Chapter 1

Processing Data

Speeches contain a wealth of information. As humans, we are taught to understand verbal and written communication – pick out the nouns, verbs, and adjectives, then combine the information to decipher meaning. Take the following excerpt from the 2010 State of the Union:

> Now, one place to start is serious financial reform. Look, I am not interested in punishing banks. I'm interested in protecting our economy. A strong, healthy financial market makes it possible for businesses to access credit and create new jobs. It channels the savings of families into investments that raise incomes. But that can only happen if we guard against the same recklessness that nearly brought down our entire economy. We need to make sure consumers and middle-class families have the information they need to make financial decisions. We can't allow financial institutions, including those that take your deposits, to take risks that threaten the whole economy.

We can infer that the excerpt was written in a time of economic hardship and the author recognizes that consumers and middle class need better information about the financial markets. Despite these clear inferences, to many, text is not be considered data. Our minds analyze the text and infer what the author is signaling, sifting for key terms and phrases. The majority of words convey little to no meaning – they help make language more fluid. We can cut to the core meaning by dropping the filler:

> ~~Now, one place to start is serious~~ financial reform. ~~Look, I am not interested in~~ punishing banks. ~~I'm interested in~~ protecting our economy. ~~A~~ strong, healthy financial market ~~makes it possible for~~ businesses ~~to access~~ credit ~~and~~ create new jobs. ~~It channels the~~ savings of families ~~into~~ investments ~~that~~ raise incomes. ~~But that can only happen if we guard against the same~~ recklessness ~~that nearly brought down our entire~~ economy. ~~We need to make sure~~ consumers ~~and~~ middle-class families ~~have the information they need to make~~financial decisions. ~~We can't allow~~ financial institutions, ~~including those that take your~~ deposits,~~to take risks that threaten the whole~~ economy.

The frequency of each unique word in each document can be tabulated, then structured into a matrix where each row represents a document and each column contains the frequency of each word. Of course, this process could be done manually, but imagine sorting through all 7,304 words in the 2010 address or scaling the process to the roughly *1.9 million words* in State of the Union addresses between 1790 and 2016. All the steps required to convert unstructured text into usable data can be done with a little bit of planning, technical imagination and data manipulation.

Every little detail about the data needs to be meticulously converted into a usable form. Let's take one line from above and dissect the changes that need to be made:

> "We need to make sure consumers and middle-class families have the information they need to make financial decisions. We can't allow financial institutions, including those that take your deposits, to take risks that threaten the whole economy."

We then turn everything into lower case so all letters of the alphabet are read the same. Stylistic elements like carriage returns and new line markers, while not visible in reading mode, should be removed.

> “we need to make sure consumers and middle-class families have the information they need to make financial decisions. we can’t allow financial institutions, including those that take your deposits, to take risks that threaten the whole economy.”

Then, we get rid of punctuation by substituting values with empty quotations (`""`). And if possible, contractions like “can’t” should be reconciled with their fully spelled counterparts.

> “we need to make sure consumers and middleclass families have the information they need to make financial decisions we cant allow financial institutions including those that take your deposits to take risks that threaten the whole economy”

Each space between each word can be used as a *delimiter* that can be used to parse a string of words.

Table 1.1: Terms

we	families	financial	those	that
need	have	decisions	that	threaten
to	the	we	take	the
make	information	cant	your	whole
sure	they	allow	deposits	economy
consumers	need	financial	to	
and	to	institutions	take	
middleclass	make	including	risks	

There are words that do not add much value as they are commonplace filler. In text processing, these words are known as *stop words*. In each domain, the list of stop words likely differs, thus data scientists may need to build a customized list. For simplicity, we rely on a stop words list that is used in mySQL – an open source relational database management system. The result is the list of remaining words.

Table 1.2: Terms after removing stop words

make	information	financial	risks
consumers	make	institutions	threaten
middleclass	financial	including	economy
families	decisions	deposits	

We can aggregate the data into a form that is meaningful to answer a research question. For example, the frequency of words may provide a clue as to what the text is about. In this case, each “financial” and “make” appear twice in the text, perhaps indicating that there is an orientation towards action (make) for financial considerations.

Table 1.3: Term Frequencies

Term	Freq	Term	Freq
financial	2	including	1
make	2	information	1
consumers	1	institutions	1
decisions	1	middleclass	1
deposits	1	risks	1
economy	1	threaten	1
families	1		

Text processing is just one aspect of readying data for use – there is far more that underlies manipulating data.

There are entire texts dedicated to data engineering and manipulating data. In this chapter, we offer a broad overview of the mechanics of data manipulation. We begin at a microscopic level, focusing on loading raw data and how to treat and transform data values into useful, analysis-ready information. Then, we take a step back to focus on the macroscopic structure of data – how to sculpt the structure (rows and columns) of a data set into a format that is appropriate for a desire use case. We close the chapter by introducing efficiency augmenting programming paradigms, namely control structures (e.g. loops, if statements) that reduce repetitive coding and user-written functions that allow scripts to be generalized and re-used.

1.1 Load and Assembly

The first step of any data project is to load and assemble a data set. The input-output (I/O) or loading of data can be tricky task for a new-to-programming rookie, requiring a particularly large investment of time to search for the right way to open a file using code. Upon finding a solution, the analyst may dismiss any concern for ordering and structuring code, resulting in ad hoc spaghetti-like code. In the course of writing this text, the authors have come across a number of precarious data practices among new programmers. In particular, we have found that:

- Analysts will often manually open files and delete unnecessary rows at the top of a file believing that functions may not be able to read in files.
- The number of columns differ between two files that need to be appended, requiring columns to be deleted or empty columns added before loading.
- Column names are not as designed, leading to manual editing of headers.
- Edits often overwrite the original file. This means that an edit were erroneous, there is no hope in restoring the original data.
- Manual edits are learned and shared as an oral history rather than a codified procedure. If processing steps are not captured in code, the habits are not reproducible, leaving data quality with analyst-specific quirks.

These manual steps become tedious when presented with hundreds if not thousands of files.

Loading and assembling data sets should follow a few ideal practices. First, the raw data should never be modified – it is the gold copy. Second, the data should be read into programming environment where all processing is conducted and the procedures are coded as a generalizable program. Third, data processing should be conducted in a way to support different use cases. A modest amount of standardizing processing should be conducted so the data is usable and stored to a new file or to a database, but more complex processing should be done with a use case in mind.

In this section, we illustrate simple workflows for loading and assembling data set. Our goal is to load gas price data from the US Energy Information Administration (EIA) and determine if prices from two ports – New York Harbor and the US Gulf Coast – are correlated. To do this, we will need to assemble a time series data set taking the following:

- Load a CSV of spot gas prices from the US Gulf Coast as a data frame.
- Load two Excel worksheets of spot gas prices from NY Harbor from two different time windows as data frames, then append one to the other.
- Combine the two into a single data frame as illustrated below, then calculate the Pearson Correlation coefficient.

1.1.1 Loading Files

There are a multitude of data storage formats in use such as extensible markup language (XML), comma separated values (CSV), Excel files among others. Each has its quirks. Some require special software to load while others simply need time and patience to crack the code. Fortunately, R is equipped to load virtually all data formats. Below is a recommended set of functions that are easy to use and flexible.

Table 1.4: Recommended functions to load an assortment of files.

Function	Package	Description
read_excel	readxl	Load Excel files into a data frame. Note that with excel files, generally need to specify which sheet to load using the *sheet* argument and may need to skip a number of rows at the beginning of worksheet to indicate the header row using *skip*.
read_csv	readr	Load any comma separated file into data frame format.
read_delim	readr	Load delimited file using any delimiter such as tab delimited (“\”̑) using the *sep* argument.
readLines	base R	Read a text file, line by line. This is helpful for working with free form text.
xmlToDataFrame	XML	Read XML into a data frame if the structure is simple and flat. Note that XML files tend to have a more complex hierarchical structure.
xmlToList	XML	If the XML is complicated, read each element as a list.
fromJSON	rjson	Read JSON into list. If JSON is a flat, non-hierarchical file, then the resulting object can be simply converted into a data frame.
read_dta	haven	Load Stata data file.
read_sas	haven	Load SAS data files.
read_spss	haven	Load SPSS data files.
load	base R	Load saved R data set that can contain multiple objects. Note that this does not need to be assigned to another object.
readRDS	base R	Read individually saved R object.

Table 1.5: Target format

date	price1	price2
2016-12-28	1.739	1.702
2016-12-29	1.729	1.703
2016-12-30	1.722	1.699

Loading CSVs. Before reading in a file, it is generally a good idea to open the file in a code editor to take a peek at the structure, but “blind loading” the file can also give clues as well. Let’s start with the CSV `doe_usgulf.csv` using the `read_csv` function from the `readr` package, assign the data to the object `gulf`, then inspect the first few rows using `head`.

```r
pacman::p_load(readr)
gulf <- read_csv("data/doe_usgulf.csv")
head(gulf, 3)
```

```
## # A tibble: 3 x 2
##   Date   `U.S. Gulf Coast Conventional Gasoline Regular Spot Price FOB (Do~
##   <chr>                                                                <dbl>
## 1 1/2/14                                                                2.52
## 2 1/3/14                                                                2.49
## 3 1/6/14                                                       2.52
```

The data were successfully loaded as a *tibble*, which is a variant of a data frame designed for more efficient data manipulation. Unlike data frames, tibbles read headers as-is, thus requiring the analyst to make cosmetic changes. Generally, it is good practice to keep column names as the same case (lower case) and avoid the use of spaces and punctuation with the exception of periods. We can edit the column headers, which can be called as a vector using `colnames(gulf)`, by assigning a new string vector that turns all letters into lower case (`tolower`) and replaces substrings (`gsub`) that are either spaces or punctuation (`[[:space:][:punct:]]`) with periods. These string manipulation commands will be covered in more detail later in this chapter.

```
colnames(gulf) <- tolower(gsub("[[:space:][:punct:]]", ".", colnames(gulf)))
```

However, as these column names are quite long, we can opt to overwrite them with something more concise. Notice that we did not need to open the CSV and manually edit the column names.

```
colnames(gulf) <- c("date", "gulf.price")
```

Loading an Excel file. Excel files tend to have a few more curve balls than CSVs. While Excel files are more readable due to aesthetic and cosmetic styling, parsing can be a challenge. Using `read_excel` from the `readxl` package, we take a first look at the first sheet (`sheet = 1`) of our Excel file `doe_ny.xlsx`, then use `head` to examine what was loaded. Unlike our CSV, our first attempt requires a tweak: The headers are located in the second row of the sheet, which has the implication that the first row needs to be skipped.

```
pacman::p_load(readxl)
attempt1 <- read_excel("data/doe_ny.xlsx", sheet = 1)
head(attempt1, 3)
```

```
## # A tibble: 3 x 4
##   Sourcekey EER_EPMRU_PF4_Y35NY_DPG                                 X__1  X__2
##   <chr>     <chr>                                                   <lgl> <lgl>
## 1 Date      New York Harbor Conventional Gasoline Regular Spot~ NA    NA
## 2 41641     2.718                                                   NA    NA
## 3 41642     2.6709999999999998                                      NA    NA
```

At this point, some may be tempted to manually delete the first row. This need not be the case as R load functions are furnished with a `skip` argument – simply specify the number of rows to skip. Below, we read in the first and second sheets of `doe_ny.xlsx`, skipping the first row in each, then rename the headers.

```
#Load first two sheets
sheet1 <- read_excel("data/doe_ny.xlsx", sheet = 1, skip = 1)
sheet2 <- read_excel("data/doe_ny.xlsx", sheet = 2, skip = 1)

#Rename columns
colnames(sheet1) <- c("date", "ny.price")
colnames(sheet2) <- c("date", "ny.price")
```

Appending data. It turns out that *sheet1* and *sheet2* are from the same time series of spot prices covering different time windows. We construct a complete data frame by appending one data frame to the other using the `rbind` function.

```
  ny <- rbind(sheet1, sheet2)
```

The above scenario is the ideal: two data frames with the same number of columns and the same column names. But what if one data frame has at least one more column than the other? The `rbind.fill` function in the `plyr` package appends the two data frames together, filling any additional missing columns with `NA` values. Below, rather than loading the entire `plyr` library, we can selectively load functions using the double colon operator "`::`".

```
  ny <- plyr::rbind.fill(sheet1, sheet2)
```

Note that if two columns in two different data frames represent the same concept but have different names, `rbind.fill` does not know to make the connection. Column names are interpreted literally, meaning that it is of the utmost importance that headers are exactly labeled.

Combining data by rows. The two data frames `gulf` and `ny` contain the same number of rows. If they are in the same exact desired order and have the same number of rows, we can use the `cbind` function to join the data sets by rows. The `cbind` function will join all columns in both data frames together, including the dates. To ensure the data is neatly organized, we keep all columns from the `gulf` data set and only the second column from the `ny` data set. The resulting data frame contains three columns containing daily spot prices from two sources.

```
#Combine data
  prices <- cbind(gulf, ny = ny[,2])
```

We calculate the correlation using the `cor` function finding a value of $\rho = 0.9896$ – the two sets of prices are highly correlated as may be expected. Note that `cor` only accepts numeric values, thus the date column (the first column) is temporarily dropped.

```
#Calculate correlation
  cor(prices[,-1])
```

```
##            gulf.price       ny
## gulf.price   1.000000 0.989617
## ny           0.989617 1.000000
```

1.2 Manipulating values

More often than not, working with raw data means working with string values. In fact, dates are often assumed to be text characters upon loading. It is easy to imagine that necessary data are embedded in a sentence. For example, below is a vector of four string values:

```
  budget <- c("Captain's Log, Stardate 1511.8. I have $10.20 for a big galactic mac.",
            "The ensign has $1,20 in her pocket.",
            "The ExO has $0.25 left after paying for overpriced warp core fuel.",
            "Chief medical officer is the high roller with $53,13.")
```

What if we need to calculate the total amount of pocket change available to buy galactic big macs? All four elements contain dollar values that can be extracted using *regular expressions* or *regex* – a series of characters that describe a regularly occurring text pattern.

To retrieve the dollar amounts embedded in the text, we first need to replace commas with a period using `str_replace_all()`, assigning the result to a new object `new`. Note that in some regions, such as Europe, commas are used as decimals rather than periods.

```
pacman::p_load(stringr)
new <- str_replace_all(budget, ",", "\\.")
```

Second, the dollar amounts need to be defined as a string pattern: a dollar sign followed by one to two digits, followed by a period, then another two digits ("`\\$\\d{1,2}\\.\\d{2}`"). Using the `str_extract_all` function in the `stringr` package, we extract all matching substrings, specifying `simplify = TRUE` to return a vector:

```
funds <- str_extract_all(new, "\\$\\d{1,2}\\.\\d{2}", simplify = TRUE)
print(funds)
```

```
##      [,1]
## [1,] "$10.20"
## [2,] "$1.20"
## [3,] "$0.25"
## [4,] "$53.13"
```

Third, we should replace dollar sign with blank, then strip out any leading white space using `str_trim()`.

```
funds <- str_replace_all(funds, "\\$","")
funds <- str_trim(funds)
print(funds)
```

```
## [1] "10.20" "1.20"  "0.25"  "53.13"
```

Lastly, we convert the character vector to numeric values using `as.numeric`, then sum the vector.

```
money <- as.numeric(funds)
print(paste0("Total galactic big mac funds = $", sum(money)))
```

```
## [1] "Total galactic big mac funds = $64.78"
```

In steps one through three, you will have noticed that the characters `"$"`, `"."`, and `"d"` were preceded by double backslash. These are known as *escaped characters* as the double backslash preceding the characters changes their meanings. In step two, a sequence of unusual characters ("`\\$\\d{1,2}\\.\\d{2}`") was used to find the "`$x.xx`" pattern, which can be broken into specific commands:

- `\\$` is a dollar sign.
- `\\d{1,2}` is a series of numerical characters that is between one to two digits long.
- `\\.` is a period.
- `\\d{2}` is a series of numerical characters that is exactly two digits long.

Mastering *regex* is a productivity multiplier, opening the possibility of ultra-precise text replacement, extraction, and other manipulation. There are plenty of scenarios where raw data is not quality controlled and mass errors plague the usefulness of the data. An analyst may spend days if not weeks or months cleaning data by hand (or rather through find and replace). With regex, haphazard cleaning is no longer an issue. To make the most of regex requires a command of both *text manipulation functions* that are designed to interpret regex as well as *regex* itself.

1.2.1 Text manipulation functions

Find and replace functionality are common in word processing and spreadsheet software, but are not particularly efficient with large data and complex string patterns. The following text manipulation functions simplify find and replace operations, whether for extracting data or editing specific values. We provide two sets of functions. The first are core functions that are furnished with R – these closely resemble string manipulation functions implemented in most programming languages. While they are powerful, they at times may require some additional trial and error to use. Alternatively, the `stringr` package simplifies text manipulation and syntax, and in some case extends functionality. For the most part, we will use `stringr` in this text.

Table 1.6: Recommended text manipulation functions.

Description	Base R	stringr
Returns either the index position of a matched string or the string containing the matched portion.	grep	str_which
Returns a logical vector indicating if a matched string was found.	grepl	str_detect
Searches for a specified pattern and replaces with user-specified substring.	gsub	str_replace_all
Remove matched pattern from string.	gsub	str_remove_all
Returns the first position of matched pattern in a string.	regexpr	str_locate
Returns the position of all matched patterns in a string.	gregexpr	str_locate_all
Extract substring based on matched pattern. Note that regmatches is used in conjunction with regexpr or gregexpr.	regmatches	str_extract_all
Splits strings into a list of values based on a delimiter.	strsplit	str_split
Extract substring based on start and end position.	substr	str_sub
Trim whitespace on either end of string (excessive spaces)	trimws	str_trim
Returns number of characters in string	nchar	str_length
Returns the number of matched patterns.		str_count
Convert to upper case	toupper	str_to_upper
Convert to lower case.	tolower	str_to_lower
Convert to title case.		str_to_title
Pad string (e.g. add leading zeros to string)	sprintf	str_pad

Complex "find" operations are possible with stringr functions. To illustrate, below we have four sentences that indicate when four US laws were signed.

```
laws <- c(". Dodd-Frank Act was signed into federal law on July 21, 2010.",
    "Glass-Steagall Act was signed into federal law by FDR on June 16, 1933",
    "Hatch Act went into effect on August 2, 1939",
    "Sarbanes-Oxley Act was signed into law on July 30, 2002")
```

Suppose we need to find acts that are named for two congressmen. The `str_detect()` function produces a Boolean vector if a regex pattern `[A-z]{1,}-[A-z]{1,}` is matched for each element in the string vector `laws`. This is particularly useful if we would like to identify and subset elements that match a text pattern.

```
  str_detect(laws, "[A-z]{1,}-[A-z]{1,}")
```

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

Otherwise, we can extract the specific senator names using `str_extract_all`, returning a vector of string values.

```
  str_extract_all(laws, "[A-z]{1,}-[A-z]{1,}", simplify = TRUE)
```

```
##      [,1]
## [1,] "Dodd-Frank"
## [2,] "Glass-Steagall"
## [3,] ""
## [4,] "Sarbanes-Oxley"
```

1.2.2 Regular Expressions (regex)

The secret ingredient of the text manipulation functions is the regex expressions that dictate a text pattern. With only a few characters, even the most complex pattern can be concisely described. There are a few basic building blocks of regex that make text search possible. There are *character classes* that define specific types of characters (e.g. numbers, letters, punctuation) quantifies that indicate how long a pattern should be, escape characters that allow certain characters to hold dual meanings, among others. But first, there are a few basic characters that make both simple and complex pattern searches possible and knowledge of these cleverly designed operators may go a long way:

(1) Alternatives (e.g. “OR” searches) can be surfaced by using a pipe “`|`”. For example, a string search for “Bob or Moe” can be represented as “Bob|Moe”.

(2) The extent of a search is denoted by parentheses `()`. For example, a string search for “Jenny” or an alternative spelling like Jenny would be represented as “Jenn(y|i)”."

(3) A search for one specific character should be placed between square brackets `[]`.

(4) The length of a match is specified using curly brackets `{}`.

In New York City, the famed avenue *Broadway* is may be written and abbreviated in a number of ways. The vector `streets` contains a few instances of spellings of Broadway mixed in with other streets that start with the letter `B`.

```
#A sampling of street names
  streets <- c("Bruckner Blvd", "Bowery", "Broadway", "Bway", "Bdway",
       "Broad Street", "Bridge Street", "B'way")

#Search for two specific options
  str_detect(streets, "Broadway|Bdway")

#Search for two variations of Broadway
  str_detect(streets, "B(road|')way")

#Search for cases where either d or apostrophe are between B and way
  str_detect(streets, "B[d']way")
```

1.2.2.1 Escaped Characters

Quite a few single characters hold a special meaning in addition to the literal meaning. To disambiguate their meaning, a backslash precedes these characters to denote the alternative meaning. A few include:

- `\n`: new line
- `\r`: carriage return
- `\t`: tab
- `\'`: single quote when in a string enclosed in single quotes (`'Nay, I can\'t'`)
- `\"`: double quote when in a string enclosed in double quotes (`"I have a \"guy\"."`)

In other cases, double backslashes should be used:

- `\\.`: period. Otherwise, un-escaped periods indicate searches for *any* single character.
- `\\$`: dollar sign. A dollar sign without backslashes indicates to find patterns at the end of a string.

1.2.2.2 Character Classes

A *character class* or *character set* is used to identify specific characters within a string. How would one represent "12.301.1034" or "?!?!?!"? One or more of the following character classes can do the job:

- `[:punct:]`: is a catch all for any and all punctuation such as periods, commas, semicolons, etc. To find a specific punctuation, simply enclose the characters between two brackets. For example, to find only commas and carrots, use "`[<>,]`".
- `[:alpha:]`: Alphabetic characters such as a, b, c, etc. With other languages including R, searches for any letter combinations are denoted as `[a-z]` for lower case, `[A-Z]` for upper case, and `[A-z]` for mixed case.
- `[:digit:]`: Numerical values. With other languages including R, it is commonly written as `\\d` or `[0-9]`. For any non-digit, write `\\D`.
- `[:alnum:]`: Alphanumeric characters (mix of letters and numbers). With other languages including R, it is indicated using to as `[0-9A-Za-z]` or `\\w`. For any non-alphanumeric character, use `\\W`.
- `[:space:]`: Spaces such as tabs, carriage returns, etc. For any white space, use `\\s`. For any non-whitespace character, use `\\S`.
- `[:graph:]`: Human readable characters including `[:alnum:]` and `[:punct:]`.
- `\\b`: Used to denote "whole words". `\\b` should be placed before and after a regex pattern. For example, `\\b\\w{10}\\b` indicates a 10 letter word.

These are only a few of the most important character classes. It is worth noting that many of the R-specific character classes may differ from other data science programming languages.

1.2.2.3 Quantifiers

Each character class on its own indicates a search for any character of that class. For example, a search for the pattern `[[:alpha]]` will return any element with an alphabetic character – it is not very specific. In practice, most character searches will involve a search for more than just one character. To indicate such a search, regex relies on *quantifiers* to indicate the length of patterns. For example, a search for a year between the year 1980 and 2000 will require exactly four digits, but a search for the speed of a gust of wind will likely vary between 1 and 3 digits. The following six quantifiers provide flexibility and specificity to accomplish targeted search tasks:

- `{n}`: match pattern n times for a preceding character class. For example `"\\d{4}"` looks for a four digit number.
- `{n, m}`: match pattern at least `n`-times and not more than `n` times for a preceding character class. For example `\\d{1,4}` looks for an one to four digit number.
- `{n, }`: match at least n times for a preceding character class. For example `\\d{4,}` searches for a number that has at least four digits.
- `*`: Wildcard, or match at least 0 times.
- `+`: Match at least once.
- `?`: Match at most once.

In the example below, quantifiers are used to extract specific number patterns with a high degree of accuracy.

```
  big_dates <- c("Octavian became Augustus on 16 Jan 27 BCE",
             "In the year 2000, a computer bug was expected to topple society.",
             "In the 5400000000 years, our sun will become a red dwarf.")

#Return a 9 digit number
  str_extract(big_dates, "\\d{9}")

#Return a 4 digit substring that is flanked by empty value at either end
  str_extract(big_dates, "\\b\\d{4}\\b")

#Match a date that follows 16 January 27 BCE
  str_extract(big_dates, "\\d{2}\\s\\w{3}\\s\\d{2}\\s\\w{3}")
```

1.2.2.4 Positions

Regex also builds in functionality to search for patterns based on location of a substring, such as at the start or end of a string. There are quite a few other position matching patterns, but the following two are the main workhorses:

- `$`: Search at the end of a string.
- `^`: Start of string when placed at the beginning of a regex pattern.

To demonstrate these patterns, we'll apply `str_extract` to three headlines from the BBC.

```
  headlines <- c("May to deliver speech on Brexit",
           "Pound falls with May's comments",
           "May: Brexit plans to be laid out in new year")

#Find elements that contain May at the beginning of the string
  str_extract(headlines, "^May")

#Find elements that contain Brexit at the beginning of the string
  str_extract(headlines, "Brexit$")
```

Together, these character combinations make it possible to accomplish complex string manipulation tasks.

1.2.2.5 DIY: Working with PII

In an increasingly digital world, data privacy is a sensitive issue that has taken center stage. At the center of it is the safeguarding of Personally identifiable information (PII). Legislation in the European Union, namely the General Data Protection Regulation or GDPR, requires companies to protect the personal data of European Union (EU) citizens associated with transactions conducted in the EU ((Burgess 2019)). The US Census Bureau, which administers the decennial census, must apply disclosure avoidance practices in order so that individuals cannot be identified ((Abowd 2018)). Thus, anonymization has become a common task when processing and using sensitive PII data.

Redaction. For example, the first element in the vector below contains hypothetical PII and sensitive information – John's social security number and balance in his savings account are shown. When presented with many lines of sensitive information, one could review each sentence and manually redact sensitive information, but given thousands if not millions of pieces of sensitive information, this is simply not feasible.

```
statement <- c( "John Doe (SSN: 012-34-5678) has $2303 in savings in his account.",
                "Georgette Smith  (SSN: 000-99-0000) owes $323 to the IRS.",
                "Alexander Doesmith (SSN: 098-76-5432) was fined $14321 for overdue books.")
```

Using a combination of regex and `stringr` we can redact sensitive information with placeholders. To remove the SSN, we need a regex pattern that captures a pattern with three digits (`\\d{3}`), a hyphen, two digits (`\\d{2}`), a hyphen, then four digits (`\\d{4}`), or when combined: `\\d{3}-\\d{2}-\\d{4}`. The matched pattern is then replaced with `XXXXX`.

```
  pacman::p_load(stringr)
  new_statement <- str_replace(statement,"\\d{3}-\\d{2}-\\d{4}", "XXXXX")
  print(new_statement)
```

```
## [1] "John Doe (SSN: XXXXX) has $2303 in savings in his account."
## [2] "Georgette Smith  (SSN: XXXXX) owes $323 to the IRS."
## [3] "Alexander Doesmith (SSN: XXXXX) was fined $14321 for overdue books."
```

Next, we replace the dollar value by matching a string that starts with the dollar sign (`\\$`) followed by at least one digit (`\\d{1,}`). And finally, the John Doe's first and last name are replaced by looking for two substrings that each have at least one uppercase letter with an unspecified length (`[A-z]{1,} [A-z]{1,}`) and are found at the beginning of the string (`^`). The resulting sentence has little to no information about the individual in question.

```
#Find a replace dollar amount
  new_statement <- str_replace(new_statement,"\\$(\\d{1,})", "XXXXX")

#Find and replace first and last name
  new_statement <- str_replace(new_statement,"^[A-z]{1,} [A-z]{1,}", "XXXXX")

  print(new_statement)
```

```
## [1] "XXXXX (SSN: XXXXX) has XXXXX in savings in his account."
## [2] "XXXXX  (SSN: XXXXX) owes XXXXX to the IRS."
## [3] "XXXXX (SSN: XXXXX) was fined XXXXX for overdue books."
```

Extraction. Alternatively, we can extract identifying information and create a structured data set. Using the same regex statements, we can apply `str_extract` to create a three variable data frame containing person name, SSN and money – all done with minimal effort.

```
  ref_table <- data.frame(name = str_extract(statement,"^[A-z]{1,} [A-z]{1,}"),
                          ssn = str_extract(statement,"\\d{3}-\\d{2}-\\d{4}"),
                          money = str_extract(statement,"\\$(\\d{1,})"))
  print(ref_table)
```

```
##                  name         ssn  money
## 1            John Doe 012-34-5678  $2303
## 2    Georgette Smith 000-99-0000   $323
## 3 Alexander Doesmith 098-76-5432 $14321
```

In either case, it is advisable to conduct randomized spot checks to determine if the regex accurately identifies the desired patterns.

1.2.3 Working with Dates

Most programming languages are unable to "auto-magically" recognize a date variable, requiring coders to specify the format using a set of date symbols. The `lubridate` package significantly lowers the bar for R users, making the process of working and manipulating dates more seamless and less prone to error.

```
pacman::p_load(lubridate)
```

Upon loading lubridate, we can convert string and numeric objects with values that represent dates into date objects. The `as_date` function is most intuitive, automatically detecting the date format and assumes that the date is recorded in UTC.

```
d0 <- as_date("2010-08-20")
```

In case `as_date` is unable to detect the date, the user can more formally define the date format using functions such as `mdy` and `ymd`. Both are able to accommodate both string and numeric formats.

```
d1 <- mdy("01/20/2010")
d2 <- mdy_hm("01/20/2010 00:00 AM")
d3 <- ymd(20100101)
```

Once data are converted into date objects, it is easy to process derivative information. To calculate duration between two dates is as simple as subtraction.

```
d1 - d3
```

```
## Time difference of 19 days
```

Rather than extracting parts of a date using regex statements, lubridate makes it possible to use simplified functions to extract each `year`, `month`, `quarter`, `day`, `hour`, `minute`, and `second`. For example, we can extract the `year` and `quarter` and concatenate into a string using `paste0`.

```
  y1 <- year(d0)
  q1 <- quarter(d0)
  paste0(y1, "Q", q1)
```

```
## [1] "2010Q3"
```

In base R, we can alternatively use the `format` function to specify an output format using date symbols. The examples below illustrate three different versions of the same month-year combination.

```
#YYYY-MM format
  format(d0, "%Y-%m")
```

```
## [1] "2010-08"
```

```
#Month Name - YYYY format
  format(d0, "%B %Y")
```

```
## [1] "August 2010"
```

```
#Month Abbreviation - YY format
  format(d0, "%b-%y")
```

```
## [1] "Aug-10"
```

Either option gets the job done, but the choice depends on one's willingness to become familiar with date symbols.

1.3 Shaping the structure of data

The ultimate goal of data retrieval and processing is to construct a data set that is ready for analysis, modeling and visualization. This process, however, can occupy between 50% to 80% of a data scientist's time (Lohr 2014) as the data need to be processed according to the use case. The number of tools available to *wrangle* data are many. But what if a standard framework can be applied make data more actionable sooner. As laid out in Wickham (2014), we can follow the principles of *tidy data*:

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

Otherwise, a *messy* data set is one that does not follow the described structure. Let's walk through a hypothetical company's financial data. Below, each month's data is a column and each row represents a different financial concept. It is a standard way of displaying balance sheets, optimized for readability but not empirical analysis. Working with this data in the current form to produce a time series forecast can prove challenging – it's messy data. The data should be reshaped into a multivariate time series: each row contains a month's data months and each financial concept as a column.

Table 1.7: Example of a messy data set.

value	Dec.1.2017	Jan.1.2018	Feb.1.2018
revenue	46188	13009	8223
cost	32000	41277	1900
heads	73	36	37

This is a relatively simple task in this case – we simply need to transpose the data into tidy form. The tidied table not only can be used for time series and cross sectional regression analysis, but are also optimal for visualization use cases.

Table 1.8: Example of a tidy data set.

date	revenue	cost	heads
12/1/17	46188	32000	73
1/1/18	13009	41277	36
2/1/18	8223	1900	37

For the much of this book, we will tend towards tidy data and use them as the preferred data structure that to enable analysis and modeling. Tidying involves moving columns, rotating, slicing, deleting – all tasks that change the shape of the data, requiring analysts to become intimately familiar with data structures. In the remainder of this section, we introduce core concepts for sculpting data into the right shape.

1.3.1 Matrix or Data Frame?

Last chapter, we introduced matrices and data frames as options for storing multivariate data. Matrices are vectors of the same variable type, but in two dimensions ($n \times m$ dimensions). Data frames are a generalization of matrices that allow for each column of data to hold different data types as well as refer to individual columns by a user-specified name.

When should each matrices and data frames be used? From a pure logistical perspective, data frames are more flexible with respect to its ability to store multiple data types. Some code libraries are built specifically for matrices and others for data frames. Ultimately, data frames are generally a safe bet. They can be subsetted, sorted, reshaped, collapsed, and merged, retaining the variable types.

Table 1.9: Matrices versus Data Frames.

	Matrix	Data Frame
Pros	Memory efficient. Good for advanced mathematical operations.	Store mixed types of data types. Allows user to refer to columns by an explicit name.
Cons	Able to store one data type at a time – leads to slightly more work required to manage multiple matrices. Columns can only be referred to by index number.	Not as memory efficient.

To illustrate data in action, we consider a data set containing all reported thefts, burglaries and robberies in Chicago between 2014 through 2018. The data are available on the Chicago Open Data Portal. A cleaned version in .Rda is available at link goes here. The file contains a data frame `crimes` with $n = 428,250$ with 17 variables.

```
load("data/chicago_crime.Rda")
```

Table 1.10: Example from crimes data set.

case.number	date	primary.type	description
HY189976	03/18/2015 09:00:00 PM	ROBBERY	ARMED: HANDGUN

case.number	date	primary.type	description
HY190019	03/18/2015 10:31:00 PM	THEFT	RETAIL THEFT
HY189725	03/18/2015 12:55:00 PM	BURGLARY	FORCIBLE ENTRY

1.3.2 Array Indexes

We can think of data frames and matrices as data organized along a grid – essentially a spreadsheet. Each cell in a data frame can be located using a coordinate set of *array indexes*, or a number that indicate the location along a row or column. The first, fifth, and millionth rows have index values of 1, 5, and 1000000. The second, ninth and one-thousandth columns have values of 2, 9, 1000. This is similar to other languages like FORTRAN, MATLAB, and Julia. In contrast, array indexes in programming languages such as Python, Perl, C, Java and Ruby start with 0 – often a point of confusion when transitioning between languages.

Why do array indexes matter? Understanding this is the gateway to subsetting data, or slicing and dicing as one pleases. To extract any part of the data frame requires referring to the array indexes in a specific format:

```
crimes[7, 6]
```

```
## # A tibble: 1 x 1
##   primary.type
##   <chr>
## 1 ROBBERY
```

where `crimes` is the name of a data frame. 7 refers to the 7th row and 6 is the 6th column – both of which are sandwiched in square brackets and separated by a comma. Executing the above state will extract the value in the seventh row and sixth column.

What if we want to extract an entire row? Simply specify the row index of interest and leave the column blank returns the desired result. The output is a single row data frame describing a robbery that occurred on "March 18, 2015 at 11:55 PM".

```
crimes[7, ]
```

To *extract multiple records by row index* depends on whether the request is sequential or piecemeal. Below, the first line extracts a range of rows from the 2nd through 4th rows in `crimes`, whereas the second extracts two non-overlapping ranges that are specified as a numeric vector.

```
crimes[2:4, ] # apply the index range to extract rows
crimes[c(1:2, 10:11), ] # specific indexes
```

The same notation can be used *to extract all street blocks that had a crime reported* by typing the number `4` after the comma in square brackets. The number 4 is the column index that contains `block` names. Rather than returning a data frame, the result is a vector containing the partially redacted street block.

```
crimes[,4]
```

In addition, data frames provide a few additional options for extracting the `block` column.

```
  crimes[, "block"] # extract column with "block" label
  crimes[["block"]] # list syntax to extract column from data frame
  crimes$block # compact version of data column manipulation
```

To *extract two or more columns* follows a familiar pattern, making use of either a range of column indexes or a vector of column names.

```
  crimes[, 3:4] # extract multiple columns
  crimes[, c("date", "block")] # multiple column labels
```

1.3.3 Subsetting

Extracting specific *subsets* is perhaps one of the most essential skills of data processing as it allows one to focus on specific aspects of the data. In addition to calling specific array indexes to subset data, we can extract specific rows based on user-specified criteria, such as "crimes that happened in 2015" or "robberies that involved a handgun". Here, we describe three approaches using logical vectors, the `which` command and the `filter` command in the `dplyr` library. The first two options are possible using the pre-built functionality of R.

The **logical vector** approach first creates a Boolean vector that describes some logical comparison, then passing the logical vector to the data frame to subset.

```
#Logical vector
  group1 <- crimes$year == 2015

#Select based on vector
  out <- crimes[group1, ]
```

While this is a perfectly valid way of subsetting data, the **which** approach offers speed gains. The logical vector is passed to the `which` function to retrieve the row index of each `TRUE` value, thereby reducing the number of elements that need to be evaluated in the subset. The relevant row indexes are then passed to the data frame, returning a result far faster than the first option.

```
#Logical vector
  group1 <- which(crimes$year == 2015)

#Select based on vector
  out <- crimes[group1, ]
```

Lastly, and perhaps more convenient, is the `filter` function in the `dplyr` package. Not only is the syntax concise, the processing time is fast. The command only requires the data frame to be specified once and variables can be specified without the dollar sign notation.

```
  pacman::p_load(dplyr)
  out <- filter(crimes, year == 2015)
```

The `dplyr` allows for *piping* of operations, meaning that multiple steps can be combined into one function call.

The pipe operator `%>%` allows the output of one step to be input into another, thereby creating a pipeline. For example, the statement below is the same as the above.

```
  out <- crimes %>% filter(year == 2015)
```

To get comfortable, let's walk through a few different scenarios.

Example 1: Find all crimes that were armed. To do this, we use `str_detect` function from `stringr` to search for the term `ARMED` from the `description` field in `crimes`, then use the result to subset using `filter`.

```
  out <- filter(crimes, str_detect(crimes$description, "ARMED"))
```

Example 2: Find all cases that where the `primary.type` of crime was either a `THEFT` or `BURGLARY`. So that the statement can be one compact line, we use the `%in%` operator to search if any the elements in the vector `c("THEFT", "BURGLARY")` are found in `primary.type`.

```
  out <- filter(crimes, primary.type %in% c("THEFT", "BURGLARY"))
```

Extract rows that meet two criteria is as simple as using logical operators. The pipe operator `"|"` should be used to represent *OR*, whereas `"&"` represents *AND*. To find all cases where an `arrest` occurred after the `year` 2016 :

```
  out <- filter(crimes, arrest == TRUE & year > 2016)
```

1.3.4 Sorting and Re-ordering

At times, data frames need to be re-arranged to make it easier to navigate the data.

Re-arranging rows and columns. For one thing, columns and rows can be re-arranged or event removed by listing column indexes in a desired order. By passing a vector of column indexes to a data frame, we can not only reorder columns but also remove them. In the example below, we first re-order columns, moving columns 10 through 17 to the beginning of the data set and moving one through nine to the end. Then, we create a new data set `crime_new` dropping columns 10 and 13 by placing a negative (`-`) before the column indexes.

```
#Re-order
  crimes <- crimes[, c(10:17, 1:9)]

#Drop columns
  crime_new <- crimes[, -c(10, 13)]
```

It is more likely that we would organize our data calling specific columns by name. The standard approach is to pass a string vector of column names, requiring each name to be placed in quotations. A more concise approach using the `select` function from `dplyr` lists the names of desired columns without quotations. Below, we illustrate how to create a data frame of geographic coordinates.

```
#base R
  crimes_geo <- crimes[, c("latitude", "longitude")]

#dplyr approach
  crimes_geo <- select(crimes, latitude, longitude)
```

To ***sort a data set*** can be done using `order`, which returns a vector of row indicies in ascending order based on the variable of interest. To sort descending, add we can either `"-"` before the variable if it is a numeric or specify `decreasing = T`. The `order` function should not be confused with its cousin the `sort` function, which is designed to order and return a vector of values.

```
#base R - sort ascending
  crimes[order(crimes$latitude, crimes$longitude), ]

#base R - sort descending (2 options)
  crimes[order(crimes$latitude, crimes$longitude, decreasing = TRUE), ]
  crimes[order(-crimes$latitude, -crimes$longitude), ]
```

The base R options are somewhat clunky, requiring repetition of the name of the data frame. In contrast, the `arrange` function in the `dplyr` package reduces redundancy. Similar to `order`, the `arrange` function sorts ascending by default.

```
#dplyr
  crimes <- arrange(crimes, latitude, longitude)
  crimes <- arrange(crimes, desc(latitude), desc(longitude))
```

1.3.5 Aggregating data

Seeing the pattern in the data can be challenging when it is too granular. Aggregating data (or “rolling up”) into coarser units shows the landscape of the forest rather than the shape of an individual tree. It is also a strategy for anonymizing data by stratum or groups of individuals while communicating insight.

Here, we introduce the `summarise` and `group_by` functions from `dplyr`. The former `summarise` is used to calculate summary metrics on a set of variables, such as the count of crimes or the average latitude of crimes. The `group_by` indicates which along which variables should aggregation occur. Together, a highly granular data set can be converted into a more compact summary set, whether a time series of activity over time or a cross sectional snap shot.

Suppose we would like to see the total number of each type of crime for each ward of Chicago. We can easily send the `crimes` data set into a pipeline like the one below. A `group_by` statement indicates that all subsequent calculations will be applied for each combination of type of crime (`primary.type`) and ward of the city (`ward`). The data is then summarized by calculating two variables: the number of crimes `num.crimes` using the count function `n` and the `arrest.rate` by taking the `mean` of the `arrest` variable. This process is a new data frame with only $n = 151$ and $k = 4$. It is clear how this can be used – turning raw data into aggregate, communicable insights.

```
  crimes  %>%
    group_by(primary.type, ward) %>%
    summarise(num.crimes = n(),
              arrest.rate = 100 * mean(arrest))
```

Any variable can be used in the `group_by` statement, separating each dimension by a comma. The `summarise` function is quite flexible encompassing a large number of summary statistics:

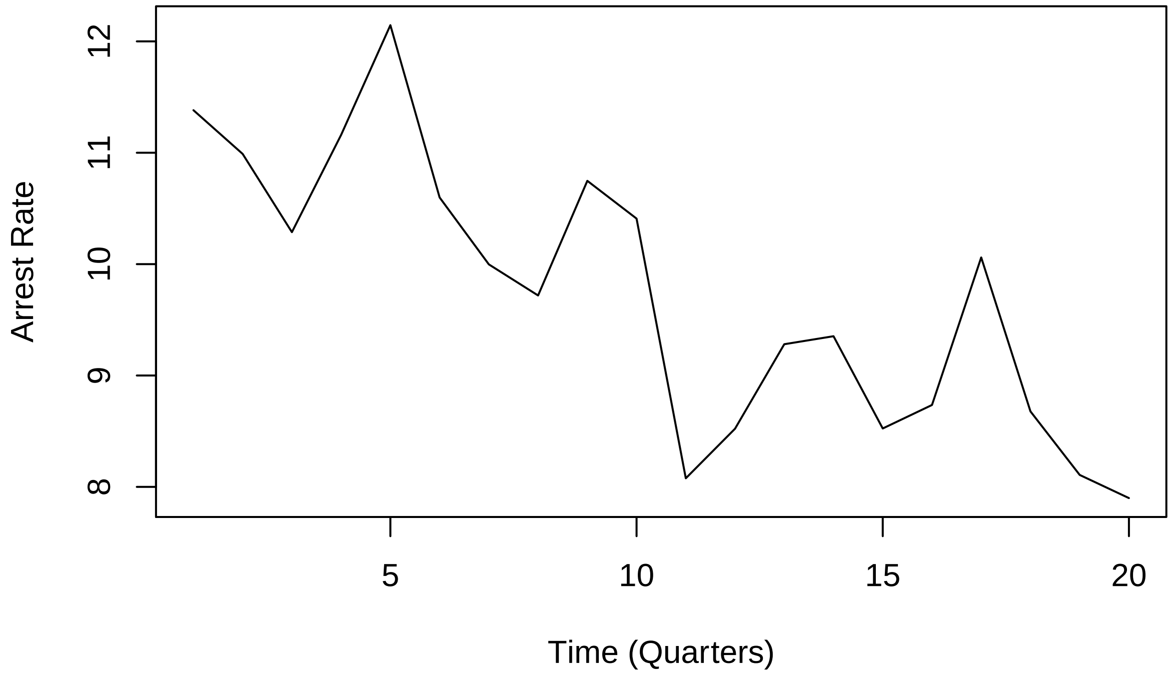
- *Central tendency*: `mean`, `median`
- *Dispersion*: `sd`, `IQR`, `mad`
- *Range*: `min`, `max`, `quantile`
- *Position*: `first`, `last`, `nth`,
- *Count*: `n`, `n_distinct`
- *Logical*: `any`, `all`

Constructing Time Series. Using these functions, we can aggregate the events data into a ***time series*** to uncover patterns and trends over time. Are crimes increasing? Do they follow a predictable seasonal pattern? Suppose we would like to visualize arrest rates in Chicago. We will follow a simple two step process: (1) *Create quarterly time variables* using a combination of `mdy_hms` and `quarter` functions from `lubridate`. The resulting time variable will be used in `group_by`. (2) *Summarize the data* into a new data frame.

```
#Convert to date form and extract the the quarter
  crimes$date1 <- mdy_hms(crimes$date)
  crimes$qtr <- quarter(crimes$date1, with_year = TRUE)

#Summarized data
  crimes_qtr <- crimes  %>%
                group_by(qtr) %>%
                summarise(arrest.rate = 100 * mean(arrest))
```

This process, as illustrated below, creates a 20-observation data frame `crimes_qtr` that shows that arrest rates have fallen over time. Tasks such as these are core processing steps that underlie most data science projects.

```
#Plot a line chart
  plot(crimes_qtr$arrest.rate,
       type = "l",
       ylab = "Arrest Rate",
       xlab = "Time (Quarters)")
```



1.3.6 Reshaping data

The shape and structure of a data set usually goes hand-in-hand with how it will be used. Data are generally stored in either *long* form and *wide* form. Long form represents each subject (or individual unit of analysis) at one point in time as a row and each variable as a column. This format also is the most analytically convenient format. In the social sciences, long format is analogous to panel data or stacked data. For example, the following is a long format data set with arrest rates associated with burglary, robberies and thefts for each ward and quarter.

Table 1.11: Example of long data.

qtr	ward	arrest.rate
2014.1	1	7.17
2014.1	13	24.51
2014.1	25	11.49
2014.1	37	30.30
2014.1	49	7.59

Wide format, in contrast, represents each subject as one row and each column is a variable. If there are multiple dimensions such as time, a variable is spread across multiple time columns. This format is more common when presenting data as it is easier to visually inspect data in a single viewport or sheet of paper. The US GDP, for example, is often published in wide form. Alternatively, the long format crime data above can be restructured into wide form, as shown below.

The shape of data has a direct implications for how analyses can be conducted and depend on the use case. Panel analyses, or analyses using data for each subject over time, require data in long form. Cross sectional analyses may require data to be in wide form. The process of converting data from one form to another is known as *reshaping*. For those who have familiarity with spreadsheet software, reshaping data essentially is the same as working with a pivot table.

Table 1.12: Example of wide data.

qtr	ward. 1	ward. 10	ward. 11	ward. 12
2014.1	7.17	14.96	11.50	12.22
2014.2	7.42	15.23	9.12	14.19
2014.3	6.69	9.41	9.76	13.25
2014.4	7.56	13.33	16.99	12.12
2015.1	10.88	21.84	10.44	13.82

To reshape data, we rely on a package called `reshape2`, which simplifies the process of reshaping into two functions: `melt` and `dcast`. The former `melt` is used to convert data into long form, whereas the latter `dcast` converts data into wide form.

```
pacman::p_load(reshape2)
```

Long to Wide. Suppose we summarized the crimes data into a quarterly data set of arrest rates by type of crime and ward. This data would be returned in long form.

```
crimes_long <- crimes  %>%
              group_by(qtr, ward, primary.type) %>%
              summarise(arrest.rate = 100 * mean(arrest),
                        num.crimes = n())
```

The `dcast` function converts data from long form to wide form and expects a data frame, a formula, and at least one value variable. Let's walk through the example below that produces a new data frame where each row is a type of crime in a ward and each column is the arrest rate for each time period.

- The formula dictates the desired structure. In the formula `ward + primary.type ~ period`, for example, variables to the left of the tilde `~` are treated as row identifiers and variables to the right are dimensions that will be spread across the columns. Multiple dimensions can be specified, each separated by an addition symbol `+`.
- The parameter `value.var` indicates the variable that will fill the cells of the new data structure, which in this case is the `arrest.rate`.

The resulting data frame has $n = 150$ rows and $k = 22$ columns.

```
crimes_wide <- dcast(crimes_long,
                     ward + primary.type ~ qtr,
                      value.var = "arrest.rate")
```

Suppose we would like to reshape both the `arrest.rate` as well as the `num.crimes`. This requires two modifications to the above code: (1) We will need to install and load the `data.table` package, which converts data frames into a class of objects for memory efficient and speedy data manipulation. From that package, we will need to use `setDT` to convert our data frame into the appropriate format. (2) Specify a string vector of variable names for the `value.var`.

The resulting data frame has $n = 150$ rows and $k = 42$ columns.

```
pacman::p_load(data.table)
crimes_widetoo <- dcast(setDT(crimes_long),
                     ward + primary.type ~ qtr,
                      value.var = c("arrest.rate", "num.crimes"))
```

Wide to Long. It is just as simple to roll the data back into long form using the `melt` function. The example below re-converts the `crimes_wide` data frame into long form. Remember that all but two variables contain `arrest.rate` for a each quarter. Moving from wide to long means that all the arrest.rate columns will be consolidated into two variables: column headers will be captured as a new time variable and column values will be arrest rates. The function expects:

- a data frame (`crimes_wide`).
- `id.vars`: a vector of variable names or column indexes that indicate row identifiers.
- `variable.name`: a name of a new variable to which the wide column names will be stored.
- `value.var`: a name of a new variable to store the values.

```
  crimes_longtoo <-   melt(crimes_wide,
                        id.vars = c("ward", "primary.type"),
                        variable.name = "qtr",
                        value.name = "arrest.rate")
```

1.4 Control Structures

Data scraped from websites and media sources are becoming increasingly more common in the digital age. Investment companies regularly use news reports to detect sentiment towards companies in order to predict the direction of stock prices and earnings reports. Economists use scraped data from e-commerce sites to measure inflation. Scraping is quite repetitive. The same set of pre-defined actions are applied to many different webpages on a website. The format of that webpage dictates the types of actions. One could manually target websites and choose the code to be applied, but a far more scalable solution is to take advantage of control structures.

Given some variable, a control structure determines how a program will treat that variable given pre-determined conditions and parameters. The webpage formats of a New York Times article and a Fox News article, for example, are fundamentally different, thus a scalable web scraping program should code that are fitting to each type in order to get the desired data. Likewise, a script designed to extract debits and credits from a financial statement may look at the header of a table, then decide how to handle each type of value. In this section, we will cover two commonly used control structures that make programming far more efficient: if statements and loops.

1.4.1 If Statement

If statements evaluate a logical statement, then execute a script based on whether the evaluated statement is true or false. If the statement is `TRUE`, then the code block is executed. Below, we have written some logic to evaluate if a manager has spent over his allocated budget for the year, returning a printed statement if spending exceeded acceptable tolerance of $400.

```
  spending <- 450
  if(spending > 400){
    #If statement true, run script between curly brackets
    print("You're over budget. Cut back.")
  }
```

```
## [1] "You're over budget. Cut back."
```

In cases where there are two or more choices, `if...else` statements would be appropriate. In addition to the `if()` statement, an `else` statement can be included to handle cases where the logical statement is `FALSE`.

```
  spending <- 399
  if(spending >= 400){
    #If statement true, run script goes here
    print("You're over budget. Cut back.")
  } else {
    #else, run script goes here
    print("You're within budget, but watch it.")
  }
```

The complexity of these statements can accommodate more complex logic:

```
  spending <- 500
  if(spending >= 450){
    print("High roller, eh? Is your debt supported by a wealthy donor?")
  } else if(spending >= 400){
    print("You're over budget. Cut back.")
  } else if(spending >= 50 && spending < 400){
    print("You're within budget. Good job.")
  } else if(spending < 50 &){
    print("You might be under-reporting your spending.")
  }
```

If statements work are designed to evaluate *one logical statement at a time* and requires a *loop* to evaluate for each element of a vector – a control structure we cover in the next subsection. For simple if statements, we can rely on the `ifelse` function that evaluates a logical condition for each element in a data frame or vector, then returns a vector with desired transformations.

```
#Vector of spending for multiple managers
  spending <- c(50, 200, 500)

#Ifelse returning overbudget flags
  out <- ifelse(spending >= 400, "Over-budget", "Within Budget")
```

1.4.2 For-loops

Many programming tasks apply the same code over and over. Rather than repeating tasks manually, loops can reduce redundant tasks by applying the same function over a range of elements, objects, etc. It is an efficiency augmenting functionality that is central to program and allows computers to perform tasks in a semi-automated way. Conceptually, for loops can be likened to an assembly line in a car factory. In order to build a car, a series of well-defined, well-timed processes need to coordinated in a serial fashion. To build 500 cars, the process needs to be executed 500 times. For-loops are essentially the same: Given a well-defined set of tasks, the manufacturing process runs 500-times.

Let's take the following toy example in which we print to console a set of numbers. The code block essentially says "print values for the range of 1 through 5", where `i` is an *index value*. When executing the statement, R will push the first value in the sequence of 1:5 into the index (in this case, it's the number 1), then the code block in between the curly brackets (`{}`) will be executed, treating `i` as if it's the number 1. Upon executing the code without error, R will advance to the next value in the sequence and repeat the process until all values in the sequence have been completed.

```
  for(i in 1:5){
    print(paste0("Car #", i, " is done."))
  }
```

```
## [1] "Car #1 is done."
## [1] "Car #2 is done."
## [1] "Car #3 is done."
## [1] "Car #4 is done."
## [1] "Car #5 is done."
```

We can do the same for a vector or list of values. In the example below, the vector `news` contains six terms. Using a for loop, we can print out each word and append the number of characters to a placeholder vector `word_length`.

```
#Vector of words
  news <- c("The","Dow","Is","Up","By","400pts")
  word_length <- c()

  for(i in news){
    print(i)
    word_length <- c(word_length, nchar(i))
  }
```

```
## [1] "The"
## [1] "Dow"
## [1] "Is"
## [1] "Up"
## [1] "By"
## [1] "400pts"
```

A few things to keep in mind when using loops. First, loops are convenient to use but there are often times more efficient functions that handle *vectorized* problems – or a function that is designed to be applied to all elements of a vector. For example, while the `word_length` example uses a loop to calculate the length of each word in the `news` vector, the `nchar` function can actually be applied to all words in the `news` vector without a loop. Second, all objects created in a loop are stored in the programming environment. While this means that all objects will be available after the loop is complete, it also means each subsequent iteration's results will overwrite the previous. To save the result of each iteration requires one to explicitly indicate how results should be stored.

If a calculation needs to be applied to each of one million files and the results need to be saved, then for-loops are a good option. *A common paradigm* involves the following:

1. Create empty placeholder object (e.g. a vector, matrix, or data frame) to store results;
2. Initialize loop; then

3. Append outputs to placeholder at the end of each loop iteration.

In the example below, we simulate 10,000 observations of a Random Walk, which is a time series in which the best forecast of today's value is yesterday's value plus an error term u_t:

$$\hat{y}_t = y_{t-1} + u_t$$

A multitude of real world phenomena can be described as random walks, such as exchange rates and stock prices. For statistical research, it is often necessary to simulate a random walk process in order to show how a new algorithm performs. Below, we simulate a random walk in which we create a placeholder data frame `df` in iteration 1. In each subsequent iteration, we add the previous period's value with a random growth selected from `alpha`, then append the result to `df`. To simulate 10,000 observations requires roughly 7 seconds to complete. What if we wanted to reduce the compute time? In the age of big data, it might be necessary to run millions of simulations, thus 7 seconds may actually be an eternity.

```
#Set placeholder data frame with n rows
  n <- 10000
  alpha <- c(-1, 1)

#Simulate Random Walk
  start <- Sys.time()
  for(i in 1:n){
    if(i == 1){
      df <- data.frame(iter = i,
                       y = sample(alpha, 1))
    } else {
      df <- rbind(df,
                  data.frame(iter = i,
                             y = df$y[i-1] + sample(alpha, 1)))
    }
 }

#Timer stop
  Sys.time() - start
```

```
## Time difference of 7.26366 secs
```

If we know the total space required for the simulation, we can set up `df` with a pre-specified number of rows and overwrite individual cells when needed. Such a simple change vastly reduces the processing time to less than 1.5 seconds – only one-quarter of the time! *Why?* Part of the answer lies in the functional differences between appending to data frames and overwriting cells in a data frame. When we use `rbind`, `R` is forced to overwrite the entire data frame with an additional row. As objects become large, the write time compounds. In contrast, overwriting individual cells only replaces specific cells, removing unnecessary read-write time.

```
#Set placeholder data frame with n rows
  n <- 10000
  df <- data.frame(iteration = 1:n,
                   y = numeric(n))
  alpha <- c(-1, 1)

#Simulate Random Walk
  start <- Sys.time()
  for(i in 1:n){
    if(i == 1){
      df$y[i] <- sample(alpha, 1)
    } else {
       df$y[i] <- df$y[i-1] + sample(alpha, 1)
    }
 }

#Timer stop
  Sys.time() - start
```

```
## Time difference of 1.825088 secs
```

Even more surprising, there are simpler functions that can simulate a random walk in far less time and with less code. In fact, a simple combination of the `sample` and `cumsum` functions can randomly draw $n = 10000$ values of u_t and calculate a vector of elements with cumulative sums (effectively a random walk), all in less than 0.01 seconds. This is all to show that loops are easy to use, but be sure to plan out your loops carefully to maximize efficiency and it is worth conducting some due diligence to identify possible alternatives.

```
#Simulate Random Walk in one line
  start <- Sys.time()
  out <- cumsum(sample(alpha, n, replace = T))
  Sys.time() - start
```

```
## Time difference of 0.003809929 secs
```

1.4.3 While

Whereas for-loops require a range of values through which to iterate, `while` statements continue to be executed until a condition is no longer true. We can imagine scenarios in which the number of required iterations is an unknown but a pre-defined criteria is known. Optimization problems with a systems of equations, for example, require that multiple candidate solutions be tested in order to find a solution that minimizes a error function – while loops are great for automating searches.

A few key ingredients are required. The `while` statement requires a logical statement (e.g. $x > 2$) – the *goal*. So as long x is greater than 2, the statement will continue to run. There also needs to be an *update step* in the script that ensures that values that are to be evaluated are tuned towards satisfying the goal so that eventually x is no longer greater than 2. Otherwise, the while loop will continue for infinitum, also known as an *infinite loop*.

To put this into perspective, we construct a while loop to search for a value of x that minimizes the formula $y = x^{\frac{3}{2}} - 8x + 100$. Before the loop, we set initial value of $x = -100$, set $step = 1$ to control the fineness of the search interval for x, create a counter variable i to keep track of the number of iterations, and a placeholder vector `vals` to store values of y. The while loop starts by evaluating if one of two conditions is true: if there are less than two iterations (`i < 2`) or if we have found a minimum value (if current value of y in iteration i is less than the previous iteration). If either is true, then the script within the curly braces is executed, otherwise the loop terminates. Notice that *step* is added to x in order to advance the loop to the next set of conditions to be evaluated.

```
#Set initial value of x and i
  x <- -100
  step <- 1
  i <- 0
  vals <- c()

#Find a minimum of the function
#where previous value is greater than current
  while(i < 2 || vals[i-1] >= vals[i]){
   vals <- c(vals, x^2 -  8*x + 100)
   i <- length(vals)
   x <- x + step
 }
```

While these loops are useful for searching for solutions, the problem needs to be well-defined and well-bounded.

1.5 Functions

Data manipulation tasks are often repeated for many different projects and it is not uncommon for two or more scripts to contain the same exact steps. Rather than tediously modifying scripts, try to live by the following philosophy: once, then never again. The idea is that if code is well-designed and documented, it can serve as re-usable tools that can be re-applied to similar problems. This is the basis of *user-written functions*: a coder can define some set of standard required inputs on which a set of steps can be applied to produce a standard output, then their work can be shared with many others who face similar problems

A function is constructed as follows. Using `function`, a set of parameters are specified upfront as the key ingredients, then a set of coded procedures are executed using those same parameters. In the example below, we construct a function `meanToo` that replicates the functionality of the `mean` function. `meanToo` requires a data frame `df` and a variable name in string format `var.name`. Within the curly brackets, we insert code that uses the input parameters as placeholders. The mean is calculated as the sum of `var.name` in data frame `df` divided by the number of rows in `df`, then assigns the result to the object `temp`. These calculations are temporary and only executed with a *local environment* – they are wiped once the function finishes running; However, the `temp` object can be extracted by passing it to `return`. All of the above steps are assigned to the `meanToo` object that is treated like any other function.

It is good practice to include documentation about what the function does and how to use it. At a minimum, include what the function is, the arguments, and the output. Typically in the culture of open source software, code should be written with others in mind: keep it transparent and re-usable.

```
meanToo <- function(df, var.name, ...){
  #
  ## Calculate mean of a variable in a data frame
  #
  # Args:
  #   df = data frame
  #   var.name = variable name (character)
  #
  # Returns:
  #   A numeric value

  #Code goes here
  temp <- sum(df[[var.name]])/nrow(df)

  #Return desired result
  return(temp)
}
```

To use the function, first run the function code above, then fill in the input parameters. Any script can be converted into a function for general-use. While it does require some time upfront to properly write a usable function, getting into the habit is worth the investment for efficiency and sharing your technical perspective.

```
  meanToo(data, "x1")
```

1.5.1 DIY: Tracking the supply of online content

Services like Google Trends illustrate the demand for online content, giving a sense of what the public are interested in and what people are reacting to at a given moment. But how about the supply of online content? The supply provides a sense if content producers feel it is worth spending any time on creating new materials.

The opioid epidemic has been a social issue that has long been brewing, but has only become the centered of public attention in recent years. As the crisis deepens, we'd expect more content to be generated. For each year, we could tediously use a web browser to manually copy the number of search results from a platform like Bing, but what if we need to track a large number of search terms. We can package searches on Bing into a simple function that serves as a wrapper that extracts the number of search results. For example, the search query below yields approximately 18,000 search results for the year 2010 (the number may change as this is only an estimate):

`https://www.bing.com/search?q=opioid%20epidemic&filters=ex1%3a%22ez5_12810_13893%22`

Breaking down the URL, we see that the substring `q=opioid%20epidemic` indicates that the search `q=` will be based on the terms that follow `q=` and spaces are encoded as `%20`. Next, the substring `filters=ex1%3a%22ez5_12810_13893%22` indicates a time range filter is applied, specifically `12810_13893` are indexes that represent the range 1/27/2005 to 1/15/2008.

To make this a seamless process, we should construct a function `bingCounts` that will retrieve the number of search results for a given `search.term` in a specified calendar year (`year.filter`). First, we will need to load packages using `require`, which is specifically designed for within-function package loading. In particular, we will use:

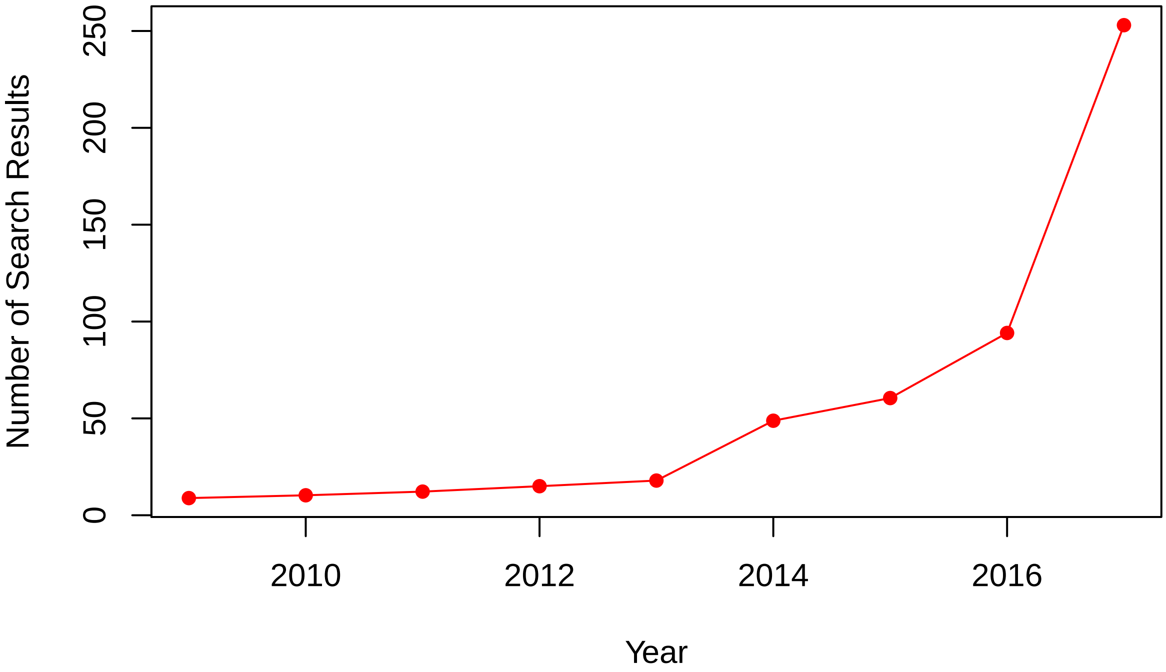
- To convert the year into indexes, we use the `lubridate` package to work with date objects.
- To efficiently construct a search URL, we rely on the `stringr` package.
- `RCurl` is used to make use of the Client URL package to make a request to the web for a webpage.
- As the webpage is in HTML format, we use the `XML` package to parse the information.

```
bingCounts <- function(search.term, year.filter){
  #
  # Retrieve number of search results for exact query
  #
  # Args:
  #   search.term = search query (character)
  #   year.term = search year.term (int)
  #
  # Returns:
  #   A numeric value

  #Load package
    require(lubridate)
    require(stringr)
    require(RCurl)
    require(XML)
  #Get date indexes
    origin <- as_date("1970-01-01")
    start.index <- as_date(paste0(year.filter, "01-01")) - origin
    end.index <- as_date(paste0(year.filter, "12-31")) - origin

  #Construct search URL by replacing placeholder terms
    search.url <- "https://www.bing.com/search?q=term&filters=ex1%3a%22ez5_tstart_tend%22"
    search.url <- str_replace(search.url, "term", gsub(" ", "%20", search.term))
    search.url <- str_replace_all(search.url, "tstart", as.character(start.index))
    search.url <- str_replace_all(search.url, "tend", as.character(end.index))

  #Get URL content as HTML
    search.html <- getURL(search.url)

  #Parse HTML
    parse.search <- htmlTreeParse(search.html, useInternalNodes = TRUE)

  #Extract result statistics
    nodes <- getNodeSet(parse.search, "//*[@id='b_tween']/span[1]")
    value <- strsplit(xmlValue(nodes[[1]])," ", fixed = TRUE)[[1]][1]
    return(as.numeric(gsub(",", "", value, fixed = TRUE)))

}
```

Rather than copying the above function multiple times, the resulting function can then be neatly wrapped into a loop to extract the number of search results for each year in a nine-year period between 2009 and 2017. We find the number of search results has grown 44-times over the period. Based on CDC data, the number of opioid-related deaths increased 2.4-times from approximately 20,400 to 49,000.[1] Thus, we can see the search queries give a measure of direction of growth, but miss the magnitude of the crisis.

```
#Set parameters
  term <- "opioid epidemic"
  results <- data.frame()

#Loop through
  for(i in 2009:2017){
    results <- rbind(results,
                     data.frame(year = i,
                                cnt = bingCounts(term, i)))
  }
```

[1] https://www.drugabuse.gov/related-topics/trends-statistics/overdose-death-rates



1.6 Best Practices

Writing just any piece of code is easy. The real challenge is to ensure that it is re-usable, functional and adopted.

Data processing is not like a vampiric feeding frenzy of the sort that might be seen on Buffy the Vampire slayer (the TV show): one simply does not arbitrarily chomp at and slice data without any thought as to what comes next. It requires discipline and finesse to acquire a raw piece of information and extract the empirical substance. It is the foundational step of any successful data science project.

Imagine that you have conducted a three-month analytics project, rushing to obtain to a result. If asked to provide backup, documentation, and a process, it may very well require an additional few months to decipher your passionate but hasty explorations. Where did this data come from? Why did I take a dot product in this equation? What happened to the raw file? Who is Bob mentioned in this email? Which directors have seen the report? Ultimately, the data scientist is in control and should be empowered to conduct data processing in a structured, scientific manner. Here are five tips that can bulletproof data processing:

- *Define a file structure and stick to it.* All work product needs to be chronicled in a coherent structure. At a minimum, create a folder for *scripts* to store all code, another for *raw* data, a folder for *outputs* from data processing, and another for *documentation*. Choose any file structure, but make sure it is consistent and always followed. Name scripts and data in a logical, intuitive format. And perhaps more importantly, create a *Read Me* file – a simple text document that explains the contents of the project and keeps a log of versions.

- *Adopt and stick to a coding style.* Each file, data field, and function should follow a style so that the logic is cohesive, readable and traceable. The Google R Style Guide (Google (2009)), for example, sets easy to follow guidelines. According to the style guide, variable names should be labeled as lower case characters with a `"."` as a separator: `variable.name`. Functions should be named in proper case without spaces: `NewFunction` or `MyFunction`. Regardless of the style guide, file names should be meaningful and the only punctuation used should be `"_"` for spaces and `"."` to accompany the file type: `gdp_2014_10_02.json`. But most importantly, the code needs to make common sense. Be vigilant with your code style as it may be the difference between a successful adopted project and a failed project.

- *Version up.* Each time code is written, modified, amended, or updated, a new version should be saved. Cloud-based services such as Github make version control simple and also allow groups of coders to work together. If storing work product on a local server or computer, simply save the file with a new version number as well as making an entry into a log or Read Me file. The log file can be as simple as a text file with time stamps indicating the date of the edit, the file name and the nature of the edit.
- *Think modularly with code.* One of the greatest human inventions is the assembly line. Each step in the process is discrete, requiring workers to specialize in one set of well-defined tasks. As a worker becomes specialized, production speed increases as they refine their ability to accomplish a specific task. If any step in the process needs to be improved, revised, or replaced, all other steps in the process are not disrupted – only one step needs to be addressed. This standardization of processes is the only way to scale. Programming and data processing are the same – the goal is to produce well-written code that can reliably accomplish and re-accomplish the same tasks using newly available data. In essence, one needs to construct code that lays the pipeline for data to flow from loading, to cleansing and standardization of values, re-structuring into usable form, and storing in an accessible format. Each module of the process should be able to be swapped out like an interchangeable piece if need be.
- *Think about data in layers.* All data should ideally stem from a single "source of truth" – one harmonized set of data should serve as the basis for all applications, whether its supporting inspectors in the field or informing high-level policy making. It is thus not uncommon to have data stored in varying degrees of processing. NASA earth science satellite imagery, for example, is stored in at least six different levels of varying amounts of processing. Level 0 data are raw satellite instrument data at full resolution. Level 1B contained reconstructed, calibrated, and processed data that serve as the core source data for downstream applications. Whereas Level 4 are outputs from specific models and analyses. Before commencing data processing, first consider what end users want to accomplish, then process the data set into a common, standard form that can serve as a lowest common denominator. It is from that source data set that all use case specific derived data should come.

It will undoubtedly require some time to first familiarize oneself with the steps of data processing before coding projects become tidy engineering feats. Nonetheless, these best practices are worthy ideals to which to strive.

Chapter References

Abowd, John. 2018. "Protecting the Confidentiality of America's Statistics: Adopting Modern Disclosure Avoidance Methods at the Census Bureau," August. https://www.census.gov/newsroom/blogs/research-matters/2018/08/protecting_the_confi.html.

Burgess, Matt. 2019. "What Is Gdpr? The Summary Guide to Gdpr Compliance in the Uk." *Wired*, January. https://www.wired.co.uk/article/what-is-gdpr-uk-eu-legislation-compliance-summary-fines-2018.

Google. 2009. "Google R Style Guide." https://google.github.io/styleguide/Rguide.xml.

Lohr, Steve. 2014. "For Big-Data Scientists, 'Janitor Work' Is Key Hurdle to Insights." *The New York Times*, August. https://www.nytimes.com/2014/08/18/technology/for-big-data-scientists-hurdle-to-insights-is-janitor-work.html.

Wickham, Hadley. 2014. "Tidy Data." *Journal of Statistical Software* 59 (10).