

# Python and R

Clay Ford, Jacob Goldstein-Greenwood, Oyinkansola Adenekan, Samantha Lomuscio

2021-11-03



# Contents

<b>Welcome</b>	<b>7</b>
<b>1 Basics</b>	<b>9</b>
1.1 Math . . . . .	9
1.2 Assignment . . . . .	10
1.3 Printing a value . . . . .	10
1.4 Packages . . . . .	11
1.5 Logic . . . . .	12
1.6 Generating a sequence of values . . . . .	13
1.7 Calculating means and medians . . . . .	13
1.8 Writing your own functions . . . . .	14
<b>2 Data Structures</b>	<b>15</b>
2.1 One-dimensional data . . . . .	15
2.2 Two-dimensional data . . . . .	19
2.3 Three-dimensional and higher data . . . . .	22
<b>3 Importing Data</b>	<b>25</b>
3.1 CSV . . . . .	25
3.2 XLS/XLSX (Excel) . . . . .	26
3.3 JSON . . . . .	27
3.4 XML . . . . .	29

<b>4</b>	<b>Data Manipulation</b>	<b>31</b>
4.1	Names of variables and their types . . . . .	31
4.2	Access variables . . . . .	33
4.3	Rename variables . . . . .	39
4.4	Create, replace and remove variables . . . . .	42
4.5	Create strings from numbers . . . . .	43
4.6	Create numbers from strings . . . . .	45
4.7	Change case . . . . .	46
4.8	Drop duplicate rows . . . . .	47
4.9	Randomly sample rows . . . . .	48
<b>5</b>	<b>Combine, Reshape and Merge</b>	<b>53</b>
5.1	Combine rows . . . . .	53
5.2	Combine columns . . . . .	54
5.3	Reshaping data . . . . .	55
5.4	Merge/Join . . . . .	61
<b>6</b>	<b>Aggregation and Group Operations</b>	<b>67</b>
6.1	Cross tabulation . . . . .	67
6.2	Group summaries . . . . .	69
6.3	Centering and Scaling . . . . .	71
<b>7</b>	<b>Basic Plotting and Visualization</b>	<b>73</b>
7.1	Histograms . . . . .	73
7.2	Barplots . . . . .	73
7.3	Scatterplot . . . . .	73
7.4	Stripcharts . . . . .	74
7.5	Boxplots . . . . .	74
7.6	Conditional or Faceted plots . . . . .	74

<i>CONTENTS</i>	5
<b>8 Statistical Inference and Modeling</b>	<b>75</b>
8.1 Comparing group means . . . . .	75
8.2 Simple linear regression . . . . .	75
8.3 Multiple regression . . . . .	76
8.4 Logistic regression . . . . .	76



# 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.





# 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.

#### R

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

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

## 1.2 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 it 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 `?assignOps`.

### Python

```
> x = 12
```

### R

```
> x <- 12
```

## 1.3 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
```

**R**

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

## 1.4 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****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 auto-complete 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 the `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*.

## 1.5 Logic

Python and R share the same operators for making comparisons:

- `==` (equals)
- `!=` (not equal to)
- `<` (less than)
- `<=` (less than or equal to)
- `>` (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 and xor (exclusive OR).

Python

R

## 1.6 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])
```

R

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

## 1.7 Calculating means and medians

The **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)
101.66666666666667
```

Median, using function from **NumPy** library

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

## 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
```

## 1.8 Writing your own functions

### Python

### R

## 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. 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.

```
> l = [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
+ l[0]
+
```

```

+ #extract last element
1
> l[-1]
+
+ # extract 2nd and 3rd elements
'hello'
> l[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')

```

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
1      2
2      3
3      hello
dtype: object
> ser1[0]
+
+ # extract 2nd and 3rd elements
'1'
> ser1[1:3]
1      2
2      3
dtype: object

```

To extract the last element of a pandas Series using `-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.

```

> import pandas as pd
+ import numpy as np
+
+ ser2 = pd.Series(data, index=['a', 'b', 'c', 'd'])
+ print(ser2)
a      1
b      2
c      3
d      hello
dtype: object

```

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
'2'
> ser2.loc["b":]
+
+ # extract element in row "d"
b      2
c      3
d      hello
dtype: object
> ser2.d
+
+ # extract element in row "b"
'hello'
> ser2.b
'2'

```

One thing to note is that mathematical operations cannot be carried out on lists, but 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.

## 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.

```
> # 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
```

```
'world'
> arr2d[:,1:3]
array([[ '2', '3'],
       ['5', '6']], dtype='<U11')
```

A pandas DataFrame is made using the `pandas.DataFrame` function in a similar way to the pandas Series.

```
> import pandas as pd
+ import numpy as np
+
+ data = np.array([[1,2,3,"hello"],[4,5,6,"world"]])
+ df = pd.DataFrame(data)
+ print(df)
   0  1  2      3
0  1  2  3  hello
1  4  5  6  world
```

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

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

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
+
```

```
+ data = np.array([[1,2,3,"hello"],[4,5,6,"world"]])
+ df = pd.DataFrame(data, index=["a","b"])
+ print(df)
   0  1  2    3
a  1  2  3  hello
b  4  5  6  world
```

We can change the column names using the `columns` attribute in the `pandas.DataFrame` function:

```
> import pandas as pd
+ import numpy as np
+
+ data = np.array([[1,2,3,"hello"],[4,5,6,"world"]])
+ df = pd.DataFrame(data, index=["a","b"], columns=["column 1","column 2", "column 3", "column 4"])
+ print(df)
   column 1 column 2 column 3 column 4
a         1         2         3   hello
b         4         5         6   world
```

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.

## 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.

```
> # matrix; populated down by column by default
> m <- matrix(data = c(1,3,5,7), nrow = 2, ncol = 2)
> m
     [,1] [,2]
[1,]    1    5
[2,]    3    7
>
> # data frame
> d <- data.frame(name = c("Rob", "Cindy"),
+                 age = c(35, 37))
> d
```

```

      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

### R

The `array()` function in R can create three-dimensional and higher data structures. 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 are simply the numbers 1 through 12.

```

> a1 <- array(data = 1:12, dim = c(2,3,2))
> a1
, , 1

      [,1] [,2] [,3]
[1,]    1    3    5
[2,]    2    4    6

```

```
, , 2

      [,1] [,2] [,3]
[1,]    7    9   11
[2,]    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*.

```
> dimnames(a1) <- list("X" = c("x1", "x2"),
+                      "Y" = c("y1", "y2", "y3"),
+                      "Z" = c("z1", "z2"))
> a1
, , Z = z1

      Y
X     y1 y2 y3
x1    1  3  5
x2    2  4  6

, , Z = z2

      Y
X     y1 y2 y3
x1    7  9 11
x2    8 10 12
```

The `as.data.frame.table()` function can collapse an array into a two-dimensional 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    4
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
```



## Chapter 3

# Importing 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. The examples below highlight one way that we frequently see used.

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

### 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.

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

	Grade	Year	Mean Scale Score
0	3	2006	700
1	4	2006	699
2	5	2006	691

## R

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

```
> d <- read.csv("data/ny_math_test.csv")
> d[1:3, c("Grade", "Year", "Mean.Scale.Score")]
  Grade Year Mean.Scale.Score
1     3 2006              700
2     4 2006              699
3     5 2006              691
```

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"]]
>
```

## 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.

```
> library(readxl)
> d_xls <- read_excel("data/ny_math_test.xlsx")
> d_xls[1:3, c("Grade", "Year", "Mean Scale Score")]
# A tibble: 3 x 3
  Grade Year `Mean Scale Score`
  <chr> <dbl>           <dbl>
1 3     2006             700
2 4     2006             699
3 5     2006             691
```

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 (**J**ava**S**cript **O**bject **N**otation) 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 **json\_normalize()** 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
0	3	2006	700
1	4	2006	699
2	5	2006	691

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

## 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)

```

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.

```

> colnames(d_json$data) <- d_json$meta$view$columns$fieldname
> d_json <- as.data.frame(d_json$data)
> d_json[1:3,c("grade", "year", "mean_scale_score")]

```

	grade	year	mean_scale_score
1	3	2006	700
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 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
1      4  2006                699
2      5  2006                691
```

### 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')
> d_list <- as_list(d_xml)
```

```
> d <- dplyr::bind_rows(d_list$response$row)
> d[] <- lapply(d, unlist)
> 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.

## Chapter 4

# Data Manipulation

This chapter looks at various strategies for 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
```

### 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   mpg     32 non-null      float64
1   cyl     32 non-null      int64
2   disp    32 non-null      float64
3   hp      32 non-null      int64
4   drat    32 non-null      float64
5   wt      32 non-null      float64
6   qsec    32 non-null      float64
7   vs      32 non-null      int64
8   am      32 non-null      int64
9   gear    32 non-null      int64
10  carb    32 non-null      int64
dtypes: float64(5), int64(6)
memory usage: 2.9 KB
```

By default, the `verbose` argument is set to `False`. Then, the function lists the unique data types in the `DataFrame`, and it prints how much memory the `DataFrame` takes up. This setting excludes the table describing each column.

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



**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.

```
> str(mtcars)
'data.frame':   32 obs. of  11 variables:
 $ mpg : num  21 21 22.8 21.4 18.7 18.1 14.3 24.4 22.8 19.2 ...
 $ cyl : num   6  6  4  6  8  6  8  4  4  6 ...
 $ disp: num  160 160 108 258 360 ...
 $ hp  : num  110 110 93 110 175 105 245 62 95 123 ...
 $ 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 Access variables

How to work with a specific column of data.

**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
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
13     15.2
14     10.4
15     10.4
16     14.7
17     32.4
18     30.4
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
```

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

```
> mtcars['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
13     15.2
14     10.4
15     10.4
16     14.7
17     32.4
18     30.4
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.

```
> mtcars['mpg'].values
array([21. , 21. , 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, 10.4, 14.7, 32.4, 30.4, 33.9, 21.5, 15.5,
       15.2, 13.3, 19.2, 27.3, 26. , 30.4, 15.8, 19.7, 15. , 21.4])
```

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
13 15.2
14 10.4
15 10.4
16 14.7
17 32.4
18 30.4
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
```

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 array, 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
```

## 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 return a data frame instead of a vector (if used with a data frame).

```
> mtcars["mpg"]
      mpg
1  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 `rows, 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
1  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 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.

```

> mtcars.rename(columns={"cyl": "cylinder", "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"
+
+ # using indexing
+ column_names[2] = "DISP"
+
```



```
+ mtcars.columns = column_names
+ mtcars.columns
Index(['mpg', 'cyl', 'DISP', 'hp', 'axle_ratio', 'wt', 'qsec', 'vs', 'am',
      'gear', 'carb'],
      dtype='object')
```

You can

## 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 “cylinder”.

```
> names(mtcars)[2]
[1] "cyl"
> names(mtcars)[2] <- "cylinders"
> names(mtcars)
[1] "mpg"      "cylinders" "disp"      "hp"      "drat"      "wt"
[7] "qsec"      "vs"        "am"        "gear"     "carb"
```

Variable names can also be changed by conditional match. Below we find the variable name that matches “drat” and change to “axle\_ratio”.

```
> names(mtcars)[names(mtcars) == "drat"]
[1] "drat"
> names(mtcars)[names(mtcars) == "drat"] <- "axle_ratio"
> names(mtcars)
[1] "mpg"      "cylinders" "disp"      "hp"      "axle_ratio"
[6] "wt"      "qsec"      "vs"        "am"      "gear"
[11] "carb"
```

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

```
> names(mtcars)[c(6,8)] <- c("weight", "engine")
>
> # or
> # names(mtcars)[names(mtcars) %in% c("wt", "vs")] <- c("weight", "engine")
>
> names(mtcars)
[1] "mpg"      "cylinders" "disp"      "hp"      "axle_ratio"
[6] "weight"   "qsec"      "engine"    "am"      "gear"
[11] "carb"
```

See also the `rename()` function in the **dplyr** package.

## 4.4 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'])
```

	mpg	cyl	DISP	hp	axle_ratio	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

## 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
```

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

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

To remove a variable, assign it NULL.

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

## 4.5 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'>
```

## 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" "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 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
```

TODO: add zip code conversion using `str_pad()` (or base R option?)

## 4.6 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.]
```

## 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 `sub` function to find “lbs.” and replace with nothing.

```
> 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
```

## 4.7 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 word in a list.

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

The `title` function changes all fo the `col_names = [col.upper() for col in mtcars.columns]`

## R

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

```
> names(mtcars) <- toupper(names(mtcars))
> names(mtcars)
[1] "MPG"      "CYLINDERS" "DISP"      "HP"      "AXLE_RATIO"
[6] "WEIGHT"   "QSEC"      "ENGINE"    "AM"      "GEAR"
[11] "CARB"     "AM_CH"     "AM_FAC"
```

```
> names(mtcars) <- tolower(names(mtcars))
> names(mtcars)
[1] "mpg"      "cylinders" "disp"      "hp"      "axle_ratio"
[6] "weight"   "qsec"      "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.

```
> stringr::str_to_title(names(mtcars))
[1] "Mpg"      "Cylinders" "Disp"      "Hp"      "Axle_ratio"
[6] "Weight"   "Qsec"      "Engine"    "Am"      "Gear"
[11] "Carb"     "Am_ch"     "Am_fac"
```

## 4.8 Drop duplicate rows

How to find and drop duplicate elements.

### Python

## 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])
> # 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 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

```

## 4.9 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`.

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.

```
> # sample 5 rows from mtcars
+ mtcars.sample(n=5, replace=True)
+
+ # sample 20% of rows from mtcars
```

	MPG	CYL	DISP	HP	AXLE_RATIO	WT	QSEC	VS	AM	GEAR	CARB	KPL
4	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2	3.975340
0	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	4.464286
6	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4	3.039966
0	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4	4.464286
3	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1	4.549320

```
> mtcars.sample(frac = 0.20, random_state=1)
```

	MPG	CYL	DISP	HP	AXLE_RATIO	...	VS	AM	GEAR	CARB	KPL
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	75.7	52	4.93	...	1	1	4	2	6.462585
23	13.3	8	350.0	245	3.73	...	0	0	3	4	2.827381
17	32.4	4	78.7	66	4.08	...	1	1	4	1	6.887755

```
[6 rows x 12 columns]
```

The numpy function `random.choice()` can be used to sample from a DataFrame.

## 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
1	15.0	8	301.0	335	3.54	3.570	14.60	0	1	5	8	1
2	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1	1
3	32.4	4	78.7	66	4.08	2.200	19.47	1	1	4	1	1

```

4 26.0      4 120.3  91      4.43  2.140 16.70      0  1  5  2  1
5 21.5      4 120.1  97      3.70  2.465 20.01      1  0  3  1  0
      am_fac
1  manual
2  manual
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
1 17.3           8 275.8 180     3.07  3.730 17.60      0  0  3  3  0
2 26.0           4 120.3  91     4.43  2.140 16.70      0  1  5  2  1
3 18.7           8 360.0 175     3.15  3.440 17.02      0  0  3  2  0
4 21.5           4 120.1  97     3.70  2.465 20.01      1  0  3  1  0
5 18.1           6 225.0 105     2.76  3.460 20.22      1  0  3  1  0
6 13.3           8 350.0 245     3.73  3.840 15.41      0  0  3  4  0
      am_fac
1 automatic
2  manual
3 automatic
4 automatic
5 automatic
6 automatic

```

To sample with replacement, set `replace = TRUE`.

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

```

> # sample 5 rows from mtcars
> # get random row numbers
> i <- sample(nrow(mtcars), size = 5)
> # use i to select rows
> mtcars[i,]
      mpg cylinders  disp  hp axle_ratio weight  qsec engine am gear carb am_ch
7 14.3           8 360.0 245     3.21  3.57 15.84      0  0  3  4  0
11 17.8          6 167.6 123     3.92  3.44 18.90      1  0  4  4  0
24 13.3           8 350.0 245     3.73  3.84 15.41      0  0  3  4  0
14 15.2           8 275.8 180     3.07  3.78 18.00      0  0  3  3  0
12 16.4           8 275.8 180     3.07  4.07 17.40      0  0  3  3  0
      am_fac
7  automatic
11 automatic
24 automatic

```

```
14 automatic
12 automatic
```

```
> # sample about 20% of rows from mtcars
> # generate random values on range of [0,1]
> i <- runif(nrow(mtcars))
> # 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
6  18.1         6 225.0 105    2.76   3.460 20.22     1  0   3    1    0
14 15.2         8 275.8 180    3.07   3.780 18.00     0  0   3    3    0
15 10.4         8 472.0 205    2.93   5.250 17.98     0  0   3    4    0
20 33.9         4  71.1  65    4.22   1.835 19.90     1  1   4    1    1
30 19.7         6 145.0 175    3.62   2.770 15.50     0  1   5    6    1
32 21.4         4 121.0 109    4.11   2.780 18.60     1  1   4    2    1
   am_fac
6  automatic
14 automatic
15 automatic
20   manual
30   manual
32   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))
> mtcars[i < 0.20,]
   mpg cylinders  disp  hp axle_ratio weight  qsec engine am gear carb am_ch
6  18.1         6 225.0 105    2.76   3.46 20.22     1  0   3    1    0
15 10.4         8 472.0 205    2.93   5.25 17.98     0  0   3    4    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    6    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

#### 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

### 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
```

```
> # example of recycled rows (d1 is repeated twice)
> d1 <- data.frame(x = 10:13, y = letters[1:4])
> d2 <- data.frame(x = c(23,34,45,44,99,99,99,99))
> cbind(d1, d2)
   x y  x
1 10 a 23
2 11 b 34
3 12 c 45
4 13 d 44
5 10 a 99
6 11 b 99
7 12 c 99
8 13 d 99
```

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:

	name	time_1_score	time_2_score
1	larry	3	0
2	moe	6	3
3	curly	2	1

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

#### 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 output columns in long data:

```
> df_long <- reshape(df_wide,
+                     direction = 'long',
+                     idvar = c('id', 'sex'), # column(s) that uniquely identifies
+                     varying = c('wk1', 'wk2', 'wk3'), # variables that contain t
+                     v.names = 'val', # desired name of column in long data that
+                     timevar = 'week') # desired name of column in long data that
> df_long
  id sex week val
1.m.1 1  m   1  16
2.m.1 2  m   1  12
3.f.1 3  f   1   8
1.m.2 1  m   2   7
2.m.2 2  m   2  19
3.f.2 3  f   2  15
1.m.3 1  m   3  15
2.m.3 2  m   3  10
3.f.3 3  f   3   7
```

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,
+                             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     1  m   wk3    15
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
9     3  f   wk3     7
```

`pivot_longer()` is particularly useful (a) when dealing with wide data that con-



tain 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_excitement
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,
+                               cols = -c('animal', 'lives_in_water'),
+                               names_to = c('month', '.value'), # ".value" is placeholder for str
+                               names_pattern = '(.+)_(.+)') # specify structure of wide column names
> animals_long_1
# A tibble: 6 x 5
  animal    lives_in_water month playfulness excitement
  <chr>      <lgl>      <chr>      <dbl>      <dbl>
1 dolphin    TRUE      jan         6          7
2 dolphin    TRUE      feb        5.5         7
3 porcupine FALSE     jan        3.5        3.5
4 porcupine FALSE     feb        4.5        3.5
5 capybara  FALSE     jan         4          4
6 capybara  FALSE     feb         5          4
```

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,
+                               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>      <lgl>          <chr> <chr>    <dbl>
1 dolphin   TRUE            jan  playfulness 6
2 dolphin   TRUE            feb  playfulness 5.5
3 dolphin   TRUE            jan  excitement  7
4 dolphin   TRUE            feb  excitement  7
5 porcupine FALSE          jan  playfulness 3.5
6 porcupine FALSE          feb  playfulness 4.5
7 porcupine FALSE          jan  excitement  3.5
8 porcupine FALSE          feb  excitement  3.5
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

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  m   1  12
3.f.1 3  f   1   8
1.m.2 1  m   2   7
2.m.2 2  m   2  19
3.f.2 3  f   2  15
1.m.3 1  m   3  15
2.m.3 2  m   3  10
3.f.3 3  f   3   7
```

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

```
> df_wide <- reshape(df_long,
+                     direction = 'wide',
+                     idvar = c('id', 'sex'), # column(s) that determine which rows show
+                     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 contents of
> df_wide
  id sex val_1 val_2 val_3
1.m.1 1  m   16    7   15
2.m.1 2  m   12   19   10
3.f.1 3  f    8   15    7
```

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

```
> library(tidyverse)
> df_wide_PW <- pivot_wider(df_long,
+                           id_cols = c('id', 'sex'),
+                           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>
1     1 m     16     7     15
2     2 m     12    19     10
3     3 f      8    15      7
```

`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
  animal  lives_in_water month playfulness excitement
<chr>    <lgl>         <chr>         <dbl>         <dbl>
1 dolphin TRUE         jan           6           7
2 dolphin TRUE         feb          5.5          7
3 porcupine FALSE       jan          3.5          3.5
4 porcupine FALSE       feb          4.5          3.5
5 capybara FALSE       jan           4           4
6 capybara FALSE       feb           5           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` allow fo
> animals_wide
# A tibble: 3 x 6
  animal    lives_in_water jan_playfulness feb_playfulness jan_excitement
  <chr>      <lgl>             <dbl>           <dbl>           <dbl>
1 dolphin   TRUE                6              5.5             7
2 porcupine FALSE                3.5           4.5             3.5
3 capybara  FALSE                4              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   TRUE            jan  excitement 7
4 dolphin   TRUE            feb  excitement 7
5 porcupine FALSE            jan  playfulness 3.5
6 porcupine FALSE            feb  playfulness 4.5
7 porcupine FALSE            jan  excitement 3.5
8 porcupine FALSE            feb  excitement 3.5
9 capybara  FALSE            jan  playfulness 4
10 capybara  FALSE            feb  playfulness 5
11 capybara  FALSE            jan  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>      <lgl>             <dbl>           <dbl>           <dbl>
1 dolphin   TRUE                6              5.5             7
2 porcupine FALSE                3.5           4.5             3.5
3 capybara  FALSE                4              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
> x
  merge_var val_x
1         a    12
2         b    94
3         c    92
> # y
> y
  merge_var val_y
1         c    78
2         d    32
3         e    30
```

### 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		merge_var	val_y		merge_var	val_y
a	12	+ (left join on merge_var)	c	78	=	a	12
b	94		d	32		b	94
c	92		e	30		c	92
x			y				

### 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

## 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
```

### 5.4.2 Right Join

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

merge_var	val_x	+	merge_var	val_y	=	merge_var	val_y
a	12		c	78		c	78
b	94		d	32		d	32
c	92		e	30		e	30

$x$ 
 $y$

## Python

```
> import pandas as pd
+ pd.merge(x, y, how = 'right')
merge_var val_x val_y
0         c    92.0    78.0
1         d     NaN    32.0
2         e     NaN    30.0
```

R

```
> # all.y = T results in a right join
> merge(x, y, by = 'merge_var', all.y = T)
  merge_var val_x val_y
1         c    92    78
2         d     NA    32
3         e     NA    30
```

### 5.4.3 Inner Join

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

merge_var	val_x	+ (inner join on merge_var)	merge_var	val_y	=	merge_var	val_y
a	12		c	78		c	92
b	94		d	32			
c	92		e	30			
$x$			$y$				

Python

```
> import pandas as pd
+ pd.merge(x, y, how = 'inner')
  merge_var  val_x  val_y
0         c   92.0   78.0
```

R

```
> # by default, merge() executes an inner join
> # (more specifically, a natural join, which is a kind of
> # inner join in which the merge-criterion column is not
> # repeated, despite being initially present in both tables)
```

```
> merge(x, y, by = 'merge_var')
  merge_var val_x val_y
1         c    92    78
```

#### 5.4.4 Outer Join

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

merge_var	val_x		merge_var	val_y		merge_var
a	12	+  (outer join on merge_var)	c	78	=	a
b	94		d	32		b
c	92		e	30		c
x			y			d
						e

#### 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
```

#### 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
1         a    12    NA
2         b    94    NA
3         c    92    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 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. The `groupby()`, 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)
sex      f  m
nationality
Canadian    1  1
Egyptian    0  1
French      2  0
> dat.groupby(by = 'nationality').nationality.count()
nationality
Canadian    2
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
           m      1
Egyptian    m      1
French      f      2
Name: nationality, dtype: int64

```

## 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)
      sex
nationality f m
Canadian   1 1
Egyptian   0 1

```

French 2 0

## 6.2 Group summaries

Computing statistical summaries per group.

### Python

### R

The `aggregate()` function allows a user to easily 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 variables (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.

```
> # One grouping variable
> # Calculating mean of `mpg` in each `cyl` group
> aggregate(x = mtcars$mpg,
+           by = list(cyl = mtcars$cyl),
+           FUN = "mean")
  cyl      x
1   4 26.66364
2   6 19.74286
3   8 15.10000
```

Adding `drop=FALSE` ensures all combinations of levels are returned 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      x
1   4  0 26.0
2   6  0 21.0
3   8  0 19.2
4   4  1 33.9
```

```
5  6  1 21.4
6  8  1  NA
```

```
> # Or, specify the variable to summarize and the grouping variables in formula notation
> aggregate(mpg ~ cyl, data = mtcars, FUN = mean)
> 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]
   cyl    vs avg_mpg
<dbl> <dbl> <dbl>
1     4     0    26
2     4     1   26.7
3     6     0   20.6
4     6     1   19.1
5     8     0   15.1
```

A benefit of `summarize()` is that it allows a user 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(displ)^(1/mean(hp)))
`summarise()` has grouped output by 'cyl'. You can override using the `groups` argument
# A tibble: 5 x 4
# Groups:   cyl [3]
   cyl    vs avg_mpg complicated_summary_calculation
<dbl> <dbl> <dbl> <dbl>
1     4     0    26                8.51
```

2	4	1	26.7	8.07
3	6	0	20.6	8.41
4	6	1	19.1	8.81
5	8	0	15.1	7.48

## 6.3 Centering and Scaling

*Centering* refers to rescaling a column or vector of values such that their mean is 0. 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 0 and their standard deviation is 1. 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

### R

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

To center a variable (without scaling it), call `scale()` with the `center` argument set to `TRUE` and the `scale` argument set to `FALSE`. The variable's mean will be subtracted off of each of the variable values. (Note: If desired, the `center` argument can be set to a numeric value instead of `TRUE/FALSE`; 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 `scale()` with the `center` and `scale` 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 `center` argument, the `scale` argument can also be set to a numeric value instead of `TRUE/FALSE`; 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 better understand data. Visualization in Python and R is a gigantic and evolving topic. We don't pretend to present a comprehensive comparison.

### 7.1 Histograms

Visualizing the distribution of numeric data.

**Python**

**R**

### 7.2 Barplots

Visualizing the distribution of categorical data.

**Python**

**R**

### 7.3 Scatterplot

Visualizing the relationship between two numeric variables.

**Python**

**R**

## 7.4 Stripcharts

Visualizing the relationship between a numeric variable and a categorical variable.

**Python**

**R**

## 7.5 Boxplots

Visualizing the relationship between a numeric variable and a categorical variable via 5 number summaries.

**Python**

**R**

## 7.6 Conditional or Faceted plots

Two or more plots of subsets of data.

## Chapter 8

# Statistical Inference and Modeling

This chapter looks at performing and interpreting common statistical analyses.

### 8.1 Comparing group means

Comparing the means of two or more groups to see if or how they differ. Two means can be analyzed with a t test. Three or more can be analyzed with ANOVA. Both the t test and ANOVA are special cases of a linear model.

**Python**

**R**

### 8.2 Simple linear regression

Analyzing if or how the variability a numeric variable depends on another numeric variable.

**Python**

**R**

### **8.3 Multiple regression**

Analyzing if or how the variability a numeric variable depends on multiple numeric variables.

**Python**

**R**

### **8.4 Logistic regression**

Analyzing if or how the variability of a binary variable depends on one or more predictor variables.

**Python**

**R**