions range from the ad-hoc, requiring human effort, to fully automated systems that are able to convert entire web sites into structured information, with limitations."  
  
p\_text[6]  
## [1] "Web scraping may be against the terms of use of some websites. The enforceability of these terms is unclear.[4] While outright duplication of original expression will in many cases be illegal, in the United States the courts ruled in Feist Publications v. Rural Telephone Service that duplication of facts is allowable. U.S. courts have acknowledged that users of \"scrapers\" or \"robots\" may be held liable for committing trespass to chattels,[5][6] which involves a computer system itself being considered personal property upon which the user of a scraper is trespassing. The best known of these cases, eBay v. Bidder's Edge, resulted in an injunction ordering Bidder's Edge to stop accessing, collecting, and indexing auctions from the eBay web site. This case involved automatic placing of bids, known as auction sniping. However, in order to succeed on a claim of trespass to chattels, the plaintiff must demonstrate that the defendant intentionally and without authorization interfered with the plaintiff's possessory interest in the computer system and that the defendant's unauthorized use caused damage to the plaintiff. Not all cases of web spidering brought before the courts have been considered trespass to chattels.[7]"

This is because the text in this list format are contained in <ul> nodes. To capture the text in lists, we can use the same steps as above but we select specific nodes which represent HTML lists components. We can approach extracting list text two ways.

First, we can pull all list elements (<ul>). When scraping all <ul> text, the resulting data structure will be a character string vector with each element representing a single list consisting of all list items in that list. In our running example there are 21 list elements as shown in the example that follows. You can see the first list scraped is the table of contents and the second list scraped is the list in the [Techniques](https://en.wikipedia.org/wiki/Web_scraping#Techniques) section.

ul\_text <- scraping\_wiki %>%  
 html\_nodes("ul") %>%  
 html\_text()  
  
length(ul\_text)  
## [1] 21  
  
ul\_text[1]  
## [1] "\n1 Techniques\n2 Legal issues\n3 Notable tools\n4 See also\n5 Technical measures to stop bots\n6 Articles\n7 References\n8 See also\n"  
  
# read the first 200 characters of the second list  
substr(ul\_text[2], start = 1, stop = 200)  
## [1] "\nHuman copy-and-paste: Sometimes even the best web-scraping technology cannot replace a human’s manual examination and copy-and-paste, and sometimes this may be the only workable solution when the web"

An alternative approach is to pull all <li> nodes. This will pull the text contained in each list item for all the lists. In our running example there's 146 list items that we can extract from this Wikipedia page. The first eight list items are the list of contents we see towards the top of the page. List items 9-17 are the list elements contained in the "[Techniques](https://en.wikipedia.org/wiki/Web_scraping#Techniques)" section, list items 18-44 are the items listed under the "[Notable Tools](https://en.wikipedia.org/wiki/Web_scraping#Notable_tools)" section, and so on.

li\_text <- scraping\_wiki %>%  
 html\_nodes("li") %>%  
 html\_text()  
  
length(li\_text)  
## [1] 147  
  
li\_text[1:8]  
## [1] "1 Techniques" "2 Legal issues"   
## [3] "3 Notable tools" "4 See also"   
## [5] "5 Technical measures to stop bots" "6 Articles"   
## [7] "7 References" "8 See also"

At this point we may believe we have all the text desired and proceed with joining the paragraph (p\_text) and list (ul\_text or li\_text) character strings and then perform the desired textual analysis. However, we may now have captured *more* text than we were hoping for. For example, by scraping all lists we are also capturing the listed links in the left margin of the webpage. If we look at the 104-136 list items that we scraped, we'll see that these texts correspond to the left margin text.

li\_text[104:136]  
## [1] "Main page" "Contents" "Featured content"   
## [4] "Current events" "Random article" "Donate to Wikipedia"  
## [7] "Wikipedia store" "Help" "About Wikipedia"   
## [10] "Community portal" "Recent changes" "Contact page"   
## [13] "What links here" "Related changes" "Upload file"   
## [16] "Special pages" "Permanent link" "Page information"   
## [19] "Wikidata item" "Cite this page" "Create a book"   
## [22] "Download as PDF" "Printable version" "Català"   
## [25] "Deutsch" "Español" "Français"   
## [28] "Íslenska" "Italiano" "Latviešu"   
## [31] "Nederlands" "日本語" "Српски / srpski"

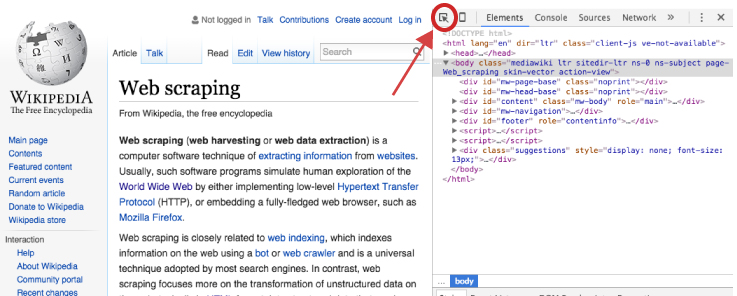
If we desire to scrape every piece of text on the webpage than this won't be of concern. In fact, if we want to scrape all the text regardless of the content they represent there is an easier approach. We can capture all the content to include text in paragraph (<p>), lists (<ul>, <ol>, and <li>), and even data in tables (<table>) by using <div>. This is because these other elements are usually a subsidiary of an HTML division or section so pulling all <div> nodes will extract all text contained in that division or section regardless if it is also contained in a paragraph or list.

all\_text <- scraping\_wiki %>%  
 html\_nodes("div") %>%   
 html\_text()

### Scraping Specific HTML Nodes

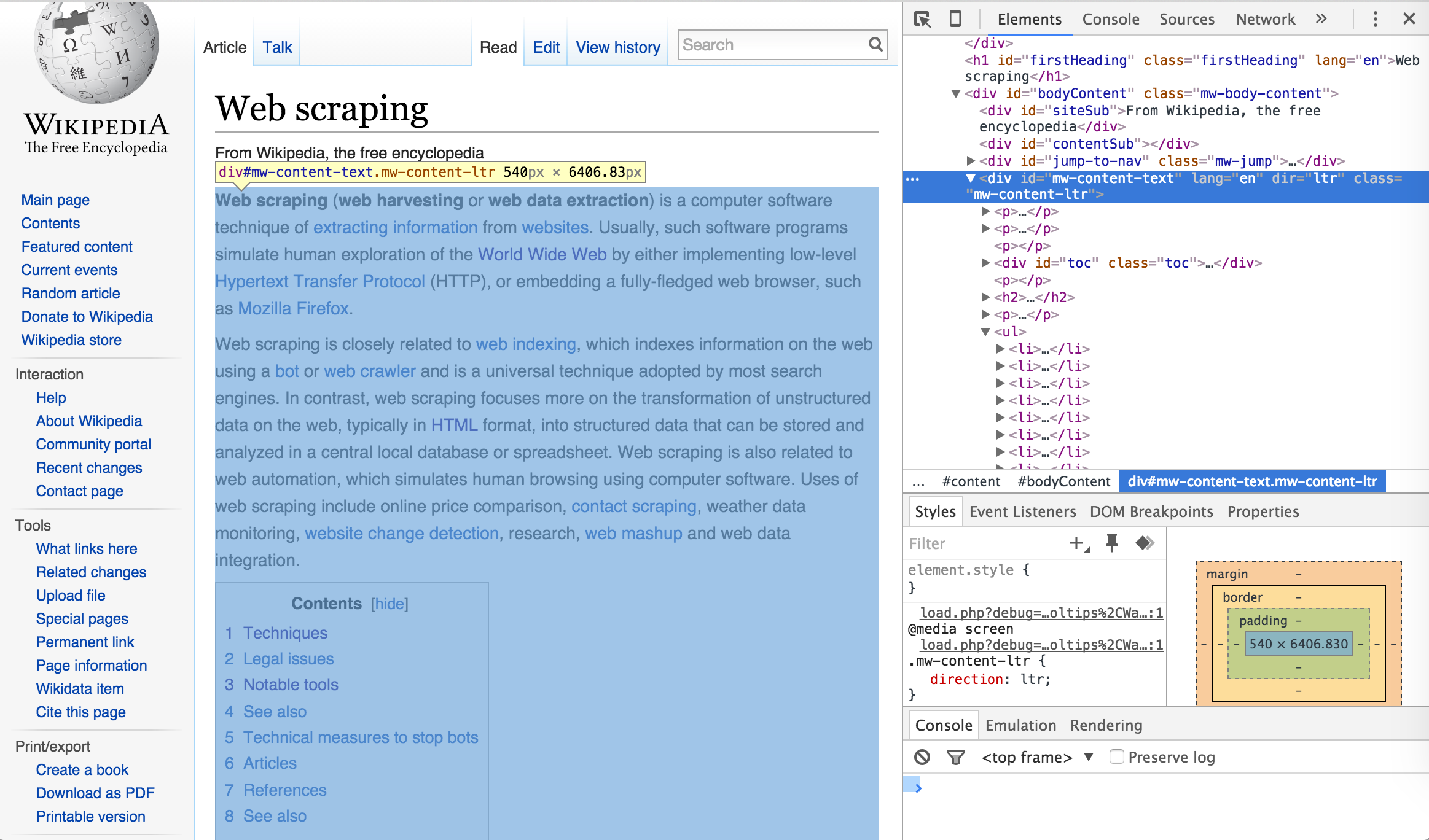
However, if we are concerned only with specific content on the webpage then we need to make our HTML node selection process a little more focused. To do this we, we can use our browser's developer tools to examine the webpage we are scraping and get more details on specific nodes of interest. If you are using Chrome or Firefox you can open the developer tools by clicking F12 (Cmd + Opt + I for Mac) or for Safari you would use Command-Option-I. An additional option which is recommended by Hadley Wickham is to use [selectorgadget.com](http://selectorgadget.com/), a Chrome extension, to help identify the web page elements you need[[3]](#footnote-81).

Once the developer's tools are opened your primary concern is with the element selector. This is located in the top lefthand corner of the developers tools window.



Developer Tools: Element Selector

Once you've selected the element selector you can now scroll over the elements of the webpage which will cause each element you scroll over to be highlighted. Once you've identified the element you want to focus on, select it. This will cause the element to be identified in the developer tools window. For example, if I am only interested in the main body of the Web Scraping content on the Wikipedia page then I would select the element that highlights the entire center component of the webpage. This highlights the corresponding element <div id="bodyContent" class="mw-body-content"> in the developer tools window as the following illustrates.



Selecting Content of Interest

I can now use this information to select and scrape all the text from this specific <div> node by calling the ID name ("#mw-content-text") in html\_nodes()[[4]](#footnote-85). As you can see below, the text that is scraped begins with the first line in the main body of the Web Scraping content and ends with the text in the [See Also](https://en.wikipedia.org/wiki/Web_scraping#See_also_2) section which is the last bit of text directly pertaining to Web Scraping on the webpage. Explicitly, we have pulled the specific text associated with the web content we desire.

body\_text <- scraping\_wiki %>%  
 html\_nodes("#mw-content-text") %>%   
 html\_text()  
  
# read the first 207 characters  
substr(body\_text, start = 1, stop = 207)  
## [1] "Web scraping (web harvesting or web data extraction) is a computer software technique of extracting information from websites. Usually, such software programs simulate human exploration of the World Wide Web"  
  
# read the last 73 characters  
substr(body\_text, start = nchar(body\_text)-73, stop = nchar(body\_text))  
## [1] "See also[edit]\n\nData scraping\nData wrangling\nKnowledge extraction\n\n\n\n\n\n\n\n\n"

Using the developer tools approach allows us to be as specific as we desire. We can identify the class name for a specific HTML element and scrape the text for only that node rather than all the other elements with similar tags. This allows us to scrape the main body of content as we just illustrated or we can also identify specific headings, paragraphs, lists, and list components if we desire to scrape only these specific pieces of text:

# Scraping a specific heading  
scraping\_wiki %>%  
 html\_nodes("#Techniques") %>%   
 html\_text()  
## [1] "Techniques"  
  
# Scraping a specific paragraph  
scraping\_wiki %>%  
 html\_nodes("#mw-content-text > p:nth-child(20)") %>%   
 html\_text()  
## [1] "In Australia, the Spam Act 2003 outlaws some forms of web harvesting, although this only applies to email addresses.[20][21]"  
  
# Scraping a specific list  
scraping\_wiki %>%  
 html\_nodes("#mw-content-text > div:nth-child(22)") %>%   
 html\_text()  
## [1] "\n\nApache Camel\nArchive.is\nAutomation Anywhere\nConvertigo\ncURL\nData Toolbar\nDiffbot\nFirebug\nGreasemonkey\nHeritrix\nHtmlUnit\nHTTrack\niMacros\nImport.io\nJaxer\nNode.js\nnokogiri\nPhantomJS\nScraperWiki\nScrapy\nSelenium\nSimpleTest\nwatir\nWget\nWireshark\nWSO2 Mashup Server\nYahoo! Query Language (YQL)\n\n"  
  
# Scraping a specific reference list item  
scraping\_wiki %>%  
 html\_nodes("#cite\_note-22") %>%   
 html\_text()  
## [1] "^ \"Web Scraping: Everything You Wanted to Know (but were afraid to ask)\". Distil Networks. 2015-07-22. Retrieved 2015-11-04. "

### Cleaning up

With any webscraping activity, especially involving text, there is likely to be some clean up involved. For example, in the previous example we saw that we can specifically pull the list of [**Notable Tools**](https://en.wikipedia.org/wiki/Web_scraping#Notable_tools); however, you can see that in between each list item rather than a space there contains one or more \n which is used in HTML to specify a new line. We can clean this up quickly with a little [character string manipulation](#string_manipulation).

library(magrittr)  
  
scraping\_wiki %>%  
 html\_nodes("#mw-content-text > div:nth-child(22)") %>%   
 html\_text()  
## [1] "\n\nApache Camel\nArchive.is\nAutomation Anywhere\nConvertigo\ncURL\nData Toolbar\nDiffbot\nFirebug\nGreasemonkey\nHeritrix\nHtmlUnit\nHTTrack\niMacros\nImport.io\nJaxer\nNode.js\nnokogiri\nPhantomJS\nScraperWiki\nScrapy\nSelenium\nSimpleTest\nwatir\nWget\nWireshark\nWSO2 Mashup Server\nYahoo! Query Language (YQL)\n\n"  
  
scraping\_wiki %>%  
 html\_nodes("#mw-content-text > div:nth-child(22)") %>%   
 html\_text() %>%   
 strsplit(split = "\n") %>%  
 unlist() %>%  
 .[. != ""]  
## [1] "Apache Camel" "Archive.is"   
## [3] "Automation Anywhere" "Convertigo"   
## [5] "cURL" "Data Toolbar"   
## [7] "Diffbot" "Firebug"   
## [9] "Greasemonkey" "Heritrix"   
## [11] "HtmlUnit" "HTTrack"   
## [13] "iMacros" "Import.io"   
## [15] "Jaxer" "Node.js"   
## [17] "nokogiri" "PhantomJS"   
## [19] "ScraperWiki" "Scrapy"   
## [21] "Selenium" "SimpleTest"   
## [23] "watir" "Wget"   
## [25] "Wireshark" "WSO2 Mashup Server"   
## [27] "Yahoo! Query Language (YQL)"

Similarly, as we saw in our example above with scraping the main body content (body\_text), there are extra characters (i.e. \n, \, ^) in the text that we may not want. Using a [little regex](#regex) we can clean this up so that our character string consists of only text that we see on the screen and no additional HTML code embedded throughout the text.

library(stringr)  
  
# read the last 700 characters  
substr(body\_text, start = nchar(body\_text)-700, stop = nchar(body\_text))  
## [1] " 2010). \"Intellectual Property: Website Terms of Use\". Issue 26: June 2010. LK Shields Solicitors Update. p. 03. Retrieved 2012-04-19. \n^ National Office for the Information Economy (February 2004). \"Spam Act 2003: An overview for business\" (PDF). Australian Communications Authority. p. 6. Retrieved 2009-03-09. \n^ National Office for the Information Economy (February 2004). \"Spam Act 2003: A practical guide for business\" (PDF). Australian Communications Authority. p. 20. Retrieved 2009-03-09. \n^ \"Web Scraping: Everything You Wanted to Know (but were afraid to ask)\". Distil Networks. 2015-07-22. Retrieved 2015-11-04. \n\n\nSee also[edit]\n\nData scraping\nData wrangling\nKnowledge extraction\n\n\n\n\n\n\n\n\n"  
  
# clean up text  
body\_text %>%  
 str\_replace\_all(pattern = "\n", replacement = " ") %>%  
 str\_replace\_all(pattern = "[\\^]", replacement = " ") %>%  
 str\_replace\_all(pattern = "\"", replacement = " ") %>%  
 str\_replace\_all(pattern = "\\s+", replacement = " ") %>%  
 str\_trim(side = "both") %>%  
 substr(start = nchar(body\_text)-700, stop = nchar(body\_text))  
## [1] "012-04-19. National Office for the Information Economy (February 2004). Spam Act 2003: An overview for business (PDF). Australian Communications Authority. p. 6. Retrieved 2009-03-09. National Office for the Information Economy (February 2004). Spam Act 2003: A practical guide for business (PDF). Australian Communications Authority. p. 20. Retrieved 2009-03-09. Web Scraping: Everything You Wanted to Know (but were afraid to ask) . Distil Networks. 2015-07-22. Retrieved 2015-11-04. See also[edit] Data scraping Data wrangling Knowledge extraction"

So there we have it, text scraping in a nutshell. Although not all encompassing, this section covered the basics of scraping text from HTML documents. Whether you want to scrape text from all common text-containing nodes such as <div>, <p>, <ul> and the like or you want to scrape from a specific node using the specific ID, this section provides you the basic fundamentals of using rvest to scrape the text you need. In the next section we move on to scraping data from HTML tables.

## Scraping HTML table data

Another common structure of information storage on the Web is in the form of HTML tables. This section reiterates some of the information from the [previous section](#scraping_HTML_text); however, we focus solely on scraping data from HTML tables. The simplest approach to scraping HTML table data directly into R is by using either the [rvest package](#scraping_tables_rvest) or the [XML package](#scraping_tables_xml). To illustrate, I will focus on the [BLS employment statistics webpage](http://www.bls.gov/web/empsit/cesbmart.htm) which contains multiple HTML tables from which we can scrape data.

### Scraping HTML tables with rvest

Recall that HTML elements are written with a start tag, an end tag, and with the content in between: <tagname>content</tagname>. HTML tables are contained within <table> tags; therefore, to extract the tables from the BLS employment statistics webpage we first use the html\_nodes() function to select the <table> nodes. In this case we are interested in all table nodes that exist on the webpage. In this example, html\_nodes captures 15 HTML tables. This includes data from the 10 data tables seen on the webpage but also includes data from a few additional tables used to format parts of the page (i.e. table of contents, table of figures, advertisements).

library(rvest)  
  
webpage <- read\_html("http://www.bls.gov/web/empsit/cesbmart.htm")  
  
tbls <- html\_nodes(webpage, "table")  
  
head(tbls)  
## {xml\_nodeset (6)}  
## [1] <table id="main-content-table">&#13;\n\t<tr>&#13;\n\t\t<td id="secon ...  
## [2] <table id="Table1" class="regular" cellspacing="0" cellpadding="0" x ...  
## [3] <table id="Table2" class="regular" cellspacing="0" cellpadding="0" x ...  
## [4] <table id="Table3" class="regular" cellspacing="0" cellpadding="0" x ...  
## [5] <table id="Table4" class="regular" cellspacing="0" cellpadding="0" x ...  
## [6] <table id="Exhibit1" class="regular" cellspacing="0" cellpadding="0" ...

Remember that html\_nodes() does not parse the data; rather, it acts as a CSS selector. To parse the HTML table data we use html\_table(), which would create a list containing 15 data frames. However, rarely do we need to scrape *every* HTML table from a page, especially since some HTML tables don't catch any information we are likely interested in (i.e. table of contents, table of figures, footers).

More often than not we want to parse specific tables. Lets assume we want to parse the second and third tables on the webpage:

* Table 2. Nonfarm employment benchmarks by industry, March 2014 (in thousands) and
* Table 3. Net birth/death estimates by industry supersector, April – December 2014 (in thousands)

This can be accomplished two ways. First, we can assess the previous tbls list and try to identify the table(s) of interest. In this example it appears that tbls list items 3 and 4 correspond with Table 2 and Table 3, respectively. We can then subset the list of table nodes prior to parsing the data with html\_table(). This results in a list of two data frames containing the data of interest.

# subset list of table nodes for items 3 & 4  
tbls\_ls <- webpage %>%  
 html\_nodes("table") %>%  
 .[3:4] %>%  
 html\_table(fill = TRUE)  
  
str(tbls\_ls)  
## List of 2  
## $ :'data.frame': 147 obs. of 6 variables:  
## ..$ CES Industry Code : chr [1:147] "Amount" "00-000000" "05-000000" ...  
## ..$ CES Industry Title: chr [1:147] "Percent" "Total nonfarm" ...  
## ..$ Benchmark : chr [1:147] NA "137,214" "114,989" "18,675" ...  
## ..$ Estimate : chr [1:147] NA "137,147" "114,884" "18,558" ...  
## ..$ Differences : num [1:147] NA 67 105 117 -50 -12 -16 -2.8 ...  
## ..$ NA : chr [1:147] NA "(1)" "0.1" "0.6" ...  
## $ :'data.frame': 11 obs. of 12 variables:  
## ..$ CES Industry Code : chr [1:11] "10-000000" "20-000000" "30-000000" ...  
## ..$ CES Industry Title: chr [1:11] "Mining and logging" "Construction" ...  
## ..$ Apr : int [1:11] 2 35 0 21 0 8 81 22 82 12 ...  
## ..$ May : int [1:11] 2 37 6 24 5 8 22 13 81 6 ...  
## ..$ Jun : int [1:11] 2 24 4 12 0 4 5 -14 86 6 ...  
## ..$ Jul : int [1:11] 2 12 -3 7 -1 3 35 7 62 -2 ...  
## ..$ Aug : int [1:11] 1 12 4 14 3 4 19 21 23 3 ...  
## ..$ Sep : int [1:11] 1 7 1 9 -1 -1 -12 12 -33 -2 ...  
## ..$ Oct : int [1:11] 1 12 3 28 6 16 76 35 -17 4 ...  
## ..$ Nov : int [1:11] 1 -10 2 10 3 3 14 14 -22 1 ...  
## ..$ Dec : int [1:11] 0 -21 0 4 0 10 -10 -3 4 1 ...  
## ..$ CumulativeTotal : int [1:11] 12 108 17 129 15 55 230 107 266 29 ...

An alternative approach, which is more explicit, is to use the [element selector process described in the previous section](#scraping_specific_nodes) to call the table ID name.

# empty list to add table data to  
tbls2\_ls <- list()  
  
# scrape Table 2. Nonfarm employment...  
tbls2\_ls$Table1 <- webpage %>%  
 html\_nodes("#Table2") %>%   
 html\_table(fill = TRUE) %>%  
 .[[1]]  
  
# Table 3. Net birth/death...  
tbls2\_ls$Table2 <- webpage %>%  
 html\_nodes("#Table3") %>%   
 html\_table() %>%  
 .[[1]]  
  
str(tbls2\_ls)  
## List of 2  
## $ Table1:'data.frame': 147 obs. of 6 variables:  
## ..$ CES Industry Code : chr [1:147] "Amount" "00-000000" "05-000000" ...  
## ..$ CES Industry Title: chr [1:147] "Percent" "Total nonfarm" ...  
## ..$ Benchmark : chr [1:147] NA "137,214" "114,989" "18,675" ...  
## ..$ Estimate : chr [1:147] NA "137,147" "114,884" "18,558" ...  
## ..$ Differences : num [1:147] NA 67 105 117 -50 -12 -16 -2.8 ...  
## ..$ NA : chr [1:147] NA "(1)" "0.1" "0.6" ...  
## $ Table2:'data.frame': 11 obs. of 12 variables:  
## ..$ CES Industry Code : chr [1:11] "10-000000" "20-000000" "30-000000" ...  
## ..$ CES Industry Title: chr [1:11] "Mining and logging" "Construction" ...  
## ..$ Apr : int [1:11] 2 35 0 21 0 8 81 22 82 12 ...  
## ..$ May : int [1:11] 2 37 6 24 5 8 22 13 81 6 ...  
## ..$ Jun : int [1:11] 2 24 4 12 0 4 5 -14 86 6 ...  
## ..$ Jul : int [1:11] 2 12 -3 7 -1 3 35 7 62 -2 ...  
## ..$ Aug : int [1:11] 1 12 4 14 3 4 19 21 23 3 ...  
## ..$ Sep : int [1:11] 1 7 1 9 -1 -1 -12 12 -33 -2 ...  
## ..$ Oct : int [1:11] 1 12 3 28 6 16 76 35 -17 4 ...  
## ..$ Nov : int [1:11] 1 -10 2 10 3 3 14 14 -22 1 ...  
## ..$ Dec : int [1:11] 0 -21 0 4 0 10 -10 -3 4 1 ...  
## ..$ CumulativeTotal : int [1:11] 12 108 17 129 15 55 230 107 266 29 ...

One issue to note is when using rvest's html\_table() to read a table with split column headings as in *Table 2. Nonfarm employment...*. html\_table will cause split headings to be included and can cause the first row to include parts of the headings. We can see this with Table 2. This requires a little clean up.

head(tbls2\_ls[[1]], 4)  
## CES Industry Code CES Industry Title Benchmark Estimate Differences NA  
## 1 Amount Percent <NA> <NA> NA <NA>  
## 2 00-000000 Total nonfarm 137,214 137,147 67 (1)  
## 3 05-000000 Total private 114,989 114,884 105 0.1  
## 4 06-000000 Goods-producing 18,675 18,558 117 0.6  
  
# remove row 1 that includes part of the headings  
tbls2\_ls[[1]] <- tbls2\_ls[[1]][-1,]  
  
# rename table headings  
colnames(tbls2\_ls[[1]]) <- c("CES\_Code", "Ind\_Title", "Benchmark",  
 "Estimate", "Amt\_Diff", "Pct\_Diff")  
  
head(tbls2\_ls[[1]], 4)  
## CES\_Code Ind\_Title Benchmark Estimate Amt\_Diff Pct\_Diff  
## 2 00-000000 Total nonfarm 137,214 137,147 67 (1)  
## 3 05-000000 Total private 114,989 114,884 105 0.1  
## 4 06-000000 Goods-producing 18,675 18,558 117 0.6  
## 5 07-000000 Service-providing 118,539 118,589 -50 (1)

### Scraping HTML tables with XML

An alternative to rvest for table scraping is to use the [XML](https://cran.r-project.org/web/packages/XML/index.html) package. The XML package provides a convenient readHTMLTable() function to extract data from HTML tables in HTML documents. By passing the URL to readHTMLTable(), the data in each table is read and stored as a data frame. In a situation like our running example where multiple tables exists, the data frames will be stored in a list similar to rvest's html\_table.

library(XML)  
  
url <- "http://www.bls.gov/web/empsit/cesbmart.htm"  
  
# read in HTML data  
tbls\_xml <- readHTMLTable(url)  
  
typeof(tbls\_xml)  
## [1] "list"  
  
length(tbls\_xml)  
## [1] 15

You can see that tbls\_xml captures the same 15 <table> nodes that html\_nodes captured. To capture the same tables of interest we previously discussed (*Table 2. Nonfarm employment...* and *Table 3. Net birth/death...*) we can use a couple approaches. First, we can assess str(tbls\_xml) to identify the tables of interest and perform normal [list subsetting](#lists_subset). In our example list items 3 and 4 correspond with our tables of interest.

head(tbls\_xml[[3]])  
## V1 V2 V3 V4 V5 V6  
## 1 00-000000 Total nonfarm 137,214 137,147 67 (1)  
## 2 05-000000 Total private 114,989 114,884 105 0.1  
## 3 06-000000 Goods-producing 18,675 18,558 117 0.6  
## 4 07-000000 Service-providing 118,539 118,589 -50 (1)  
## 5 08-000000 Private service-providing 96,314 96,326 -12 (1)  
## 6 10-000000 Mining and logging 868 884 -16 -1.8  
  
head(tbls\_xml[[4]], 3)  
## CES Industry Code CES Industry Title Apr May Jun Jul Aug Sep Oct Nov Dec  
## 1 10-000000 Mining and logging 2 2 2 2 1 1 1 1 0  
## 2 20-000000 Construction 35 37 24 12 12 7 12 -10 -21  
## 3 30-000000 Manufacturing 0 6 4 -3 4 1 3 2 0  
## CumulativeTotal  
## 1 12  
## 2 108  
## 3 17

Second, we can use the which argument in readHTMLTable() which restricts the data importing to only those tables specified numerically.

# only parse the 3rd and 4th tables  
emp\_ls <- readHTMLTable(url, which = c(3, 4))  
  
str(emp\_ls)  
## List of 2  
## $ Table2:'data.frame': 145 obs. of 6 variables:  
## ..$ V1: Factor w/ 145 levels "00-000000","05-000000",..: 1 2 3 4 5 6 7 8 ...  
## ..$ V2: Factor w/ 143 levels "Accommodation",..: 130 131 52 116 102 74 ...  
## ..$ V3: Factor w/ 145 levels "1,010.3","1,048.3",..: 40 35 48 37 145 140 ...  
## ..$ V4: Factor w/ 145 levels "1,008.4","1,052.3",..: 41 34 48 36 144 142 ...  
## ..$ V5: Factor w/ 123 levels "-0.3","-0.4",..: 113 68 71 48 9 19 29 11 ...  
## ..$ V6: Factor w/ 56 levels "-0.1","-0.2",..: 30 31 36 30 30 16 28 14 29 ...  
## $ Table3:'data.frame': 11 obs. of 12 variables:  
## ..$ CES Industry Code : Factor w/ 11 levels "10-000000","20-000000",..:1 ...  
## ..$ CES Industry Title: Factor w/ 11 levels "263","Construction",..: 8 2 ...  
## ..$ Apr : Factor w/ 10 levels "0","12","2","204",..: 3 7 1 ...  
## ..$ May : Factor w/ 10 levels "129","13","2",..: 3 6 8 5 7 ...  
## ..$ Jun : Factor w/ 10 levels "-14","0","12",..: 5 6 7 3 2 ...  
## ..$ Jul : Factor w/ 10 levels "-1","-2","-3",..: 6 5 3 10 ...  
## ..$ Aug : Factor w/ 9 levels "-19","1","12",..: 2 3 9 4 8 ...  
## ..$ Sep : Factor w/ 9 levels "-1","-12","-2",..: 5 8 5 9 1 ...  
## ..$ Oct : Factor w/ 10 levels "-17","1","12",..: 2 3 6 5 9 ...  
## ..$ Nov : Factor w/ 8 levels "-10","-15","-22",..: 4 1 7 5 ...  
## ..$ Dec : Factor w/ 8 levels "-10","-21","-3",..: 4 2 4 7 ...  
## ..$ CumulativeTotal : Factor w/ 10 levels "107","108","12",..: 3 2 6 4 ...

The third option involves explicitly naming the tables to parse. This process uses the [element selector process described in the previous section](#scraping_specific_nodes) to call the table by name. We use getNodeSet() to select the specified tables of interest. However, a key difference here is rather than copying the table ID names you want to copy the XPath. You can do this with the following: After you've highlighted the table element of interest with the element selector, right click the highlighted element in the developer tools window and select Copy XPath. From here we just use readHTMLTable() to convert to data frames and we have our desired tables.

library(RCurl)  
  
# parse url  
url\_parsed <- htmlParse(getURL(url), asText = TRUE)  
  
# select table nodes of interest  
tableNodes <- getNodeSet(url\_parsed, c('//\*[@id="Table2"]', '//\*[@id="Table3"]'))  
  
# convert HTML tables to data frames  
bls\_table2 <- readHTMLTable(tableNodes[[1]])  
bls\_table3 <- readHTMLTable(tableNodes[[2]])  
  
head(bls\_table2)  
## V1 V2 V3 V4 V5 V6  
## 1 00-000000 Total nonfarm 137,214 137,147 67 (1)  
## 2 05-000000 Total private 114,989 114,884 105 0.1  
## 3 06-000000 Goods-producing 18,675 18,558 117 0.6  
## 4 07-000000 Service-providing 118,539 118,589 -50 (1)  
## 5 08-000000 Private service-providing 96,314 96,326 -12 (1)  
## 6 10-000000 Mining and logging 868 884 -16 -1.8  
  
head(bls\_table3, 3)  
## CES Industry Code CES Industry Title Apr May Jun Jul Aug Sep Oct Nov Dec  
## 1 10-000000 Mining and logging 2 2 2 2 1 1 1 1 0  
## 2 20-000000 Construction 35 37 24 12 12 7 12 -10 -21  
## 3 30-000000 Manufacturing 0 6 4 -3 4 1 3 2 0  
## CumulativeTotal  
## 1 12  
## 2 108  
## 3 17

A few benefits of XML's readHTMLTable that are routinely handy include:

* We can specify names for the column headings
* We can specify the classes for each column
* We can specify rows to skip

For instance, if you look at bls\_table2 above notice that because of the split column headings on *Table 2. Nonfarm employment...* readHTMLTable stripped and replaced the headings with generic names because R does not know which variable names should align with each column. We can correct for this with the following:

bls\_table2 <- readHTMLTable(tableNodes[[1]],   
 header = c("CES\_Code", "Ind\_Title", "Benchmark",  
 "Estimate", "Amt\_Diff", "Pct\_Diff"))  
  
head(bls\_table2)  
## CES\_Code Ind\_Title Benchmark Estimate Amt\_Diff Pct\_Diff  
## 1 00-000000 Total nonfarm 137,214 137,147 67 (1)  
## 2 05-000000 Total private 114,989 114,884 105 0.1  
## 3 06-000000 Goods-producing 18,675 18,558 117 0.6  
## 4 07-000000 Service-providing 118,539 118,589 -50 (1)  
## 5 08-000000 Private service-providing 96,314 96,326 -12 (1)  
## 6 10-000000 Mining and logging 868 884 -16 -1.8

Also, for bls\_table3 note that the net birth/death values parsed have been converted to factor levels. We can use the colClasses argument to correct this.

str(bls\_table3)  
## 'data.frame': 11 obs. of 12 variables:  
## $ CES Industry Code : Factor w/ 11 levels "10-000000","20-000000",..: 1 2 ...  
## $ CES Industry Title: Factor w/ 11 levels "263","Construction",..: 8 2 7 ...  
## $ Apr : Factor w/ 10 levels "0","12","2","204",..: 3 7 1 5 ...  
## $ May : Factor w/ 10 levels "129","13","2",..: 3 6 8 5 7 9 ...  
## $ Jun : Factor w/ 10 levels "-14","0","12",..: 5 6 7 3 2 7 ...  
## $ Jul : Factor w/ 10 levels "-1","-2","-3",..: 6 5 3 10 1 7 ...  
## $ Aug : Factor w/ 9 levels "-19","1","12",..: 2 3 9 4 8 9 5 ...  
## $ Sep : Factor w/ 9 levels "-1","-12","-2",..: 5 8 5 9 1 1 ...  
## $ Oct : Factor w/ 10 levels "-17","1","12",..: 2 3 6 5 9 4 ...  
## $ Nov : Factor w/ 8 levels "-10","-15","-22",..: 4 1 7 5 8 ...  
## $ Dec : Factor w/ 8 levels "-10","-21","-3",..: 4 2 4 7 4 6 ...  
## $ CumulativeTotal : Factor w/ 10 levels "107","108","12",..: 3 2 6 4 5 ...  
  
bls\_table3 <- readHTMLTable(tableNodes[[2]],   
 colClasses = c("character","character",   
 rep("integer", 10)))  
  
str(bls\_table3)  
## 'data.frame': 11 obs. of 12 variables:  
## $ CES Industry Code : Factor w/ 11 levels "10-000000","20-000000",..: 1 2 ...  
## $ CES Industry Title: Factor w/ 11 levels "263","Construction",..: 8 2 7 ...  
## $ Apr : int 2 35 0 21 0 8 81 22 82 12 ...  
## $ May : int 2 37 6 24 5 8 22 13 81 6 ...  
## $ Jun : int 2 24 4 12 0 4 5 -14 86 6 ...  
## $ Jul : int 2 12 -3 7 -1 3 35 7 62 -2 ...  
## $ Aug : int 1 12 4 14 3 4 19 21 23 3 ...  
## $ Sep : int 1 7 1 9 -1 -1 -12 12 -33 -2 ...  
## $ Oct : int 1 12 3 28 6 16 76 35 -17 4 ...  
## $ Nov : int 1 -10 2 10 3 3 14 14 -22 1 ...  
## $ Dec : int 0 -21 0 4 0 10 -10 -3 4 1 ...  
## $ CumulativeTotal : int 12 108 17 129 15 55 230 107 266 29 ...

Between rvest and XML, scraping HTML tables is relatively easy once you get fluent with the syntax and the available options. This section covers just the basics of both these packages to get you moving forward with scraping tables. In the next section we move on to working with application program interfaces (APIs) to get data from the web.

## Working with APIs

An application-programming interface (API) in a nutshell is a method of communication between software programs. APIs allow programs to interact and use each other's functions by acting as a middle man. Why is this useful? Lets say you want to pull weather data from the [NOAA](http://www.ncdc.noaa.gov/cdo-web/webservices). You have a few options:

* You could query the data and download the spreadsheet or manually cut-n-paste the desired data and then import into R. Doesn't get you any coolness points.
* You could use some webscraping techniques previously covered to parse the desired data. Golf clap. The downfall of this strategy is if NOAA changes their website structure down the road your code will need to be adjusted.
* Or, you can use the [rnoaa](https://ropensci.org/tutorials/rnoaa_tutorial.html) package which allows you to send specific instructions to the NOAA API via R, the API will then perform the action requested and return the desired information. The benefit of this strategy is if the NOAA changes its website structure it won't impact the API data retreival structure which means no impact to your code. Standing ovation!

Consequently, APIs provide consistency in data retrieval processes which can be essential for recurring analyses. Luckily, the use of APIs by organizations that collect data are [growing exponentially](http://www.programmableweb.com/api-research). This is great for you and I as more and more data continues to be at our finger tips. So what do you need to get started?

### Prerequisites?

Each API is unique; however, there are a few fundamental pieces of information you'll need to work with an API. First, the reason you're using an API is to request specific types of data from a specific data set from a specific organization. You at least need to know a little something about each one of these:

1. The URL for the organization and data you are pulling. Most pre-built API packages already have this connection established but when using httr you'll need to specify.
2. The data set you are trying to pull from. Most organizations have numerous data sets to peruse so you need to make yourself familiar with the names of the available data sets.
3. The data content. You'll need to specify the specific data variables you want the API to retrieve so you'll need to be familiar with, or have access to, the data library.

In addition to these key components you will also, typically, need to provide a form of identification and/or authorization. This is done via:

1. API key (aka token). A key is used to identify the user along with track and control how the API is being used (guard against malicious use). A key is often obtained by supplying basic information (i.e. name, email) to the organization and in return they give you a multi-digit key.
2. [OAuth](http://oauth.net/). OAuth is an authorization framework that provides credentials as proof for access to certain information. Multiple forms of credentials exist and OAuth can actually be a fairly confusing topic; however, the httr package has simplified this greatly which we demonstrate at the end of this section.

Rather than dwell on these components, they'll likely become clearer as we progress through examples. So, let's move on to the fun stuff.

### Existing API Packages

Like everything else you do in R, when looking to work with an API your first question should be "Is there a package for that?" R has an extensive list of packages in which API data feeds have been hooked into R. You can find a slew of them scattered throughout the [CRAN Task View: Web Technologies and Services](https://cran.r-project.org/web/views/WebTechnologies.html) web page, on the [rOpenSci](https://ropensci.org/packages/) web page, and some more [here](http://stats.stackexchange.com/questions/12670/data-apis-feeds-available-as-packages-in-r).

To give you a taste for how these packages typically work, I'll quickly cover three packages:

* [blsAPI](#blsAPI) for pulling U.S. Bureau of Labor Statistics data
* [rnoaa](#rnoaa) for pulling NOAA climate data
* [rtimes](#rtimes) for pulling data from multiple APIs offered by the New York Times

#### blsAPI

The [blsAPI](https://cran.r-project.org/web/packages/blsAPI/index.html) allows users to request data for one or multiple series through the U.S. Bureau of Labor Statistics API. To use the blsAPI app you only need knowledge on the data; no key or OAuth are required. I lllustrate by pulling [Mass Layoff Statistics](http://www.bls.gov/mls/mlsover.htm) data but you will find all the available data sets and their series code information [here](http://www.bls.gov/help/hlpforma.htm).

The key information you will be concerned about is contained in the series identifier. For the Mass Layoff data the the series ID code is MLUMS00NN0001003. Each component of this series code has meaning and can be adjusted to get specific Mass Layoff data. The BLS provides this [breakdown](http://www.bls.gov/help/hlpforma.htm#ML) for what each component means along with the available list of codes for this data set. For instance, the **S00** (MLUM**S00**NN0001003) component represents the [division/state](http://download.bls.gov/pub/time.series/ml/ml.srd). S00 will pull for all states but I could change to D30 to pull data for the Midwest or S39 to pull for Ohio. The **N0001** (MLUMS00N**N0001**003) component represents the [industry/demographics](http://download.bls.gov/pub/time.series/ml/ml.irc). N0001 pulls data for all industries but I could change to N0008 to pull data for the food industry or C00A2 for all persons age 30-44.

I simply call the series identifier in the blsAPI() function which pulls the JSON data object. We can then use the fromJSON() function from the rjson package to convert to an R data object (a list in this case). You can see that the raw data pull provides a list of 4 items. The first three provide some metadata info (status, response time, and message if applicable). The data we are concerned about is in the 4th (Results$series$data) list item which contains 31 observations.

library(rjson)  
library(blsAPI)  
  
# supply series identifier to pull data (initial pull is in JSON data)  
layoffs\_json <- blsAPI('MLUMS00NN0001003')   
  
# convert from JSON into R object  
layoffs <- fromJSON(layoffs\_json)   
  
List of 4  
 $ status : chr "REQUEST\_SUCCEEDED"  
 $ responseTime: num 38  
 $ message : list()  
 $ Results :List of 1  
 ..$ series:List of 1  
 .. ..$ :List of 2  
 .. .. ..$ seriesID: chr "MLUMS00NN0001003"  
 .. .. ..$ data :List of 31  
 .. .. .. ..$ :List of 5  
 .. .. .. .. ..$ year : chr "2013"  
 .. .. .. .. ..$ period : chr "M05"  
 .. .. .. .. ..$ periodName: chr "May"  
 .. .. .. .. ..$ value : chr "1383"

One of the inconveniences of an API is we do not get to specify how the data we receive is formatted. This is a minor price to pay considering all the other benefits APIs provide. Once we understand the received data format we can typically re-format using a little [list subsetting](#lists_subset) which we previously covered and looping which we'll cover in a [future chapter](#loop_functions).

# create empty data frame to fill   
layoff\_df <- data.frame(NULL)  
  
# extract data of interest from each nested year-month list   
for(i in seq\_along(layoffs$Results$series[[1]]$data)) {  
 df <- data.frame(layoffs$Results$series[[1]]$data[i][[1]][1:4])  
 layoff\_df <- rbind(layoff\_df, df)  
}  
  
head(layoff\_df)  
## year period periodName value  
## 1 2013 M05 May 1383  
## 2 2013 M04 April 1174  
## 3 2013 M03 March 1132  
## 4 2013 M02 February 960  
## 5 2013 M01 January 1528  
## 6 2012 M13 Annual 17080

blsAPI also allows you to pull multiple data series and has optional arguments (i.e. start year, end year, etc.). You can see other options at help(package = blsAPI).

#### rnoaa

The [rnoaa](https://ropensci.org/tutorials/rnoaa_tutorial.html) package allows users to request climate data from multiple data sets through the [National Climatic Data Center API](http://www.ncdc.noaa.gov/cdo-web/webservices/v2). Unlike blsAPI, the rnoaa app requires you to have an API key. To request a key go [here](http://www.ncdc.noaa.gov/cdo-web/token) and provide your email; a key will immediately be emailed to you.

key <- "vXTdwNoAVx..." # truncated

With the key in hand, we can begin pulling data. The NOAA provides a comprehensive [metadata library](http://www.ncdc.noaa.gov/homr/reports) to familiarize yourself with the data available. Let's start by pulling all the available NOAA climate stations near my residence. I live in Montgomery county Ohio so we can find all the stations in this county by inserting the [FIPS code](http://www.census.gov/geo/reference/codes/cou.html). Furthermore, I'm interested in stations that provide data for the [GHCND data set](https://www.ncdc.noaa.gov/oa/climate/ghcn-daily/) which contains records on numerous daily variables such as "maximum and minimum temperature, total daily precipitation, snowfall, and snow depth; however, about two thirds of the stations report precipitation only." See ?ncdc\_stations for other data sets available via rnoaa.

library(rnoaa)  
  
stations <- ncdc\_stations(datasetid='GHCND',   
 locationid='FIPS:39113',  
 token = key)  
  
stations$data  
## Source: local data frame [23 x 9]  
##   
## elevation mindate maxdate latitude  
## (dbl) (chr) (chr) (dbl)  
## 1 294.1 2009-02-09 2014-06-25 39.6314  
## 2 251.8 2009-03-01 2016-01-16 39.6807  
## 3 295.7 2009-03-25 2012-09-08 39.6252  
## 4 298.1 2009-08-24 2012-07-20 39.8070  
## 5 304.5 2010-04-02 2016-01-12 39.6949  
## 6 283.5 2012-07-01 2016-01-16 39.7373  
## 7 301.4 2012-07-29 2016-01-16 39.8795  
## 8 317.3 2012-09-08 2016-01-12 39.8329  
## 9 298.1 2012-09-07 2016-01-15 39.6247  
## 10 250.5 2012-09-11 2016-01-08 39.7180  
## .. ... ... ... ...  
## Variables not shown: name (chr), datacoverage (dbl), id (chr),  
## elevationUnit (chr), longitude (dbl)

So we see that several stations are available from which to pull data. To actually pull data from one of these stations we need the station ID. The station I want to pull data from is the Dayton International Airport station. We can see that this station provides data from 1948-present and I can get the station ID as illustrated. Note that I use some dplyr for data manipulation here; we will cover dplyr in a later [chapter](#dplyr) but this just illustrates the fact that we received the data via the API.

library(dplyr)  
  
stations$data %>%   
 filter(name == "DAYTON INTERNATIONAL AIRPORT, OH US") %>%   
 select(mindate, maxdate, id)  
## Source: local data frame [1 x 3]  
##   
## mindate maxdate id  
## (chr) (chr) (chr)  
## 1 1948-01-01 2016-01-15 GHCND:USW00093815

To pull all available GHCND data from this station we'll use ncdc(). We simply supply the data to pull, the start and end dates (ncdc restricts you to a one year limit), station ID, and your key. We can see that this station provides a full range of data types.

climate <- ncdc(datasetid='GHCND',   
 startdate = '2015-01-01',   
 enddate = '2016-01-01',   
 stationid='GHCND:USW00093815',  
 token = key)  
  
climate$data  
## Source: local data frame [25 x 8]  
##   
## date datatype station value fl\_m fl\_q  
## (chr) (chr) (chr) (int) (chr) (chr)  
## 1 2015-01-01T00:00:00 AWND GHCND:USW00093815 72   
## 2 2015-01-01T00:00:00 PRCP GHCND:USW00093815 0   
## 3 2015-01-01T00:00:00 SNOW GHCND:USW00093815 0   
## 4 2015-01-01T00:00:00 SNWD GHCND:USW00093815 0   
## 5 2015-01-01T00:00:00 TAVG GHCND:USW00093815 -38 H   
## 6 2015-01-01T00:00:00 TMAX GHCND:USW00093815 28   
## 7 2015-01-01T00:00:00 TMIN GHCND:USW00093815 -71   
## 8 2015-01-01T00:00:00 WDF2 GHCND:USW00093815 240   
## 9 2015-01-01T00:00:00 WDF5 GHCND:USW00093815 240   
## 10 2015-01-01T00:00:00 WSF2 GHCND:USW00093815 130   
## .. ... ... ... ... ... ...  
## Variables not shown: fl\_so (chr), fl\_t (chr)

Since we recently had some snow here let's pull data on snow fall for 2015. We adjust the limit argument (by default ncdc limits results to 25) and identify the data type we want. By sorting we see what days experienced the greatest snowfall (don't worry, the results are reported in mm!).

snow <- ncdc(datasetid='GHCND',   
 startdate = '2015-01-01',   
 enddate = '2015-12-31',   
 limit = 365,  
 stationid='GHCND:USW00093815',  
 datatypeid = 'SNOW',  
 token = key)  
  
snow$data %>%   
 arrange(desc(value))  
## Source: local data frame [365 x 8]  
##   
## date datatype station value fl\_m fl\_q  
## (chr) (chr) (chr) (int) (chr) (chr)  
## 1 2015-03-01T00:00:00 SNOW GHCND:USW00093815 114   
## 2 2015-02-21T00:00:00 SNOW GHCND:USW00093815 109   
## 3 2015-01-25T00:00:00 SNOW GHCND:USW00093815 71   
## 4 2015-01-06T00:00:00 SNOW GHCND:USW00093815 66   
## 5 2015-02-16T00:00:00 SNOW GHCND:USW00093815 30   
## 6 2015-02-18T00:00:00 SNOW GHCND:USW00093815 25   
## 7 2015-02-14T00:00:00 SNOW GHCND:USW00093815 23   
## 8 2015-01-26T00:00:00 SNOW GHCND:USW00093815 20   
## 9 2015-02-04T00:00:00 SNOW GHCND:USW00093815 20   
## 10 2015-02-12T00:00:00 SNOW GHCND:USW00093815 20   
## .. ... ... ... ... ... ...  
## Variables not shown: fl\_so (chr), fl\_t (chr)

This is just an intro to rnoaa as the package offers a slew of data sets to pull from and functions to apply. It even offers built in plotting functions. Use help(package = "rnoaa") to see all that rnoaa has to offer.

#### rtimes

The [rtimes](https://cran.r-project.org/web/packages/rtimes/index.html) package provides an interface to Congress, Campaign Finance, Article Search, and Geographic APIs offered by the New York Times. The data libraries and documentation for the several APIs available can be found [here](http://developer.nytimes.com/docs/). To use the Times' API you'll need to get an API key [here](http://developer.nytimes.com/apps/register).

article\_key <- "4f23572d8..." # truncated  
cfinance\_key <- "ee0b7cef..." # truncated  
congress\_key <- "57b3e8a3..." # truncated

Lets start by searching NY Times articles. With the presendential elections upon us, we can illustrate by searching the least controversial candidate...Donald Trump. We can see that there are 4,566 article hits for the term "Trump". We can get more information on a particular article by subsetting.

library(rtimes)  
  
# article search for the term 'Trump'  
articles <- as\_search(q = "Trump",   
 begin\_date = "20150101",   
 end\_date = '20160101',  
 key = article\_key)  
  
# summary  
articles$meta  
## hits time offset  
## 1 4565 28 0  
  
# pull info on 3rd article  
articles$data[3]  
## [[1]]  
## <NYTimes article>Donald Trumpâs Strongest Supporters: A Certain Kind of Democrat  
## Type: News  
## Published: 2015-12-31T00:00:00Z  
## Word count: 1469  
## URL: http://www.nytimes.com/2015/12/31/upshot/donald-trumps-strongest-supporters-a-certain-kind-of-democrat.html  
## Snippet: In a survey, he also excels among low-turnout voters and among the less affluent and the less educated, so the question is: Will they show up to vote?

We can use the campaign finance API and functions to gain some insight into Trumps compaign income and expenditures. The only special data you need is the [FEC ID](http://www.fec.gov/finance/disclosure/candcmte_info.shtml?tabIndex=2) for the candidate of interest.

trump <- cf\_candidate\_details(campaign\_cycle = 2016,   
 fec\_id = 'P80001571',  
 key = cfinance\_key)  
  
# pull summary data  
trump$meta  
## id name party  
## 1 P80001571 TRUMP, DONALD J REP  
## fec\_uri  
## 1 http://docquery.fec.gov/cgi-bin/fecimg/?P80001571  
## committee mailing\_address mailing\_city  
## 1 /committees/C00580100.json 725 FIFTH AVENUE NEW YORK  
## mailing\_state mailing\_zip status total\_receipts  
## 1 NY 10022 O 1902410.45  
## total\_from\_individuals total\_from\_pacs total\_contributions  
## 1 92249.33 0 96298.97  
## candidate\_loans total\_disbursements begin\_cash end\_cash  
## 1 1804747.23 1414674.29 0 487736.16  
## total\_refunds debts\_owed date\_coverage\_from date\_coverage\_to  
## 1 0 1804747.23 2015-04-02 2015-06-30  
## independent\_expenditures coordinated\_expenditures  
## 1 1644396.8 0

rtimes also allows us to gain some insight into what our locally elected officials are up to with the Congress API. First, I can get some informaton on my Senator and then use that information to see if he's supporting my interest. For instance, I can pull the most recent bills that he is co-sponsoring.

# pull info on OH senator  
senator <- cg\_memberbystatedistrict(chamber = "senate",   
 state = "OH",   
 key = congress\_key)  
senator$meta  
## id name role gender party  
## 1 B000944 Sherrod Brown Senator, 1st Class M D  
## times\_topics\_url twitter\_id youtube\_id seniority  
## 1 SenSherrodBrown SherrodBrownOhio 9  
## next\_election  
## 1 2018  
## api\_url  
## 1 http://api.nytimes.com/svc/politics/v3/us/legislative/congress/members/B000944.json  
  
# use member ID to pull recent bill sponsorship  
bills <- cg\_billscosponsor(memberid = "B000944",   
 type = "cosponsored",   
 key = congress\_key)  
head(bills$data)  
## Source: local data frame [6 x 11]  
##   
## congress number  
## (chr) (chr)  
## 1 114 S.2098  
## 2 114 S.2096  
## 3 114 S.2100  
## 4 114 S.2090  
## 5 114 S.RES.267  
## 6 114 S.RES.269  
## Variables not shown: bill\_uri (chr), title (chr), cosponsored\_date  
## (chr), sponsor\_id (chr), introduced\_date (chr), cosponsors (chr),  
## committees (chr), latest\_major\_action\_date (chr),  
## latest\_major\_action (chr)

It looks like the most recent bill Sherrod is co-sponsoring is S.2098 - Student Right to Know Before You Go Act. Maybe I'll do a NY Times article search with as\_search() to find out more about this bill...an exercise for another time.

So this gives you some flavor of how these API packages work. You typically need to know the data sets and variables requested along with an API key. But once you get these basics its pretty straight forward on requesting the data. Your next question may be, what if the API that I want to get data from does not yet have an R package developed for it?

### httr for All Things Else

Although numerous R API packages are available, and cover a wide range of data, you may eventually run into a situation where you want to leverage an organization's API but an R package does not exist. Enter [httr](https://cran.r-project.org/web/packages/httr/index.html). httr was developed by Hadley Wickham to easily work with web APIs. It offers multiple functions (i.e. HEAD(), POST(), PATCH(), PUT() and DELETE()); however, the function we are most concerned with today is Get(). We use the Get() function to access an API, provide it some request parameters, and receive an output.

To give you a taste for how the httr package works, I'll quickly cover how to use it for a basic key-only API and an OAuth-required API:

* [Key-only API](#key_only) is illustrated by pulling U.S. Department of Education data available on [data.gov](https://api.data.gov/docs/)
* [OAuth-required API](#oauth) is illustrated by pulling tweets from my personal Twitter feed

#### Key-only API

To demonstrate how to use the httr package for accessing a key-only API, I'll illustrate with the [College Scorecard API](https://api.data.gov/docs/ed/) provided by the Department of Education. First, you'll need to [request your API key](https://api.data.gov/signup/).

# truncated key  
edu\_key <- "fd783wmS3Z..."

We can now proceed to use httr to request data from the API with the GET() function. I went to North Dakota State University (NDSU) for my undergrad so I'm interested in pulling some data for this school. I can use the provided [data library](https://collegescorecard.ed.gov/data/documentation/) and [query explanation](https://github.com/18F/open-data-maker/blob/api-docs/API.md) to determine the parameters required. In this example, the URL includes the primary path ("<https://api.data.gov/ed/collegescorecard/>"), the API version ("v1"), and the endpoint ("schools"). The question mark ("?") at the end of the URL is included to begin the list of query parameters, which only includes my API key and the school of interest.

library(httr)  
  
URL <- "https://api.data.gov/ed/collegescorecard/v1/schools?"  
  
# import all available data for NDSU  
ndsu\_req <- GET(URL, query = list(api\_key = edu\_key,  
 school.name = "North Dakota State University"))

This request provides me with every piece of information collected by the U.S. Department of Education for NDSU. To retrieve the contents of this request I use the content() function which will output the data as an R object (a list in this case). The data is segmented into two main components: *metadata* and *results*. I'm primarily interested in the results.

The results branch of this list provides information on lat-long location, school identifier codes, some basic info on the school (city, number of branches, school website, accreditor, etc.), and then student data for the years 1997-2013.

ndsu\_data <- content(ndsu\_req)  
  
names(ndsu\_data)  
## [1] "metadata" "results"  
  
names(ndsu\_data$results[[1]])  
## [1] "2008" "2009" "2006" "ope6\_id" "2007" "2004"   
## [7] "2013" "2005" "location" "2002" "2003" "id"   
## [13] "1996" "1997" "school" "1998" "2012" "2011"   
## [19] "2010" "ope8\_id" "1999" "2001" "2000"

To see what kind of student data categories are offered we can assess a single year. You can see that available data includes earnings, academics, student info/demographics, admissions, costs, etc. With such a large data set, which includes many embedded lists, sometimes the easiest way to learn the data structure is to peruse names at different levels.

# student data categories available by year  
names(ndsu\_data$results[[1]]$`2013`)  
## [1] "earnings" "academics" "student" "admissions" "repayment"   
## [6] "aid" "cost" "completion"  
  
# cost categories available by year  
names(ndsu\_data$results[[1]]$`2013`$cost)  
## [1] "title\_iv" "avg\_net\_price" "attendance" "tuition"   
## [5] "net\_price"  
  
# Avg net price cost categories available by year  
names(ndsu\_data$results[[1]]$`2013`$cost$avg\_net\_price)  
## [1] "other\_academic\_year" "overall" "program\_year"   
## [4] "public" "private"

So if I'm interested in comparing the rise in cost versus the rise in student debt I can simply subset for this data once I've identified its location and naming structure. Note that for this subsetting we use the [magrittr](#pipe) package and the [`sapply](#apply_family) function; both we cover in later chapters but this is just meant to illustrate the types of data available through this API.

library(magrittr)  
  
# subset list for annual student data only  
ndsu\_yr <- ndsu\_data$results[[1]][c(as.character(1996:2013))]  
  
# extract median debt data for each year  
ndsu\_yr %>%  
 sapply(function(x) x$aid$median\_debt$completers$overall) %>%   
 unlist()  
## 1997 1998 1999 2000 2001 2002 2003 2004   
## 13388.0 13856.0 14500.0 15125.0 15507.0 15639.0 16251.0 16642.5   
## 2005 2006 2007 2008 2009 2010 2011 2012   
## 17125.0 17125.0 17125.0 17250.0 19125.0 21500.0 23000.0 24954.5   
## 2013   
## 25050.0  
  
# extract net price for each year  
ndsu\_yr %>%   
 sapply(function(x) x$cost$avg\_net\_price$overall) %>%   
 unlist()  
## 2009 2010 2011 2012 2013   
## 13474 12989 13808 15113 14404

Quite simple isn't it...at least once you've learned how the query requests are formatted for a particular API.

#### OAuth-required API

At the outset I mentioned how OAuth is an authorization framework that provides credentials as proof for access. Many APIs are open to the public and only require an API key; however, some APIs require authorization to account data (think personal Facebook & Twitter accounts). To access these accounts we must provide proper credentials and OAuth authentication allows us to do this. This section is not meant to explain the details of OAuth (for that see [this](http://hueniverse.com/2007/09/05/explaining-oauth/), [this](https://en.wikipedia.org/wiki/OAuth), and [this](http://hueniverse.com/oauth/)) but, rather, how to use httr in times when OAuth is required.

I'll demonstrate by accessing the Twitter API using my Twitter account. The first thing we need to do is identify the OAuth endpoints used to request access and authorization. To do this we can use oauth\_endpoint() which typically requires a *request* URL, *authorization* URL, and *access* URL. httr also included some baked-in endpoints to include LinkedIn, Twitter, Vimeo, Google, Facebook, and GitHub. We can see the Twitter endpoints using the following:

twitter\_endpts <- oauth\_endpoints("twitter")  
twitter\_endpts  
## <oauth\_endpoint>  
## request: https://api.twitter.com/oauth/request\_token  
## authorize: https://api.twitter.com/oauth/authenticate  
## access: https://api.twitter.com/oauth/access\_token

Next, I register my application at <https://apps.twitter.com/>. One thing to note is during the registration process, it will ask you for the *callback url*; be sure to use "<http://127.0.0.1:1410>". Once registered, Twitter will provide you with keys and access tokens. The two we are concerned about are the API key and API Secret.

twitter\_key <- "BZgukbCol..." # truncated  
twitter\_secret <- "YpB8Xy..." # truncated

We can then bundle the consumer key and secret into one object with oauth\_app(). The first argument, appname is simply used as a local identifier; it does not need to match the name you gave the Twitter app you developed at <https://apps.twitter.com/>.

We are now ready to ask for access credentials. Since Twitter uses OAuth 1.0 we use oauth1.0\_token() function and incorporate the endpoints identified and the oauth\_app object we previously named twitter\_app.

twitter\_token <- oauth1.0\_token(endpoint = twitter\_endpts, twitter\_app)  
  
Waiting for authentication in browser...  
Press Esc/Ctrl + C to abort  
Authentication complete.

Once authentication is complete we can now use the API. I can pull all the tweets that show up on my personal timeline using the GET() function and the access cridentials I stored in twitter\_token. I then use content() to convert to a list and I can start to analyze the data.

In this case each tweet is saved as an individual list item and a full range of data are provided for each tweet (i.e. id, text, user, geo location, favorite count, etc). For instance, we can see that the first tweet was by [FiveThirtyEight](http://fivethirtyeight.com/) concerning American politics and, at the time of this analysis, has been favorited by 3 people.

# request Twitter data  
req <- GET("https://api.twitter.com/1.1/statuses/home\_timeline.json",  
 config(token = twitter\_token))  
  
# convert to R object  
tweets <- content(req)  
  
# available data for first tweet on my timeline  
names(tweets[[1]])  
 [1] "created\_at" "id"   
 [3] "id\_str" "text"   
 [5] "source" "truncated"   
 [7] "in\_reply\_to\_status\_id" "in\_reply\_to\_status\_id\_str"   
 [9] "in\_reply\_to\_user\_id" "in\_reply\_to\_user\_id\_str"   
[11] "in\_reply\_to\_screen\_name" "user"   
[13] "geo" "coordinates"   
[15] "place" "contributors"   
[17] "is\_quote\_status" "retweet\_count"   
[19] "favorite\_count" "entities"   
[21] "extended\_entities" "favorited"   
[23] "retweeted" "possibly\_sensitive"   
[25] "possibly\_sensitive\_appealable" "lang"   
  
# further analysis of first tweet on my timeline  
tweets[[1]]$user$name  
[1] "FiveThirtyEight"  
  
tweets[[1]]$text  
[1] "\U0001f3a7 A History Of Data In American Politics (Part 1): William Jennings Bryan to Barack Obama https://t.co/oCKzrXuRHf https://t.co/6CvKKToxoH"  
  
tweets[[1]]$favorite\_count  
[1] 3

This provides a fairly simple example of incorporating OAuth authorization. The httr provides several examples of accessing common social network APIs that require OAuth. I recommend you go through several of these examples to get familiar with using OAuth authorization; see them at demo(package = "httr"). The most difficult aspect of creating your own connections with APIs is gaining an understanding of the API and the arguments they leverage. This obviously requires time and energy devoted to digging into the API documentation and data library. Next its just a matter of trial and error (likely more the latter than the former) to learn how to translate these arguments into httr function calls to pull the data of interest.

Also, note that httr provides several other useful functions not covered here for communicating with APIs (i.e. POST(), BROWSE()). For more on these other httr capabilities see this [quickstart vignette](https://cran.r-project.org/web/packages/httr/vignettes/quickstart.html).

## Additional Resources

As I stated in the outset, this chapter is meant to provide an introduction to basic web scraping capabilities in R. This area is vast and complex and this chapter will far from provide you expertise level insight. To advance your knowledge in webscraping with R [*Automated Data Collection with R*](http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX%22) and [*XML and Web Technologies for Data Sciences with R*](http://www.amazon.com/XML-Web-Technologies-Data-Sciences/dp/1461478995) offer the most detailed resources available. But this chapter should be enough to get your curiousity piqued and to start pulling data from the tangled masses of online data.

# Exporting Data

Although getting data into R is essential, getting data out of R can be just as important. Whether you need to export data or analytic results simply to store, share, or feed into another system it is generally a straight forward process. This section will cover how to export data to [text files](#export_txt), [Excel files](#export_excel) (along with some additional formatting capabilities), and [save to R data objects](#save_object). In addition to the the commonly used base R functions to perform data importing, I will also cover functions from the popular readr and xlsx packages along with a lesser known but useful r2excel package for Excel formatting.

## Writing data to text files

As mentioned in the importing data section, text files are a popular way to hold and exchange tabular data as almost any data application supports exporting data to the CSV (or other text file) formats. Consequently, exporting data to a text file is a pretty standard operation. Plus, since you've already learned how to import text files you pretty much have the basics required to write to text files...we just use a slightly different naming convention.

Similar to the examples provided in the importing text files section, the two main groups of functions that I will demonstrate to write to text files include base R functions and readr package functions.

### Base R functions

write.table() is the multipurpose work-horse function in base R for exporting data. The functions write.csv() and write.delim() are special cases of write.table() in which the defaults have been adjusted for efficiency. To illustrate these functions let's work with a data frame that we wish to export to a CSV file in our working directory.

df <- data.frame(var1 = c(10, 25, 8),   
 var2 = c("beer", "wine", "cheese"),   
 var3 = c(TRUE, TRUE, FALSE),  
 row.names = c("billy", "bob", "thornton"))  
  
df  
## var1 var2 var3  
## billy 10 beer TRUE  
## bob 25 wine TRUE  
## thornton 8 cheese FALSE

To export df to a CSV file we can use write.csv(). Additional arguments allow you to exclude row and column names, specify what to use for missing values, add or remove quotations around character strings, etc.

# write to a csv file  
write.csv(df, file = "export\_csv")  
  
# write to a csv and save in a different directory  
write.csv(df, file = "/folder/subfolder/subsubfolder/export\_csv")  
  
# write to a csv file with added arguments  
write.csv(df, file = "export\_csv", row.names = FALSE, na = "MISSING!")

In addition to CSV files, we can also write to other text files using write.table and write.delim().

# write to a tab delimited text files  
write.delim(df, file = "export\_txt")  
  
# provides same results as read.delim  
write.table(df, file = "export\_txt", sep="\t")

### readr package

The readr package uses write functions similar to base R. However, readr write functions are about twice as fast and they do not write row names. One thing to note, where base R write functions use the file = argument, readr write functions use path =.

library(readr)  
  
# write to a csv file  
write\_csv(df, path = "export\_csv2")  
  
# write to a csv and save in a different directory  
write\_csv(df, path = "/folder/subfolder/subsubfolder/export\_csv2")  
  
# write to a csv file without column names  
write\_csv(df, path = "export\_csv2", col\_names = FALSE)  
  
# write to a txt file without column names  
write\_delim(df, path = "export\_txt2", col\_names = FALSE)

## Writing data to Excel files

As previously mentioned, many organizations still rely on Excel to hold and share data so exporting to Excel is a useful bit of knowledge. And rather than saving to a .csv file to send to a co-worker who wants to work in Excel, its more efficient to just save R outputs directly to an Excel workbook. Since I covered importing data with the xlsx package, I'll also cover exporting data with this package. However, the readxl package which I demonstrated in the importing data section does not have a function to export to Excel. But there is a lesser known package called r2excel that provides exporting and formatting functions for Excel which I will cover.

### xlsx package

Saving a data frame to a .xlsx file is as easy as saving to a .csv file:

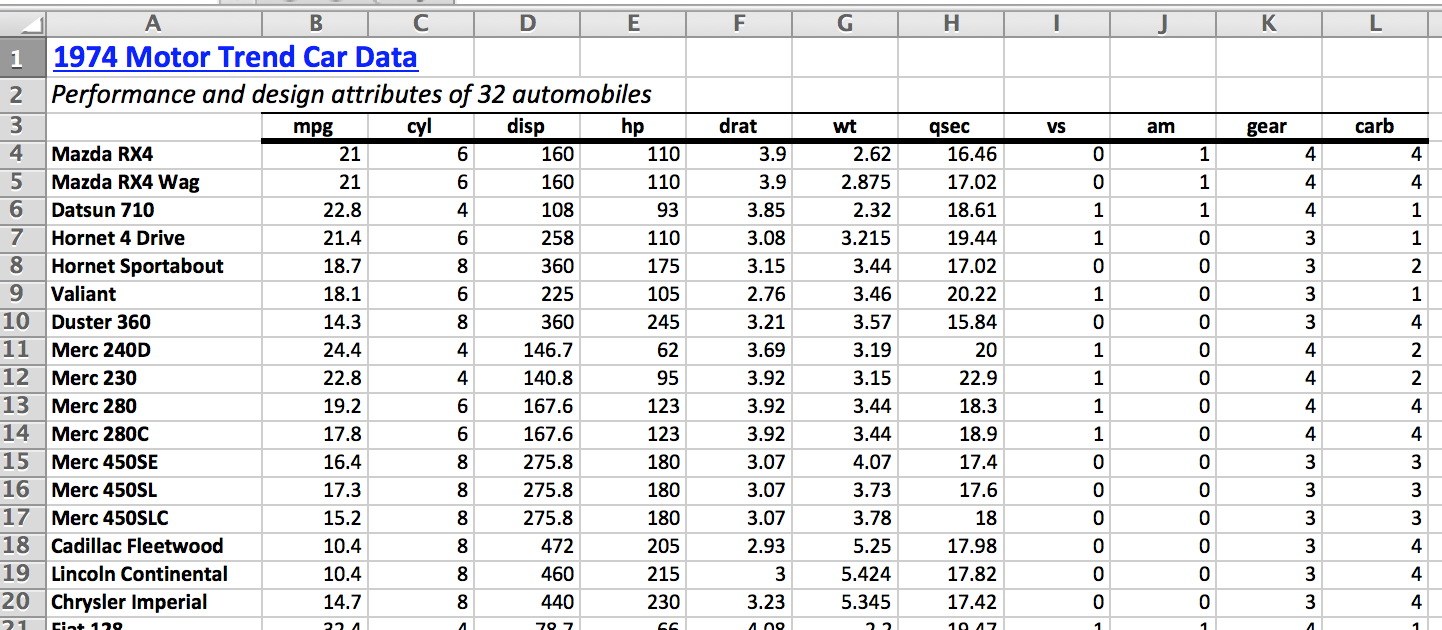
library(xlsx)  
  
# write to a .xlsx file  
write.xlsx(df, file = "output\_example.xlsx")  
  
# write to a .xlsx file without row names  
write.xlsx(df, file = "output\_example.xlsx", row.names = FALSE)

In some cases you may wish to create a .xlsx file that contains multiple data frames. In this you can just create an empty workbook and save the data frames on seperate worksheets within the same workbook:

# create empty workbook  
multiple\_df <- createWorkbook()  
  
# create worksheets within workbook  
car\_df <- createSheet(wb = multiple\_df, sheetName = "Cars")  
iris\_df <- createSheet(wb = multiple\_df, sheetName = "Iris")  
  
# add data frames to worksheets; for this example I use the  
# built in mtcars and iris data frames  
addDataFrame(x = mtcars, sheet = car\_df)  
addDataFrame(x = iris, sheet = iris\_df)  
  
# save as a .xlsx file   
saveWorkbook(multiple\_df, file = "output\_example\_2.xlsx")

By default this saves the row and column names but this can be adjusted by adding col.names = FALSE and/or row.names = FALSE to the addDataFrame() function. There is also the ability to do some formatting with the xlsx package. The following provides several examples of how you can edit titles, subtitles, borders, column width, etc.[[5]](#footnote-144) Although at first glance this can appear tedious for simple Excel editing, the real benefits present themselves when you integrate this editing into automated analyses.

# create new workbook  
wb <- createWorkbook()  
  
#--------------------  
# DEFINE CELL STYLES   
#--------------------  
# title and subtitle styles  
title\_style <- CellStyle(wb) +   
 Font(wb, heightInPoints = 16,  
 color = "blue",   
 isBold = TRUE,   
 underline = 1)  
  
subtitle\_style <- CellStyle(wb) +   
 Font(wb, heightInPoints = 14,  
 isItalic = TRUE,  
 isBold = FALSE)  
  
# data table styles  
rowname\_style <- CellStyle(wb) +  
 Font(wb, isBold = TRUE)  
  
colname\_style <- CellStyle(wb) +  
 Font(wb, isBold = TRUE) +  
 Alignment(wrapText = TRUE, horizontal = "ALIGN\_CENTER") +  
 Border(color = "black",  
 position = c("TOP", "BOTTOM"),  
 pen = c("BORDER\_THIN", "BORDER\_THICK"))  
  
#-------------------------  
# CREATE & EDIT WORKSHEET   
#-------------------------  
# create worksheet  
Cars <- createSheet(wb, sheetName = "Cars")  
  
# helper function to add titles  
xlsx.addTitle <- function(sheet, rowIndex, title, titleStyle) {  
 rows <- createRow(sheet, rowIndex = rowIndex)  
 sheetTitle <- createCell(rows, colIndex = 1)  
 setCellValue(sheetTitle[[1,1]], title)  
 setCellStyle(sheetTitle[[1,1]], titleStyle)  
}  
  
# add title and sub title to worksheet  
xlsx.addTitle(sheet = Cars, rowIndex = 1,   
 title = "1974 Motor Trend Car Data",  
 titleStyle = title\_style)  
  
xlsx.addTitle(sheet = Cars, rowIndex = 2,   
 title = "Performance and design attributes of 32 automobiles",  
 titleStyle = subtitle\_style)  
  
# add data frame to worksheet  
addDataFrame(mtcars, sheet = Cars, startRow = 3, startColumn = 1,  
 colnamesStyle = colname\_style,   
 rownamesStyle = rowname\_style)  
  
# change row name column width  
setColumnWidth(sheet = Cars, colIndex = 1, colWidth = 18)  
  
# save workbook  
saveWorkbook(wb, file = "output\_example\_3.xlsx")

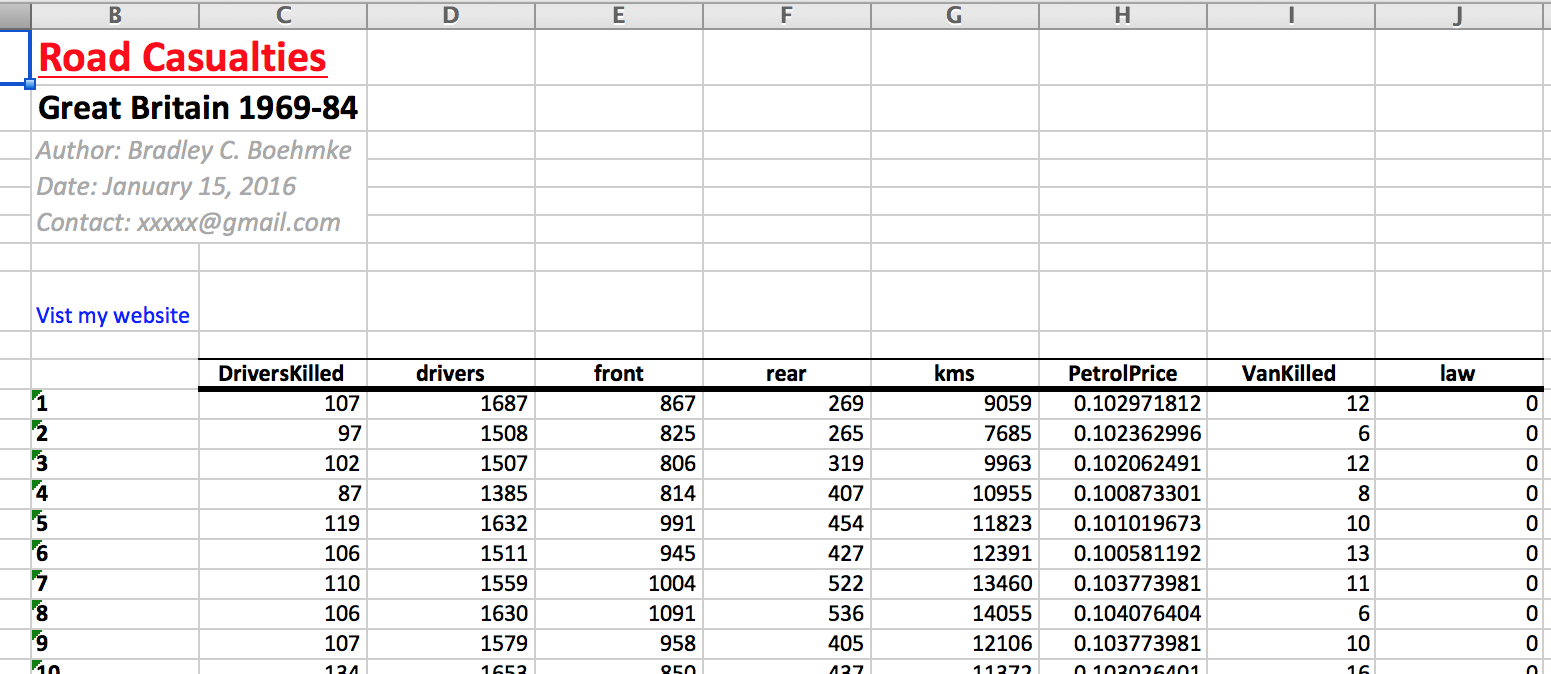


Formatted Excel Output Example 1

### r2excel package

Although Formatting Excel files using the xlsx package is possible, the last section illustrated that it is a bit cumbersome. For this reason, [A. Kassambara](https://github.com/kassambara) created the r2excel package which depends on the xlsx package but provides easy to use functions for Excel formatting. The following provides a simple example but you can find many additional formatting functions [here](http://www.sthda.com/english/wiki/r2excel-read-write-and-format-easily-excel-files-using-r-software)

# install.packages("devtools")  
devtools::install\_github("kassambara/r2excel")  
library(r2excel)  
  
# create new workbook  
wb <- createWorkbook()  
  
# create worksheet  
Casualties <- createSheet(wb, sheetName = "Casualties")  
  
# add title  
xlsx.addHeader(wb, sheet = Casualties,   
 value = "Road Casualties",  
 level = 1,   
 color = "red",   
 underline = 1)  
  
# add subtitle  
xlsx.addHeader(wb, sheet = Casualties,   
 value = "Great Britain 1969-84",  
 level = 2,   
 color = "black")  
  
# add author information  
author = paste("Author: Bradley C. Boehmke \n",  
 "Date: January 15, 2016 \n",  
 "Contact: xxxxx@gmail.com", sep = "")  
  
xlsx.addParagraph(wb, sheet = Casualties,  
 value = author,   
 isItalic = TRUE,   
 colSpan = 2,   
 rowSpan = 4,   
 fontColor = "darkgray",   
 fontSize = 14)  
  
# add hyperlink  
xlsx.addHyperlink(wb, sheet = Casualties,   
 address = "http://bradleyboehmke.github.io/",   
 friendlyName = "Vist my website", fontSize = 12)  
  
xlsx.addLineBreak(sheet = Casualties, 1)  
  
# add data frame to worksheet, I'm using the built in  
# Seatbelt data which you can view at data(Seatbelt)  
xlsx.addTable(wb, sheet = Casualties, data = Seatbelts, startCol = 2)  
   
# save the workbook to an Excel file  
saveWorkbook(wb, file = "output\_example\_4.xlsx")



Formatted Excel Output Example 2

## Saving data as an R object file

Sometimes you may need to save data or other R objects outside of your workspace. You may want to share R data/objects with co-workers, transfer between projects or computers, or simply archive them. There are three primary ways that people tend to save R data/objects: as .RData, .rda, or as .rds files.

.rda is just short for .RData, therefore, these file extensions represent the same underlying object type. You use the .rda or .RData file types when you want to save several, or all, objects and functions that exist in your global environment. On the other hand, if you only want to save a single R object such as a data frame, function, or statistical model results its best to use .rds file type. You can use .rda or .RData to save a single object but the benefit of .rds is it only saves a representation of the object and not the name whereas .rda and .RData save the both the object and its name. As a result, with .rds the saved object can be loaded into a named object within R that is different from the name it had when originally saved. The following illustrates how you save R objects with each type.

# save() can be used to save multiple objects in you global environment,  
# in this case I save two objects to a .RData file  
x <- stats::runif(20)  
y <- list(a = 1, b = TRUE, c = "oops")  
save(x, y, file = "xy.RData")  
  
# save.image() is just a short-cut for ‘save my current workspace’,  
# i.e. all objects in your global environment  
save.image()  
  
# save a single object to file  
saveRDS(x, "x.rds")  
  
# restore it under a different name  
x2 <- readRDS("x.rds")  
identical(x, x2)  
[1] TRUE

## Additional resources

The following provides additional resources for exporting data:

* [R data import/export manual](https://cran.r-project.org/doc/manuals/R-data.html)
* [WriteXLS package](https://cran.r-project.org/web/packages/WriteXLS/WriteXLS.pdf)
* [XLConnect package](https://cran.r-project.org/web/packages/XLConnect/vignettes/XLConnect.pdf)

1. In [Automated Data Collection with R](http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX%22) Munzert et al. state that "[t]he first way to get data from the web is almost too banal to be considered here and actually not a case of web scraping in the narrower sense." [↑](#footnote-ref-55)
2. An example is provided in [Automated Data Collection with R](http://www.amazon.com/Automated-Data-Collection-Practical-Scraping/dp/111883481X/ref=pd_sim_14_1?ie=UTF8&dpID=51Tm7FHxWBL&dpSrc=sims&preST=_AC_UL160_SR108%2C160_&refRID=1VJ1GQEY0VCPZW7VKANX%22) in which they use a similar approach to extract desired CSV files scattered throughout the [Maryland State Board of Elections websiteMaryland State Board of Elections website](http://www.elections.state.md.us/elections/2012/election_data/index.html). [↑](#footnote-ref-67)
3. You can learn more about selectors at [flukeout.github.io](http://flukeout.github.io/) [↑](#footnote-ref-81)
4. You can simply assess the name of the ID in the highlighted element or you can right click the highlighted element in the developer tools window and select *Copy selector*. You can then paste directly into `html\_nodes() as it will paste the exact ID name that you need for that element. [↑](#footnote-ref-85)
5. This example was derived from [STHDA](http://www.sthda.com/english/). Additional options, such as adding plot outputs can be found at STHDA and also in the *XML and Web Technologies for Data Sciences with R* book. [↑](#footnote-ref-144)