

Contents

Welcome 7						
1	Bas	ics	9			
	1.1	Math	9			
	1.2	Missing values	10			
	1.3	Assignment	10			
	1.4	Printing a value	11			
	1.5	Packages	11			
	1.6	Logic	18			
	1.7	Generating a sequence of values	20			
	1.8	Calculating means and medians	21			
	1.9	Writing your own functions	22			
2	Dat	a Structures	27			
	2.1	One-dimensional data	27			
	2.2	Two-dimensional data	31			
	2.3	Three-dimensional and higher data	34			
	2.4	General data structures	37			
3	Imp	port, Export, and Save Data	41			
	3.1	CSV	41			
	3.2	XLS/XLSX (Excel)	42			
	3.3	JSON	43			
	3.4	XML	45			
	3.5	Exporting/Writing/Saving data and variables	46			

4 CONTENTS

4	Dat	a Manipulation	49
	4.1	Names of variables and their types $\ \ldots \ \ldots \ \ldots \ \ldots$	49
	4.2	Select variables	51
	4.3	Filter/Subset variables	57
	4.4	Rename variables	62
	4.5	Create, replace and remove variables \hdots	65
	4.6	Create strings from numbers	66
	4.7	Create numbers from strings	69
	4.8	Combine strings	70
	4.9	Finding and replacing patterns within strings	71
	4.10	Change case	74
	4.11	Drop duplicate rows	75
	4.12	Format dates	77
	4.13	Randomly sample rows	82
5	Con	abine, Reshape and Merge	87
	5.1	Combine rows	87
	5.2	Combine columns	89
	5.3	Reshaping data	90
	5.4	$Merge/Join \ \dots $	96
6	\mathbf{Agg}	regation and Group Operations	101
	6.1	Cross tabulation	101
	6.2	Group summaries	103
	6.3	Centering and Scaling	106
7	Basi	ic Plotting and Visualization	109
	7.1	Histograms	110
	7.2	Barplots	113
	7.3	Scatterplot	117
	7.4	Stripcharts	120
	7.5	Boxplots	123
	7.6	Facet plots	126

CONTENTS	F
CONTERNIS	5
CONTENIO	· ·

8	Sele	ected Topics in Statistical Inference	135
	8.1	Comparing group means	135
	8.2	Comparing group proportions	144
	8.3	Linear modeling	147
	8.4	Logistic regression	151

6 CONTENTS

Welcome

This book provides parallel examples in Python and R to help users of one platform more easily learn how the other platform "works" when it comes to data analysis.

This work is licensed under a Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License.

8 CONTENTS

Chapter 1

Basics

This chapter covers the very basics of Python and R.

1.1 Math

Mathematical operators are the same except for exponents, integer division, and remainder division (modulo).

Python

Python uses ** for exponentiation, // for integer division, and % for remainder division.

```
> 3**2
9
> 5 // 2
2
> 5 % 2
1
```

In Python, the + operator can also be used to combine strings. See this TBD section.

${\bf R}$

Python uses $\hat{\ }$ for exponentiation, %/% for integer division, and %% for remainder division.

```
> 3^2
[1] 9
> 5 %/% 2
[1] 2
> 5 %% 2
[1] 1
```

1.2 Missing values

Python and R represent missing values differently, and the distinction is worth keeping in mind, as missing values will crop up throughout this book—some code examples intake or output data that are entirely or partially missing. In Python, a standard indicator for a missing value in a data set is NaN.* In R, missing values are generally indicated by NA. NaN does appear in R as well, but R reserves NaN to indicate values that are not technically absent but that are not defined and/or that can't be represented with numbers; e.g., Inf/Inf.

1.3 Assignment

Python uses = for assignment, while R can use either = or <- for assignment. The latter "assignment arrow" is preferred in most R style guides to distinguish between assignment and setting the value of a function argument. According to R's documentation, "The operator <- can be used anywhere, whereas the operator = is only allowed at the top level (e.g., in the complete expression typed at the command prompt) or as one of the subexpressions in a braced list of expressions." See <code>?assignOps</code>.

Python

```
> x = 12
```

 \mathbf{R}

```
> x <- 12
```

1.4 Printing a value

To see the value of an object created via assignment, you can simply enter the object at the console and hit enter for both Python and R, though it is common in Python to explicitly use the print() function.

Python

```
> x
12
```

\mathbf{R}

```
> x
[1] 12
```

1.5 Packages

User-created functions can be bundled and distributed as packages. Packages need to be installed only once. Thereafter they're "imported" (Python) or "loaded" (R) in each new session when needed.

Packages with large user bases are often updated to add functionality and fix bugs. The updates are not automatically installed. Staying apprised of library/package updates can be challenging. Some suggestions are following developers on Twitter, signing up for newsletters, or periodically checking to see what updates are available.

Packages often depend on other packages. These are known as "dependencies." Sometimes packages are updated to accommodate changes to other packages they depend on.

Python

When you download Python, you gain access to The Python Standard Library. This library includes several datatypes and functions for storing data, performing mathematical operations, and beyond. Commonly used datatypes include *list* and *range*. As you can see below, you do not need to import data types from the Standard Python Library.

```
> my_list = []
+ for idx in range(5):
+ my_list.append(idx)
+ print(my_list)
[0, 1, 2, 3, 4]
```

Libraries contain modules, groups of functions. To use functions from modules in the Python Standard Library, users must import the appropriate module. Examples include math and itertools, which both include several functions for a range of operations.

```
> import math
+ one = 1
+ two = 2
+ print(math.pow(two, one))
2.0
```

Users can also download 100s of libraries outside of the Standard Python Library. Python libraries are also called packages. Popular libraries include numpy, used for operations on arrays/vectors and pandas, used for data analysis. The following code is an example of importing a Python library, NumPy, into a Python script.

```
> import numpy as np
+
+ my_array = np.array([1,2,3])
+ print(my_array)
[1 2 3]
```

To use Python libraries outside the Python Standard Library, they must be installed in the Python environment.

Anaconda is the most popular Python library manager. Anaconda allows you to use and create Python virtual environments and install libraries to these environments. A Python virtual environment is a collection of libraries isolated from other virtual environments. These environments allow users to seamlessly organize programming projects.

You can download Anaconda from the following link: https://www.anaconda.com/products/individual. When you download Anaconda, you have access to the Anaconda Navigator, a graphical user interface, and the Anaconda Prompt, a command prompt. Anaconda comes with an automatic environment called "base".

The following screenshot illustrates how to install a library to an environment using the Anaconda GUI. Using the drop down menu, navigate to "Not installed". Then, select the desired library from the list. The search bar can be

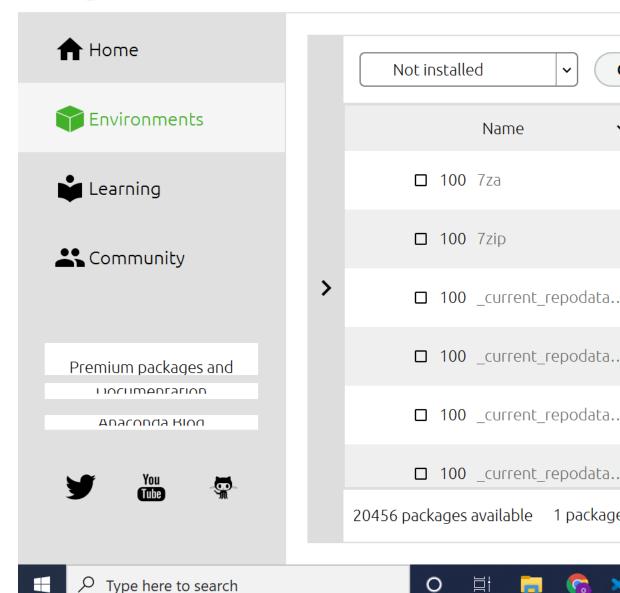
1.5. PACKAGES 13

used to search for libraries. Finally, click the green "Apply" button to install the package.

Anaconda Navigator

File Help





1.5. PACKAGES 15

The following screenshot illustrates how to install a library to the "base" environment using the Anaconda Command Prompt.

16 CHAPTER 1. BASICS Anaconda Prompt (anaconda3) (base) C:\Users\oyina>conda install numpy_











1.5. PACKAGES 17

Sometimes the commands to download libraries are not as simple as shown in the above example. The Anaconda website provides commands for how to download popular Python libraries.

\mathbf{R}

The main repository for R packages is the Comprehensive R Archive Network (CRAN). Another repository is Bioconductor, which provides tools for working with genomic data. Many packages are also distributed on GitHub.

To install packages from CRAN use the install.packages() function. In RStudio, you can also go to Tools...Install Packages... for a dialog that will autocomplete package names as you type.

```
> # install the vcd package, a package for Visualizing Categorical Data
> install.packages("vcd")
>
> # load the package
> library(vcd)
>
> # see which packages on your computer have updates available
> old.packages()
>
> # download and install available package updates;
> # set ask = TRUE to verify installation of each package
> update.packages(ask = FALSE)
```

To install R packages from GitHub use the install_github() function from the **devtools** package. You need to include the username of the repo owner followed by a forward slash and the name of the package. Typing two colons between a package and a function in the package allows you to use that function without loading the package. That's how we use install_github() below.

```
> install.packages("devtools")
> devtools::install_github("username/packagename")
```

Occasionally when installing package updates you will be asked, "Do you want to install from sources the package which needs compilation?" R packages on CRAN are *compiled* for Mac and Windows operating systems. That can take a day or two after a package has been submitted to CRAN. If you try to install a package that has not been compiled then you'll get asked the question above. If you click *Yes*, R will try to compile the package on your computer. This will only work if you have the required build tools on your computer. For Windows this means having Rtools installed. Mac users should already have the necessary build tools. Unless you absolutely need the latest version of a package, it's probably fine to click *No*.

1.6 Logic

Python and R share the same relational operators for making comparisons:

```
== (equals)
!= (not equal to)
< (less than)</li>
<= (less than or equal to)</li>
> (greater than)
>= (greater than or equal to)
```

Likewise they share the same operators for logical AND and OR:

```
• & (AND)
• | (OR)
```

However R also has && and || operators for programming control-flow.

Python and R have different operators for negation. Python uses not. R uses !.

Python

These Python operators can be used to compare arrays to single values or other arrays. This operation returns an array containing true and false values.

```
> import numpy as np
+
+ # Comparison of array to single value
+ x1 = np.array([1,5,9,12,11,6])
+ x1 < 8
+
+ # Comparison of array to another array
array([ True, True, False, False, False, True])
> x2 = np.array([2,4,6,14,15,7])
+ x1 > x2
array([False, True, True, False, False, False])
```

We can make multiple comparisons with the AND (and) and OR (or) operators. An important thing to note is that the and operator is inclusive, meaning that all statements must be true to return True. The or operator is exclusive, meaning that at least one of the statements joined by or must be true to return True.

1.6. LOGIC 19

```
> x=5
+ y= 4
+
+ x > 6 and y < 10
False
> x > 6 or y < 10
True</pre>
```

True and False operators have numeric values of 1 and 0, respectively. We can sum and average these values.

${\bf R}$

R's relational operators allow comparisons between a vector and a single value, or comparisons between two vectors. The result is a vector of TRUE/FALSE values.

```
> # vector compared with value
> x1 <- c(1, 5, 9, 12, 11, 6)
> x1 < 8
[1] TRUE TRUE FALSE FALSE TRUE
>
> # vector compared with vector
> x2 <- c(2, 4, 6, 14, 15, 7)
> x1 > x2
[1] FALSE TRUE TRUE FALSE FALSE FALSE
```

Comparisons with NA (missing value) results in NA.

```
> x1 <- c(1, 5, 9, NA, 11, 6)
> x1 < 8
[1] TRUE TRUE FALSE NA FALSE TRUE
```

Multiple comparisons can be made with AND (&) and OR (|) operators.

```
> x2 > 3 & x2 < 10

[1] FALSE TRUE TRUE FALSE FALSE TRUE
> x2 < 3 | x2 > 10

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

TRUE/FALSE values in R have numeric values of 1/0. This allows us to sum and average them. (Note: an average of 0 and 1 values is the proportion of 1's.)

```
> # sum of values greater than 10
> sum(x2 > 10)
[1] 2
>
> # proportion of values greater than 10
> mean(x2 > 10)
[1] 0.33333333
```

Use the ! operator for negation. This allows to check for something that is NOT TRUE.

```
> # which value are NOT less than 6
> !x2 < 6
[1] FALSE FALSE TRUE TRUE TRUE TRUE</pre>
```

See the ?Comparison and ?Logic help pages for more information.

1.7 Generating a sequence of values

In Python, one option for generating a sequence of values is arange() from NumPy. In R, a common approach is to use seq(). The sequences can be incremented by indicating a step argument in arange() or a by argument in seq(). Be aware that the end of the start/stop interval in arange() is open, but both sides of the from/to interval in seq() are closed.

Python

```
> import numpy as np
+ x = np.arange(start = 1, stop = 11, step = 2)
+ x
array([1, 3, 5, 7, 9])
```

 \mathbf{R}

```
> x <- seq(from = 1, to = 11, by = 2)
> x
[1] 1 3 5 7 9 11
```

1.8 Calculating means and medians

The \mathbf{NumPy} Python library has functions for calculating means and medians, and base R has functions for doing the same.

Python

Mean, using function from NumPy library

```
> import numpy as np
+ x = [90, 105, 110]
+ x_avg = np.mean(x)
+ print(x_avg)
101.6666666666666667
```

Median, using function from **NumPy** library

```
> x = [98, 102, 20, 22, 304]
+ x_med = np.median(x)
+ print(x_med)
98.0
```

 \mathbf{R}

Mean, using function from base R

```
> x <- c(90, 105, 110)
> x_avg <- mean(x)
> x_avg
[1] 101.6667
```

Median, using function from base R

```
> x <- c(98, 102, 20, 22, 304)
> x_med <- median(x)
> x_med
[1] 98
```

1.9 Writing your own functions

Python and R allow and encourage users to create their own functions. Functions can be created, named, and stored in memory and used throughout a session. Or they can be created on-the-fly "anonymously" and used once.

Python

Functions in Python are defined by using the def keyword followed by the name we choose for our function with its arguments inside parentheses. We must include a return() statement after the body of our function to indicate the end of the function. The return statement takes an optional argument in its parentheses that will be the output of the function. Here we create a function to calculate the standard error of a mean (SEM) and call it SEM.

```
> def SEM(x):
+ import numpy as np # import statement included inside the function to ensure it's
+ s = x.std(ddof=1) # find standard deviation of the data, specify delta degrees of
+ n = x.shape[0] # extract the length of the input array
+ sem = s / np.sqrt(n) # calculate the SEM
+ return(sem) # return the calculated SEM value
```

Now let's try our function out on some test data.

```
> d = np.array([3,4,4,7,9,6,2,5,7])
+ SEM(d)
0.7412035591181296
```

Oftentimes functions have built-in error-checking that returns messages describing the error. Here we show a simple error-check to ensure that the argument passed to our function is a number.

```
> def SEM(x):
+  import numpy as np
+
+  if np.issubdtype(x.dtype,np.number)==False:
+  raise ValueError("Data must be numeric")
```

Python functions can return more than one result. It will output the results into a tuple. A tuple is a data structure very similar to a list, but it is immutable - we cannot change the order of the entries. Here we make our function return both the mean and the SEM of our data.

```
> def SEM(x):
+   import numpy as np
+
+   if np.issubdtype(x.dtype,np.number)==False:
+     raise ValueError("Data must be numeric")
+
+     s = x.std(ddof=1)
+     n = x.shape[0]
+     sem = s / np.sqrt(n)
+
+     m = np.mean(x)
+     return(sem,m)
```

\mathbf{R}

Functions in R can be created and named using function(). Add arguments inside the parentheses. Longer functions with multiple lines can be wrapped in curly braces {}.

Below we create a function to calculate the standard error of a mean (SEM) and name it sem. It takes one argument: x, a vector of numbers. Both the function name and argument name(s) can be whatever we like, as long as they follow R's naming conventions.

```
> sem <- function(x){
+    s <- sd(x)
+    n <- length(x)
+    s/sqrt(n)
+ }</pre>
```

Now we can try it out on some test data.

```
> d <- c(3,4,4,7,9,6,2,5,7)
> sem(d)
[1] 0.7412036
```

Functions that will be used on different data and/or by different users often need built-in error-checking to return informative error messages. This simple example checks if the data are not numeric and returns a special error message.

```
> sem <- function(x){
+    if(!is.numeric(x)) stop("x must be numeric")
+    s <- sd(x)
+    n <- length(x)
+    s/sqrt(n)
+ }
> sem(c(1, 4, 6, "a"))
Error in sem(c(1, 4, 6, "a")): x must be numeric
```

R functions can also return more than one result. Below we return a list that holds the mean and SEM, but we could also return a vector, a data frame, or other data structure. Notice we also add an additional argument, ..., known as the three dots argument. This allows us to pass arguments for sd and mean directly through our own function. Below we pass through na.rm = TRUE to drop missing values.

```
> sem <- function(x, ...){
+    if(!is.numeric(x)) stop("x must be numeric")
+    s <- sd(x, ...)
+    n <- length(x)
+    se <- s/sqrt(n)
+    mean <- mean(x, ...)
+    list(mean = mean, SEM = se)
+ }
>    d <- c(1, 4, 6, 8, NA, 4, 4, 8, 6)
> sem(d, na.rm = TRUE)
$mean
[1] 5.125
$SEM
[1] 0.7855339
```

Functions can also be created on-the-fly as "anonymous" functions. This simply means the functions are not saved as objects in memory. These are often used with R's family of apply functions. As before, the functions can be created with

function(). We can also use the backslash $\$ as a shorthand for function(). We demonstrate both below with a data frame.

```
> # generate some example data
> d <- data.frame(x1 = c(3, 5, 7, 1, 5, 4),
                x2 = c(6, 9, 8, 9, 2, 5),
+
                x3 = c(1, 9, 9, 7, 8, 4))
> d
 x1 x2 x3
  3 6 1
2 5 9 9
  7
     8
  1
     9
        7
5
  5
     2 8
6 4 5 4
```

Now find the standard error of the mean for the three columns using an anonymous function with lapply(). The "l" means the result will be a list. We apply the function to each column of the data frame.

```
> lapply(d, function(x)sd(x)/sqrt(length(x)))
$x1
[1] 0.8333333

$x2
[1] 1.118034

$x3
[1] 1.308094
```

We can also use the backslash as a shorthand for function().

```
> lapply(d, \(x)sd(x)/sqrt(length(x)))
$x1
[1] 0.8333333

$x2
[1] 1.118034

$x3
[1] 1.308094
```

Chapter 2

Data Structures

This chapter compares and contrasts data structures in Python and R.

2.1 One-dimensional data

A one-dimensional data structure can be visualized as a column in a spreadsheet or as a list of values.

Python

There are many ways to organize one-dimensional data in Python. Three of the most common one-dimensional data structures are lists, numpy arrays, and pandas Series. All three are ordered and mutable, and can contain data of different types.

Lists in Python do not need to be explicitly declared; they are indicated by the use of square brackets.

```
> 1 = [1,2,3,'hello']
```

Values in lists can be accessed by using square brackets. Python indexing begins at 0, so to extract the first element, we would use the index 0. Python also allows for negative indexing; using an index of -1 will return the last value in the list. Indexing a range in Python is not inclusive of the last index.

```
> # extract first element
+ 1[0]
+
```

```
+ #extract last element
1
> 1[-1]
+
+ # extract 2nd and 3rd elements
'hello'
> 1[1:3]
[2, 3]
```

Numpy arrays, on the other hand, need to be declared using the numpy.array() function, and the numpy package needs to be imported.

```
> import numpy as np
+
+ arr = np.array([1,2,3,'hello'])
+ print(arr)
['1' '2' '3' 'hello']
```

Accessing data in a numpy array is the same as indexing a list.

```
> # extract first element
+ arr[0]
+
+ # extract last element
'1'
> arr[-1]
+
+ # extract 2nd and 3rd elements
'hello'
> arr[1:3]
array(['2', '3'], dtype='<U11')</pre>
```

Pandas Series also need to be declared using the pandas.Series() function. Like numpy, the pandas package must be imported as well. The pandas package is built on numpy, so we can input data into a pandas Series using a numpy array. We can extract data from the Series by using the index similar to indexing a list and numpy array.

```
> import pandas as pd
+ import numpy as np
+
+ data = np.array([1,2,3,"hello"])
+ ser1 = pd.Series(data)
+ print(ser1)
```

```
# extract first element
0
         1
         2
1
2
         3
     hello
dtype: object
> ser1[0]
+ # extract 2nd and 3rd elements
111
> ser1[1:3]
1
    2
2
     3
dtype: object
```

To extract the last element of a pandas Series using $\neg 1$, we need to use the iloc function.

```
> ser1.iloc[-1]
'hello'
```

We can relabel the indices of the Series to whatever we like using the index attribute within the Series function.

We can then use our own specified indices to select and index our data. Indexing with our labels can be done in two ways. One similar to indexing arrays and lists with square brackets using the .loc function, and the other follows this form: Series.label_name.

```
+ # extract element in row b
+ ser2.loc["b"]
```

```
+ # extract elements from row b to the end
121
> ser2.loc["b":]
 # extract element in row "d"
         2
b
С
         3
d
     hello
dtype: object
> ser2.d
+ # extract element in row "b"
'hello'
> ser2.b
121
```

One thing to note is that mathematical operations cannot be carried out on lists, but they can be carried out on numpy arrays and pandas Series. In general, lists are better for short data sets that you will not be operating on mathematically. Numpy arrays and pandas Series are better for long data sets, and for data sets that will be operated on mathematically.

\mathbf{R}

In R a one-dimensional data structure is called a *vector*. We can create a vector using the c() function. A vector in R can only contain one type of data (all numbers, all strings, etc). The columns of data frames are vectors. If multiple types of data are put into a vector, the data will be coerced according to the hierarchy logical < integer < double < complex < character. This means if you mix, say, integers and character data, all the data will be coerced to character.

```
> x1 <- c(23, 43, 55)
> x1
[1] 23 43 55
>
> # all values coerced to character
> x2 <- c(23, 43, 'hi')
> x2
[1] "23" "43" "hi"
```

Values in a vector can be accessed by position using indexing brackets. R indexes elements of a vector starting at 1. Index values are inclusive. For example, 2:3 selects the second and third elements.

```
> # extract the 2nd value
> x1[2]
[1] 43
>
> # extract the 2nd and 3rd value
> x1[2:3]
[1] 43 55
```

2.2 Two-dimensional data

Two-dimensional data are rectangular in nature, consisting of rows and columns. These can be the type of data you might find in a spreadsheet with a mix of data types in columns; they can also be matrices as you might encounter in matrix algebra.

Python

In Python, two common two-dimensional data structures include the *numpy* array and the *pandas DataFrame*.

A two-dimensional numpy array is made in a similar way to the one-dimensional array using the numpy.array function.

```
> import numpy as np
+
+ arr2d = np.array([[1,2,3,"hello"],[4,5,6,"world"]])
+ print(arr2d)
[['1' '2' '3' 'hello']
  ['4' '5' '6' 'world']]
```

Selecting data for a two-dimensional numpy array follows the same form as indexing a one-dimensional array.

```
> import numpy as np
+
+ # extract first element
+ arr2d[0,0]
+
+ # extract last element
'1'
> arr2d[-1, -1]
+
+ # extract 2nd and 3rd columns
```

A pandas Data Frame is made using the ${\tt pandas}\,.{\tt DataFrame}$ function in a similar way to the pandas Series.

Selecting data from a DataFrame is similar to that of the Series.

```
> # extract first element
+ df.loc[0,0]
+ # extract column 1
111
> df.loc[0]
+ # extract row 1
0
        1
         2
1
2
         3
3
     hello
Name: 0, dtype: object
> df.loc[0,0]
111
```

Like the pandas Series, we can change the indices and the column names of the DataFrame and can use those to select and index our data.

We change the indices again using the index attribute in the pandas.DataFrame function:

```
> import pandas as pd
+ import numpy as np
+
```

We can change the column names using the ${\tt columns}$ attribute in the ${\tt pandas.DataFrame}$ function:

One thing to note is that numpy arrays can actually have N dimensions, whereas pandas DataFrames can only have two. Numpy arrays will be the better choice for data with more than two dimensions.

\mathbf{R}

Two-dimensional data structures in R include the *matrix* and *data frame*. A matrix can contain only one data type. A data frame can contain multiple vectors, each of which can consist of different data types.

Create a matrix with the matrix() function. Create a data frame with the data.frame() function. Most imported data comes into R as a data frame.

```
name age
1 Rob 35
2 Cindy 37
```

Values in a matrix and data frame can be accessed by position using indexing brackets. The first number(s) refers to rows; the second number(s) refers to columns. Leaving row or column numbers empty selects all rows or columns.

```
> # extract value in row 1, column 2
> m[1,2]
[1] 5
>
> # extract values in row 2
> d[2,]
   name age
2 Cindy 37
```

2.3 Three-dimensional and higher data

Three-dimensional and higher data can be visualized as multiple rectangular structures stratified by extra variables. These are sometimes referred to as arrays. Analysts usually prefer two-dimensional data frames to arrays. Data frames can accommodate multidimensional data by including the additional dimensions as variables.

Python

To create a three-dimensional and higher data structure in Python, we again use a numpy array. We can think of the three-dimensional array as a stack of two-dimensional arrays. We construct this in the same way as the one- and two-dimensional arrays.

We can also construct a three-dimensional numpy array using the reshape function on an existing array. The argument of reshape is where you input your desired dimensions - strata, rows, columns. Here, the arange function is used to create a numpy array containing the numbers 1 through 12 (to recreate the same array shown above).

Indexing the three-dimensional array follows the same format as the two-dimensional arrays. Since we can think of the three-dimensional array as a stack of two-dimensional arrays, we can extract each "stacked" two-dimensional array. Here we extract the first of the "stacked" two-dimensional arrays:

We can also extract entire rows and columns, and individual array elements:

```
> # extract 1st row of 2nd strata (second "stacked" 2-D array)
+ arr3d[1, 0]
+
+ # extract 1st column of 2nd strata
array([7, 8, 9])
> arr3d[1, :, 0]
+
+ # extract the number 6 (1st strata, 2nd row, 3rd column)
array([7, 10])
> arr3d[0, 1, 2]
```

The three-dimensional arrays can be converted to two-dimensional arrays again using the reshape function:

```
> arr3d_2d = arr3d.reshape(4,3)
+ arr3d_2d
array([[ 1,  2,  3],
```

```
[ 4, 5, 6],
[ 7, 8, 9],
[10, 11, 12]])
```

\mathbf{R}

The array() function in R can create three-dimensional and higher data structures. Arrays are like vectors and matrices in that they can only contain one data type. In fact matrices and arrays are sometimes described as vectors with instructions on how to layout the data.

We can specify the dimension number and size using the dim argument. Below we specify 2 rows, 3 columns, and 2 strata using a vector: c(2,3,2). This creates a three-dimensional data structure. The data in the example are simply the numbers 1 through 12.

```
> a1 \leftarrow array(data = 1:12, dim = c(2,3,2))
> a1
, , 1
       [,1] [,2] [,3]
[1,]
          1
                 3
[<mark>2</mark>,]
          2
                 4
, , 2
       [,1] [,2] [,3]
[1,]
          7
                 9
                       11
[<mark>2,]</mark>
          8
                10
                       12
```

Values in arrays can be accessed by position using indexing brackets.

```
> # extract value in row 1, column 2, strata 1
> a1[1,2,1]
[1] 3
>
> # extract column 2 in both strata
> # result is returned as matrix
> a1[,2,]
       [,1] [,2]
[1,] 3 9
[2,] 4 10
```

The dimensions can be named using the dimnames() function. Notice the names must be a *list*.

The as.data.frame.table() function can collapse an array into a twodimensional structure that may be easier to use with standard statistical and graphical routines. The responseName argument allows you to provide a suitable column name for the values in the array.

```
> as.data.frame.table(a1, responseName = "value")
   X Y Z value
1 x1 y1 z1
               1
2 x2 y1 z1
               2
3 x1 y2 z1
               3
4 x2 y2 z1
5 x1 y3 z1
               5
6 x2 y3 z1
              6
7 x1 y1 z2
               7
8 x2 y1 z2
               8
9 x1 y2 z2
               9
10 x2 y2 z2
              10
11 x1 y3 z2
              11
12 x2 y3 z2
              12
```

2.4 General data structures

Both R and Python provide general "catch-all" data structures that can contain any number, shape, and type of data.

Python

The most general data structures in Python include the *list* and the *tuple*. Both lists and tuples are ordered collections of objects called *elements*. The elements can be other lists/tuples, arrays, integers, objects, etc.

Lists are mutable objects; elements can be reordered or deleted and new elements can be added after the list has been created. Tuples, on the other hand, are immutable; once a tuple is created it cannot be changed.

Lists are created using square brackets. Here we create a list and add an element to the list after it is created using the append function.

```
> lst = [1, 2, 'a', 'b', [3, 4, 5]]
+ lst
[1, 2, 'a', 'b', [3, 4, 5]]
> lst.append('c')
+ lst
[1, 2, 'a', 'b', [3, 4, 5], 'c']
```

Tuples are created using parenthesis. Here we create a tuple.

```
> tuple = (1, 2, 'a', 'b', [3, 4, 5])
+ tuple
(1, 2, 'a', 'b', [3, 4, 5])
```

Let's try to use the append function to explore the immutability of the tuple. We expect to get an error.

```
> tuple.append('c')
Error in py_call_impl(callable, dots$args, dots$keywords): AttributeError: 'tuple' obj
Detailed traceback:
  File "<string>", line 1, in <module>
```

We can refer to specific list/tuple elements by using square brackets. In the square brackets we put the index number of the element. The element in the first position is at index 0.

```
> # Extract the first element of the list and the tuple
+ lst[0]
1
> tuple[0]
+
+ # Extract the last element of each
```

```
1
> lst[-1]
'c'
> tuple[-1]
[3, 4, 5]
```

\mathbf{R}

The most general data structure in R is the *list*. A list is an ordered collection of objects, which are referred to as the *components*. The components can be vectors, matrices, arrays, data frames, and other lists. The components are always numbered but can also have names. The results of statistical functions are often returned as lists.

We can create lists with the list() function. The list below contains three components: a vector named "x", a matrix named "y", and a data frame named "z". Notice the m and d objects were created in the two-dimensional data section earlier in this chapter.

```
> 1 \leftarrow list(x = c(1,2,3),
           y = m,
             z = d
> 1
$x
[1] 1 2 3
$y
     [,1] [,2]
        1
[2,]
        3
              7
$z
   name age
    Rob 35
2 Cindy 37
```

We can refer to list components by their order number or name (if present). To use order number, use indexing brackets. Single brackets returns a list. Double brackets return the component itself.

```
> # second element returned as list
> 1[2]
$y
     [,1] [,2]
```

```
[1,] 1 5
[2,] 3 7
>
> # second element returned as itself (matrix)
> 1[[2]]
      [,1] [,2]
[1,] 1 5
[2,] 3 7
```

Use the \$ operator to refer to components by name. This returns the component itself.

```
> 1$y
    [,1] [,2]
[1,]    1    5
[2,]    3    7
```

Finally it is worth noting that a data frame is a special case of a list consisting of components with the same length. The is.list() function returns TRUE if an object is a list and FALSE otherwise.

```
> # object d is data frame
> d
    name age
1   Rob   35
2 Cindy   37
> str(d)
'data.frame':   2 obs. of   2 variables:
$ name: chr "Rob" "Cindy"
$ age : num   35   37
>
> # but a data frame is a list
> is.list(d)
[1] TRUE
```

Chapter 3

Import, Export, and Save Data

This chapter reviews importing external data into Python and R, including CSV, Excel, and other structured data files. There is often more than one way to import data into Python and R. Each example below highlights one way per file type.

The data set we use for demonstration is the New York State Math Test Results by Grade from 2006 - 2011, downloaded from data.gov on September 30, 2021.

The final section presents approaches to exporting and saving data.

3.1 CSV

Comma separated value (CSV) files are text files with fields separated by commas. They are useful for "rectangular" data, where rows represent observations and columns represent variables or features.

Python

The pandas function read_csv() is a common approach to importing CSV files into Python.

```
    1
    4
    2006
    699

    2
    5
    2006
    691
```

\mathbf{R}

There are many ways to import a csv file. A common way is to use the base R function read.csv().

Notice the spaces in the column names have been replaced with periods.

Two packages that provide alternatives to read.csv() are readr and data.table. The readr function read_csv() returns a tibble. The data.table function fread() returns a data.table.

3.2 XLS/XLSX (Excel)

Excel files are native to Microsoft Excel. Prior to 2007, Excel files had an extension of XLS. With the launch of Excel 2007, the extension was changed to XLSX. Excel files can have multiple sheets of data. This needs to be accounted for when importing into Python and R.

Python

The pandas function read_excel() is a common approach to importing Excel files into Python. The sheet_name argument allows you to specify which sheet you want to import. You can specify sheet by its (zero-indexed) ordering or by its name. Since this Excel file only has one sheet we do not need to use the argument. In addition, specifying sheet_name=None will read in all sheets and return a dict data structure where the *key* is the sheet name and the *value* is a DataFrame.

```
> import pandas as pd
> d = pd.read_excel('data/ny_math_test.xlsx')
> d.loc[0:2, ["Grade", "Year", "Mean Scale Score"]]
>
```

3.3. JSON 43

\mathbf{R}

readxl is a well-documented and actively maintained package for importing Excel files into R. The workhorse function is read_excel(). The sheet argument allows you to specify which sheet you want to import. You can specify sheet by its ordering or by its name. Since this Excel file only has one sheet we do not need to use the argument.

The result is a *tibble*, a tidyverse data frame.

It's worth noting we can use the range argument to specify a range of cells to import. For example, if the top left corner of the data was B5 and the bottom right corner of the data was J54, we could enter range="B5:J54" to just import that section of data.

3.3 JSON

JSON (JavaScript Object Notation) is a flexible format for storing data. JSON files are text and can be viewed in any text editor. Because of their flexibility JSON files can be quite complex in the way they store data. Therefore there is no one-size-fits-all method for importing JSON files into Python or R.

Python

Below is one approach to importing our "ny_math_test.json" example file. We first import Python's built-in **json** package and use its loads() function to read in the lines of the json file. The file is accessed using the open function and its associated read method.

Next we use the **pandas** function <code>json_normalize()</code> to convert the 'data' structure of the json data into a DataFrame.

Finally we add column names to the DataFrame.

```
> import json
+ # load data using Python JSON module
+ with open('data/ny_math_test.json','r') as f:
      data = json.loads(f.read())
+ import pandas as pd
+ d json = pd.json normalize(data, record path =['data'])
+ # add column names
+ names = list()
+ for i in range(23):
   names.append(data['meta']['view']['columns'][i]['name'])
+ d json.columns = names
+ d_json.loc[0:2, ["Grade", "Year", "Mean Scale Score"]]
  Grade Year Mean Scale Score
      3 2006
      4 2006
                           699
      5 2006
                           691
```

Again, this is just one approach that assumes we want a DataFrame.

\mathbf{R}

jsonlite is one of several R packages available for importing JSON files into
R. The read_json() function takes a JSON file and returns a list or data
frame depending on the structure of the data file and its arguments. We set
simplifyVector = TRUE so the data is simplified into a matrix.

```
> library(jsonlite)
> d_json <- read_json('data/ny_math_test.json', simplifyVector = TRUE)</pre>
```

The d_json object is a list with two elements: "meta" and "data". The "data" element is a matrix that contains the data of interest. The "meta" element contains the column names for the data (among much else). Notice we had to "drill down" in the list to find the column names. We assign column names to the matrix using the colnames() function and then convert the matrix to a data frame using the as.data.frame() function.

3.4. XML 45

```
2 4 2006 699
3 5 2006 691
```

3.4 XML

XML (eXtensible Markup Language) is a markup language that was designed to store data. XML files are text and can be viewed in any text editor or a web browser. Because of their flexibility, XML files can be quite complex in the way they store data. Therefore there is no one-size-fits-all approach for importing XML files into Python or R.

Python

The pandas library provides the read_xml function for importing XML files. The ny_math_test.xml file identifies records with nodes named "row". The 168 rows are nested in one node also called "row". Therefore we use the xpath argument to specify that we want to elect all row elements that are descendant of the single row element.

```
> import pandas as pd
+ d_xml = pd.read_xml('data/ny_math_test.xml', xpath="row//row")
+ d_xml.loc[0:2, ["grade", "year", "mean_scale_score"]]
  grade
         year mean_scale_score
0
      3
         2006
                             700
      4
         2006
                             699
1
      5
         2006
                             691
```

\mathbf{R}

xml2 is a relatively small but powerful package for importing and working with XML files. The read_xml() function imports an XML file and returns a list of pointers to XML nodes. There are a number of ways to proceed once you import an XML file, such as using the xml_find_all() function to find nodes that match an xpath expression. Below we take a simple approach and convert the XML nodes into a list using the as_list() function that is part of the xml2 package. Once we have the XML nodes in a list, we can use the bind_rows() function in the dplyr package to create a data frame. Notice we have to drill down into the list to select the element that contains the data. After this we need to do one more thing: unlist each the columns into vectors. We do this by applying the unlist function to each column of d. We save the result by assigning to d[], which overwrites each element (or column) of d with the unlisted result.

```
> library(xml2)
> d_xml <- read_xml('data/ny_math_test.xml')</pre>
> d_list <- as_list(d_xml)</pre>
> d <- dplyr::bind_rows(d_list$response$row)</pre>
> d[] <- lapply(d, unlist)</pre>
> d[1:3,c("grade", "year", "mean_scale_score")]
# A tibble: 3 x 3
  grade year mean_scale_score
  <chr> <chr> <chr>
1 3
        2006 700
2 4
        2006
               699
3 5
        2006
              691
```

The result is a *tibble*, a tidyverse data frame. We would most likely want to proceed to converting certain columns to numeric.

3.5 Exporting/Writing/Saving data and variables

There are several ways to export/write/save files from Python and R. The following examples highlight some of these ways.

Python

The pandas function to_csv() saves a pandas DataFrame as a csv file.

```
> # pass a file name to the function
+ d.to_csv("data.csv")
```

The Python package **pickle** allows you to write (save) any object from the Python environment and read (load) any object you have written into the Python environment.

The following code writes to a pickle file. The first line opens the file object being written to. In the open function, 'file_name' specifies the file path of the file object. Then, 'wb' stands for 'write binary', which means the file is being written in binary form (1s and 0s). After the *as* keyword, 'file_', is the user selected name of the file object.

The second line uses the pickle.dump() function. This function requires two arguments: the object being written and the name of the file object.

```
> import pickle
+
+ # define the file name
+ file_name = 'data.pickle'
+
+ # write the variable to the file system
+ with open(file_name, 'wb') as file_:
+ pickle.dump(d, file_)
```

The following code reads to a pickle file. The first line opens the file object being read from. In the open function, 'data.pickle' specifies the file path of the file object. Then, 'rb' stands for 'read binary', which means the file is being read in binary form (1s and 0s). After the as keyword, 'my_file', is the user selected name of the file object.

The second line uses the pickle.load() function. This function requires one argument: the name of the file object.

```
+ # read the specified file from the file system and load into variable
+ with open('data.pickle', 'rb') as my_file:
+ d = pickle.load(my_file)
```

\mathbf{R}

To export a matrix or data frame to a CSV file, use the write.csv() function. To export to a file with a different field separator, such as a tab, use write.table(). The minimal arguments for write.csv() are the object and the file name. To export a data frame named dat to a file named dat.csv to your current working directory:

```
> write.csv(dat, file = "dat.csv")
```

By default a column for row names or numbers is included in the exported csv file. To turn that off, set row.names = FALSE, like so:

```
> write.csv(dat, file = "dat.csv", row.names = FALSE)
```

To append a matrix or data frame to an existing csv file, set append = TRUE.

See also sink(), cat(), and writeLines() for sending text and output to a file.

To save and load R objects for future use in R, there are two options:

- 1. Save and load a single object using saveRDS() and readRDS().
- 2. Save multiple objects using save() and load().

Save and load a single object The minimal arguments for saveRDS() are the object and a file name with an .rds extension. For example, to save a single data frame named dat to your current working directory as dat.rds:

```
> saveRDS(dat, file = "dat.rds")
```

To load the rds file into R from your current working directory, use the readRDS() function. Notice we must assign the result of readRDS() to an object. The object name need not match the file name.

```
> d <- readRDS("dat.rds")</pre>
```

The advantage of saving and loading native R objects is the preservation of characteristics such as factors, attributes, classes, etc. Any object can be saved, including model objects, functions, vectors, lists, etc.

Save multiple objects The minimal arguments for save() are the objects to save and a file name with a .rda extension. Objects can also be specified as a character vector to the list argument. For example, to save a data frame named dat, a model object named m, and a plot object called p, to your current working directory as work.rda:

```
> save(dat, m, p, file = "work.rda")
```

Or with objects specified as a character vector:

```
> save(list = c("dat", "m", "p"), file = "work.rda")
```

To load the rda file from your current working directory, use the load() function. Notice we do not assign the result to an object name. The result of the load() function is to load the objects into your global environment.

```
> load("work.rda")
```

Upon successful execution of the load() function, the dat, m, and p objects will be loaded into your global environment. Any objects already in your global environment with the same name will be overwritten without warning.

You can also save *everything* in your global environment into a rda file using the <code>save.image()</code> function. It works just like the <code>save()</code> function except you do not specify which objects to save. You simply provide a file name. If you do not specify a file name, a default name of .Rdata is used. To load the file use the <code>load()</code> function. Again, all objects will be loaded into the gloabal environment, overwriting any existing objects with the same name.

Chapter 4

Data Manipulation

This chapter looks at various strategies for filtering, selecting, modifying and deriving variables in data. Unless otherwise stated, examples are for DataFrames (Python) and data frames (R) and use the mtcars data frame that is included with R.

```
> # Python
+ import pandas
+ mtcars = pandas.read_csv('data/mtcars.csv')

> # R
> data(mtcars)
> # drop row names to match Python version of data
> rownames(mtcars) <- NULL</pre>
```

4.1 Names of variables and their types

View and inspect the names of variables and their type (numeric, string, logical, etc.) This is useful to ensure that variables have the expected type.

Python

The .info() function in pandas lists information on the DataFrame.

Setting the argument verbose to True prints the name of the columns, their length excluding NULL values, and their data type (dtype) in a table. The function lists the unique data types in the DataFrame, and it prints how much memory the DataFrame takes up.

```
> mtcars.info(verbose=True)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Data columns (total 11 columns):
    Column Non-Null Count Dtype
0
            32 non-null
                            float64
    mpg
            32 non-null
1
    cyl
                            int64
2
    disp
            32 non-null
                            float64
            32 non-null
                            int64
3
    hp
4
    drat
            32 non-null
                            float64
                          float64
5
    wt
            32 non-null
6
            32 non-null
                          float64
    qsec
7
    ٧s
            32 non-null
                            int64
            32 non-null
                            int64
8
    am
    gear
            32 non-null
                            int64
10 carb
            32 non-null
                            int64
dtypes: float64(5), int64(6)
memory usage: 2.9 KB
```

Setting verbose to False excludes the table describing each column.

```
> mtcars.info(verbose=False)
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 32 entries, 0 to 31
Columns: 11 entries, mpg to carb
dtypes: float64(5), int64(6)
memory usage: 2.9 KB
```

If a DataFrame has 100 or fewer columns, the <code>verbose</code> argument defaults to <code>True.</code>

\mathbf{R}

The str() function in R lists the names of the variables, their type, the first few values, and the dimensions of the data frame.

```
$ drat: num 3.9 3.9 3.85 3.08 3.15 2.76 3.21 3.69 3.92 3.92 ...
$ wt : num 2.62 2.88 2.32 3.21 3.44 ...
$ qsec: num 16.5 17 18.6 19.4 17 ...
$ vs : num 0 0 1 1 0 1 0 1 1 1 ...
$ am : num 1 1 1 0 0 0 0 0 0 0 ...
$ gear: num 4 4 4 3 3 3 3 4 4 4 ...
$ carb: num 4 4 1 1 2 1 4 2 2 4 ...
```

To see just the names of the data frame, use the names() function.

```
> names(mtcars)
[1] "mpg" "cyl" "disp" "hp" "drat" "wt" "qsec" "vs" "am" "gear"
[11] "carb"
```

To see just the dimensions of the data frame, use the dim() function. It returns the number of rows and columns, respectively.

```
> dim(mtcars)
[1] 32 11
```

4.2 Select variables

How to select specific columns of data frames.

Python

The period operator . provides access to a column in a DataFrame as a vector. This returns pandas Series. A pandas series can do everything a numpy array can do.

```
> mtcars.mpg
      21.0
0
      21.0
1
2
      22.8
3
      21.4
4
      18.7
5
      18.1
6
      14.3
7
      24.4
8
      22.8
9
      19.2
10
      17.8
```

```
16.4
11
12
      17.3
13
      15.2
14
      10.4
      10.4
15
16
      14.7
      32.4
17
18
      30.4
19
      33.9
20
      21.5
21
      15.5
      15.2
22
23
      13.3
24
      19.2
25
      27.3
26
      26.0
27
      30.4
28
      15.8
29
      19.7
      15.0
30
31
      21.4
Name: mpg, dtype: float64
```

Indexing also provides access to columns as a pandas Series. Single and double quotations both work.

```
> mtcars['mpg']
      21.0
1
      21.0
2
      22.8
3
      21.4
4
      18.7
5
      18.1
6
      14.3
7
      24.4
8
      22.8
9
      19.2
10
      17.8
      16.4
11
      17.3
12
      15.2
13
14
      10.4
      10.4
15
      14.7
16
17
      32.4
```

```
30.4
18
19
      33.9
20
      21.5
21
      15.5
22
      15.2
23
      13.3
24
      19.2
25
      27.3
26
      26.0
27
      30.4
28
      15.8
29
      19.7
30
      15.0
31
      21.4
Name: mpg, dtype: float64
```

Operations on numpy arrays are faster than operations on pandas Series. But using pandas series should be fine, in terms of performance, in many cases. This is important for large data sets on which many operations are performed. The .values function returns a numpy array.

Double indexing returns a pandas DataFrame, instead of a numpy array or pandas series.

```
> mtcars[['mpg']]
     mpg
0
    21.0
1
    21.0
2
    22.8
3
    21.4
4
    18.7
5
    18.1
6
    14.3
7
    24.4
8
    22.8
9
    19.2
10 17.8
11 16.4
12 17.3
```

```
15.2
13
14
   10.4
15
   10.4
16
   14.7
   32.4
17
18 30.4
19
   33.9
20
   21.5
21
   15.5
22
  15.2
23
   13.3
   19.2
24
25 27.3
26 26.0
27
   30.4
28 15.8
29 19.7
30 15.0
31 21.4
```

The head() and tail() functions return the first 5 or last 5 values. Use the n argument to change the number of values. This function works on numpy arrays, pandas series and pandas DataFrames.

```
> # first 6 values
+ mtcars.mpg.head()
0    21.0
1    21.0
2    22.8
3    21.4
4    18.7
Name: mpg, dtype: float64
```

```
> # last row of DataFrame
+ mtcars.tail(n=1)
    mpg cyl disp hp drat wt qsec vs am gear carb
31 21.4 4 121.0 109 4.11 2.78 18.6 1 1 4 2
```

\mathbf{R}

The dollar sign operator, \$, provides access to a column in a data frame as a vector.

```
> mtcars$mpg
[1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

Double-indexing brackets also provide access to columns as a vector.

```
> mtcars[["mpg"]]
[1] 21.0 21.0 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 17.8 16.4 17.3 15.2 10.4
[16] 10.4 14.7 32.4 30.4 33.9 21.5 15.5 15.2 13.3 19.2 27.3 26.0 30.4 15.8 19.7
[31] 15.0 21.4
```

Single-indexing brackets work as well, but they return a data frame instead of a vector (if used with a data frame).

```
> mtcars["mpg"]
   mpg
  21.0
2 21.0
3 22.8
4 21.4
5 18.7
6 18.1
7 14.3
8 24.4
9 22.8
10 19.2
11 17.8
12 16.4
13 17.3
14 15.2
15 10.4
16 10.4
17 14.7
18 32.4
19 30.4
20 33.9
21 21.5
22 15.5
23 15.2
24 13.3
25 19.2
26 27.3
27 26.0
28 30.4
```

```
29 15.8
30 19.7
31 15.0
32 21.4
```

Single-indexing brackets also allow selection of rows when used with a comma. The syntax is ${\tt rows}$, ${\tt columns}$

```
> # first three rows
> mtcars[1:3, "mpg"]
[1] 21.0 21.0 22.8
```

Finally single-indexing brackets allow us to select multiple columns. Request columns either by name or position using a vector.

```
> mtcars[c("mpg", "cyl")]
    mpg cyl
  21.0
          6
2 21.0
          6
3 22.8
          4
4
  21.4
          6
5
  18.7
          8
6 18.1
          6
7 14.3
          8
8 24.4
          4
9 22.8
          4
10 19.2
          6
11 17.8
          6
12 16.4
          8
13 17.3
          8
14 15.2
          8
15 10.4
          8
16 10.4
          8
17 14.7
          8
18 32.4
          4
19 30.4
          4
20 33.9
          4
21 21.5
          4
22 15.5
          8
23 15.2
          8
24 13.3
          8
25 19.2
          8
26 27.3
          4
27 26.0
          4
28 30.4
          4
```

```
29 15.8 8
30 19.7 6
31 15.0 8
32 21.4 4
> # same as mtcars[1:2]
```

The head() and tail() functions return the first 6 or last 6 values. Use the n argument to change the number of values. They work with vectors or data frames.

```
> # first 6 values
> head(mtcars$mpg)
[1] 21.0 21.0 22.8 21.4 18.7 18.1

> # last row of data frame
> tail(mtcars, n = 1)
    mpg cyl disp hp drat wt qsec vs am gear carb
32 21.4 4 121 109 4.11 2.78 18.6 1 1 4 2
```

4.3 Filter/Subset variables

How to view rows of a data frame that meet certain conditions.

Python

We can filter rows of a DataFrame based on a condition to subset. The data type returned depends on the filtration method.

The following code returns a DataFrame, not a Series, as there is more than one column selected from the DataFrame. Use a list, square brackets [], to subset more than one column.

```
> mtcars[mtcars["mpg"] > 30][["mpg", "cyl"]]
         mpg    cyl
17    32.4     4
18    30.4     4
19    33.9     4
27    30.4     4
```

Both pandas Series and NumPy arrays can be used for faster performance or vector operations. Many functions require a vector as input.

The following code returns one column, mpg, as a pandas Series. A pandas Series is one column from a pandas DataFrame.

```
> mtcars[mtcars["mpg"] > 30]["mpg"]
17      32.4
18      30.4
19      33.9
27      30.4
Name: mpg, dtype: float64
```

The following code also returns a pandas Series, but using the . operator to select for a column, rather than square brackets [].

```
> mtcars[mtcars["mpg"] > 30].mpg
17     32.4
18     30.4
19     33.9
27     30.4
Name: mpg, dtype: float64
```

Both of the following lines of code return NumPy arrays using the .values function. df1 is one dimension, for the one column, and df1 is two dimensions, for the two columns.

```
> df1 = mtcars[mtcars["mpg"] > 30]["mpg"].values
+ df2 = mtcars[mtcars["mpg"] > 30][["mpg", "cyl"]].values
```

You can also filter with multiple row conditions.

```
> mtcars[mtcars["mpg"] > 30][mtcars["hp"] < 66]</pre>
          cyl disp hp drat
                                   wt
                                                             carb
                                        qsec
                                              ٧s
                                                  am
                                                      gear
18
    30.4
               75.7
                     52
                         4.93
                               1.615
                                       18.52
                                               1
                                                   1
                                                          4
                                                                2
19 33.9
            4 71.1 65 4.22 1.835
                                      19.90
                                                   1
                                                                1
<string>:1: UserWarning: Boolean Series key will be reindexed to match DataFrame index
```

\mathbf{R}

In base R, we can use subsetting brackets or the subset() function to select rows based on some condition. Below we demonstrate both approaches to view only those rows with "mpg" greater than 30. First we begin with subsetting brackets.

The subsetting brackets take three arguments:

1. i: the condition to subset on.

- 2. j: the columns to show. If none specified, all columns are returned
- 3. drop: an optional logical argument (TRUE/FALSE) to determine whether or not to coerce the output to the lowest possible dimension. The default is TRUE.

We rarely type the first two argument names, i and j, when using subsetting brackets.

This example returns only the rows with mpg > 30 and all columns. Notice we need to preface mpg with mtcars\$ to tell R where to find the "mpg" column and that we need to provide a comma after the condition.

```
> mtcars[mtcars$mpg > 30, ]
   mpg cyl disp hp drat   wt qsec vs am gear carb
18 32.4   4 78.7  66 4.08 2.200 19.47  1  1   4   1
19 30.4   4 75.7  52 4.93 1.615 18.52  1  1   4   2
20 33.9   4 71.1  65 4.22 1.835 19.90  1  1  4   1
28 30.4   4 95.1 113 3.77 1.513 16.90  1  1  5   2
```

We can select what columns to see in the second argument as a vector. Notice we only need to specify the column names as a character vector. We can also use numbers corresponding to the column number as well as conditional statements.

```
> mtcars[mtcars$mpg > 30, c("mpg", "wt", "gear")]
    mpg    wt gear
18 32.4 2.200    4
19 30.4 1.615    4
20 33.9 1.835    4
28 30.4 1.513    5
```

Show first three columns.

```
> mtcars[mtcars$mpg > 30, 1:3]
    mpg cyl disp
18 32.4    4 78.7
19 30.4    4 75.7
20 33.9    4 71.1
28 30.4    4 95.1
```

Show columns with names consisting of only two characters. The nchar() function counts the number of characters in a string. The expression nchar(names(mtcars)) == 2 returns a vector of TRUE/FALSE values where TRUE indicates the column name is only two characters in length.

Notice when we specify only one column, the brackets return a vector.

```
> mtcars[mtcars$mpg > 30, "mpg"]
[1] 32.4 30.4 33.9 30.4
```

To get a data frame, set the drop argument to FALSE.

The subset() function allows us to refer to column names without using the \$ extractor function or quoting column names. It also has a drop argument but its default is FALSE. It has four arguments:

- 1. x: the data frame to subset.
- 2. subset: the condition to subset on.
- 3. select: the columns to select.
- 4. drop: an optional logical argument (TRUE/FALSE) to determine whether or not to coerce the output to the lowest possible dimension. The default is FALSE.

We rarely type the first three argument names, x, subset and select, when using subset().

Below we replicate the previous examples using subset().

```
> # rows where mpg > 30 and all columns
> subset(mtcars, mpg > 30)
   mpg cyl disp hp drat
                            wt qsec vs am gear carb
         4 78.7 66 4.08 2.200 19.47
18 32.4
                                      1
                                        1
                                                   1
19 30.4
         4 75.7 52 4.93 1.615 18.52
                                     1 1
                                              4
                                                   2
20 33.9
         4 71.1 65 4.22 1.835 19.90
                                                   1
28 30.4 4 95.1 113 3.77 1.513 16.90 1 1
```

```
> # rows where mpg > 30 and the mpg, wt, and gear columns
> subset(mtcars, mpg > 30, c(mpg, wt, gear))
   mpg
          wt gear
18 32.4 2.200
19 30.4 1.615
20 33.9 1.835
                4
28 30.4 1.513
> # rows where mpg > 30 and the first three columns
> subset(mtcars, mpg > 30, 1:3)
   mpg cyl disp
        4 78.7
18 32.4
19 30.4
         4 75.7
20 33.9 4 71.1
28 30.4 4 95.1
> # rows where mpg > 30 and columns consisting of two characters
> subset(mtcars, mpg > 30, nchar(names(mtcars)) == 2)
          wt vs am
   hp
18 66 2.200 1 1
19 52 1.615 1 1
20 65 1.835 1 1
28 113 1.513 1 1
> # rows where mpg > 30 and mpg column, as a vector
> subset(mtcars, mpg > 30, mpg, drop = TRUE)
[1] 32.4 30.4 33.9 30.4
> # rows where mpg > 30 and mpg column, as a data frame
> subset(mtcars, mpg > 30, mpg)
   mpg
18 32.4
19 30.4
20 33.9
28 30.4
```

Another difference between subsetting brackets and the subset() function is how they handle missing values. Subsetting brackets return missing values while subset() does not. We demonstrate with a toy data frame. Notice the "x" column has a missing value.

```
x y
1 1 12
2 NA 21
3 5 34
```

When we condition on x < 3, the subsetting bracket approach returns a row with NA values.

The subset() approach ignores the missing value.

```
> subset(dframe, x < 3)
    x y
1 1 12</pre>
```

To replicate the <code>subset()</code> result with the subsetting brackets, we need to include an additional condition to only show rows where x is NOT missing. We can do that with the <code>is.na()</code> function. The <code>is.na()</code> function returns TRUE if a value is missing and FALSE otherwise. If we preface with <code>!</code>, we get TRUE if a value is NOT missing and FALSE otherwise.

```
> dframe[dframe$x < 3 & !is.na(dframe$x),]
   x  y
1  1  12</pre>
```

See also the filter() function in the dplyr package and the enhanced subsetting brackets in the data.table package.

4.4 Rename variables

How to rename variables or "column headers".

Python

Column names can be changed using the function .rename(). Below, we change the column names "cyl" and "wt" to "cylinder" and "WT", respectively.

> m	tcars.	rename(col	umns={"	cyl":	"cylin	der", "	wt":"WT	"})			
	mpg	cylinder	disp	hp	drat	WT	qsec	vs	am	gear	carb
0	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
1	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
2	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
3	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
4	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
5	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
6	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
7	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
8	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
9	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4
10	17.8	6	167.6	123	3.92	3.440	18.90	1	0	4	4
11	16.4	8	275.8	180	3.07	4.070	17.40	0	0	3	3
12	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
13	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
14	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
15	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
16	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
17	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
18	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
19	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
20	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
21	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
22	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
23	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
24	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
25	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
26	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
27	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
28	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
29	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
30	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
31	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

Alternatively, column names can be changed by replacing the vector of column names with a new vector. Below, we create a vector of columns that replaces "drat" with "axle_ratio" using conditional match and indexing and "disp" with "DISP" using indexing.

```
> column_names = mtcars.columns.values
+
+ # using conditional match
+ column_names[column_names == "drat"] = "axle_ratio"
+
```

\mathbf{R}

Variable names can be changed by their index (ie, order of columns in the data frame). Below the second column is "cyl". We change the name to "cylinders".

Variable names can also be changed by conditional match. Below we find the variable name that matches "drat" and change to "axle_ratio".

More than one variable name can be changed using a vector of positions or matches.

See also the rename() function in the dplyr.

4.5 Create, replace and remove variables

We often need to create variables that are functions of other variables, or replace existing variables with an updated version.

Python

Adding a new variable using the indexing notation and assigning a result adds a new column.

```
> # add column for Kilometer per liter
+ mtcars['kpl'] = mtcars.mpg/2.352
```

Doing the same with an *existing* column name updates the values in a column.

```
> # update to liters per 100 Kilometers
+ mtcars['kpl'] = 100/mtcars.kpl
```

Alternatively, the . notation can be used to update the values in a column.

```
> # update to liters per 50 Kilometers
+ mtcars.kpl = 50/mtcars.kpl
```

To remove a column, use the .drop() function.

```
> # drop the kpl variable
+ mtcars.drop(columns=['kpl'])
         cyl
                DISP
                       hp axle_ratio
                                                                    carb
     mpg
                                           wt
                                                qsec
                                                      ٧s
                                                          am
                                                              gear
0
              160.0 110
                                                                 4
   21.0
            6
                                 3.90 2.620
                                              16.46
                                                       0
                                                           1
                                                                       4
1
   21.0
            6 160.0 110
                                 3.90
                                       2.875
                                               17.02
                                                                       4
2
   22.8
            4 108.0
                      93
                                 3.85
                                        2.320
                                              18.61
                                                           1
                                                                 4
                                                                       1
3
    21.4
            6 258.0 110
                                 3.08
                                        3.215
                                               19.44
                                                           0
                                                                 3
                                                                       1
4
   18.7
            8 360.0 175
                                 3.15
                                        3.440
                                              17.02
                                                           0
                                                                 3
                                                                       2
                                                       0
5
   18.1
            6 225.0 105
                                        3.460
                                               20.22
                                                                 3
                                                                       1
                                 2.76
6
   14.3
            8 360.0 245
                                                                 3
                                 3.21
                                        3.570
                                              15.84
                                                       0
                                                           0
                                                                       4
7
    24.4
            4 146.7
                       62
                                 3.69
                                        3.190
                                               20.00
                                                       1
                                                           0
                                                                 4
                                                                       2
                                                                 4
                                                                       2
8
    22.8
            4 140.8
                       95
                                 3.92
                                        3.150
                                               22.90
                                                           0
9
    19.2
            6 167.6 123
                                 3.92
                                        3.440
                                               18.30
                                                       1
                                                                 4
                                                                       4
                                                                       4
10 17.8
            6 167.6
                     123
                                 3.92
                                        3.440
                                              18.90
                                                           0
                                                                 4
                                                       1
            8 275.8 180
                                                                       3
11 16.4
                                 3.07
                                       4.070 17.40
                                                           0
```

12	17.3	8	275.8	180	3.07	3.730	17.60	0	0	3	3
13	15.2	8	275.8	180	3.07	3.780	18.00	0	0	3	3
14	10.4	8	472.0	205	2.93	5.250	17.98	0	0	3	4
15	10.4	8	460.0	215	3.00	5.424	17.82	0	0	3	4
16	14.7	8	440.0	230	3.23	5.345	17.42	0	0	3	4
17	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1
18	30.4	4	75.7	52	4.93	1.615	18.52	1	1	4	2
19	33.9	4	71.1	65	4.22	1.835	19.90	1	1	4	1
20	21.5	4	120.1	97	3.70	2.465	20.01	1	0	3	1
21	15.5	8	318.0	150	2.76	3.520	16.87	0	0	3	2
22	15.2	8	304.0	150	3.15	3.435	17.30	0	0	3	2
23	13.3	8	350.0	245	3.73	3.840	15.41	0	0	3	4
24	19.2	8	400.0	175	3.08	3.845	17.05	0	0	3	2
25	27.3	4	79.0	66	4.08	1.935	18.90	1	1	4	1
26	26.0	4	120.3	91	4.43	2.140	16.70	0	1	5	2
27	30.4	4	95.1	113	3.77	1.513	16.90	1	1	5	2
28	15.8	8	351.0	264	4.22	3.170	14.50	0	1	5	4
29	19.7	6	145.0	175	3.62	2.770	15.50	0	1	5	6
30	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8
31	21.4	4	121.0	109	4.11	2.780	18.60	1	1	4	2

\mathbf{R}

Adding a new variable name after the dollar sign notation and assigning a result adds a new column.

```
> # add column for Kilometer per liter
> mtcars$kpl <- mtcars$mpg/2.352</pre>
```

Doing the same with an existing variable updates the values in a column.

```
> # update to liters per 100 Kilometers
> mtcars$kpl <- 100/mtcars$kpl</pre>
```

To remove a variable, assign it NULL.

```
> # drop the kpl variable
> mtcars$kpl <- NULL</pre>
```

See also the $\mathtt{mutate}()$ function in the \mathtt{dplyr} package.

4.6 Create strings from numbers

You may have data that is numeric but that needs to be treated as a string.

Python

You can change the data type of a column in a DataFrame using the astype function

```
> mtcars['am'] = mtcars['am'].astype(str)
+ type(mtcars.am[0]) # check the type of the first item in 'am' column
<class 'str'>
```

A potential number-to-string conversion task in Python might be formatting 5-digit American zip codes. Some zip codes begin with 0, but if stored as a numeric value, the 0 is dropped. For example, consider the following pandas DataFrame. Notice the leading 0 is dropped from two of the zip codes.

```
> zc = pandas.read_csv('data/zc.csv')
+ print(zc)
    state         zip
0     VT     5001
1     VA     22901
2     NH     3282
```

One way to fix this is using the string zfill() method. First we convert the numeric column to string type using the method we just demonstrated. Then we access the "zip" column using zc.zip and the zfill() method using str.zfill with the width parameter set to 5. This pads the string with "0" on the left to make each value 5 characters wide.

If we knew we were importing zip codes using read_csv, we could also use the dtype argument to specify which storage type to use for the "zip" column. Below we pass a dictionary that maps the "str" type to the "zip" column. The result is a properly formatted zip code column.

```
> zc = pandas.read_csv('data/zc.csv', dtype = {'zip': 'str'})
+ print(zc)
    state    zip
0    VT    05001
1    VA    22901
2    NH    03282
```

 \mathbf{R}

The as.character() function takes a vector and converts it to string format.

```
> head(mtcars$am)
[1] 1 1 1 0 0 0
> head(as.character(mtcars$am))
[1] "1" "1" "0" "0" "0"
```

Note we just demonstrated conversion. To save the conversion we need to assign the result to the data frame.

```
> # add new string variable am_ch
> mtcars$am_ch <- as.character(mtcars$am)
> head(mtcars$am_ch)
[1] "1" "1" "1" "0" "0" "0"
```

The factor() function can also be used to convert a numeric vector into a categorical variable. The result is not exactly a string, however. A factor is made of integers with character labels. Factors are useful for character data that have a fixed set of levels (eg, "grade 1", grade 2", etc)

```
> # convert to factor
> head(mtcars$am)
[1] 1 1 1 0 0 0
> head(factor(mtcars$am))
[1] 1 1 1 0 0 0
Levels: 0 1
>
> # convert to factor with labels
> head(factor(mtcars$am, labels = c("automatic", "manual")))
[1] manual manual manual automatic automatic
Levels: automatic manual
```

Again we just demonstrated factor conversion. To save the conversion we need to assign to the data frame.

```
> # create factor variable am_fac
> mtcars$am_fac <- factor(mtcars$am, labels = c("automatic", "manual"))
> head(mtcars$am_fac)
[1] manual manual manual automatic automatic automatic
Levels: automatic manual
```

A common number-to-string conversion task in R is formatting 5-digit American zip codes. Some zip codes begin with 0, but if stored as a numeric value, the 0 is dropped.

```
> zip_codes <- c(03766, 03748, 22901, 03264)
> zip_codes
[1] 3766 3748 22901 3264
```

We need to store the zip code as a character value so the 0 is preserved. One way to do this is via the sprintf() function in base R. The first argument is the format string or conversion specification. A conversion specification begins with "%". The following "0" and "5" says to format the zip_codes vector as a 5-digit string padded by zeroes on the left. The final "i" says we're working with integer values.

```
> sprintf("%05i", zip_codes)
[1] "03766" "03748" "22901" "03264"
```

See also the str_pad() function in the stringr package.

4.7 Create numbers from strings

String variables that ought to be numbers usually have some character data in the values such as units (eg, "4 cm"). To create numbers from strings it's important to remove any character data that cannot be converted to a number.

Python

The astype(float) or astype(int) function will coerce strings to numerical representation.

For demonstration, let's say we have the following numpy array.

```
> import numpy as np
+ weight = np.array(["125 lbs.", "132 lbs.", "156 lbs."])
```

The astype(float) function throws an error due to the presence of strings. The astype() function is for numpy arrays.

```
> try:
+ weight.astype(float)
+ except ValueError:
+ print("ValueError: could not convert string to float: '125 lbs.'")
ValueError: could not convert string to float: '125 lbs.'
```

One way to approach this is to first remove the strings from the objects and then use astype(float). Below we use the strip() function to find "lbs." using a list comprehension.

```
> # [] indicates a list in python
+ # np.array() changes the list back into an array
+ weight = np.array([w.strip(" lbs.") for w in weight])
```

Now we can use the astype() function to change the elements in weight from str to float.

```
> weight.astype(float)
array([125., 132., 156.])
```

\mathbf{R}

The as.numeric() function will attempt to coerce strings to numeric type *if* possible. Any non-numeric values are coerced to NA.

For demonstration, let's say we have the following vector.

```
> weight <- c("125 lbs.", "132 lbs.", "156 lbs.")
```

The as.numeric() function returns all NA due to presence of character data.

```
> as.numeric(weight)
Warning: NAs introduced by coercion
[1] NA NA NA
```

There are many ways to approach this. A common approach is to first remove the characters and then use as.numeric(). Below we use the gsub() function to find "lbs." and replace with nothing (find-and-replace procedures are discussed more below).

```
> weightN <- gsub("lbs.", "", weight)
> as.numeric(weightN)
[1] 125 132 156
```

The parse_number() function in the **readr** package can often take care of these situations automatically.

```
> readr::parse_number(weight)
[1] 125 132 156
```

4.8 Combine strings

String concatenation—turning 'Jane' and 'Smith' into 'Jane Smith'—is easily done in both languages.

Python

The + operator can combine strings in Python.

```
> species = 'yellow-bellied sea snake'
+ tail_shape = 'paddle-shaped'
+
+ statement = 'The ' + species + ' has a ' + tail_shape + ' tail that helps it swim.'
+ print(statement)
The yellow-bellied sea snake has a paddle-shaped tail that helps it swim.
```

\mathbf{R}

The paste() and paste() functions combine strings in R. The former concatenates strings and places spaces between them; the latter concatenates sans spaces.

```
> species <- 'rainbow boa'
> appearance <- 'iridescent'
> location <- 'Central and South America'
>
> statement1 <- paste('The', species, 'has an', appearance, 'sheen.')
> statement1
[1] "The rainbow boa has an iridescent sheen."
>
> # Note that spaces must be provided explicitly when using pasteO()
> statement2 <- pasteO('The ', species, ' is found in ', location)
> statement2
[1] "The rainbow boa is found in Central and South America"
```

4.9 Finding and replacing patterns within strings

This section reviews key functions in Python and R for finding and replacing character patterns. The functions we discuss can search for fixed character patterns (e.g., "Meredith Rollins" to case-sensitively match that name and that name alone) or regular expression (regex) patterns (e.g., $\$ to capture all instances of >=1 word character). Note that in R, meta characters, like w (to match word characters) and d (to match digits), are escaped with two backslashes (e.g., $\$ and $\$ In Python, regex patterns are generally headed by r, which allows meta characters in the regex itself to be escaped with just one $\$ (e.g., $r"\$). Regex is an enormous topic, and we don't discuss it at

any length here, but you can learn more about regular expressions—and how they're implemented in different programming languages—at these resources: https://www.regular-expressions.info/; https://regexone.com/

Python

The **re** module provides a set of functions for searching and manipulating strings. The **search()** function does exactly as its name suggests: It identifies matches for a fixed or regex character pattern in a string. **sub()** searches for and replaces character patterns (fixed or regex). The **count** argument in **sub()** allows a user to specify how many instances of the matched pattern they want to to replace; e.g., use **count = 1** to replace just the first instance of a match.

```
> import re
+ statement = 'Pencils with an HB graphite grade are commonly used for writing. An HB
+ # Search for "HB" using fixed and regex patterns
+ search_result1 = re.search(pattern = "HB", string = statement)
+ print(search_result1)
<re.Match object; span=(16, 18), match='HB'>
> search_result2 = re.search(pattern = r"[H,B]{2}", string = statement)
+ print(search_result2)
+ # Replace all instances of "HB"
<re.Match object; span=(16, 18), match='HB'>
> all_replaced = re.sub(pattern = 'HB', repl = 'HB (hard black)', string = statement)
+ print(all_replaced)
+ # Replace just the first instance of HB
Pencils with an HB (hard black) graphite grade are commonly used for writing. An HB (hard black)
> one_replaced = re.sub(pattern = 'HB', repl = 'HB (hard black)', string = statement,
+ print(one_replaced)
+ # Search and replace using a regex pattern instead of a fixed string
Pencils with an HB (hard black) graphite grade are commonly used for writing. An HB per
> regex_replaced = re.sub(pattern = r'(?<=\.)\s{1}', repl = '\n', string = statement)
+ print(regex_replaced)
Pencils with an HB graphite grade are commonly used for writing.
An HB pencil is approximately equal to a #2 pencil.
```

\mathbf{R}

The standard-issue string-search function is grep(); it returns the index of the elements in a set of one or more strings for which a pattern match was found.

(grepl() acts similarly but returns a vector of TRUE/FALSE indicating whether a match was found in each string passed to the function.) The functions sub() and gsub() can be used to find and replace instances of a pattern: The former replaces just the first instance; the latter replaces all instances. The search pattern can be provided as a raw character string or as a regular expression.

```
> statements <- c('Great Pencil Co. primarily sells pencils of the following grades: HB; B; and 3
                  'Great Pencil Co. has its headquarters in Maine, and Great Pencil Co. has suppl
>
> # Search for pattern and return indexex of elements for which match is found
> grep(pattern = 'pencil', x = statements) # When searched for case sensitively, "pencil" is only
[1] 1
> grep(pattern = '(?i)pencil', x = statements) # When searched for case insensitively, "P/pencil
[1] 1 2
> # Replace the first instance of a pattern (Co. --> Company)
> revised <- sub(pattern = 'Co.', replacement = 'Company', x = statements)
[1] "Great Pencil Company primarily sells pencils of the following grades: HB; B; and 3B."
[2] "Great Pencil Company has its headquarters in Maine, and Great Pencil Co. has supplied the No
> # Replace all instances of a pattern (; --> ,)
> revised2 <- gsub(pattern = ';', replacement = ',', x = revised)
> revised2
[1] "Great Pencil Company primarily sells pencils of the following grades: HB, B, and 3B."
[2] "Great Pencil Company has its headquarters in Maine, and Great Pencil Co. has supplied the No
> # Find and replace a pattern using regex (3B --> 2B)
> final <- sub(pattern = '\d{1}', replacement = '2', x = revised2)
> final
[1] "Great Pencil Company primarily sells pencils of the following grades: HB, B, and 2B."
[2] "Great Pencil Company has its headquarters in Maine, and Great Pencil Co. has supplied the No
```

Those functions can be used to trim excess (or all) white space in character strings.

```
> spaced_string <- c('This string started out with too many spaces.')
> # Replace all instances of >=2 spaces with single spaces
> gsub(pattern = '\\s{2,}', replacement = ' ', x = spaced_string)
[1] "This string started out with too many spaces."
> # Remove all white space
> collapse_these <- c('9:00 - 10:15', '10:15 - 11:30', '11:30 - 12:00')
> gsub(pattern = '\\s', replacement = '', x = collapse_these)
[1] "9:00-10:15" "10:15-11:30" "11:30-12:00"
```

The package stringi also provides an array of string-search and string-

manipulation functions, including stri_detect(), stri_replace(), and stri_extract(), all of which easily handle fixed and regex search patterns. For example:

```
> library(stringi)
> user_dat <- data.frame(name = c('Shire, Jane E', 'Winchester, Marcus L', 'Fox, Sal')
> user_dat
                  name id number
         Shire, Jane E
                          aaa101
2 Winchester, Marcus L
                          aaa102
              Fox, Sal
                          aaa103
> # Say we want to use regex patterns and the stringi package to eliminate the 'aaa' p
> # the user IDs and then add middle initials---for those users who have them---to the
> user_dat$id_number <- stri_replace(user_dat$id_number, regex = '\\w{3}(?=\\d+)', rep
> user_dat$middle_initial <- stri_extract(user_dat$name, regex = '\\b\\w{1}\\b')
> user_dat
                  name id_number middle_initial
         Shire, Jane E
                             101
                                              Ε
                                              L
2 Winchester, Marcus L
                             102
              Fox, Sal
                             103
                                            <NA>
```

4.10 Change case

How to change the case of strings. The most common case transformations are lower case, upper case, and title case.

Python

The lower(), upper(), and title() functions convert case to lower, upper, and title, respectively. We can use a list comprehension to apply these functions to each string in a list.

```
> col_names = [col.upper() for col in mtcars.columns]
+ mtcars.columns = col_names
```

\mathbf{R}

The tolower() and toupper() functions convert case to lower and upper, respectively.

```
> names(mtcars) <- toupper(names(mtcars))</pre>
> names(mtcars)
                                               "HP"
 [1] "MPG"
                   "CYLINDERS"
                                 "DISP"
                                                             "AXLE_RATIO"
 [6] "WEIGHT"
                                                             "GEAR"
                   "QSEC"
                                 "ENGINE"
                                               "AM"
[11] "CARB"
                   "AM_CH"
                                 "AM_FAC"
> names(mtcars) <- tolower(names(mtcars))</pre>
> names(mtcars)
                   "cylinders" "disp"
                                               "hp"
                                                             "axle ratio"
 [1] "mpg"
 [6] "weight"
                   "gsec"
                                 "engine"
                                               "am"
                                                             "gear"
[11] "carb"
                   "am ch"
                                 "am fac"
```

The **stringr** package provides a convenient title case conversion function, **str_to_title()**, which capitalizes the first letter of each string.

4.11 Drop duplicate rows

How to find and drop duplicate elements.

Python

The duplicated() function determines which rows of a DataFrame are duplicates of previous rows.

First, we create a DataFrame with a duplicate row by using the pandas concat() function. concat() combines DataFrames by rows or columns, row by default.

```
> # create DataFrame with duplicate rows
+ import pandas as pd
+ mtcars2 = pd.concat([mtcars.iloc[0:3,0:6], mtcars.iloc[0:1,0:6]])
```

The duplicated() function returns a logical vector. TRUE indicates a row is a duplicate of a previous row.

```
> # create DataFrame with duplicate rows
+ mtcars2.duplicated()
0 False
```

```
1 False
2 False
0 True
dtype: bool
```

\mathbf{R}

The duplicated() function "determines which elements of a vector or data frame are duplicates of elements with smaller subscripts". (from ?duplicated)

```
> # create data frame with duplicate rows
> mtcars2 <- rbind(mtcars[1:3,1:6], mtcars[1,1:6])</pre>
> # last row is duplicate of first
> mtcars2
  mpg cylinders disp hp axle_ratio weight
1 21.0
              6 160 110 3.90 2.620
2 21.0
              6 160 110
                               3.90 2.875
3 22.8
              4 108 93
                               3.85 2.320
4 21.0
              6 160 110
                               3.90 2.620
```

The duplicated() function returns a logical vector. TRUE indicates a row is a duplicate of a previous row.

```
> # last row is duplicate
> duplicated(mtcars2)
[1] FALSE FALSE TRUE
```

The TRUE/FALSE vector can be used to extract or drop duplicate rows. Since TRUE in indexing brackets will keep a row, we can use! to negate the logicals and keep those that are "NOT TRUE"

```
> # drop the duplicate and update the data frame
> mtcars3 <- mtcars2[!duplicated(mtcars2),]
> mtcars3
    mpg cylinders disp hp axle_ratio weight
1 21.0     6 160 110     3.90    2.620
2 21.0     6 160 110     3.90    2.875
3 22.8     4 108 93     3.85    2.320
```

```
> # extract and investigate the duplicate row
> mtcars2[duplicated(mtcars2),]
  mpg cylinders disp hp axle_ratio weight
4 21 6 160 110 3.9 2.62
```

The anyDuplicated() function returns the row number of duplicate rows.

```
> anyDuplicated(mtcars2)
[1] 4
```

4.12 Format dates

With formatted dates we can calculate elapsed time, extract components of a date, properly order names of months, and more.

Python

The Python module datetime can be used to create various date and time objects. Here we will discuss 4 of the main classes within datetime that are most useful.

The first class we will go over is the date() class. This creates a "date" object whose only attributes are year, month, day.

Here we create a date object using the date class. The attributes are specified as integers in the argument of date() in this order date(year, month, day).

```
> import datetime as dt
+
+ x = dt.date(2001, 4, 12)
+ print(x)
2001-04-12
```

To get today's date, we can use the date.today() function:

```
> today = dt.date.today()
+ print(today)
2022-03-23
```

Note that the output of both x and today are only year-month-day because they are date objects.

We can extract each of these attributes (year, month, day) from the date object as follows:

```
> today.year
2022
> today.month
3
> today.day
23
```

Next we will discuss the time() class. This class creates time objects containing information about only a time. The attributes that go into the time() class are hours, minutes, seconds in that order. Like the date class, these attributes must be inputted as integers.

```
> y = dt.time(11, 34, 56)
+ print(y)
11:34:56
```

If you want a time object containing only hours and minutes, only seconds, etc. you can specify the attributes by name when creating the time object.

```
> only_hrs = dt.time(hour = 10)
+ only_mins = dt.time(minute = 55)
+
+ print(only_hrs)
10:00:00
> print(only_mins)
00:55:00
```

Again similar to the date class, we can extract hour, minute, and second attributes from the time objects:

```
> y.hour
11
> y.minute
34
> y.second
56
> y.microsecond
0
```

Now we will talk about the datetime() class that creates a datetime object containing information about both date and time. The attributes must be inputted as integers and are year, month, day, hour, minute, second, in that order. Like the date and time classes, we can speficy specific attributes in the argument using the attrubute names as well. If we don't specify any time components, the datetime object defaults to time 00:00:00.

```
> # Input attributes in order
+ z = dt.datetime(1981, 4, 12, 11, 34, 56)
+ print(z)
+ # Input attributes using attribute names (any order)
1981-04-12 11:34:56
```

```
> z2= dt.datetime(year = 2021, day = 6, month = 12, hour = 6)
+ print(z2)
+
+ # No time attributes
2021-12-06 06:00:00
> z3 = dt.datetime(1981, 4, 12)
+ print(z3)
1981-04-12 00:00:00
```

Again, we can extract attributes in exactly the same way as the date and time classes.

```
> z.year
1981
> z.day
12
> z.hour
11
```

The final class we will discuss is the timedelta class. This class is used to store date/time differences between date objects.

The default settings for a timedelta object are as follows: timedelta(weeks=0, days=0, hours=0,minutes=0, seconds=0, milliseconds=0, microseconds=0)

Here is an example of how to add and subtract dates aand times using these objects.

```
> # Create a datetime object for the current time that we will increment
+ d1 = dt.datetime.now()
+ print(d1)
+
+ # Add 550 days to our datetime object
2022-03-23 22:55:44.630640
> d2 = d1 + dt.timedelta(days = 550)
+ print(d2)
+
+ # Subtract 5 hours from our datetime object
2023-09-24 22:55:44.630640
> d3 = d1 - dt.timedelta(hours = 5)
+ print(d3)
2022-03-23 17:55:44.630640
```

Finally, we will discuss how to convert strings to datetime objects and vice versa.

The attribute strftime() converts datetime objects to strings. In the argument of strftime() can specify the format you would like.

```
> d1
datetime.datetime(2022, 3, 23, 22, 55, 44, 630640)
> d1.strftime("%A %m %Y")
'Wednesday 03 2022'
> d1.strftime("%a %m %y")
'Wed 03 22'
```

The attribute strptime() converts strings into datetime objects. In the argument of strptime() you must specify the string and then the format of the string.

```
> d4 = "27/10/98 11:03:9.033"
+
+ d1.strptime(d4, "%d/%m/%y %H:%M:%S.%f")
datetime.datetime(1998, 10, 27, 11, 3, 9, 33000)
```

\mathbf{R}

Dates in R can be stored as a Date class or a Date-Time class. Dates are stored as the number of days since January 1, 1970. Date-Times are stored as the number of seconds since January 1, 1970. With dates stored in this manner we can calculate elapsed time in units such as days, weeks, hours, minutes, and so forth.

Below are the dates of the first five NASA Columbia Space Shuttle flights entered as a character vector.

R does not immediately recognize these as a Date class. To format as a Date class, we can either use the base R as.Date() function or one of the convenience functions in the **lubridate** package. The as.Date() function requires a specified POSIX conversion specification as documented in ?strptime. Below the conversion code "%d %B %Y" says Date is entered as two digit day of month (%d), full month name (%B), and year with century (%Y).

```
> date1 <- as.Date(date, format = "%d %B %Y")
> date1
[1] "1981-04-12" "1981-11-12" "1982-03-22" "1982-06-27" "1982-11-11"
```

The dates now print in year-month-date format, however they are stored internally as number of days since January 1, 1970. This can be seen by using as.numeric() on the "date1" vector.

```
> as.numeric(date1)
[1] 4119 4333 4463 4560 4697
```

The **lubridate** package provides a series of functions that are permutations of the letters "m", "d", and "y" to represent the order of date components. To format the original "date" vector, we use the dmy() function since the date components are ordered as day, month and year. Notice we must load the **lubridate** package to use this function.

```
> library(lubridate)
> date2 <- dmy(date)
> date2
[1] "1981-04-12" "1981-11-12" "1982-03-22" "1982-06-27" "1982-11-11"
```

When dates are formatted we can easily extract information such as day of week or month. For example to extract the day of week of the launches as an ordered factor, we can use the **lubridate** function wday() with label=TRUE and abbr = FALSE.

```
> wday(date2, label = TRUE, abbr = FALSE)
[1] Sunday Thursday Monday Sunday Thursday
7 Levels: Sunday < Monday < Tuesday < Wednesday < Thursday < ... < Saturday</pre>
```

To calculate elapsed time between launches in days we can use the base R diff() function.

```
> diff(date2)
Time differences in days
[1] 214 130 97 137
```

To store a date as a Date-Time class we need to include a time component. Below are the first five Columbia launch dates with times. UTC refers to Universal Coordinated Time.

To format as a Date-Time class we can use either the base R as.POSIXct() function or one of the convenience functions in the **lubridate** package. To use as.POSIXct() we need to include additional POSIX conversion specifications for the hour, minute and second of launch. The "%H:%M:%S" specification refers to hours, minutes and seconds. The tz argument specifies the time zone of the times.

When we use as.numeric() on the "datetime1" vector we see it is stored as number of seconds since January 1, 1970.

```
> as.numeric(datetime1)
[1] 355924804 374425800 385660800 394038000 405865140
```

Using **lubridate** we can append _hms() to any of the "mdy" functions to format dates with time components as a Date-Time class. Notice the default time zone in **lubridate** is UTC.

```
> datetime2 <- dmy_hms(datetime)
> datetime2
[1] "1981-04-12 12:00:04 UTC" "1981-11-12 15:10:00 UTC"
[3] "1982-03-22 16:00:00 UTC" "1982-06-27 15:00:00 UTC"
[5] "1982-11-11 12:19:00 UTC"
```

To calculate elapsed time between launches in hours, we can use the **lubridate** function time_length() with the unit set to "hours". Below we use diff() and then pipe to time_length().

```
> diff(datetime2) |> time_length(unit = "hours")
[1] 5139.166 3120.833 2327.000 3285.317
```

For more information on working with dates and times in R, see the vignette accompanying the **lubridate** package.

4.13 Randomly sample rows

How to take a random sample of rows from a data frame. The sample is usually either a fixed size or a proportion.

Python

The pandas package provide a function for taking a sample of fixed size or a proportion. To sample with replacement, set replace = TRUE.

Additionally, the random sample will change every time the code is run. To always generate the same "random" sample, set random_state to any positive integer.

To create a sample with a fixed number of rows, use the n argument.

```
> # sample 5 rows from mtcars
+ mtcars.sample(n=5, replace=True)
     MPG CYL
                          AXLE_RATIO
                                             ۷S
                                                     GEAR CARB
                                                                      KPL
                DISP
                       ΗP
                                                 AM
30
   15.0
            8 301.0 335
                                  3.54
                                              0
                                                  1
                                                         5
                                                              8
                                                                 3.188776
2
    22.8
            4
              108.0
                       93
                                  3.85
                                        . . .
                                              1
                                                  1
                                                         4
                                                              1
                                                                 4.846939
18
    30.4
            4
                                                  1
                                                              2
                                                                 6.462585
                75.7
                       52
                                  4.93
                                              1
                                                         4
    22.8
            4
                                                  0
                                                         4
                                                              2
8
              140.8
                       95
                                  3.92
                                              1
                                                                4.846939
29 19.7
            6 145.0 175
                                  3.62
                                              0
                                                  1
                                                         5
                                                              6 4.187925
[5 rows x 12 columns]
```

To create a sample of a proportion, use the frac argument.

```
> # sample 20% of rows from mtcars
+ mtcars.sample(frac = 0.20, random_state=1)
     MPG CYL
                DISP
                       HP AXLE RATIO
                                              VS
                                                      GEAR CARB
                                                                       KPL
                                        . . .
                                                  AM
27
    30.4
            4
                95.1 113
                                  3.77
                                         . . .
                                               1
                                                   1
                                                         5
                                                              2
                                                                 6.462585
3
    21.4
            6 258.0 110
                                  3.08
                                               1
                                                   0
                                                         3
                                                              1
                                                                 4.549320
22 15.2
            8 304.0 150
                                  3.15
                                               0
                                                   0
                                                         3
                                                              2
                                                                 3.231293
                                        . . .
18 30.4
            4
                                  4.93
                                                         4
                                                              2 6.462585
                75.7
                       52
                                               1
                                                   1
23 13.3
              350.0 245
                                  3.73
                                               0
                                                   0
                                                         3
                                                                 2.827381
            8
   32.4
                                               1
                                                   1
                                                         4
                                                              1
                                                                 6.887755
17
                78.7
                       66
                                  4.08
[6 rows x 12 columns]
```

The numpy function random.choice() in combination with the loc() function can be used to sample from a DataFrame.

The random.choice() function creates a random sample according to the given parameters. The loc() function is used to access rows and columns by index.

```
> # import the numpy package
+ import numpy as np
+
+ # create a random sample of size 5 with replacement
```

```
+ random_sample = np.random.choice(len(mtcars), (5,), replace=True)
+ # use random_sample to sample from mtcars
+ mtcars.loc[random_sample,]
                                                                           KPL
     MPG CYL
                 DISP
                         ΗP
                              AXLE_RATIO
                                                 ۷S
                                                     AM
                                                          GEAR CARB
8
    22.8
             4
                140.8
                                                                   2
                         95
                                    3.92
                                                  1
                                                      0
                                                             4
                                                                      4.846939
10
    17.8
             6
                167.6
                        123
                                    3.92
                                                  1
                                                      0
                                                             4
                                                                  4
                                                                      3.784014
                                           . . .
    19.2
             8
                400.0
                        175
                                    3.08
                                                             3
                                                                   2
                                                                      4.081633
                                           . . .
19
    33.9
             4
                 71.1
                         65
                                    4.22
                                                  1
                                                      1
                                                             4
                                                                      7.206633
                                           . . .
                                                                   1
   16.4
                275.8
                                                      0
                                                             3
11
             8
                        180
                                    3.07
                                                  0
                                                                   3
                                                                      3.486395
[5 rows x 12 columns]
```

The random sample will change every time the code is run. To always generate the same "random" sample, use the random.seed() function with any positive integer.

```
> # setting seed to always get same random sample
+ np.random.seed(123)
+ # create a random sample of size 5 with replacement
+ sample = np.random.choice(len(mtcars), (5,), replace=True)
+ mtcars.loc[sample,]
     MPG CYL
                 DISP
                        HP
                            AXLE_RATIO
                                           WT
                                                 QSEC
                                                       VS AM
                                                               GEAR
                                                                     CARB
                                                                                 KPL
30
    15.0
            8
               301.0
                       335
                                   3.54
                                         3.57
                                                14.60
                                                        0
                                                           1
                                                                  5
                                                                        8
                                                                           3.188776
    15.2
            8
               275.8
                       180
                                   3.07
                                         3.78
                                                18.00
                                                        0
                                                           0
                                                                  3
                                                                        3
13
                                                                           3.231293
                                                                  5
    15.0
            8
               301.0
                       335
                                   3.54
                                         3.57
                                                14.60
                                                                        8
                                                                           3.188776
2
    22.8
               108.0
                                   3.85
                                         2.32
                                                                  4
            4
                        93
                                               18.61
                                                                           4.846939
                                                        1
                                                           1
                                                                        1
               351.0
                                                                  5
28
   15.8
            8
                       264
                                   4.22 3.17
                                               14.50
                                                        0
                                                           1
                                                                           3.358844
```

\mathbf{R}

There are many ways to sample rows from a data frame in R. The **dplyr** package provides a convenience function, **slice_sample()**, for taking either a fixed sample size or a proportion.

```
> # sample 5 rows from mtcars
> dplyr::slice_sample(mtcars, n = 5)
   mpg cylinders disp hp axle_ratio weight qsec engine am gear carb am_ch
               8 440.0 230
1 14.7
                                  3.23 5.345 17.42
                                                          0
                                                             0
                                                                  3
                                                                              0
2 17.3
               8 275.8 180
                                                          0
                                                             0
                                                                  3
                                                                        3
                                                                              0
                                  3.07
                                        3.730 17.60
3 30.4
                  95.1 113
                                  3.77
                                        1.513 16.90
                                                          1
                                                             1
                                                                  5
                                                                        2
                                                                              1
4 32.4
                                                                  4
                  78.7
                        66
                                  4.08
                                       2.200 19.47
                                                          1
                                                             1
                                                                        1
                                                                              1
               8 275.8 180
                                  3.07 4.070 17.40
                                                          0
                                                                        3
                                                                              0
5 16.4
```

```
am_fac
1 automatic
2 automatic
3
    manual
4
    manual
5 automatic
> # sample 20% of rows from mtcars
> dplyr::slice_sample(mtcars, prop = 0.20)
  mpg cylinders disp hp axle_ratio weight qsec engine am gear carb am_ch
         4 108.0 93 3.85 2.320 18.61 1 1
1 22.8
                                                       4
2 24.4
                            3.69 3.190 20.00
           4 146.7 62
                                               1 0
                                                                 0
3 14.3
           8 360.0 245
                           3.21 3.570 15.84
                                               0 0
                                               1 0
4 21.4
           6 258.0 110
                            3.08 3.215 19.44
                                                      3
                                                            1
                                                                 0
                                               0 1
           8 351.0 264
5 15.8
                            4.22 3.170 14.50
                                                      5
                                                           4
                                                                 1
                                               1 1
                                                     4 2
6 21.4
           4 121.0 109
                            4.11 2.780 18.60
                                                                 1
    am_fac
1
    manual
2 automatic
3 automatic
4 automatic
5
    manual
    manual
```

To sample with replacement, set replace = TRUE.

The base R functions sample() and runif() can be combined to sample fixed sizes or approximate proportions.

```
> # sample 5 rows from mtcars
> # get random row numbers
> i <- sample(nrow(mtcars), size = 5)</pre>
> # use i to select rows
> mtcars[i,]
   mpg cylinders disp hp axle_ratio weight qsec engine am gear carb am_ch
18 32.4
          4 78.7 66 4.08 2.200 19.47
                                                 1 1 4
5 18.7
            8 360.0 175
                              3.15 3.440 17.02
                                                   0 0
                                                           3
                                                               2
                                                                     0
25 19.2
                                                   0 0 3
            8 400.0 175
                              3.08 3.845 17.05
                                                                     0
20 33.9
            4 71.1 65
                             4.22 1.835 19.90
                                                   1 1 4
                                                             1
                                                                     1
10 19.2
            6 167.6 123
                          3.92 3.440 18.30
                                                   1 0 4
                                                                     0
     am_fac
18
     manual
5 automatic
25 automatic
20
     manual
10 automatic
```

```
> # sample about 20% of rows from mtcars
> # generate random values on range of [0,1]
> i <- runif(nrow(mtcars))</pre>
> # use i < 0.20 logical vector to
> # select rows that correspond to TRUE
> mtcars[i < 0.20,]
    mpg cylinders disp hp axle_ratio weight qsec engine am gear carb am_ch
                                  3.69 3.190 20.00
8
  24.4
                4 146.7
                         62
                                                             0
9 22.8
                                  3.92 3.150 22.90
                                                                       2
                4 140.8 95
                                                          1
                                                             0
                                                                             0
                                                                       2
25 19.2
                8 400.0 175
                                  3.08 3.845 17.05
                                                          0 0
                                                                  3
                                                                             0
26 27.3
                4 79.0 66
                                  4.08 1.935 18.90
                                                          1 1
                                                                  4
                                                                       1
                                                                             1
                4 120.3 91
                                                                  5
                                                                       2
27 26.0
                                  4.43 2.140 16.70
                                                          0 1
                                                                             1
30 19.7
                6 145.0 175
                                  3.62 2.770 15.50
                                                          0 1
                                                                  5
                                                                       6
                                                                             1
      am fac
8
  automatic
9 automatic
25 automatic
26
      manual
27
      manual
30
      manual
```

The random sample will change every time the code is run. To always generate the same "random" sample, use the set.seed() function with any positive integer.

```
> # always get the same random sample
> set.seed(123)
> i <- runif(nrow(mtcars))</pre>
> mtcars[i < 0.20,]
    mpg cylinders disp hp axle_ratio weight qsec engine am gear carb am_ch
                                                              0
6 18.1
                6 225.0 105
                                   2.76
                                           3.46 20.22
                                                           1
                                                                    3
                                                                         1
                                                                               0
15 10.4
                8 472.0 205
                                   2.93
                                           5.25 17.98
                                                           0 0
                                                                    3
                                                                               0
18 32.4
                4 78.7 66
                                   4.08
                                           2.20 19.47
                                                           1 1
                                                                    4
                                                                         1
                                                                               1
30 19.7
                6 145.0 175
                                   3.62
                                           2.77 15.50
                                                           0 1
                                                                    5
                                                                               1
      am_fac
6 automatic
15 automatic
18
      manual
30
      manual
```

Chapter 5

Combine, Reshape and Merge

This chapter looks at various strategies for combining, reshaping, and merging data

5.1 Combine rows

Combining rows may be thought of as "stacking" rectangular data structures.

Python

The pandas function **concat** function "binds" rows. It takes a list of pandas DataFrame objects. The second argument **axis** specifies a row bind when 0 and a column bind when 1. The default value is 0. The column names of the DataFrames should match, otherwise the DataFrame fills with NaNs. You can bind rows with different column types.

```
> import pandas as pd
+
+ d1 = pd.DataFrame({'x':[4,5,6], 'y':['a','b','c']})
+ d2 = pd.DataFrame({'x':[3,2,1], 'y':['d','e','f']})
+
+ # create list of DataFrame objects
+ frames = [d1, d2]
+ combined_df = pd.concat(frames)
+
```

The following code is an example of when column names do not match, resulting in NaNs in the DataFrame.

```
+ # DataFrame with different column names
+ d1 = pd.DataFrame({'x':[4,5,6], 'z':['a','b','c']})
+ d2 = pd.DataFrame({'x':[3,2,1], 'y':['d','e','f']})
+ # create list of DataFrame objects
+ frames = [d1, d2]
+ combined_df = pd.concat(frames)
+ combined_df
       z
  Х
            У
0
  4
       a NaN
       b NaN
2
  6
       c NaN
  3 NaN
            d
1 2 NaN
             е
2 1 NaN
            f
```

\mathbf{R}

The rbind() function "binds" rows. It takes two or more objects. To row bind data frames, the column names must match, otherwise an error is returned. If columns being stacked have differing variable types, the values will be coerced according to logical < integer < double < complex < character. (E.g., if you stack a set of rows with type logical in column J on a set of rows with type character in column J, the output will have column J as type character.)

```
> d1 <- data.frame(x = 4:6, y = letters[1:3])
> d2 <- data.frame(x = 3:1, y = letters[4:6])
> rbind(d1, d2)
    x y
1 4 a
```

```
2 5 b
3 6 c
4 3 d
5 2 e
6 1 f
```

See also the bind_rows() function in the dplyr package.

5.2 Combine columns

Combining columns may be thought of as setting rectangular data structures next to each other.

Python

The concat function also "binds" columns. It takes two or more objects. The second argument axis specifies a row bind when 0 and a column bind when 1. The default value is 0. To column bind data frames, the number of rows must match; otherwise, the function throws an error.

\mathbf{R}

The cbind() function "binds" columns. It takes two or more objects. To column bind data frames, the number of rows must match; otherwise, the object with fewer rows will have rows "recycled" (if possible) or an error will be returned.

```
> d1 <- data.frame(x = 10:13, y = letters[1:4])
> d2 <- data.frame(x = c(23,34,45,44))
> cbind(d1, d2)
        x y x
1 10 a 23
2 11 b 34
3 12 c 45
4 13 d 44
```

See also the bind_cols() function in the dplyr package.

5.3 Reshaping data

The next two sections discuss how to reshape data from wide to long and from long to wide. "Wide" data are structured such that multiple values associated with a given unit (e.g., a person, a cell culture, etc.) are placed in the same row:

Long data, conversely, are structured such that all values are contained in one column, with another column identifying what value is given in any particular row ("time 1," "time 2," etc.):

```
id time score
```

```
3
1 larry
            1
                   0
2 larry
            2
                   6
3
    moe
            1
                   3
4
    moe
            2
                   2
5 curly
            1
6 curly
            2
                   1
```

Shifting between these two data formats is often necessary for implementing certain statistical techniques or representing data with particular visualizations.

5.3.1 Wide to long

Python

\mathbf{R}

In base R, the reshape() function can take data from wide to long or long to wide. The **tidyverse** also provides reshaping functions: pivot_longer() and pivot_wider(). The **tidyverse** functions have a degree of intuitiveness and usability that may make them the go-to reshaping tools for many R users. We give examples below using both base R and **tidyverse**.

Say we begin with a wide data frame, df_wide, that looks like this:

```
id sex wk1 wk2 wk3
1 1 m 16 7 15
2 2 m 12 19 10
3 3 f 8 15 7
```

To lengthen a data frame using reshape(), a user provides arguments specifying the columns that identify values' origins (person, cell culture, etc.), the columns containing values to be lengthened, and the desired names for new columns in long data:

```
> df_long <- reshape(df_wide,</pre>
                          direction = 'long',
                          idvar = c('id', 'sex'), # column(s) that uniquely identifies/y each rou
                          varying = c('wk1', 'wk2', 'wk3'), # variables that contain the values
                          v.names = 'val', # desired name of column in long data that will conta-
                          timevar = 'week') # desired name of column in long data that will iden:
> df_long
      id sex week val
1.m.1
      1
           m
                1
                   16
           m
2.m.1 2
                1
                   12
```

```
3.f.1
       3
                  1
                       8
1.m.2
                  2
                      7
       1
            m
2.m.2
            m
                  2
                     19
3.f.2 3
            f
                  2
                     15
                  3
1.m.3
                     15
                  3
2.m.3
       2
                     10
            \mathbf{m}
3.f.3 3
```

The **tidyverse** function for taking data from wide to long is pivot_longer(). To lengthen df_wide using pivot_longer(), a user would write:

```
> library(tidyverse)
> df_long_PL <- pivot_longer(df_wide,</pre>
                                 cols = -c('id', 'sex'), # columns that contain the valu
                                 names_to = 'week', # desired name of column in long dat
                                 values_to = 'val') # desired name of column in long dat
> df_long_PL
# A tibble: 9 x 4
     id sex
              week
                       val
  <int> <chr> <chr> <int>
1
      1 m
              wk1
                        16
2
      1 m
               wk2
                         7
3
                        15
              wk3
      1 m
4
      2 m
              wk1
                        12
5
      2 m
              wk2
                        19
6
      2 m
              wk3
                        10
7
      3 f
              wk1
                         8
8
      3 f
              wk2
                        15
                         7
9
      3 f
              wk3
```

pivot_longer() is particularly useful (a) when dealing with wide data that contain multiple sets of repeated measures in each row that need to be lengthened separately (e.g., two monthly height measurements and two monthly weight measurements for each person) and (b) when column names and/or column values in the long data need to be extracted from column names of the wide data using regular expressions.

For example, say we begin with a wide data frame, animals_wide, in which every row contains two values for each of two different measures:

	animal	lives_in_water	jan_playfulness	feb_playfulness	jan_excitement
1	dolphin	TRUE	6.0	5.5	7.0
2	porcupine	FALSE	3.5	4.5	3.5
3	capybara	FALSE	4.0	5.0	4.0
	feb excite	ement			

```
1 7.0
2 3.5
3 4.0
```

pivot_longer() can be used to convert this data frame to a long format where there is one column for each of the measures, playfulness and excitement:

```
> animals_long_1 <- pivot_longer(animals_wide,</pre>
                                cols = -c('animal', 'lives_in_water'),
                                names_to = c('month', '.value'), # ".value" is placeholder for sta
                                names_pattern = '(.+)_(.+)') # specify structure of wide column no
> animals_long_1
# A tibble: 6 x 5
  animal
            lives_in_water month playfulness excitement
  <chr>
                                        <dbl>
                                                    <dbl>
            <lgl>
                            <chr>>
1 dolphin
            TRUE
                            jan
                                          6
                                                      7
2 dolphin
                                          5.5
                                                      7
            TRUE
                            feb
3 porcupine FALSE
                                          3.5
                                                      3.5
                            jan
4 porcupine FALSE
                            feb
                                          4.5
                                                      3.5
5 capybara FALSE
                                          4
                                                      4
                            jan
6 capybara FALSE
                                                      4
                            feb
```

Alternatively, pivot_longer() can be used to convert this data frame to a long format where there is one column containing all the playfulness and excitement values:

```
> animals_long_2 <- pivot_longer(animals_wide,</pre>
                                 cols = -c('animal', 'lives_in_water'),
                                names_to = c('month', 'measure'),
                                names_pattern = '(.+)_{(.+)}',
                                 values_to = 'val')
> animals_long_2
# A tibble: 12 x 5
   animal
            lives_in_water month measure
                                               val
   <chr>
                           <chr> <chr>
                                             <dbl>
            <lgl>
 1 dolphin TRUE
                           jan playfulness
                                               6
 2 dolphin
            TRUE
                           feb playfulness
                                               5.5
                           jan excitement
                                               7
 3 dolphin
            TRUE
 4 dolphin
            TRUE
                           feb excitement
                                               7
 5 porcupine FALSE
                           jan playfulness
                                               3.5
 6 porcupine FALSE
                           feb playfulness
                                               4.5
 7 porcupine FALSE
                                               3.5
                           jan excitement
 8 porcupine FALSE
                                               3.5
                           feb excitement
 9 capybara FALSE
                           jan playfulness
                                               4
10 capybara FALSE
                           feb playfulness
                                               5
```

```
11 capybara FALSE jan excitement 4
12 capybara FALSE feb excitement 4
```

5.3.2 Long to wide

Python

 \mathbf{R}

Say we begin with a long data frame, df_long, that looks like this:

```
> df_long
     id sex week val
1.m.1 1
          m
               1
                  16
2.m.1 2
                  12
               1
3.f.1
      3
          f
               1
                  8
                  7
1.m.2
               2
      1
          m
               2
2.m.2 2
                  19
          m
3.f.2 3 f
               2 15
1.m.3 1
               3 15
          m
               3
2.m.3
      2
                  10
          m
               3
3.f.3 3
```

To take data from long to wide with base R's reshape(), a user would write:

```
> df_wide <- reshape(df_long,</pre>
                      direction = 'wide',
                      idvar = c('id', 'sex'), # column(s) that determine which rows sho
                      v.names = 'val', # column containing values to widen
                      timevar = 'week', # column from which resulting wide column names
                      sep = '_') # the `sep` argument allows a user to specify how the
> df_wide
      id sex val_1 val_2 val_3
                        7
                 16
                             15
2.m.1
       2
                 12
                       19
                             10
           \mathbf{m}
3.f.1 3
                 8
                       15
```

The tidyverse function for taking data from long to wide is pivot_wider(). To widen df_long using pivot_longer(), a user would write:

```
values_from = 'val',
                             names_from = 'week',
                             names_prefix = 'week_') # `names_prefix` specifies a string to paste
> df_wide_PW
# A tibble: 3 x 5
     id sex week_1 week_2 week_3
  <int> <chr> <int> <int> <int> <int>
                  16
                          7
                                 15
1
      1 m
      2 m
                  12
                         19
                                 10
      3 f
                   8
                         15
```

pivot_wider() offers a lot of usability when widening relatively complicated long data structures. For example, say we want to widen both of the long versions of the animals data frame created above.

To widen the version of the long data that has a column for each of the measures (playfulness and excitement):

```
> animals_long_1
# A tibble: 6 x 5
            lives_in_water month playfulness excitement
  animal
                                    <dbl>
                                                  <dbl>
  <chr>
            <lgl>
                           <chr>
1 dolphin
            TRUE
                                        6
                                                    7
                           jan
2 dolphin
            TRUE
                           feb
                                         5.5
                                                    7
3 porcupine FALSE
                           jan
                                         3.5
                                                    3.5
4 porcupine FALSE
                                         4.5
                                                    3.5
                           feb
5 capybara FALSE
                                         4
                                                    4
                           jan
6 capybara FALSE
                           feb
                                                    4
> animals_wide <- pivot_wider(animals_long_1,
                              id_cols = c('animal',
                                          'lives_in_water'),
                              values_from = c('playfulness',
                                               'excitement'),
                              names_from = 'month',
                              names_glue = '{month}_{.value}')
                              \# `names_glue` allows for customization
                              # of column names using "glue";
                              # see https://glue.tidyverse.org/
> animals_wide
# A tibble: 3 x 6
  animal
            lives_in_water jan_playfulness feb_playfulness jan_excitement
                                                                     <dbl>
  <chr>
            <1g1>
                                     <dbl>
                                                     <dbl>
1 dolphin
           TRUE
                                       6
                                                       5.5
                                                                      7
2 porcupine FALSE
                                       3.5
                                                       4.5
                                                                      3.5
3 capybara FALSE
                                                       5
                                                                       4
# ... with 1 more variable: feb_excitement <dbl>
```

To widen the version of the long data that has one column containing all the values of playfulness and excitement together:

```
> animals_long_2
# A tibble: 12 x 5
   animal
             lives_in_water month measure
                                                 val
   <chr>
             <lgl>
                            <chr> <chr>
                                               <dbl>
 1 dolphin
             TRUE
                            jan
                                  playfulness
                                                 6
 2 dolphin
             TRUE
                            feb
                                  playfulness
                                                 5.5
 3 dolphin
                                  excitement
                                                 7
             TRUE
                            jan
 4 dolphin
             TRUE
                                  excitement
                                                 7
                            feb
 5 porcupine FALSE
                            jan
                                  playfulness
                                                 3.5
 6 porcupine FALSE
                                                 4.5
                            feb playfulness
 7 porcupine FALSE
                            jan
                                  excitement
                                                 3.5
 8 porcupine FALSE
                            feb
                                  excitement
                                                 3.5
 9 capybara FALSE
                            jan
                                  playfulness
                                                 4
10 capybara FALSE
                                  playfulness
                                                 5
                            feb
                            jan
11 capybara FALSE
                                  excitement
                                                 4
12 capybara FALSE
                            feb
                                  excitement
                                                 4
> animals_wide <- pivot_wider(animals_long_2,
                              id_cols = c('animal', 'lives_in_water'),
                              values_from = 'val',
                              names_from = c('month', 'measure'),
                              names_sep = '_')
> animals_wide
# A tibble: 3 x 6
  animal
            lives_in_water jan_playfulness feb_playfulness jan_excitement
  <chr>
            <1g1>
                                      <dbl>
                                                      <dbl>
1 dolphin
                                                        5.5
            TRUE
                                       6
                                                                       7
2 porcupine FALSE
                                       3.5
                                                        4.5
                                                                       3.5
3 capybara FALSE
                                                        5
                                                                       4
# ... with 1 more variable: feb_excitement <dbl>
```

5.4 Merge/Join

The merge/join examples below all make use of the following sample data frames:

```
> x
  merge_var val_x
1     a    12
2     b    94
3     c    92
> y
```

```
5.4. MERGE/JOIN
```

97

5.4.1 Left Join

A left join of x and y keeps all rows of x and merges rows of y into x where possible based on the merge criterion:

merge_var	val_x
а	12
b	94
С	92

+ (left join on merge_var)

merge_var	val_y
С	78
d	32
е	30

merge_var val_
a 12
b 94
c 92

X

у

Python

```
> import pandas as pd
+ pd.merge(x, y, how = 'left')
  merge_var val_x val_y
0         a 12.0      NaN
1         b 94.0      NaN
2         c 92.0      78.0
```

 \mathbf{R}

```
> # all.x = T results in a left join
> merge(x, y, by = 'merge_var', all.x = T)
  merge_var val_x val_y
1          a     12     NA
2          b     94     NA
3          c     92     78
```

merge

d e

5.4.2 Right Join

A right join of x and y keeps all rows of y and merges rows of x into y wherever possible based on the merge criterion:

merge_var	val_x 12	+ (right join on merge_var)	merge_var	78	=
b	94		d	32	
С	92	merge_var)	е	30	

Python

 \mathbf{R}

5.4.3 Inner Join

An inner join of x and y returns merged rows for which a match can be found on the merge criterion $in\ both\ tables$:

val y

78

32 30

merge_var	val_x		merge_var
а	12	+ (inner join on merge_var)	С
b	94		d
С	92	merge_var	е

merge_var val

x y

Python

 \mathbf{R}

5.4.4 Outer Join

An outer join of x and y keeps all rows from both tables, merging rows wherever possible based on the merge criterion:

merge

a b c d

е

merge_var	val_x		merge_var	val_y	
а	12	+	С	78	=
b	94	(outer join on merge_var)	d	32	
С	92	merge_var)	е	30	
	x	-		у	

Python

```
> import pandas as pd
+ pd.merge(x, y, how = 'outer')
  merge_var val_x val_y
0          a 12.0     NaN
1          b 94.0     NaN
2          c 92.0     78.0
3          d     NaN      32.0
4          e     NaN      30.0
```

\mathbf{R}

```
> # all = T (or all.x = T AND all.y = T) results in an outer join
> merge(x, y, by = 'merge_var', all = T)
 merge_var val_x val_y
       a 12
1
                  NA
2
        b 94
                  NA
        c 92
3
                  78
4
        d
           NA
                  32
5
        e NA
                  30
```

Chapter 6

Aggregation and Group Operations

This chapter looks at manipulating and summarizing data by groups.

6.1 Cross tabulation

Cross tabulation is the process of determining frequencies per group (or determining values based on frequencies, like proportions), with groups delineated by one or more variables (e.g., nationality and sex).

The Python and R examples of cross tabulation below both make use of the following dataset, dat:

```
> dat
  nationality sex
1   Canadian  m
2   French  f
3   French  f
4   Egyptian  m
5   Canadian  f
```

Python

The **pandas** package contains a **crosstab()** function for cross tabulation with two or more variables. Alternatively, the **groupby()** function, also in **pandas**, facilitates cross tabulation by one or more variables when used in combination with **count()**.

```
> import pandas as pd
+ pd.crosstab(dat.nationality, dat.sex)
nationality
Canadian
             1 1
Egyptian
             0 1
French
             2 0
> dat.groupby(by = 'nationality').nationality.count()
nationality
Canadian
Egyptian
            1
French
            2
Name: nationality, dtype: int64
> dat.groupby(by = ['nationality', 'sex']).nationality.count()
+ # Or: dat.groupby(by = ['nationality', 'sex']).sex.count()
nationality sex
Canadian
             f
                    1
                    1
                    1
Egyptian
             m
                    2
French
             f
Name: nationality, dtype: int64
```

\mathbf{R}

The table() function performs cross tabulation in R. A user can enter a single grouping variable or enter multiple grouping variables separated by a comma(s). The xtabs() function also computes cross-tabs; a user enters the variables to be used for grouping in formula notation.

```
> table(dat$nationality)
Canadian Egyptian
                   French
       2
               1
                         2
> table(dat$nationality, dat$sex)
           f m
  Canadian 1 1
  Egyptian 0 1
  French
          2 0
> xtabs(formula = ~nationality + sex, data = dat)
nationality f m
   Canadian 1 1
   Egyptian 0 1
```

```
French 2 0
```

6.2 Group summaries

Computing statistical summaries per group.

Python

The groupby() function from Pandas splits up a data set based on one or more grouping variables. Summarizing functions—like mean(), sum(), and so on—can then be applied to those groups. In the first example below, we use groupby() to group rows of the mtcars dataset by the number of cylinders each car has; from there, we select just the mpg column and call mean(), thus producing the average miles per gallon within each cylinder group. In the second example, we again group observations by cyl, but instead of then selecting just the mpg column, we directly call mean(); this gives the mean for each variable in the data set within each cylinder group. Finally, in the third example, we group by two variables—cyl and vs—and then use the describe() function to generate a set of descriptive statistics for mpg within each cylinder*vs group (e.g., mean, SD, minimum, etc.).

```
> import pandas as pd
+ mean_mpg_by_cyl = mtcars.groupby(by = 'cyl')['mpg'].mean()
+ print(mean_mpg_by_cyl)
cyl
4.0
       26.663636
6.0
       19.742857
8.0
       15.100000
Name: mpg, dtype: float64
> means_all_vars = mtcars.groupby(by = 'cyl').mean()
+ print(means_all_vars)
                       disp
                                                              gear
                                                                        carb
           {\tt mpg}
                                     hp
                                                     am
cyl
4.0
     26.663636
               105.136364
                              82.636364
                                                          4.090909
                                                                    1.545455
                                               0.727273
                             122.285714
                                                          3.857143
6.0
     19.742857
                183.314286
                                               0.428571
     15.100000
                353.100000
                             209.214286
                                               0.142857
                                                          3.285714
                                                                    3.500000
[3 rows x 10 columns]
> mpg_by_cyl_vs = mtcars.groupby(by = ['cyl', 'vs'])['mpg'].describe()
+ print(mpg_by_cyl_vs)
         count
                                 std
                                        min
                                                25%
                                                       50%
                                                               75%
                                                                     max
                      mean
cyl vs
```

```
4.0 0.0
       1.0 26.000000
                             NaN 26.0 26.000 26.00 26.00
                                                            26.0
   1.0
         10.0 26.730000 4.748111 21.4 22.800
                                               25.85
                                                     30.40
                                                            33.9
6.0 0.0
          3.0 20.566667 0.750555
                                 19.7
                                       20.350
                                               21.00
                                                     21.00
                                                            21.0
   1.0
         4.0 19.125000 1.631717 17.8 18.025
                                               18.65 19.75
                                                            21.4
8.0\ 0.0\ 14.0\ 15.100000\ 2.560048\ 10.4\ 14.400\ 15.20\ 16.25\ 19.2
```

\mathbf{R}

The aggregate() function can be used to generate by-group statistical summaries based on one or more grouping variables. Grouping variables can be declared as a list in the function's by argument. Alternatively, the grouping variable(s) and the variable to be summarized can be passed to aggregate() in formula notation: var_to_be_aggregated ~ grouping_var_1 + ... + grouping_var_N. The summarizing function (e.g., mean(); median(); etc.) is declared in the FUN argument.

Adding drop=FALSE ensures all combinations of levels are returned even if no data exist at that combination. Below the final row is NA since there are no 8-cylinder cars with a "straight" engine (vs = 1).

```
> # Two or more grouping variables
> # Calculating max of `mpg` in each `cyl`*`vs` group
> aggregate(x = mtcars$mpg,
           by = list(cyl = mtcars$cyl, vs = mtcars$vs),
           FUN = "max", drop = FALSE)
 cyl vs
           х
1
   4 0 26.0
2
   6 0 21.0
3
   8 0 19.2
   4 1 33.9
5
   6 1 21.4
 8 1 NA
```

```
> # Or, specify the variable to summarize and the grouping variables in formula notation > aggregate(mpg ~ cyl + vs, data = mtcars, FUN = max)
```

The **tidyverse** also offers a summarizing function, summarize() (or summarise(), for the Britons), which is in the **dplyr** package. After grouping a data frame/tibble (with, e.g., **dplyr**'s group_by() function), a user passes it to summarize(), specifying in the function call how the summary statistic should be calculated.

```
> library(dplyr)
> mtcars %>%
   group_by(cyl, vs) %>%
   summarize(avg mpg = mean(mpg))
`summarise()` has grouped output by 'cyl'. You can override using the `.groups` argument.
# A tibble: 5 x 3
# Groups: cyl [3]
          vs avg_mpg
   cyl
  <dbl> <dbl>
               <dbl>
1
     4
                26
     4
          1
                26.7
2
3
     6
         0
                20.6
     6
                19.1
4
           1
5
     8 0
                15.1
```

summarize() makes it easy to specify relatively complicated summary calculations without needing to write an external function.

```
> mtcars %>%
    group_by(cyl, vs) %>%
    summarize(avg_mpg = mean(mpg),
              complicated_summary_calculation =
                min(mpg)^0.5 *
                mean(wt)^0.5 +
                mean(disp)^(1/mean(hp)))
`summarise()` has grouped output by 'cyl'. You can override using the `.groups` argument.
# A tibble: 5 x 4
# Groups: cyl [3]
          vs avg_mpg complicated_summary_calculation
    cyl
  <dbl> <dbl>
                <dbl>
                                                <dbl>
1
      4
           0
                 26
                                                 8.51
      4
                26.7
2
           1
                                                 8.07
3
      6
            0
                20.6
                                                 8.41
4
      6
                 19.1
                                                 8.81
           1
5
     8
                15.1
                                                 7.48
           0
```

6.3 Centering and Scaling

Centering refers to subtracting a constant, such as the mean, from every one of set of values. This is sometimes performed to aid interpretation of linear model coefficients.

Scaling refers to rescaling a column or vector of values such that their mean is zero and their standard deviation is one. This is sometimes performed to put multiple variables on the same scale and is often recommended for procedures such as principal components analysis (PCA).

Python

The scale() function from the **preprocessing** module of the **scikit-learn** package provides one-step centering and scaling. To center a variable at zero without scaling it, use scale() with with_mean = True and with_std = False (both are True by default).

```
> from sklearn import preprocessing
+
+ centered_mpg = preprocessing.scale(mtcars.mpg, with_mean = True, with_std = False)
+ centered_mpg.mean()
-3.1086244689504383e-15
```

To scale a variable after centering it (so that its mean is zero and its standard deviation is one), use scale() with with mean = True and with std = True.

```
> from sklearn import preprocessing
+
+ scaled_mpg = preprocessing.scale(mtcars.mpg, with_mean = True, with_std = True)
+ scaled_mpg.mean()
-4.996003610813204e-16
> scaled_mpg.std()
1.0
```

\mathbf{R}

The scale() function can both center and scale variables.

To center a variable without scaling it, call <code>scale()</code> with the <code>center</code> argument set to <code>TRUE</code> and the <code>scale</code> argument set to <code>FALSE</code>. The variable's mean will be subtracted off of each of the variable values. (Note: If desired, the <code>center</code> argument can be set to a numeric value instead of <code>TRUE/FALSE</code>; in that case, each variable value will have the argument value subtracted off of it.)

```
> centered_mpg <- scale(mtcars$mpg, center = T, scale = F)
> mean(centered_mpg)
[1] 4.440892e-16
```

To scale a variable (while also centering it), call <code>scale()</code> with the <code>center</code> and <code>scale</code> arguments set to TRUE (these are the default argument values). The variable's mean will be subtracted off of each of the variable values, and each value will then be divided by the variable's standard deviation. (Note: As with the <code>center</code> argument, the <code>scale</code> argument can also be set to a numeric value instead of <code>TRUE/FALSE</code>; in that case, the divisor will be the argument value instead of the standard deviation.)

```
> scaled_mpg <- scale(mtcars$mpg, center = T, scale = T)
> mean(scaled_mpg)
[1] 7.112366e-17
> sd(scaled_mpg)
[1] 1
```

Chapter 7

Basic Plotting and Visualization

This chapter looks at creating basic plots to explore and understand data. Visualization in Python and R is a gigantic and evolving topic. We don't pretend to present a comprehensive comparison.

The plots below make use of the **palmerpenguins** data set, which contains various measurements for 344 penguins across three islands in the Antarctic Palmer Archipelago. The data were collected by Kristen Gorman and colleagues, and they were made available under a CC0 public domain license by Allison Horst, Alison Hill, and Kristen Gorman.

For the R sections below, we discuss how to generate plots using base R and using **ggplot2**.

Here's a glimpse at the data set:

```
> head(penguins)
# A tibble: 6 x 8
  species island bill_length_mm bill_depth_mm flipper_length_~ body_mass_g sex
                                         <dbl>
  <fct>
          <fct>
                          <dbl>
                                                          <int>
                                                                       <int> <fct>
1 Adelie Torge~
                           39.1
                                          18.7
                                                             181
                                                                        3750 male
2 Adelie Torge~
                           39.5
                                          17.4
                                                             186
                                                                        3800 fema~
3 Adelie Torge~
                           40.3
                                          18
                                                             195
                                                                        3250 fema~
          Torge~
4 Adelie
                           NA
                                          NA
                                                             NA
                                                                          NA <NA>
5 Adelie Torge~
                           36.7
                                          19.3
                                                             193
                                                                        3450 fema~
6 Adelie Torge~
                           39.3
                                          20.6
                                                             190
                                                                        3650 male
# ... with 1 more variable: year <int>
```

7.1 Histograms

Visualizing the distribution of numeric data.

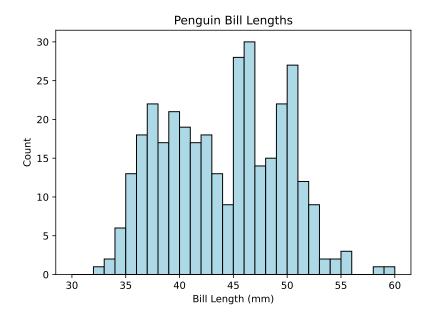
Python

The Python plotting library Matplotlib's hist() function computes and plots a histogram. There are many parameters that can be specified within the hist() function so that you can customize the output histogram plot to best fit your needs. Some parameters include the number of bins, the upper and lower bounds on each bin, weights, colors, and more.

Below we show a histogram of the bill length from the dataset. We specified 30 bins each of which is light blue with a black outline of linewidth 1. The hist() defaults to no outline which can make it difficult to distinguish bins clearly, so we add in the bin outlines here.

One thing to note is that the bins are left inclusive and right exclusive. For example, if a particular bin spans the range of 1 to 3, the bin will include the value 1 but will exclude the value 2 (and will include all values between 1 and 3). In short, bin ranges are as follows [x1,x2) where x1 is the starting point of the bin and x2 is the ending point of the bin.

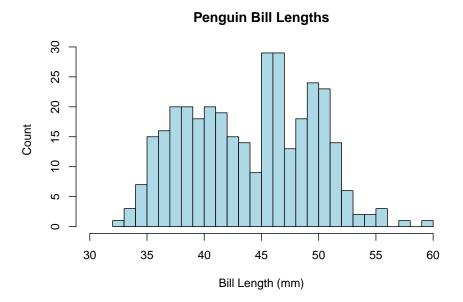
Notice the semicolon at the end of the plt.hist() function. This suppresses the printing of the array generated to create the histogram.



 \mathbf{R}

Base R's hist() function generates histograms, and features of the histogram—like the bar color, number of bins/breaks, and so on—can be easily customized as below.

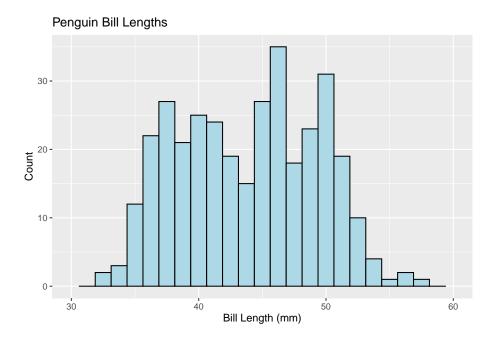
```
> hist(penguins$bill_length_mm, breaks = 25, col = 'lightblue', xlim = c(30, 60),
+ main = 'Penguin Bill Lengths', xlab = 'Bill Length (mm)', ylab = 'Count')
```



The **ggplot2** method for generating histograms follows the standard **ggplot2** syntax: Initialize a plot with **ggplot()**, and then add layers thereto, specifying aesthetic properties along the way. Here, the layer to add is **geom_histogram()**.

```
> ggplot(penguins, aes(x = bill_length_mm)) +
+ geom_histogram(fill = 'lightblue', color = 'black', bins = 25) +
+ xlim(30, 60) + labs(title = 'Penguin Bill Lengths', x = 'Bill Length (mm)', y = 'C
```

7.2. BARPLOTS 113



7.2 Barplots

Visualizing the distribution of categorical data.

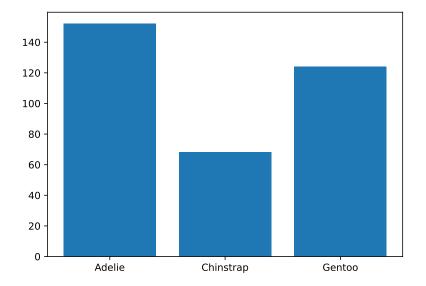
Python

For this example, we will generate a bar plot showing how many of each species—Adelie, Chinstrap, Gentoo—we have in our dataset. We go through two ways of doing this here.

First, we use the Matplotlib plotting library to create the bar plot using the function bar(). To start, we determine the number of each species, then use that data to create the bar plot.

```
> import matplotlib.pyplot as plt
+
# Determine the number of each species
+ adelie_counts = len(penguins.loc[penguins["species"]=="Adelie"])
+ chinstrap_counts = len(penguins.loc[penguins["species"]=="Chinstrap"])
+ gentoo_counts = len(penguins.loc[penguins["species"]=="Gentoo"])
+
# Save the counts information into arrays to be inputted into the bar() function
+ spec = ["Adelie", "Chinstrap", "Gentoo"]
```

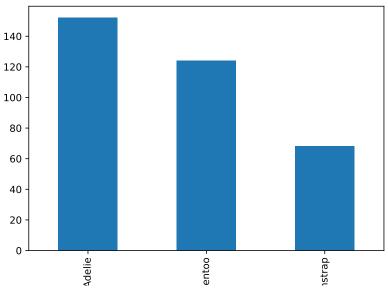
```
+ counts = [adelie_counts,chinstrap_counts,gentoo_counts]
+
+ plt.clf() # clears the figure to ensure that multiple plots are not overlaid
+ plt.bar(spec,counts)
<BarContainer object of 3 artists>
> plt.show()
```



Our data is stored in a pandas Dataframe, which has it's own built-in plotting module, plot. Here we create the same bar plot by using the pandas bar() function.

```
> plt.clf()
+ penguins["species"].value_counts().plot.bar()
+ plt.show()
```

7.2. BARPLOTS 115



One thing to note here is that we generated the same bar plot containing the same information with way less effort. Using the built-in pandas plotting routine proved to be the more efficient method here.

\mathbf{R}

To form barplots, we'll first take the **penguins** data set and create a summary data frame containing the statistics we're looking to plot. Here, that's simply the sample size of each species in the data set.

```
> species_counts <- as.data.frame(xtabs(~ species, data = penguins))
> species_counts
          species Freq
1     Adelie     152
2 Chinstrap     68
3     Gentoo     124
```

We can plot those values using the barplot() function in base R, specifying arguments along the way to customize the title/axis labeling, bar colors, and range of the y axis. To add values above the bars, we can follow barplot() with a text() call as below.

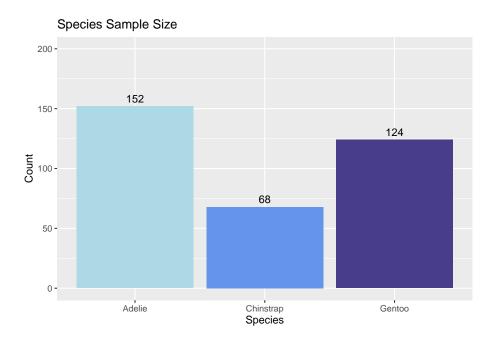
```
> penguin_plot <- barplot(Freq ~ species, data = species_counts, col = c('lightblue', 'cornflower
+ main = 'Species Sample Size',</pre>
```

```
+ xlab = 'Species', ylab = 'Count', ylim = c(0, 200))
> text(x = penguin_plot, y = species_counts$Freq + 10,
+ labels = species_counts$Freq)
```

Species Sample Size 152 124 168 Adelie Chinstrap Gentoo

To recreate the barplot above with **ggplot2**, one can add a **geom_bar()** layer to a plot initialized with **ggplot()**.

Species



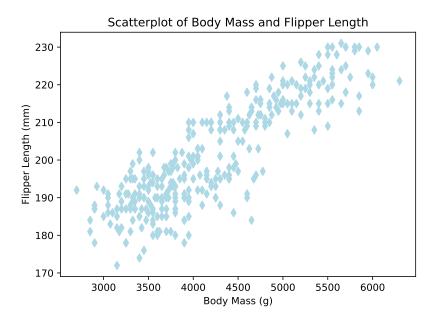
7.3 Scatterplot

Visualizing the relationship between two numeric variables.

Python

The scatter() function, part of matplotlib, can produce scatterplots in Python. The x and y arguments specify the points to plot. In the example below, we also use the c and marker arguments to customize the point color and point shape, respectively. The xlabel(), ylabel(), and title() functions customize plot labels.

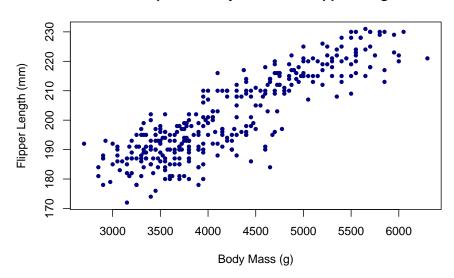
```
> import matplotlib.pyplot as plt
+
+ penguins_no_na = penguins.dropna() # remove NA rows, as we only want to plot present data
+ plt.clf() # clear the plotting space to prevent plot overlap
+ plt.scatter(x = penguins_no_na['body_mass_g'], y = penguins_no_na['flipper_length_mm'], c = 'le'
+ plt.xlabel('Body Mass (g)')
+ plt.ylabel('Flipper Length (mm)')
+ plt.title('Scatterplot of Body Mass and Flipper Length')
+
+ plt.show()
```



 \mathbf{R}

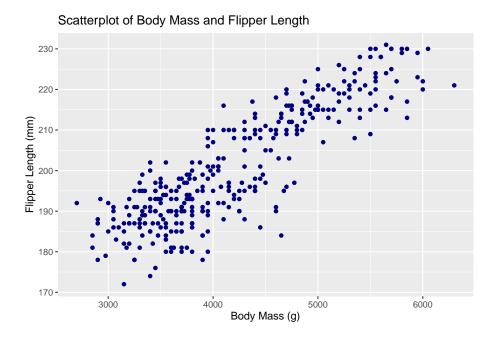
Scatterplots can be generated in base R with the plot() function. The pch argument below modifies the point shape (e.g., 20 = solid circle; 24 = unfilled triangle; etc.)

Scatterplot of Body Mass and Flipper Length



To generate a scatterplot with **ggplot2**, initialize a plot with **ggplot()**, then add a layer of points with **geom_point()**.

```
> ggplot(penguins, aes(x = body_mass_g, y = flipper_length_mm)) +
+ geom_point(color = 'navy') +
+ labs(title = 'Scatterplot of Body Mass and Flipper Length',
+ x = 'Body Mass (g)', y = 'Flipper Length (mm)')
```



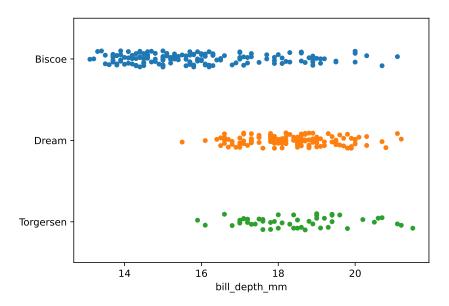
7.4 Stripcharts

Stripcharts, or strip plots, are one-dimensional scatterplots. Like boxplots, they reveal the distribution of a numeric variable within levels of a categorical variable.

Python

The **seaborn** package provides the **stripplot()** function. Specify which variables you want on the x and y axes. Below we specify island on the y axis to see the distribution of bill_depth_mm horizontally. Specify your Pandas data frame using the data argument. Finally create the plot using plt.show() from **matplotlib**.

```
> import seaborn as sns
+ import matplotlib.pyplot as plt
+ sns.stripplot(x="bill_depth_mm", y="island", data=penguins)
+ plt.show()
```

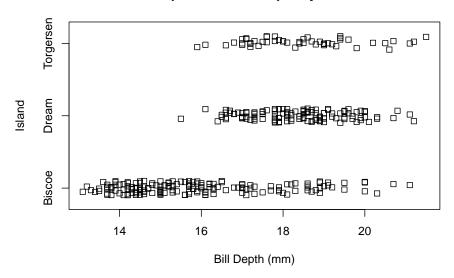


The stripplot help page provides more examples.

\mathbf{R}

Base R offers the stripchart() function. To indicate the numeric variable and the grouping variable, you can use formula notation: numeric_var ~ grouping_var. Adding methhod = 'jitter' to the set of arguments spreads the points out slightly within each level of the grouping variable, making it easier to see points that might otherwise be obscured by overlap.

Stripchart of Bill Depth by Island

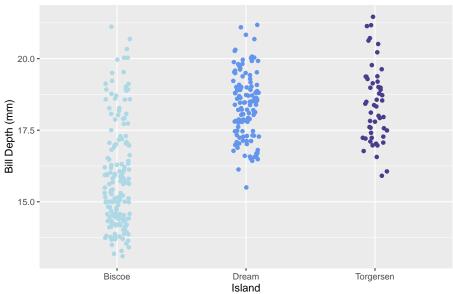


Stripcharts can also be made with **ggplots2**'s geom_jitter() function, as shown below. You can control the amount of jitter with a position argument in geom_jitter().

```
> ggplot(penguins, aes(x = island, y = bill_depth_mm)) +
+ geom_jitter(aes(color = island), position = position_jitter(0.1)) +
+ scale_color_manual(values = c('lightblue', 'cornflowerblue', 'darkslateblue')) + #
+ labs(title = 'Stripchart of Bill Depth by Island',
+ x = 'Island', y = 'Bill Depth (mm)') +
+ theme(legend.position = 'none')
Warning: Removed 2 rows containing missing values (geom_point).
```

7.5. BOXPLOTS 123





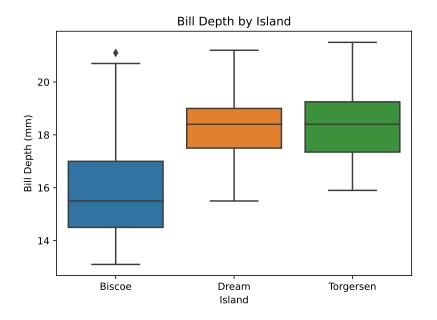
7.5 Boxplots

Visualizing the relationship between a numeric variable and a categorical variable via five-number summaries.

Python

The boxplot() function in seaborn generates boxplots, and matplotlib can be used for the aesthetics of the plot.

```
> import seaborn as sns
+ import matplotlib.pyplot as plt
+ plt.figure()
+ sns.boxplot(x="island", y="bill_depth_mm", data=penguins)
+ plt.xlabel("Island")
+ plt.ylabel("Bill Depth (mm)")
+ plt.title("Bill Depth by Island")
+ plt.show()
```



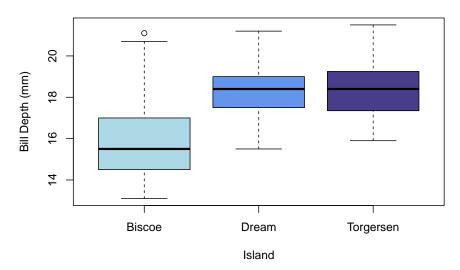
 ${\bf R}$

The boxplot() function in base R generates boxplots, and a user specifies the grouping variable and the numeric variable to be plotted in formula notation: y ~ grouping_var.

```
> boxplot(bill_depth_mm ~ island, data = penguins, col = c('lightblue', 'cornflowerblue')
+ main = 'Boxplot of Bill Depth by Island', xlab = 'Island', ylab = 'Bill Depth
```

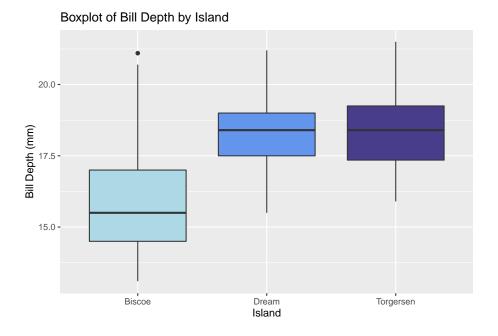
7.5. BOXPLOTS 125





To generate a boxplot with **ggplot2**, add a **geom_boxplot()** layer to a plot initialized with **ggplot()**.

```
> ggplot(penguins, aes(x = island, y = bill_depth_mm)) +
+ geom_boxplot(aes(fill = island)) +
+ scale_fill_manual(values = c('lightblue', 'cornflowerblue', 'darkslateblue')) +
+ labs(title = 'Boxplot of Bill Depth by Island',
+ x = 'Island', y = 'Bill Depth (mm)') +
+ theme(legend.position = 'none')
```



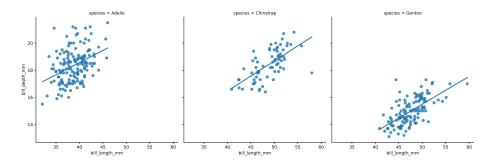
7.6 Facet plots

Facet plots (also called trellis plots, lattice plots, and conditional plots) are comprised of multiple smaller plots, where each subplot contains a subset of the overall data, with subsets defined by one or more faceting variables.

Python

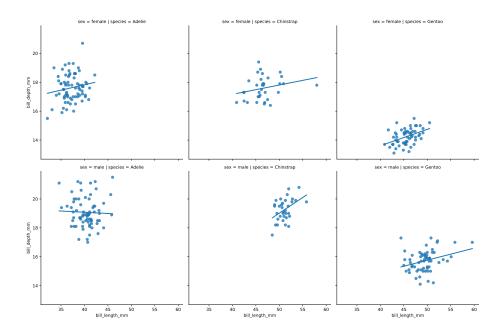
The seaborn package provides several functions for creating facet plots, including relplot(), displot(), catplot(), and lmplot(). Below we demonstrate the lmplot() function which allows you to create scatter plots at certain levels of categorical variable. Specify your x and y variables as character strings using the x and y arguments, respectively. Specing the grouping variable using either the col or row arguments. By default a linear-squares lines is added to the plot. Setting ci = None suppresses the confidence interval ribbon.

```
> plt.clf()
+ sns.lmplot(x = "bill_length_mm", y = "bill_depth_mm", col = "species",
+ data = penguins, ci = None)
```



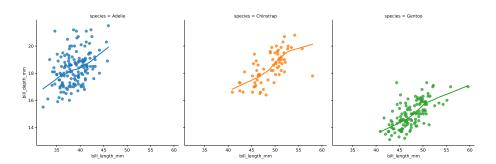
To facet by two variables, provide variables to both the col and row arguments.

```
> plt.clf()
+ sns.lmplot(x = "bill_length_mm", y = "bill_depth_mm",
+ col = "species", row = "sex",
+ data = penguins, ci = None)
```



Set lowess = True for smooth trend lines. Notice also that color can be mapped to the same variable used for faceting.

```
> plt.clf()
+ sns.lmplot(x = "bill_length_mm", y = "bill_depth_mm",
+ col = "species", hue = "species",
+ data = penguins, ci = None, lowess = True)
```

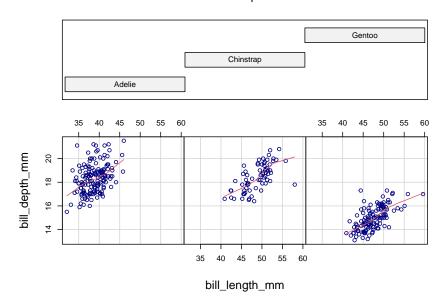


The lmplot help page showcases other examples.

 \mathbf{R}

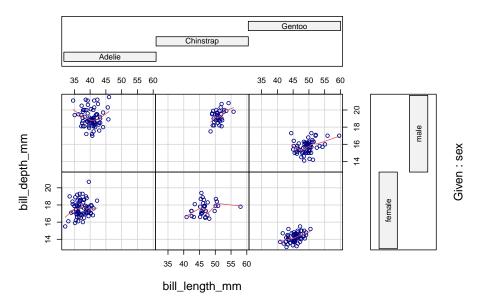
The <code>coplot()</code> function in base R produces conditioning plots using formula notation: y ~ x | <code>grouping_var</code>. The <code>rows</code> and <code>columns</code> arguments control layout. Below we specify one row of plots. The <code>panel</code> argument controls what action is carried out in each plot. The default is a scatterplot. Below we use the base R <code>panel.smooth</code> function to create scatter plots with a smooth trend line.

Given: species

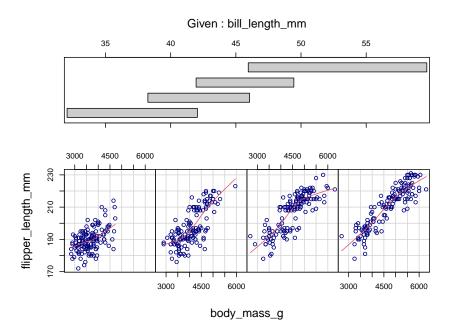


To condition on two variables, use formula notation with syntax: $y \sim x \mid grp_var1 * grp_var2$.

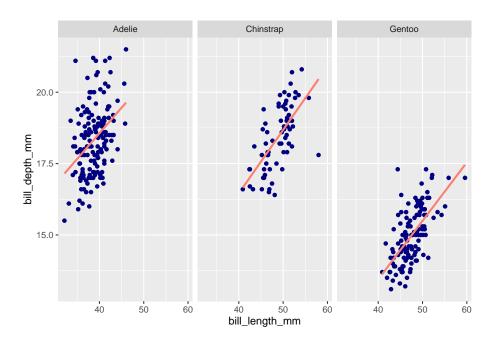




The labels of the conditioning variables unfortunately use a lot of real estate in the margins, and there is no easy way to modify that. However, this design works quite well when we condition on a *numeric variable*. The coplot() function automatically creates overlapping group intervals to condition on, and the stacked layout of the labels helps us visualize how the relationship between y and x changes between the groups. To manually set the number of groups, use the number argument. Below we specify 4 groups to be generated for "bill_length_mm".



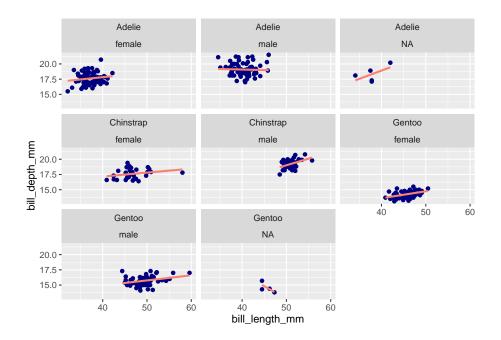
Alternatively, **ggplot2** provides a intuitive and easy-to-use method for generating facet plots: A user specifies the aesthetics of the plot using standard **ggplot2** syntax (i.e., as a series of added layers) and then adds an additional call, facet_wrap() (or facet_grid(); differences are discussed below), specifying the faceting variable(s) to split up the plots by.

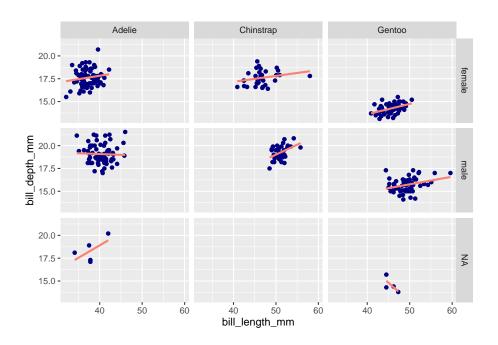


```
> # Use formula notation, a character vector, or vars() to specify faceting
> # variables; e.g., ~species, c('species'), or vars(species)
```

The number of rows and columns can be manually specified with nrow and ncol arguments in facet_wrap(). By default, the x and y axes of all facet plots will be on the same scale. The axis ranges can be set to vary freely by adding scales = 'free' as an argument (or, alternatively, scales = 'free_x' or scales = 'free_y' to free just the x or y axis).

Both facet_wrap() and facet_grid() can be used to make facet plots. When faceting based on multiple variables (e.g., species and sex), facet_wrap() will drop group combinations for which there are no data points, whereas facet_grid() will generate a plot for all possible group combinations:





> # Note that facet_grid() has separate `rows` and `cols` arguments for specifying
> # faceting variables

Chapter 8

Selected Topics in Statistical Inference

This chapter looks at performing selected statistical analyses. It is not comprehensive. The focus is on implementation using Python and R. Good statistical practice is more than knowing which function to use. At a minimum we recommend reading the article, Ten Simple Rules for Effective Statistical Practice [Kass et al., 2016].

8.1 Comparing group means

Many research studies compare mean values of some quantity of interest between two or more groups. A t test analyzes two group means. An Analysis of Variance, or ANOVA, analyzes three or more group means. Both the t test and ANOVA are special cases of a linear model.

To demonstrate the t test, we examine fictitious data on 15 scores between two groups of subjects. The "control" group was tested as-is while the "treated" group experienced a particular intervention. Of interest is (1) whether or not the mean scores differ meaningfully between the treated and control groups, and (2) if they do differ, how are they different?

To demonstrate the ANOVA test, we use data from *The Analysis of Biological Data (3rd ed)*[Whitlock and Schluter, 2020] on the mass of pine cones (in grams) from three different environments in North America. Of interest is (1) whether or not the mean mass of pine cones differ meaningfully between the three locations, and (2) if they do differ, how are they different?

We usually assess the first question in each scenario with a hypothesis test and p-value. The null hypothesis is no difference between the means. The p-value is

the probability of the observed differences between the groups (or more extreme differences) assuming the null hypothesis is true. A small p-value, traditionally less then 0.05, provides evidence against the null. For example, a p-value of 0.01 says there's a 1% chance of sampling data as different as this (or more different) if there really was no difference between the groups. Note that p-values don't tell you how two or more statistics differ. See the ASA Statement on p-values.

We assess the second question in each scenario by calculating confidence intervals on the difference in means. This is more informative than a p-value. A confidence interval gives us information on the uncertainty, direction and magnitude of a difference in means. For example, a 95% confidence interval of [2, 15] tells us the data is consistent with a difference anywhere between 2 and 15 and that the mean of one group appears to be at least 2 units larger than the mean of the other group. Note that a 95% confidence interval does not mean there is a 95% probability that the true value is in the interval. The confidence interval either captured the true value or it did not. We don't know. However the process of calculating the confidence interval works roughly 95% of the time.

Python

t-test

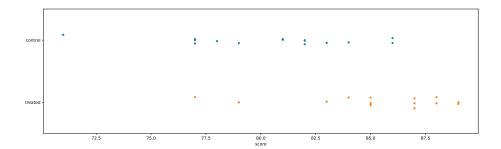
Our data is available as a **Pandas** dataframe. It's small enough to view in its entirety.

```
> ch8_d1
    score
              group
0
     77.0
            control
1
     81.0
            control
2
     77.0
            control
3
     86.0
            control
     81.0
4
            control
5
     77.0
            control
6
     82.0
            control
7
     83.0
            control
8
     82.0
            control
9
     79.0
           control
10
     86.0
            control
     82.0
11
            control
     78.0
12
            control
13
     71.0
            control
14
     84.0
            control
15
     85.0
           treated
16
     85.0
            treated
17
     89.0
            treated
18
     88.0
           treated
```

```
19
     87.0
          treated
20
     89.0
          treated
21
     88.0 treated
22
     85.0 treated
23
     77.0
          treated
24
     87.0
          treated
25
     85.0
          treated
26
     84.0 treated
27
     79.0 treated
28
     83.0 treated
     87.0 treated
```

A stripchart is one of many ways to visualize numeric data between two groups. Here we use the seaborn function stripplot(). It appears the treated group had higher scores.

```
> import seaborn as sns
+ import matplotlib.pyplot as plt
+ plt.clf()
+ sns.stripplot(x="score", y="group", data=ch8_d1)
+ plt.show()
```



One way to perform a t test in Python is via the CompareMeans() function and its associated methods available in the **statsmodels** package. Below we import **statsmodels.stats.api** as "sms".

```
> import statsmodels.stats.api as sms
```

We first extract the data we want to compare as pandas Series.

```
> d_control = ch8_d1.query('group == "control"')['score']
+ d_treated = ch8_d1.query('group == "treated"')['score']
```

Next we create Descriptive statistics objects using the DescrStatsW() function.

```
> control = sms.DescrStatsW(d_control)
+ treated = sms.DescrStatsW(d_treated)
```

Descriptive statistics objects have attributes such as mean and std (standard deviation). Below we print the mean and standard deviation of each group. We also round the standard deviation to three decimal places and place a line break before printing the standard deviation.

```
> print("control mean:", control.mean, "\ncontrol std:", round(control.std, 3))
control mean: 80.4
control std: 3.844
> print("treated mean:", treated.mean, "\ntreated std:", round(treated.std, 3))
treated mean: 85.2
treated std: 3.331
```

Next we create a CompareMeans means object using the CompareMeans() function. The required inputs are Descriptive statistics objects. We save the result as "ttest".

```
> ttest = sms.CompareMeans(control, treated)
```

Now we can use various methods with the "ttest" object. To see the result of a two sample t test assuming unequal variances, along with a confidence interval on the differences, use the summary method with usevar='unequal'.

<pre>> print(ttest.summary(usevar='unequal'))</pre>						
Test for equality of means						
	coef	std err	t	P> t	[0.025	0.975]
subset #1	-4.8000	1.359	-3.531	0.001	-7.587	-2.013

The p-value of 0.001 is small, providing good evidence that the difference in means we witnessed reflects a real difference in the population. The confidence interval on the difference in means tells us the data is consistent with a difference between -7 and -2. It appears we can expect the control group to score at least 2 points lower than the treated group.

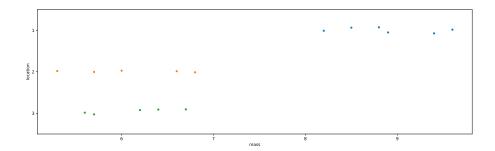
ANOVA

Our data is available as a **Pandas** dataframe. It's small enough to view in its entirety.

```
> ch8_d2
    mass location
0
     9.6
1
     9.4
                 1
2
     8.9
                  1
3
     8.8
                  1
     8.5
5
     8.2
                 1
6
     6.8
                 2
7
     6.6
                 2
                 2
8
     6.0
9
                 2
     5.7
10
     5.3
                 2
                 3
11
     6.7
12
     6.4
                 3
                 3
13
     6.2
14
     5.7
                 3
                 3
15
     5.6
```

Again we use a stripchart to visualize the three groups of data. It appears the pine cones in location 1 have a higher mass.

```
> plt.clf()
+ sns.stripplot(x="mass", y="location", data=ch8_d2)
+ plt.show()
```



We can calculate means using the groupby and mean methods.

```
> ch8_d2['mass'].groupby(ch8_d2['location']).mean()
location
1    8.90
2    6.08
3    6.12
Name: mass, dtype: float64
```

One way to perform an ANOVA test in Python is via the anova_oneway() function, also available in the **statsmodels** package.

The anova_oneway() function can perform an ANOVA on a pandas Dataframe with the first argument specifying the numeric data and the second argument the grouping variable. We also set use_var='equal' to replicate the R output below.

```
> sms.anova_oneway(ch8_d2.mass, ch8_d2.location, use_var='equal')
<class 'statsmodels.stats.base.HolderTuple'>
statistic = 50.085429769392036
pvalue = 7.786760128813737e-07
df = (2.0, 13.0)
df_num = 2.0
df_denom = 13.0
nobs_t = 16.0
n_groups = 3
means = array([8.9 , 6.08, 6.12])
nobs = array([6., 5., 5.])
vars_ = array([0.28 , 0.387, 0.217])
use_var = 'equal'
welch_correction = True
tuple = (50.085429769392036, 7.786760128813737e-07)
```

The small p-value of 0.0000007 provides strong evidence that the difference in means we witnessed reflects a real difference in the population.

A common follow-up to an ANOVA is Tukey's Honestly Significant Differences (HSD), which computes differences between all possible pairs and returns adjusted p-values and confidence intervals to account for the multiple comparisons. To carry this out in the **statsmodels** package, we need to first create a MultiComparison object using the multicomp.MultiComparison() function. Then we use the tukeyhsd() method to compare the means with corrected p-values.

The difference in means between locations 2 and 1 (2 - 1) and locations 3 and 1 (3 - 1) are about -2.8. The difference in means between locations 3 and 2 (3 - 1)

- 2) is inconclusive. It seems to be small but we're not sure if the difference is positive or negative.

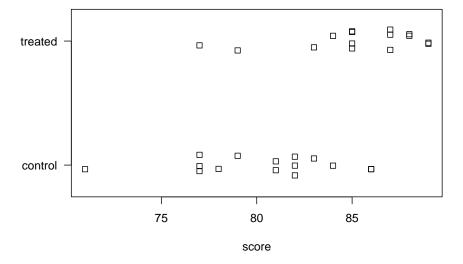
${\bf R}$

t-test

The str() function allows to take a quick look at the data frame ch8_d1. One column contains the scores, the other column indicates which group the subject was in (control vs treated).

A stripchart is one of many ways to visualize numeric data between two groups. Here we use the base R function stripchart(). The formula score ~ group says to plot score by group. The las = 1 argument says to rotate the y-axis labels. The method = "jitter" arguments says to randomly scatter the points vertically so they don't overplot. It appears the treated group had higher scores.

```
> stripchart(score ~ group, data = ch8_d1, las = 1, method = "jitter")
```



To calculate the means between the two groups we can use the aggregate() function. Again the formula score ~ group says to aggregate score by group. We specify mean so that we calculate the mean between the two groups. Some other functions we could specify include median, sd, or sum. The sample mean of the treated group is about 5 points higher than the control group.

```
> aggregate(score ~ group, data = ch8_d1, mean)
    group score
1 control 80.4
2 treated 85.2
```

Is this difference meaningful? What if we took more samples? Would each sample result in similar differences in the means? A t test attempts to answer this.

The t.test() function accommodates formula notation allowing us to specify that we want to calculate mean score by group.

The p-value of 0.0015 is small, providing good evidence that the difference in means we witnessed reflects a real difference in the population. The confidence interval on the difference in means tells us the data is consistent with a difference between -7 and -2. It appears we can expect the control group to score at least 2 points lower than the treated group.

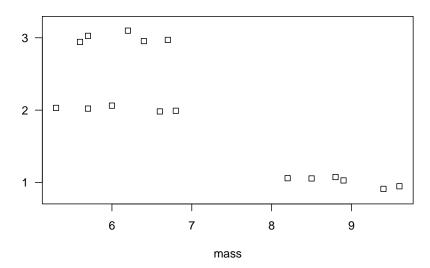
ANOVA

The str() function allows to take a quick look at the data frame ch8_d2. One column contains the mass of the pine cones, the other column indicates which location the pine cone was found.

```
> str(ch8_d2)
'data.frame': 16 obs. of 2 variables:
$ mass : num 9.6 9.4 8.9 8.8 8.5 8.2 6.8 6.6 6 5.7 ...
$ location: chr "1" "1" "1" "1" ...
```

Again we use a stripchart to visualize the three groups of data. It appears the pine cones in location 1 have a higher mass.

```
> stripchart(mass ~ location, data = ch8_d2, las = 1, method = "jitter")
```



To calculate the means between the three groups we can use the aggregate() function. Again the formula mass ~ location says to aggregate mass by location. We specify mean so that we calculate the mean between the three groups.

```
> aggregate(mass ~ location, data = ch8_d2, mean)
    location mass
1           1 8.90
2           2 6.08
3           3 6.12
```

Is this difference meaningful? ANOVA attempts to answer this.

The aov() function carries out the ANOVA test and also accommodates formula notation. It's usually preferable to save the ANOVA result into an object and call summary() on the object.

```
Residuals 13 3.816 0.294 ---
Signif. codes: 0 '***' 0.001 '**' 0.05 '.' 0.1 ' ' 1
```

The small p-value of 0.0000007 provides strong evidence that the difference in means we witnessed reflects a real difference in the population.

Unlike the t.test() output, the aov() summary does not provide confidence intervals on differences in means. That's because there are many kinds of differences we might want to assess. A common and easy procedure is Tukey's Honestly Significant Differences (HSD), which computes differences between all possible pairs and returns adjusted p-values and confidence intervals to account for the multiple comparisons. Base R provides the TukeyHSD() function for this task. Call it on the ANOVA object.

The difference in means between locations 2 and 1 (2 - 1) and locations 3 and 1 (3 - 1) are about -2.8. The difference in means between locations 3 and 2 (3 - 2) is inconclusive. It seems to be small but we're not sure if the difference is positive or negative.

8.2 Comparing group proportions

It is often of interest to compare proportions between two groups. Sometimes this is referred to as a two-sample proportion test. To demonstrate we use an exercise from the text *Introductory Statistics with R* [Dalgaard, 2008] (p.154). We are told that 210 out of 747 patients died of Rocky Mountain spotted fever in the western United States. That's a proportion of 0.281. In the eastern United States, 122 out 661 patients died. That's a proportion of 0.185. Is the difference in proportions statistically significant? In other words, assuming there is no difference in the fatality rate between the two regions, is this difference in proportions surprising?

Python

A two-sample proportion test can be carried out in Python using the test_proportions_2indep() function from the statsmodels package. The two proportions being compared must be independent.

The first argument is the number of successes or occurrences for the first proportion. The second argument is the number of total trials for the first group. The third and fourth arguments are the occurrences and total number of trials for the second group, respectively.

```
> import statsmodels.stats.api as sms
+ ptest = sms.test_proportions_2indep(210, 747, 122, 661)
```

We can extract the p-value of the test and the difference in proportions using the pvalue and diff attributes, respectively.

```
> ptest.pvalue
1.632346798072468e-05

> # rounded to 4 decimal places
+ round(ptest.diff, 4)
0.0966
```

To calculate a 95% confidence interval for the difference in proportions we need to use the confint_proportions_2indep() function.

```
> pdiff = sms.confint_proportions_2indep(210, 747, 122, 661)
+ pdiff
(0.05241555145475882, 0.13988087590630482)
```

The result is returned as a tuple with an extreme amount of precision. We recommend rounding these values to few decimal places. Here's one way using f strings. Notice we extract each element of the "pdiff" tuple and round to 5 decimal places.

```
> print(f"({round(pdiff[0],5)}, {round(pdiff[1],5)})")
(0.05242, 0.13988)
```

This results are slightly different from the R example below. That's because the test_proportions_2indep() and confint_proportions_2indep() functions use different methods. See their respective help pages to learn more about the methods available and other function arguments.

```
test_proportions_2indep help page
confint_proportions_2indep help page
```

146

 \mathbf{R}

A two-sample proportion test in R can be carried out with the prop.test() function. The first argument, x, is the number of "successes" or "occurrences" of some event for each group. The second argument, n, is the number of total trials for each group.

```
> prop.test(x = c(210, 122), n = c(747, 661))

2-sample test for equality of proportions with continuity correction

data: c(210, 122) out of c(747, 661)
X-squared = 17.612, df = 1, p-value = 2.709e-05
alternative hypothesis: two.sided
95 percent confidence interval:
    0.05138139 0.14172994
sample estimates:
    prop 1    prop 2
0.2811245 0.1845688
```

The proportion of patients who died in the western US is about 0.28. The proportion who died in the eastern US is about 0.18. The small p-value says there is a very small chance of seeing a difference as large as this (or larger) if there really was no difference in the proportions. The confidence interval on the difference of proportions ranges from 0.05 to 0.14, indicating that this fever seems to kill at least 5% more patients in the western US.

Sometimes data is presented in a 2-way table with successes and failures. We can present the preceding data in a table as follows using the matrix() function.

When the table is constructed in this fashion with "successes" in the first column and "failures" in the second column, we can feed the table directly to the prop.test() function. (Obviously "success" here means "experienced the event of interest".)

The chi-squared test statistic is reported as X-squared = 17.612. This is the same statistic reported if we ran a chi-squared test of association using the chisq.test() function.

```
> chisq.test(fever)
    Pearson's Chi-squared test with Yates' continuity correction

data: fever
X-squared = 17.612, df = 1, p-value = 2.709e-05
```

This tests the null hypothesis of no association between location in the US and fatality of the fever. The result is identical to prop.test() output, however there is no indication of the nature of association.

8.3 Linear modeling

Linear modeling attempts to assess if or how the variability a numeric variable depends on one or more predictor variables. This is often referred to as regression modeling or multiple regression. While it is relatively easy to "fit a model" and generate lots of output, the model we fit may not be very good. There are many decisions we have to make when proposing a model. Which predictors do we include? Will they interact? Do we allow for non-linear effects? Answering these kinds of questions require subject matter expertise.

We walk through a somewhat simple example using data on weekly gas consumption. The data is courtesy of the R package MASS [Venables and Ripley, 2002]. The documentation describes the data as follows:

"Mr Derek Whiteside of the UK Building Research Station recorded the weekly gas consumption and average external temperature at his own house in southeast England for two heating seasons, one of 26 weeks before, and one of 30

148

weeks after cavity-wall insulation was installed. The object of the exercise was to assess the effect of the insulation on gas consumption."

The whiteside data frame has 56 rows and 3 columns:

- Insul: A factor, before or after insulation.
- Temp: average outside temperature in degrees Celsius.
- Gas: weekly gas consumption in 1000s of cubic feet.

Below we demonstrate modeling Gas as a function of Insul, Temp, and their interaction.

Obviously this is not a comprehensive treatment of linear modeling.

Python

\mathbf{R}

The lm() function fits a linear model in R using whatever model we propose. We specify models using a special syntax. The basic construction is to first list your dependent or response variable, then a tilde (~), and then your predictor variables, or terms, separated by plus operators (+). Listing two variables separated by a colon (:) indicates we wish to fit an interaction for those variables. See ?formula for further details on formula syntax.

It's considered best practice to reference variables in a data frame and indicate the data frame using the data argument. Though not required, you'll almost always want to save the result to an object for further inquiry.

```
> m <- lm(Gas ~ Insul + Temp + Insul:Temp, data = whiteside)
```

Once you fit your model, you can extract information about it using several functions. The most commonly used include:

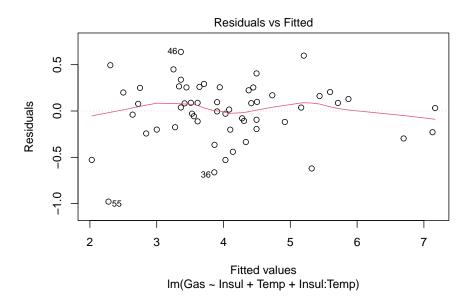
- summary(): summary of model coefficients with standard errors and test statistics
- coef(): model coefficients
- confint(): 95% confidence interval of model coefficients
- plot(): a set of four diagnostic plots

The summary() function produces the standard regression summary one typically finds described in a statistics textbook.

```
> summary(m)
Call:
lm(formula = Gas ~ Insul + Temp + Insul:Temp, data = whiteside)
Residuals:
    Min
             1Q
                  Median
                              3Q
                                     Max
-0.97802 -0.18011 0.03757 0.20930 0.63803
Coefficients:
              Estimate Std. Error t value Pr(>|t|)
(Intercept)
              6.85383 0.13596 50.409 < 2e-16 ***
InsulAfter
              Temp
              -0.39324
                         0.02249 -17.487 < 2e-16 ***
                         0.03211 3.591 0.000731 ***
InsulAfter:Temp 0.11530
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
Residual standard error: 0.323 on 52 degrees of freedom
Multiple R-squared: 0.9277, Adjusted R-squared: 0.9235
F-statistic: 222.3 on 3 and 52 DF, p-value: < 2.2e-16
```

Calling plot() on a model object produces four different diagnostic plots by default. Using the which argument we can specify which of six possible plots to create. The first one checks the constant variance assumption (ie, that our model is not dramatically over- or under-predicting values.) We hope to see residuals evenly scattered around 0. (See ?plot.lm for more details on the diagnostic plots.)

```
> plot(m, which = 1)
```

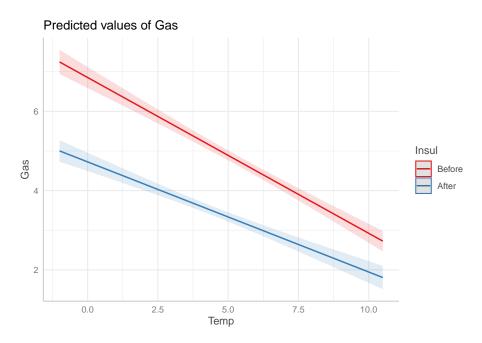


Once we fit a model and we're reasonably confident that it's a good model, we may want to visualize it. Three packages in R that help with this are **emmeans**, **effects**, and **ggeffects**. We briefly demonstrate the **ggeffects** package.

You need to first install the **ggeffects** package as it does not come with the base R installation. Once installed, load using the library() function.

Once loaded, we can get a basic visualization of our model by using the plot() and ggpredict() functions. This is particularly useful for models with interactions. Use the terms argument to specify which variables to plot. Below we list "Temp" first, which will plot "Temp" on the x axis. Then we list "Insul", the grouping variable, to indicate we want a separate fit for each level of "Insul".

```
> # install.pacakges("ggeffects")
> library(ggeffects)
Warning: package 'ggeffects' was built under R version 4.1.2
> plot(ggpredict(m, terms = c("Temp", "Insul")))
```



We see that after installing insulation, gas consumption fell considerably, and that the effect of temperature on gas consumption is less pronounced.

8.4 Logistic regression

Logistic regression attempts to assess if or how the variability a binary variable depends on one or more predictor variables. It is a type of Generalized Linear Model and is commonly used to model the *probability* of an event occurring. While it is relatively easy to "fit a model" and generate lots of output, the model we fit may not be very good. There are many decisions we have to make when proposing a model. Which predictors do we include? Will they interact? Do we allow for non-linear effects? Answering these kinds of questions require subject matter expertise.

We walk through a basic example using data on low infant birth weight. The data is courtesy of the R package MASS [Venables and Ripley, 2002]. According to the documentation, "the data were collected at Baystate Medical Center, Springfield, Mass during 1986."

We use the data as prepared in the example code found at ?birthwt.

The birthwt data frame has 189 rows and 9 columns:

- low: 1 if birth weight less than 2.5 kg, 0 otherwise
- age: mother's age in years

- lwt: mother's weight in pounds at last menstrual period
- race: mother's race (white, black, other)
- smoke: smoking status during pregnancy (1 = yes, 0 = no)
- ptd: previous premature labors (1 = yes, 0 = no)
- ht: history of hypertension (1 = yes, 0 = no)
- ui: presence of uterine irritability (1 = yes, 0 = no)
- ftv: number of physician visits during the first trimester (0, 1, 2+)

Below we demonstrate modeling low as a function of all other predictors.

Obviously this is not a comprehensive treatment of logistic regression.

Python

\mathbf{R}

The glm() function fits a generalized linear model in R using whatever model we propose. We specify models using a special syntax. The basic construction is to first list your dependent or response variable, then a tilde (~), and then your predictor variables, or terms, separated by plus operators (+). Listing two variables separated by a colon (:) indicates we wish to fit an interaction for those variables. See ?formula for further details on formula syntax.

In addition, glm() requires we specify a family argument to specify the error distribution for the dependent variable. The default is gaussian. For a logistic regression model, we need to specify binomial since our dependent variable is binary.

It's considered best practice to reference variables in a data frame and indicate the data frame using the data argument. Though not required, you'll almost always want to save the result to an object for further inquiry.

Since we're modeling low as a function of all other variables in the data frame, we could have used the following syntax, where the period symbolizes all other remaining variables:

```
> mod <- glm(low ~ ., data = birthwt, family = binomial)</pre>
```

Once you fit your logistic regression model, you can extract information about it using several functions. The most commonly used include:

- summary(): summary of model coefficients with standard errors and test statistics
- coef(): model coefficients
- confint(): 95% confidence interval of model coefficients

The summary() function produces the standard regression summary one typically finds described in a statistics textbook.

```
> summary(mod)
Call:
glm(formula = low ~ age + lwt + race + smoke + ptd + ht + ui +
    ftv, family = binomial, data = birthwt)
Deviance Residuals:
   Min
             1Q
                 Median
                               3Q
                                      Max
-1.7038 -0.8068 -0.5008
                          0.8835
                                    2.2152
Coefficients:
           Estimate Std. Error z value Pr(>|z|)
(Intercept) 0.82302 1.24471
                                0.661
                                       0.50848
           -0.03723
                       0.03870 -0.962 0.33602
age
                    0.00708 -2.211 0.02705 *
lwt
           -0.01565
                       0.53597
                                2.225 0.02609 *
raceblack
          1.19241
raceother
            0.74069
                       0.46174
                                1.604 0.10869
                       0.42502
smoke1
           0.75553
                               1.778 0.07546 .
           1.34376
                       0.48062
                               2.796 0.00518 **
ptd1
ht1
                       0.72074
                                2.654 0.00794 **
            1.91317
            0.68019
                       0.46434
                                1.465 0.14296
ui1
           -0.43638
                       0.47939 -0.910 0.36268
ftv1
ftv2+
           0.17901
                                0.392 0.69488
                       0.45638
Signif. codes: 0 '***' 0.001 '**' 0.01 '*' 0.05 '.' 0.1 ' ' 1
(Dispersion parameter for binomial family taken to be 1)
    Null deviance: 234.67 on 188 degrees of freedom
Residual deviance: 195.48 on 178 degrees of freedom
AIC: 217.48
Number of Fisher Scoring iterations: 4
```

Exponentiating coefficients in a logistic regression model produces odds ratios. To get the odds ratio for the previous premature labors variable, ptd, we do the following:

```
> exp(coef(mod)["ptd1"])
    ptd1
3.833443
```

This says the odds of having an infant with low birth weight are about 3.8 times higher for women who experienced previous premature labors versus women who did not, assuming all other variables equal.

The 3.8 value is just an estimate. We can use the confint() function to get a 95% confidence interval on the odds ratio.

```
> exp(confint(mod)["ptd1",])
    2.5 % 97.5 %
1.516837 10.128974
```

It appears the odds ratio is at least 1.5, possibly as high as 10.1 (assuming we believe this model).

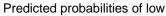
Once we fit a model and we're reasonably confident that it's a good model, we may want to visualize it. Three packages in R that help with this are **emmeans**, **effects**, and **ggeffects**. We briefly demonstrate the **ggeffects** package.

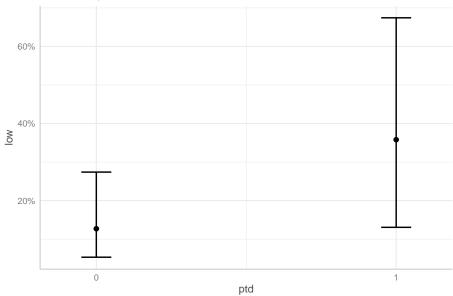
You need to first install the **ggeffects** package as it does not come with the base R installation. Once installed, load using the library() function.

Once loaded, we can get a basic visualization of our model by using the plot() and ggpredict() functions. This is particularly useful for logistic regression models because it produces model predictions on a probability scale.

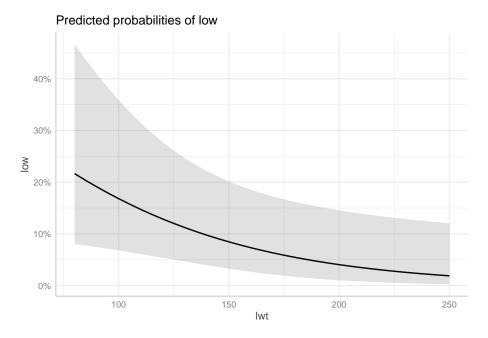
Use the terms argument to specify which variables to plot. Below we create two plots: one for ptd (previous premature labors) and one for lwt (mother's weight at last menstrual period).

```
> # install.packages("ggeffects")
> library(ggeffects)
> plot(ggpredict(mod, terms = "ptd"))
```





It looks like the probability of low infant birth weight jumps from about 12% to over 35% for mothers who previously experienced premature labors, though the error bars on the expected values are quite large.



It appears the probability of low infant birth weight drops from about 15% when a mother weighs 100 lbs to about 5% when a mother weighs around 200 lbs. The regions with the larger confidence ribbon indicate regions of higher uncertainty. There are clearly not many mothers in our data who weigh less than 100 lbs.

Bibliography

Peter Dalgaard. Introductory Statistics with R. Springer, 2 edition, 2008.

Robert E. Kass, Brian S. Caffo, Marie Davidian, Xiao-Li Meng, Bin Yu, and Nancy Reid. Ten simple rules for effective statistical practice. *PLOS Computational Biology*, 12(6), 2016. doi: 10.1371/journal.pcbi.1004961.

W. N. Venables and B. D. Ripley. Modern Applied Statistics with S. Springer, New York, fourth edition, 2002. URL https://www.stats.ox.ac.uk/pub/ MASS4/. ISBN 0-387-95457-0.

Michael Whitlock and Dolph Schluter. *The Analysis of Biological Data*. Macmillan Learning, 3 edition, 2020.