Python and R

Clay Ford, Jacob Goldstein-Greenwood, Samantha Lomuscio

2021-09-29

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 and Libraries

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

Libraries and 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.

Libraries and packages often depend on other libraries and packages. These are known as "dependencies". Sometimes libraries/packages are updated to accommodate changes to other libraries/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.

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

Occasionally when installing or updating a package 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. If you try to install a package that has not been compiled then you'll get asked if you want to compile it yourself. You can usually just answer No. If you answer 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 typically already have the necessary build tools.

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

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
> 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
```

Chapter 3

Importing Data

This chapter reviews importing external data into Python and R, including CSV, txt, and other structured data files.

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/dat.csv')
```

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/dat.csv")
>
```

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.

```
> library(readr)
> d <- read_csv("data/dat.csv")
>
> library(data.table)
> d <- fread("data/dat.csv")
```

3.2 XLS/XLSX (Excel)

Python

R

3.3 Text

Python

R

3.4 JSON

Python

R

3.5 XML

Python

R

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 22.8         4 140.8  95       3.92  3.150 22.90      1  0    4    2     0
2 21.4         6 258.0 110       3.08  3.215 19.44      1  0    3    1     0
3 17.3         8 275.8 180       3.07  3.730 17.60      0  0    3    3     0
4 15.0         8 301.0 335       3.54  3.570 14.60      0  1    5    8     1
5 19.2         8 400.0 175       3.08  3.845 17.05      0  0    3    2     0
     am_fac
1 automatic
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 24.4         4 146.7  62       3.69  3.190 20.00      1  0    4    2     0
2 19.2         6 167.6 123       3.92  3.440 18.30      1  0    4    4     0
3 10.4         8 472.0 205       2.93  5.250 17.98      0  0    3    4     0
4 26.0         4 120.3  91       4.43  2.140 16.70      0  1    5    2     1
5 30.4         4  95.1 113       3.77  1.513 16.90      1  1    5    2     1
6 18.1         6 225.0 105       2.76  3.460 20.22      1  0    3    1     0
     am_fac
1 automatic
2 automatic
3 automatic
4    manual
5    manual
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         8 440.0 230       3.23  5.345 17.42      0  0    3    4     0
7  14.3         8 360.0 245       3.21  3.570 15.84      0  0    3    4     0
20 33.9         4  71.1  65       4.22  1.835 19.90      1  1    4    1     1
21 21.5         4 120.1  97       3.70  2.465 20.01      1  0    3    1     0
28 30.4         4  95.1 113       3.77  1.513 16.90      1  1    5    2     1
      am_fac
17 automatic
7  automatic
20    manual
21 automatic
28    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
1  21.0         6 160.0 110       3.90  2.620 16.46      0  1    4    4     1
10 19.2         6 167.6 123       3.92  3.440 18.30      1  0    4    4     0
19 30.4         4  75.7  52       4.93  1.615 18.52      1  1    4    2     1
20 33.9         4  71.1  65       4.22  1.835 19.90      1  1    4    1     1
24 13.3         8 350.0 245       3.73  3.840 15.41      0  0    3    4     0
29 15.8         8 351.0 264       4.22  3.170 14.50      0  1    5    4     1
30 19.7         6 145.0 175       3.62  2.770 15.50      0  1    5    6     1
      am_fac
1     manual
10 automatic
19    manual
20    manual
24 automatic
29    manual
30    manual
```

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

```
> # always get the same random sample
> set.seed(123)
> i <- runif(nrow(mtcars))
> 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`.

```
> 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 is 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

Python

R

5.4 Reshape long to wide

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