# Python and R

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

2021-10-04

# Contents

# Welcome

This book provides parallel examples in Python and R to help users of one platform more easily transition to the other.

# 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 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 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 start/stop interval in `arange()` is *open*, but the from/to interval in `seq()` is *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
```

# 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

### 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 is 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, or matrices as you might encounter in matrix algebra.

### Python

### 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) 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 is 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

```
> import pandas
+ d = pandas.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

### 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.

```
> # read in the 2nd sheet
> 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 for importing JSON files into Python or R.

### Python

### 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 (e**X**tensible **M**arkup **L**anguage) 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

### 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

#### 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

### 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

### 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

### 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

### 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

### 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**

**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

### 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 21.0         6 160.0 110       3.90  2.875 17.02      0  1    4    4     1
2 17.3         8 275.8 180       3.07  3.730 17.60      0  0    3    3     0
3 16.4         8 275.8 180       3.07  4.070 17.40      0  0    3    3     0
4 26.0         4 120.3  91       4.43  2.140 16.70      0  1    5    2     1
5 17.8         6 167.6 123       3.92  3.440 18.90      1  0    4    4     0
     am_fac
1    manual
2 automatic
3 automatic
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 15.8         8 351.0 264       4.22  3.170 14.50      0  1    5    4     1
2 30.4         4  95.1 113       3.77  1.513 16.90      1  1    5    2     1
3 22.8         4 108.0  93       3.85  2.320 18.61      1  1    4    1     1
4 19.2         8 400.0 175       3.08  3.845 17.05      0  0    3    2     0
5 10.4         8 460.0 215       3.00  5.424 17.82      0  0    3    4     0
6 15.5         8 318.0 150       2.76  3.520 16.87      0  0    3    2     0
     am_fac
1    manual
2    manual
3    manual
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
17 14.7         8  440 230       3.23  5.345 17.42      0  0    3    4     0
24 13.3         8  350 245       3.73  3.840 15.41      0  0    3    4     0
5  18.7         8  360 175       3.15  3.440 17.02      0  0    3    2     0
2  21.0         6  160 110       3.90  2.875 17.02      0  1    4    4     1
26 27.3         4   79  66       4.08  1.935 18.90      1  1    4    1     1
      am_fac
17 automatic
24 automatic
5  automatic
2     manual
26    manual
```

```
> # 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
12 16.4         8 275.8 180       3.07  4.070 17.40      0  0    3    3     0
13 17.3         8 275.8 180       3.07  3.730 17.60      0  0    3    3     0
17 14.7         8 440.0 230       3.23  5.345 17.42      0  0    3    4     0
18 32.4         4  78.7  66       4.08  2.200 19.47      1  1    4    1     1
19 30.4         4  75.7  52       4.93  1.615 18.52      1  1    4    2     1
      am_fac
12 automatic
13 automatic
17 automatic
18    manual
19    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 Reshape wide to long

The next two sections discuss how to reshape data from wide to long and from long to wide. "Wide" data are data 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 time_3_score
1 larry            3            0            6
2   moe            6            3            3
3 curly            2            1            1
```

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

```
     id time score
1 larry    1     3
2 larry    2     0
3 larry    3     6
4   moe    1     6
5   moe    2     3
6   moe    3     3
7 curly    1     2
8 curly    2     1
9 curly    3     1
```

Shifting between these two data formats is often useful—necessarily, really—for implementing statistical techniques or representing data with particular visualizations.

### Python

### R

In base R, the `reshape()` function can take data from wide to long or long to wide. The **tidyverse** also provides functions for doing so: `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 base R and **tidyverse**.

For example, 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 convert a data frame from wide to long using `reshape()`, the user must specify the argument `direction = 'long'`. The user also declares an `idvar` argument, which specifies variable(s) that uniquely identify each row and will therefore be repeated in long output (`id` and `sex`); a `varying` argument, which specifies the repeated measurements that are to be lengthened (`wk1`, `wk2`, and `wk3`); as well as `v.names` and `timevar` arguments, which respectively indicate the desired names of (1) the column containing the values in the long data (`weekly_val`) and (2) the column that identifies each value in the long data (`week`).

```
> df_long <- reshape(df_wide,
+                    direction = 'long',
+                    idvar = c('id', 'sex'),
+                    varying = c('wk1', 'wk2', 'wk3'),
+                    v.names = 'weekly_val',
+                    timevar = 'week')
> df_long
      id sex week weekly_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 recreate the output of the `reshape()` function above using `pivot_longer()`, a user would write:

```
> library(tidyverse)
> df_long_PL <- pivot_longer(df_wide,
+                            cols = -c('id', 'sex'),
+                            names_to = 'week',
+                            values_to = 'weekly_val')
> df_long_PL
# A tibble: 9 x 4
     id sex   week  weekly_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 contain multiple different sets of repeated measures that need to be lengthened separately (e.g., two monthly height measurements and two monthly weight measurements in each row) and/or (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`, that looks like this:

```
     animal loves_water ABC_1 ABC_2 XYZ_1 XYZ_2
1   dolphin        TRUE     1     5     6     2
2 porcupine       FALSE     4     5     3     1
3    rabbit       FALSE     2     2     5     2
```

`pivot_longer()` could be used to convert this data frame to a long format in a couple of different ways:

```
> # Long data will contain one column for each measure (ABC and XYZ)
> animals_long_1 <- pivot_longer(animals_wide,
+                                cols = -c('animal', 'loves_water'),
+                                # ".value" serves as placeholder for values that will be extracted
+                                names_to = c('.value', 'measure_num'),
+                                names_pattern = '(.+)_(.)')
> animals_long_1
# A tibble: 6 x 5
  animal    loves_water measure_num   ABC   XYZ
  <chr>     <lgl>       <chr>       <dbl> <dbl>
1 dolphin   TRUE        1               1     6
2 dolphin   TRUE        2               5     2
3 porcupine FALSE       1               4     3
4 porcupine FALSE       2               5     1
5 rabbit    FALSE       1               2     5
6 rabbit    FALSE       2               2     2
> # Long data will contain one column containing values for both measures together (ABC and XYZ)
```

```
> animals_long_2 <- pivot_longer(animals_wide,
+                                cols = -c('animal', 'loves_water'),
+                                names_to = c('measure', 'measure_num'),
+                                names_pattern = '(.+)_(.)',
+                                values_to = 'measure_val')
> animals_long_2
# A tibble: 12 x 5
   animal    loves_water measure measure_num measure_val
   <chr>     <lgl>       <chr>   <chr>              <dbl>
 1 dolphin   TRUE        ABC     1                      1
 2 dolphin   TRUE        ABC     2                      5
 3 dolphin   TRUE        XYZ     1                      6
 4 dolphin   TRUE        XYZ     2                      2
 5 porcupine FALSE       ABC     1                      4
 6 porcupine FALSE       ABC     2                      5
 7 porcupine FALSE       XYZ     1                      3
 8 porcupine FALSE       XYZ     2                      1
 9 rabbit    FALSE       ABC     1                      2
10 rabbit    FALSE       ABC     2                      2
11 rabbit    FALSE       XYZ     1                      5
12 rabbit    FALSE       XYZ     2                      2
```

## 5.4 Reshape long to wide

To take data from long to wide using the `reshape()` function, the user specifies `direction = 'wide'`. The user also provides `idvar`, `v.names`, and `timevar` arguments, which serve the same purpose as they do when `reshape()` is used to lengthen data: `idvar` specifies the variable(s) that uniquely "group" values together that will be displayed in the same row in the wide data; `v.names` indicates the variable that contains the values that are to be widened; and `timevar` refers to the column in the long data that identifies each value's context (its time point, its measurement location, etc.). The contents of the `timevar` column are used to generate the widened column names, as demonstrated below.

```
> df_long
      id sex week weekly_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
> df_wide <- reshape(df_long,
+                    direction = 'wide',
+                    idvar = c('id', 'sex'),
+                    v.names = 'weekly_val',
+                    timevar = 'week',
+                    sep = '...') # the `sep` argument allows a user to specify how the contents
> df_wide
      id sex weekly_val...1 weekly_val...2 weekly_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 recreate the output of the `reshape()` function above using `pivot_longer()`, a user would write:

```
> library(tidyverse)
> df_wide_PW <- pivot_wider(df_long,
+                           id_cols = c('id', 'sex'),
+                           values_from = 'weekly_val',
+                           names_from = 'week')
> df_wide_PW
# A tibble: 3 x 5
     id sex     `1`   `2`   `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 trying to reshape more-complicated long data structures:

```
> # Convert long data with one column for each measure (ABC and XYZ) into wide format
> animals_long_1
# A tibble: 6 x 5
  animal    loves_water measure_num   ABC   XYZ
  <chr>     <lgl>       <chr>       <dbl> <dbl>
1 dolphin   TRUE        1               1     6
2 dolphin   TRUE        2               5     2
3 porcupine FALSE       1               4     3
4 porcupine FALSE       2               5     1
5 rabbit    FALSE       1               2     5
6 rabbit    FALSE       2               2     2
```

```
> animals_wide <- pivot_wider(animals_long_1,
+                             id_cols = c('animal', 'loves_water'),
+                             values_from = c('ABC', 'XYZ'),
+                             names_from = 'measure_num',
+                             names_sep = '_')
> animals_wide
# A tibble: 3 x 6
  animal    loves_water ABC_1 ABC_2 XYZ_1 XYZ_2
  <chr>     <lgl>       <dbl> <dbl> <dbl> <dbl>
1 dolphin   TRUE            1     5     6     2
2 porcupine FALSE           4     5     3     1
3 rabbit    FALSE           2     2     5     2
> # Convert long data with one column containing values for both measures together (AB
> animals_long_2
# A tibble: 12 x 5
   animal    loves_water measure measure_num measure_val
   <chr>     <lgl>       <chr>   <chr>              <dbl>
 1 dolphin   TRUE        ABC     1                      1
 2 dolphin   TRUE        ABC     2                      5
 3 dolphin   TRUE        XYZ     1                      6
 4 dolphin   TRUE        XYZ     2                      2
 5 porcupine FALSE       ABC     1                      4
 6 porcupine FALSE       ABC     2                      5
 7 porcupine FALSE       XYZ     1                      3
 8 porcupine FALSE       XYZ     2                      1
 9 rabbit    FALSE       ABC     1                      2
10 rabbit    FALSE       ABC     2                      2
11 rabbit    FALSE       XYZ     1                      5
12 rabbit    FALSE       XYZ     2                      2
> animals_wide <- pivot_wider(animals_long_2,
+                             id_cols = c('animal', 'loves_water'),
+                             values_from = 'measure_val',
+                             names_from = c('measure', 'measure_num'),
+                             names_sep = '_')
> animals_wide
# A tibble: 3 x 6
  animal    loves_water ABC_1 ABC_2 XYZ_1 XYZ_2
  <chr>     <lgl>       <dbl> <dbl> <dbl> <dbl>
1 dolphin   TRUE            1     5     6     2
2 porcupine FALSE           4     5     3     1
3 rabbit    FALSE           2     2     5     2
```

**Python**

**R**

## 5.5 Merge/Join

### 5.5.1 Left Join

**Python**

**R**

### 5.5.2 Right Join

**Python**

**R**

### 5.5.3 Inner Join

**Python**

**R**

### 5.5.4 Outer Join

**Python**

**R**