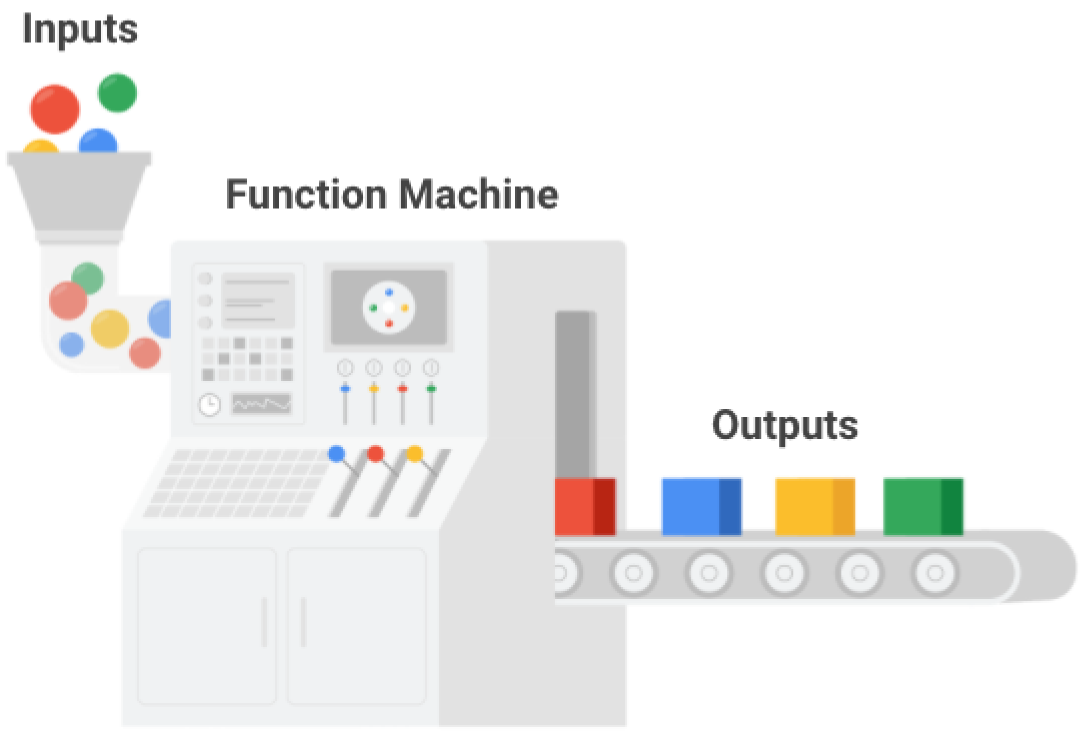
Data Import

In this reading, you will learn the basics of importing data into R. It comes with built-in datasets that are great tools for learning how to use R and practice analyzing data. You will explore the data() function and learn how to load sample datasets into RStudio. Then you will go through how to use two tidyverse packages—readr and readxl—to import files from other sources into R. You will learn how to use readr to read a .csv file, and how to use readxl to read a .xlsx file.

The data() function

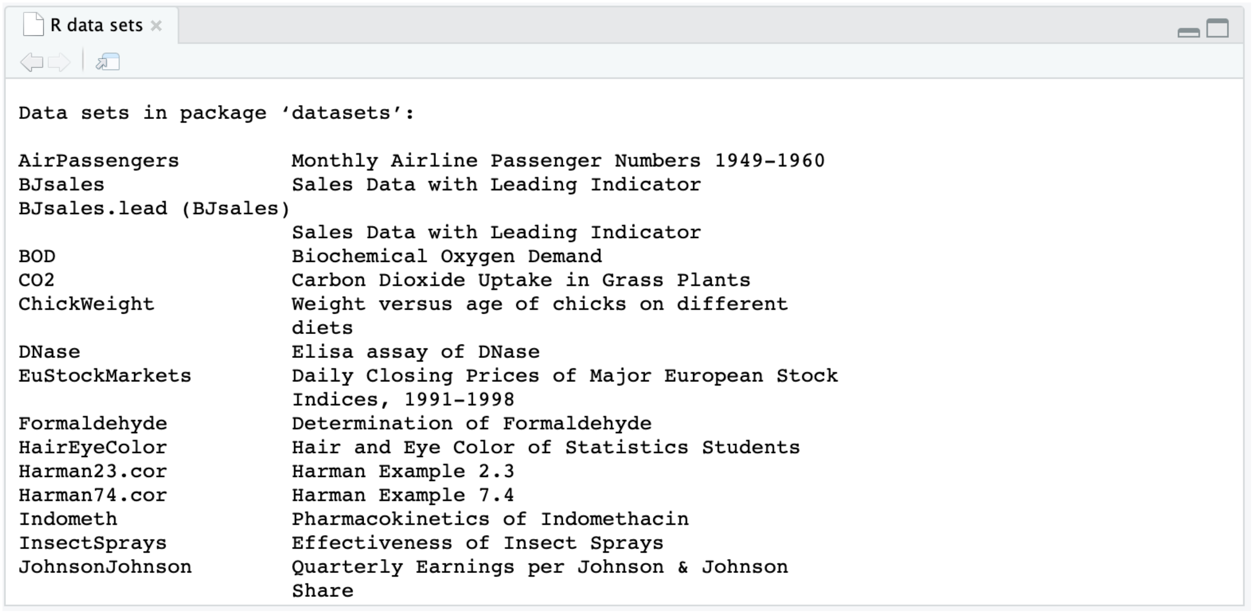


The default installation of R comes with a number of preloaded datasets that you can practice with. This is a great way to develop your R skills and learn about some important data analysis functions. Plus, many online resources and tutorials use these sample datasets to teach coding concepts in R.

You can use the **data()** function to load these datasets in R. If you run the data function without an argument, R will display a list of the available datasets.

```
data()
```

This includes the list of preloaded datasets from the *datasets* package.



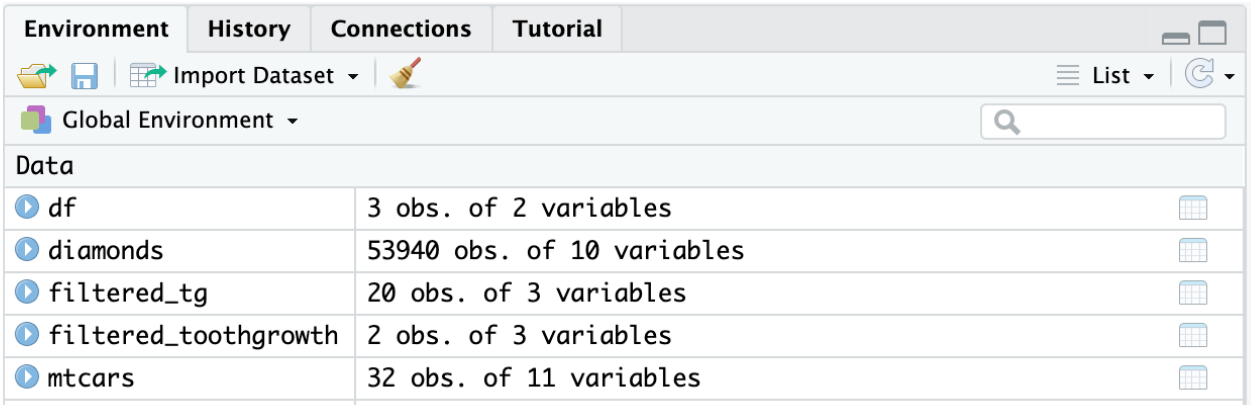
```
Data sets in package 'datasets':

AirPassengers          Monthly Airline Passenger Numbers 1949-1960
BJsales                Sales Data with Leading Indicator
BJsales.lead (BJsales)
                       Sales Data with Leading Indicator
BOD                    Biochemical Oxygen Demand
CO2                    Carbon Dioxide Uptake in Grass Plants
ChickWeight            Weight versus age of chicks on different
                       diets
DNase                  Elisa assay of DNase
EuStockMarkets         Daily Closing Prices of Major European Stock
                       Indices, 1991-1998
Formaldehyde           Determination of Formaldehyde
HairEyeColor           Hair and Eye Color of Statistics Students
Harman23.cor           Harman Example 2.3
Harman74.cor           Harman Example 7.4
Indometh               Pharmacokinetics of Indomethacin
InsectSprays           Effectiveness of Insect Sprays
JohnsonJohnson         Quarterly Earnings per Johnson & Johnson
                       Share
```

If you want to load a specific dataset, just enter its name in the parentheses of the data() function. For example, let's load the *mtcars* dataset, which has information about cars that have been featured in past issues of *Motor Trend* magazine.

```
data(mtcars)
```

When you run the function, R will load the dataset. The dataset will also appear in the Environment pane of your RStudio. The Environment pane displays the names of the data objects, such as data frames and variables, that you have in your current workspace. In this image, *mtcars* appears in the fifth row of the pane. R tells us that it contains 32 observations and 11 variables.



Data

df	3 obs. of 2 variables
diamonds	53940 obs. of 10 variables
filtered_tg	20 obs. of 3 variables
filtered_toothgrowth	2 obs. of 3 variables
mtcars	32 obs. of 11 variables

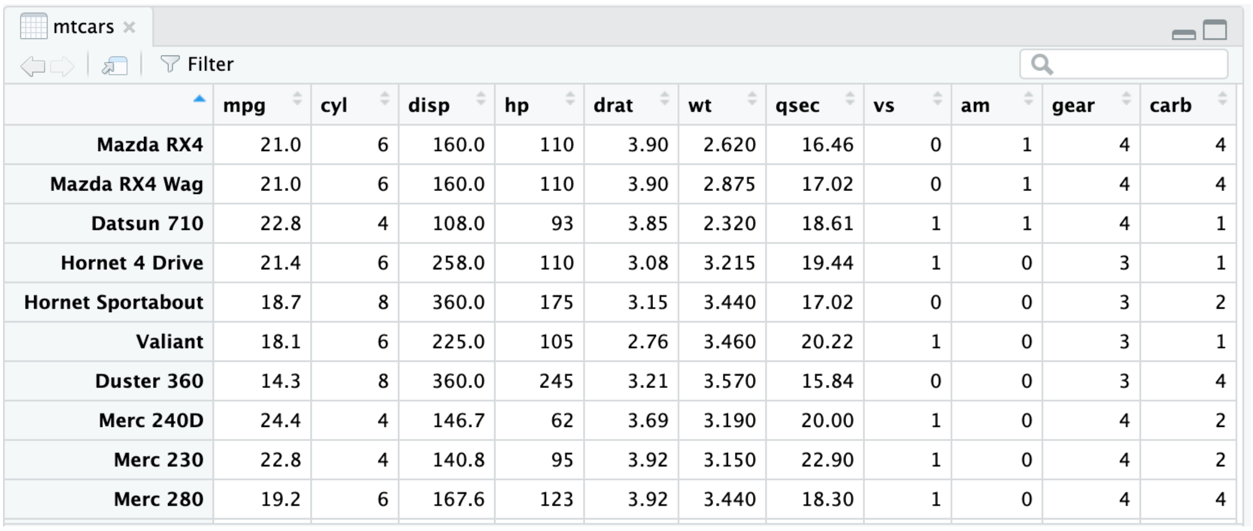
Now that the dataset is loaded, you can get a preview of it in the R console pane. Just type its name...

```
mtcars
```

...and then press ctrl (or cmnd) and enter.

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

You can also display the dataset by clicking directly on the name of the dataset in the Environment pane. So, if you click on **mtcars** in the Environment pane, R automatically runs the View() function and displays the dataset in the RStudio data viewer.



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

Try experimenting with other datasets in the list if you want some more practice.

The readr package

In addition to using R’s built-in datasets, it is also helpful to import data from other sources to use for practice or analysis. The readr package in R is a great tool for reading rectangular data. Rectangular data is data that fits nicely inside a rectangle of rows and columns, with each column referring to a single variable and each row referring to a single observation.

Here are some examples of file types that store rectangular data:

- **.csv (comma separated values)**: a .csv file is a plain text file that contains a list of data. They mostly use commas to separate (or delimit) data, but sometimes they use other characters, like semicolons.
- **.tsv (tab separated values)**: a .tsv file stores a data table in which the columns of data are separated by tabs. For example, a database table or spreadsheet data.
- **.fwf (fixed width files)**: a .fwf file has a specific format that allows for the saving of textual data in an organized fashion.
- **.log:** a .log file is a computer-generated file that records events from operating systems and other software programs.

Base R also has functions for reading files, but the equivalent functions in readr are typically *much* faster. They also produce tibbles, which are easy to use and read.

The readr package is part of the core tidyverse. So, if you've already installed the tidyverse, you have what you need to start working with readr. If not, you can install the tidyverse now.

readr functions

The goal of readr is to provide a fast and friendly way to read rectangular data. readr supports several read_ functions. Each function refers to a specific file format.

- read_csv()
  : comma-separated values (.csv) files
- read_tsv()
  : tab-separated values files
- read_delim()
  : general delimited files
- read_fwf()
  : fixed-width files
- read_table()
  : tabular files where columns are separated by white-space
- read_log()
  : web log files

These functions all have similar syntax, so once you learn how to use one of them, you can apply your knowledge to the others. This reading will focus on the read_csv() function, since .csv files are one of the most common forms of data storage and you will work with them frequently.

In most cases, these functions will work automatically: you supply the path to a file, run the function, and you get a tibble that displays the data in the file. Behind the scenes, readr parses

the overall file and specifies how each column should be converted from a character vector to the most appropriate data type.

Reading a .csv file with readr

The readr package comes with some sample files from built-in datasets that you can use for example code. To list the sample files, you can run the readr_example() function with no arguments.

```
readr_example()
```

```
[1] "challenge.csv"     "epa78.txt"         "example.log"
[4] "fwf-sample.txt"    "massey-rating.txt" "mtcars.csv"
[7] "mtcars.csv.bz2"    "mtcars.csv.zip"
```

The “mtcars.csv” file refers to the *mtcars* dataset that was mentioned earlier. Let’s use the **read_csv()** function to read the “mtcars.csv” file, as an example. In the parentheses, you need to supply the path to the file. In this case, it’s “readr_example(“mtcars.csv”).

```
read_csv(readr_example("mtcars.csv"))
```

When you run the function, R prints out a column specification that gives the name and type of each column.

```
── Column specification ────────────────────────────────────────────
cols(
  mpg = col_double(),
  cyl = col_double(),
  disp = col_double(),
  hp = col_double(),
  drat = col_double(),
  wt = col_double(),
  qsec = col_double(),
  vs = col_double(),
  am = col_double(),
  gear = col_double(),
  carb = col_double()
)
```

R also prints a tibble.

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# … with 22 more rows
```

Optional: the readxl package

To import spreadsheet data into R, you can use the readxl package. The readxl package makes it easy to transfer data from Excel into R. Readxl supports both the legacy .xls file format and the modern xml-based .xlsx file format.

The readxl package is part of the tidyverse but is not a *core* tidyverse package, so you need to load readxl in R by using the library() function.

```
library(readxl)
```

Reading an .xlsx file with readxl

Like the readr package, readxl comes with some sample files from built-in datasets that you can use for practice. You can run the code `readxl_example()` to see the list.

You can use the **read_excel()** function to read a spreadsheet file just like you used read_csv() function to read a .csv file. The code for reading the example file `“type-me.xlsx”` includes the path to the file in the parentheses of the function.

```
read_excel(readxl_example("type-me.xlsx"))
```

You can use the excel_sheets()

Opens in a new tab
 function to list the names of the individual sheets.

excel_sheets(readxl_example("type-me.xlsx"))

[1] "logical_coercion" "numeric_coercion" "date_coercion" "text_coercion"

You can also specify a sheet by name or number. Just type “sheet =” followed by the name or number of the sheet. For example, you can use the sheet named “numeric_coercion” from the list above.

```
read_excel(readxl_example("type-me.xlsx"), sheet = "numeric_coercion")
```

When you run the function, R returns a tibble of the sheet.

```
# A tibble: 7 x 2
  `maybe numeric?` explanation
  <chr>            <chr>
1 NA               "empty"
2 TRUE             "boolean true"
3 FALSE            "boolean false"
4 40534            "datetime"
5 123456           "the string \"123456\""
6 123456           "the number 123456"
7 cabbage          "\"cabbage\""
```

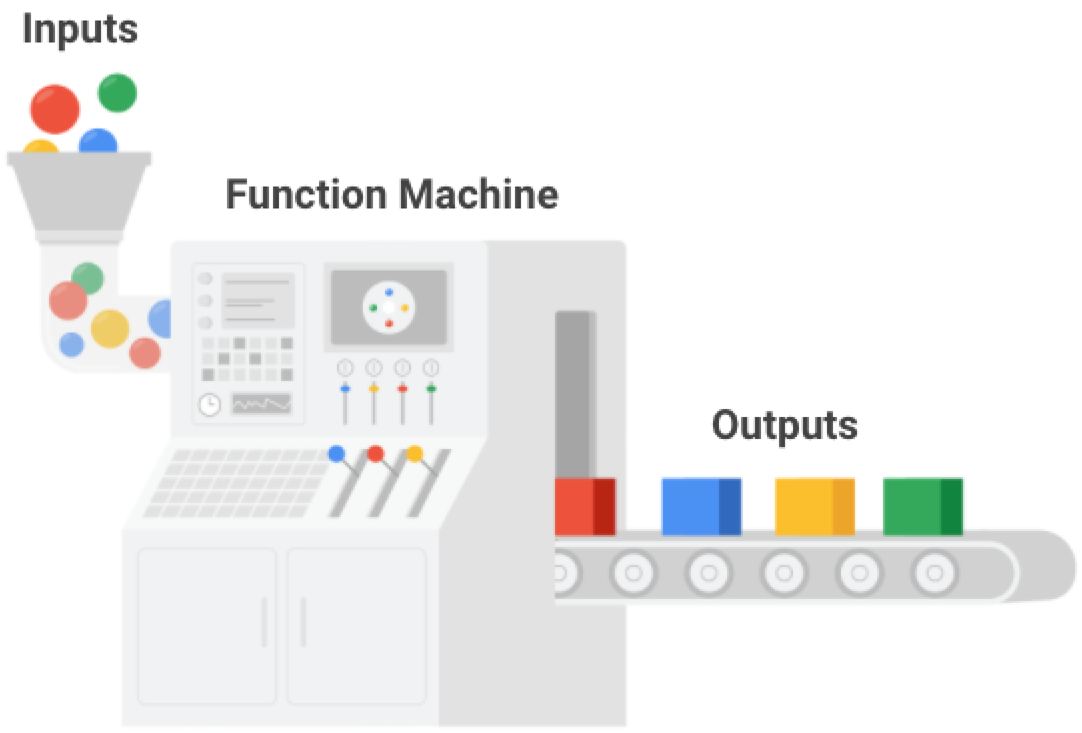
Additional resources

- If you want to learn how to use readr functions to work with more complex files, check out the Data Import chapter
 Opens in a new tab
 of the R for Data Science book. It explores some of the common issues you might encounter when reading files, and how to use readr to manage those issues.
- The readxl
 Opens in a new tab
 entry in the tidyverse documentation gives a good overview of the basic functions in readxl, provides a detailed explanation of how the package operates and the coding concepts behind them, and offers links to other useful resources.
- The R "datasets" package contains lots of useful preloaded datasets. Check out The R Datasets Package
 Opens in a new tab
 for a list. The list includes links to detailed descriptions of each dataset.

Data Import

In this reading, you will learn the basics of importing data into R. It comes with built-in datasets that are great tools for learning how to use R and practice analyzing data. You will explore the data() function and learn how to load sample datasets into RStudio. Then you will go through how to use two tidyverse packages—readr and readxl—to import files from other sources into R. You will learn how to use readr to read a .csv file, and how to use readxl to read a .xlsx file.

The data() function

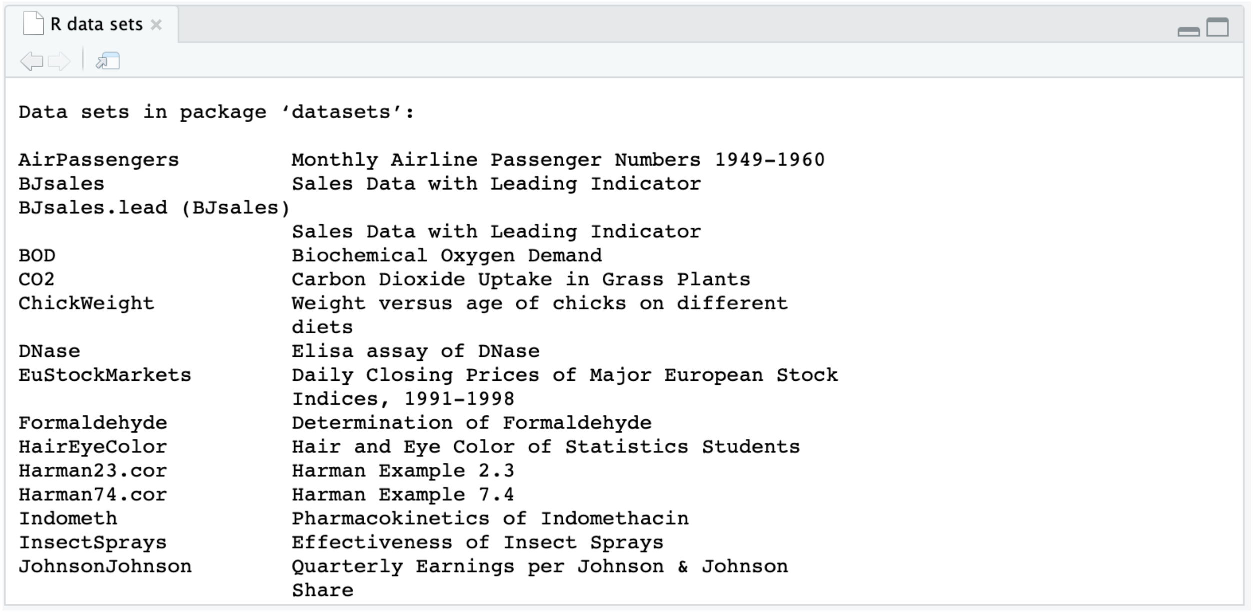


The default installation of R comes with a number of preloaded datasets that you can practice with. This is a great way to develop your R skills and learn about some important data analysis functions. Plus, many online resources and tutorials use these sample datasets to teach coding concepts in R.

You can use the **data()** function to load these datasets in R. If you run the data function without an argument, R will display a list of the available datasets.

```
data()
```

This includes the list of preloaded datasets from the *datasets* package.



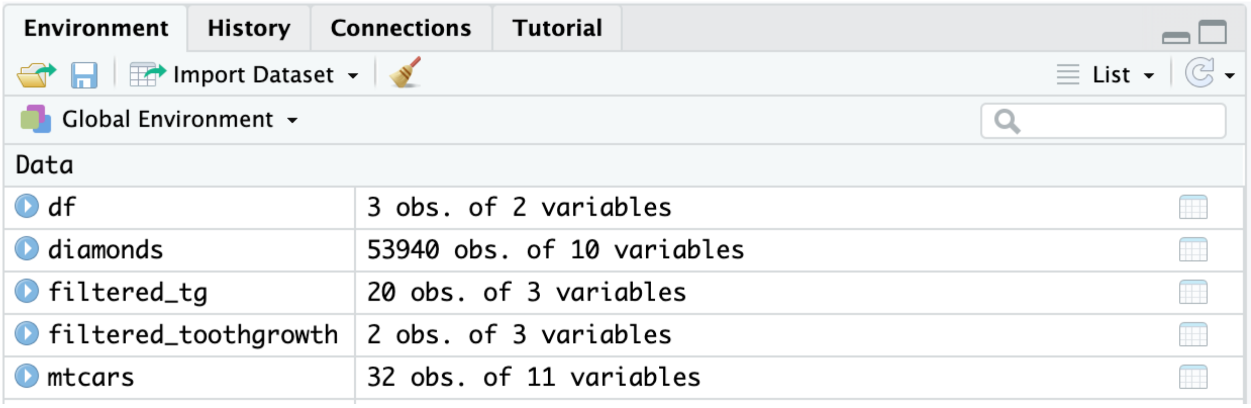
```
Data sets in package 'datasets':

AirPassengers          Monthly Airline Passenger Numbers 1949-1960
BJsales                Sales Data with Leading Indicator
BJsales.lead (BJsales)
                       Sales Data with Leading Indicator
BOD                    Biochemical Oxygen Demand
CO2                    Carbon Dioxide Uptake in Grass Plants
ChickWeight            Weight versus age of chicks on different
                       diets
DNase                  Elisa assay of DNase
EuStockMarkets         Daily Closing Prices of Major European Stock
                       Indices, 1991-1998
Formaldehyde           Determination of Formaldehyde
HairEyeColor           Hair and Eye Color of Statistics Students
Harman23.cor           Harman Example 2.3
Harman74.cor           Harman Example 7.4
Indometh               Pharmacokinetics of Indomethacin
InsectSprays           Effectiveness of Insect Sprays
JohnsonJohnson         Quarterly Earnings per Johnson & Johnson
                       Share
```

If you want to load a specific dataset, just enter its name in the parentheses of the data() function. For example, let's load the *mtcars* dataset, which has information about cars that have been featured in past issues of *Motor Trend* magazine.

```
data(mtcars)
```

When you run the function, R will load the dataset. The dataset will also appear in the Environment pane of your RStudio. The Environment pane displays the names of the data objects, such as data frames and variables, that you have in your current workspace. In this image, *mtcars* appears in the fifth row of the pane. R tells us that it contains 32 observations and 11 variables.



Data	
df	3 obs. of 2 variables
diamonds	53940 obs. of 10 variables
filtered_tg	20 obs. of 3 variables
filtered_toothgrowth	2 obs. of 3 variables
mtcars	32 obs. of 11 variables

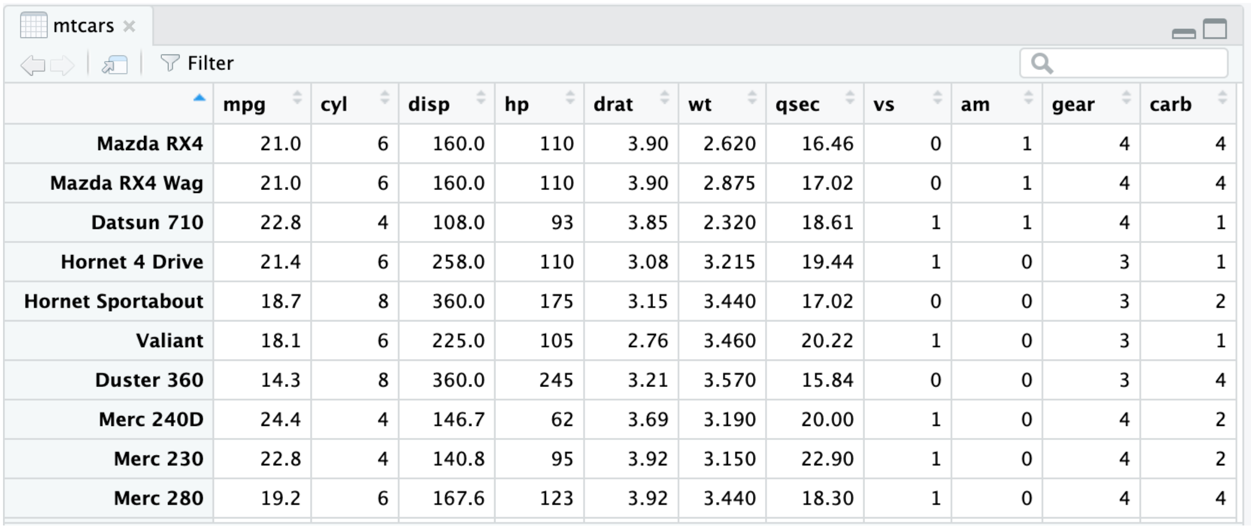
Now that the dataset is loaded, you can get a preview of it in the R console pane. Just type its name...

```
mtcars
```

...and then press ctrl (or cmnd) and enter.

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

You can also display the dataset by clicking directly on the name of the dataset in the Environment pane. So, if you click on **mtcars** in the Environment pane, R automatically runs the View() function and displays the dataset in the RStudio data viewer.



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

Try experimenting with other datasets in the list if you want some more practice.

The readr package

In addition to using R's built-in datasets, it is also helpful to import data from other sources to use for practice or analysis. The readr package in R is a great tool for reading rectangular data. Rectangular data is data that fits nicely inside a rectangle of rows and columns, with each column referring to a single variable and each row referring to a single observation.

Here are some examples of file types that store rectangular data:

- **.csv (comma separated values)**: a .csv file is a plain text file that contains a list of data. They mostly use commas to separate (or delimit) data, but sometimes they use other characters, like semicolons.
- **.tsv (tab separated values)**: a .tsv file stores a data table in which the columns of data are separated by tabs. For example, a database table or spreadsheet data.
- **.fwf (fixed width files)**: a .fwf file has a specific format that allows for the saving of textual data in an organized fashion.
- **.log:** a .log file is a computer-generated file that records events from operating systems and other software programs.

Base R also has functions for reading files, but the equivalent functions in readr are typically *much* faster. They also produce tibbles, which are easy to use and read.

The readr package is part of the core tidyverse. So, if you've already installed the tidyverse, you have what you need to start working with readr. If not, you can install the tidyverse now.

readr functions

The goal of readr is to provide a fast and friendly way to read rectangular data. readr supports several read_ functions. Each function refers to a specific file format.

- `read_csv()`
  : comma-separated values (.csv) files
- `read_tsv()`
  : tab-separated values files
- `read_delim()`
  : general delimited files
- `read_fwf()`
  : fixed-width files
- `read_table()`
  : tabular files where columns are separated by white-space
- `read_log()`
  : web log files

These functions all have similar syntax, so once you learn how to use one of them, you can apply your knowledge to the others. This reading will focus on the read_csv() function, since .csv files are one of the most common forms of data storage and you will work with them frequently.

In most cases, these functions will work automatically: you supply the path to a file, run the function, and you get a tibble that displays the data in the file. Behind the scenes, readr parses

the overall file and specifies how each column should be converted from a character vector to the most appropriate data type.

Reading a .csv file with readr

The readr package comes with some sample files from built-in datasets that you can use for example code. To list the sample files, you can run the readr_example() function with no arguments.

```
readr_example()
```

```
[1] "challenge.csv"     "epa78.txt"         "example.log"
[4] "fwf-sample.txt"    "massey-rating.txt" "mtcars.csv"
[7] "mtcars.csv.bz2"    "mtcars.csv.zip"
```

The “mtcars.csv” file refers to the *mtcars* dataset that was mentioned earlier. Let’s use the **read_csv()** function to read the “mtcars.csv” file, as an example. In the parentheses, you need to supply the path to the file. In this case, it’s “readr_example(“mtcars.csv”).

```
read_csv(readr_example("mtcars.csv"))
```

When you run the function, R prints out a column specification that gives the name and type of each column.

```
── Column specification ────────────────────────────────────────
cols(
  mpg = col_double(),
  cyl = col_double(),
  disp = col_double(),
  hp = col_double(),
  drat = col_double(),
  wt = col_double(),
  qsec = col_double(),
  vs = col_double(),
  am = col_double(),
  gear = col_double(),
  carb = col_double()
)
```

R also prints a tibble.

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# … with 22 more rows
```

Optional: the readxl package

To import spreadsheet data into R, you can use the readxl package. The readxl package makes it easy to transfer data from Excel into R. Readxl supports both the legacy .xls file format and the modern xml-based .xlsx file format.

The readxl package is part of the tidyverse but is not a *core* tidyverse package, so you need to load readxl in R by using the library() function.

```
library(readxl)
```

Reading an .xlsx file with readxl

Like the readr package, readxl comes with some sample files from built-in datasets that you can use for practice. You can run the code `readxl_example()` to see the list.

You can use the **read_excel()** function to read a spreadsheet file just like you used read_csv() function to read a .csv file. The code for reading the example file `“type-me.xlsx”` includes the path to the file in the parentheses of the function.

```
read_excel(readxl_example("type-me.xlsx"))
```

You can use the excel_sheets()

Opens in a new tab
function to list the names of the individual sheets.

excel_sheets(readxl_example("type-me.xlsx"))

[1] "logical_coercion" "numeric_coercion" "date_coercion" "text_coercion"

You can also specify a sheet by name or number. Just type "sheet =" followed by the name or number of the sheet. For example, you can use the sheet named "numeric_coercion" from the list above.

```
read_excel(readxl_example("type-me.xlsx"), sheet = "numeric_coercion")
```

When you run the function, R returns a tibble of the sheet.

```
# A tibble: 7 x 2
  `maybe numeric?` explanation
  <chr>            <chr>
1 NA               "empty"
2 TRUE             "boolean true"
3 FALSE            "boolean false"
4 40534            "datetime"
5 123456           "the string \"123456\""
6 123456           "the number 123456"
7 cabbage          "\"cabbage\""
```

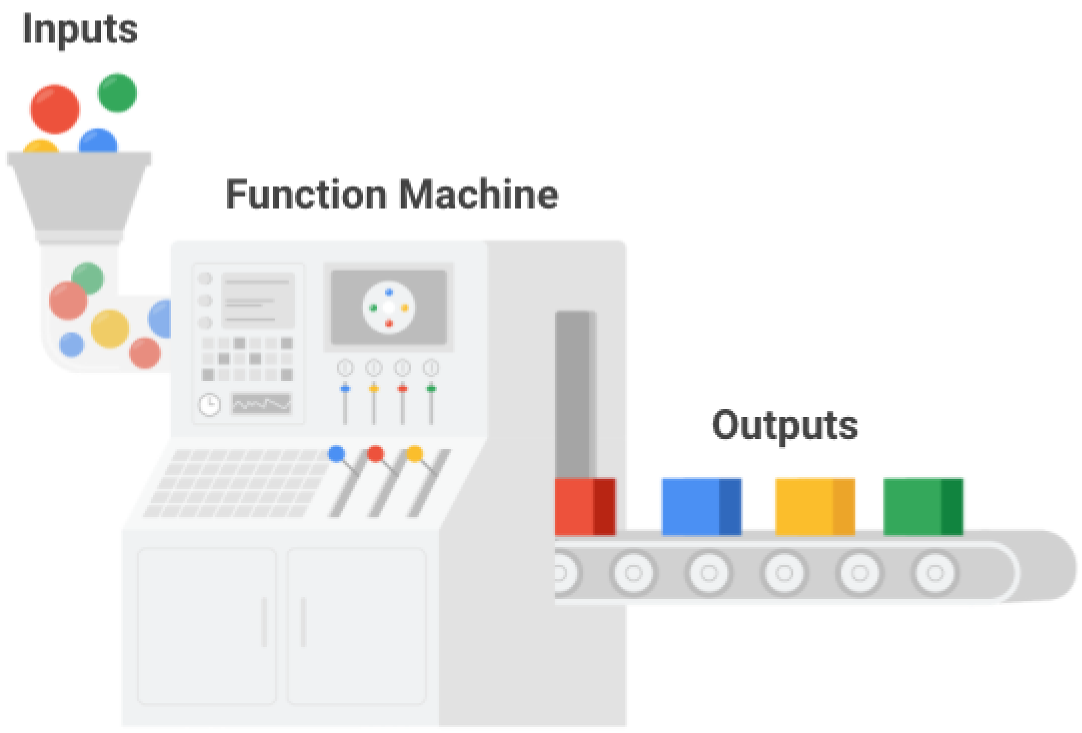
Additional resources

- If you want to learn how to use readr functions to work with more complex files, check out the Data Import chapter
  Opens in a new tab
  of the R for Data Science book. It explores some of the common issues you might encounter when reading files, and how to use readr to manage those issues.
- The readxl
  Opens in a new tab
  entry in the tidyverse documentation gives a good overview of the basic functions in readxl, provides a detailed explanation of how the package operates and the coding concepts behind them, and offers links to other useful resources.
- The R "datasets" package contains lots of useful preloaded datasets. Check out The R Datasets Package
  Opens in a new tab
  for a list. The list includes links to detailed descriptions of each dataset.

Data Import

In this reading, you will learn the basics of importing data into R. It comes with built-in datasets that are great tools for learning how to use R and practice analyzing data. You will explore the data() function and learn how to load sample datasets into RStudio. Then you will go through how to use two tidyverse packages—readr and readxl—to import files from other sources into R. You will learn how to use readr to read a .csv file, and how to use readxl to read a .xlsx file.

The data() function

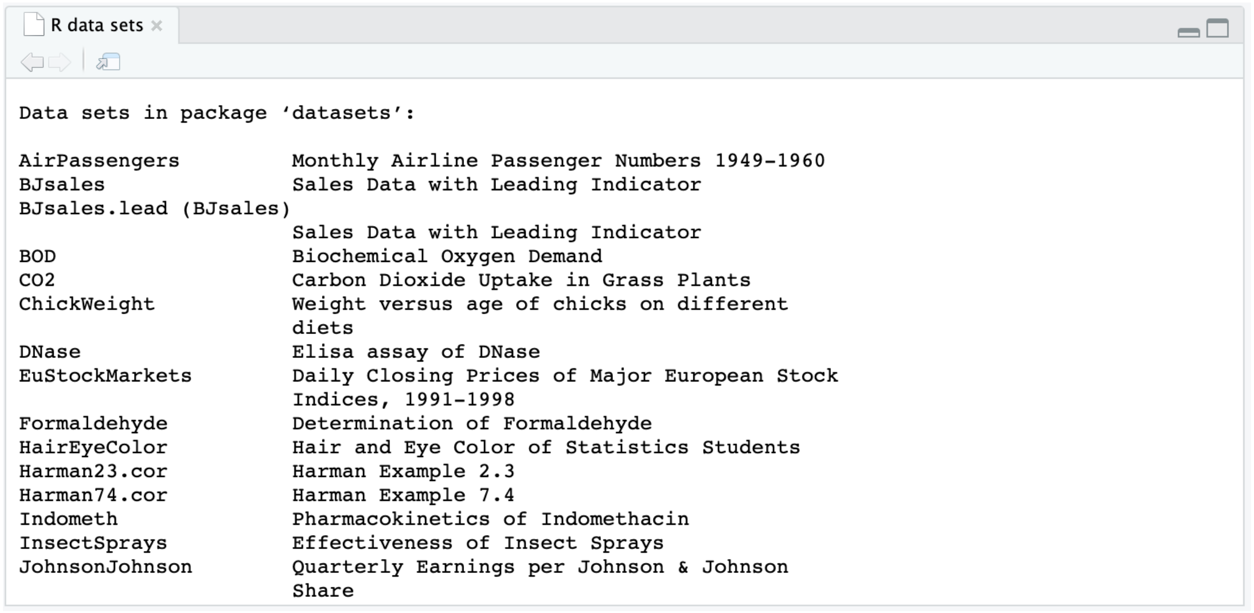


The default installation of R comes with a number of preloaded datasets that you can practice with. This is a great way to develop your R skills and learn about some important data analysis functions. Plus, many online resources and tutorials use these sample datasets to teach coding concepts in R.

You can use the **data()** function to load these datasets in R. If you run the data function without an argument, R will display a list of the available datasets.

```
data()
```

This includes the list of preloaded datasets from the *datasets* package.



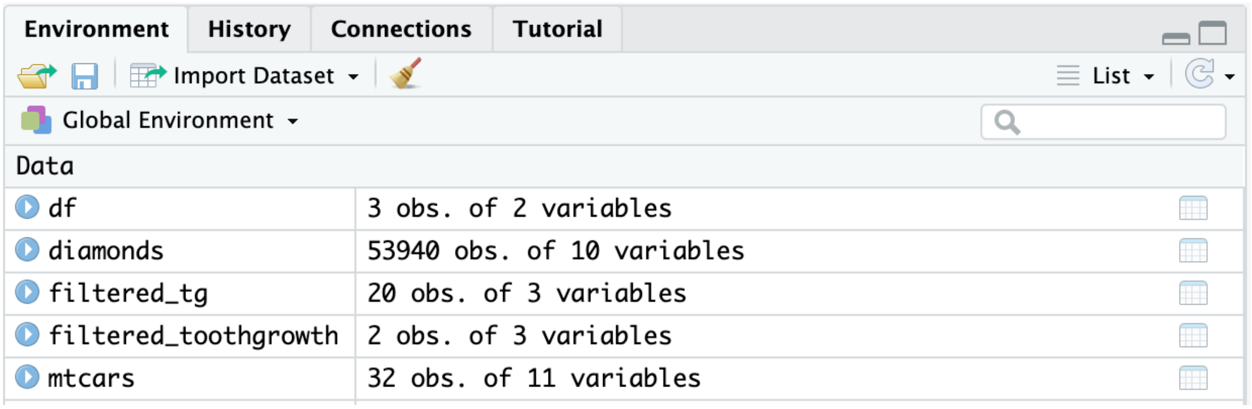
```
Data sets in package ‘datasets’:

AirPassengers          Monthly Airline Passenger Numbers 1949-1960
BJsales                Sales Data with Leading Indicator
BJsales.lead (BJsales)
                       Sales Data with Leading Indicator
BOD                    Biochemical Oxygen Demand
CO2                    Carbon Dioxide Uptake in Grass Plants
ChickWeight            Weight versus age of chicks on different
                       diets
DNase                  Elisa assay of DNase
EuStockMarkets         Daily Closing Prices of Major European Stock
                       Indices, 1991-1998
Formaldehyde           Determination of Formaldehyde
HairEyeColor           Hair and Eye Color of Statistics Students
Harman23.cor           Harman Example 2.3
Harman74.cor           Harman Example 7.4
Indometh               Pharmacokinetics of Indomethacin
InsectSprays           Effectiveness of Insect Sprays
JohnsonJohnson         Quarterly Earnings per Johnson & Johnson
                       Share
```

If you want to load a specific dataset, just enter its name in the parentheses of the data() function. For example, let’s load the *mtcars* dataset, which has information about cars that have been featured in past issues of *Motor Trend* magazine.

```
data(mtcars)
```

When you run the function, R will load the dataset. The dataset will also appear in the Environment pane of your RStudio. The Environment pane displays the names of the data objects, such as data frames and variables, that you have in your current workspace. In this image, *mtcars* appears in the fifth row of the pane. R tells us that it contains 32 observations and 11 variables.



Data	
df	3 obs. of 2 variables
diamonds	53940 obs. of 10 variables
filtered_tg	20 obs. of 3 variables
filtered_toothgrowth	2 obs. of 3 variables
mtcars	32 obs. of 11 variables

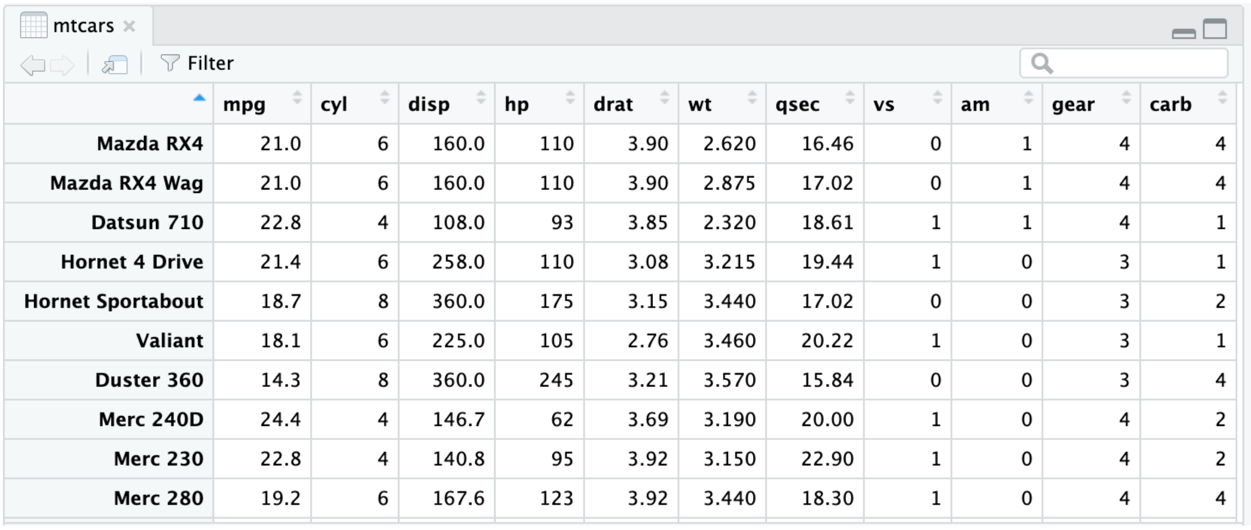
Now that the dataset is loaded, you can get a preview of it in the R console pane. Just type its name...

```
mtcars
```

...and then press ctrl (or cmnd) and enter.

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

You can also display the dataset by clicking directly on the name of the dataset in the Environment pane. So, if you click on **mtcars** in the Environment pane, R automatically runs the View() function and displays the dataset in the RStudio data viewer.



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

Try experimenting with other datasets in the list if you want some more practice.

The readr package

In addition to using R's built-in datasets, it is also helpful to import data from other sources to use for practice or analysis. The readr package in R is a great tool for reading rectangular data. Rectangular data is data that fits nicely inside a rectangle of rows and columns, with each column referring to a single variable and each row referring to a single observation.

Here are some examples of file types that store rectangular data:

- **.csv (comma separated values)**: a .csv file is a plain text file that contains a list of data. They mostly use commas to separate (or delimit) data, but sometimes they use other characters, like semicolons.
- **.tsv (tab separated values)**: a .tsv file stores a data table in which the columns of data are separated by tabs. For example, a database table or spreadsheet data.
- **.fwf (fixed width files)**: a .fwf file has a specific format that allows for the saving of textual data in an organized fashion.
- **.log:** a .log file is a computer-generated file that records events from operating systems and other software programs.

Base R also has functions for reading files, but the equivalent functions in readr are typically *much* faster. They also produce tibbles, which are easy to use and read.

The readr package is part of the core tidyverse. So, if you've already installed the tidyverse, you have what you need to start working with readr. If not, you can install the tidyverse now.

readr functions

The goal of readr is to provide a fast and friendly way to read rectangular data. readr supports several read_ functions. Each function refers to a specific file format.

- `read_csv()`
  : comma-separated values (.csv) files
- `read_tsv()`
  : tab-separated values files
- `read_delim()`
  : general delimited files
- `read_fwf()`
  : fixed-width files
- `read_table()`
  : tabular files where columns are separated by white-space
- `read_log()`
  : web log files

These functions all have similar syntax, so once you learn how to use one of them, you can apply your knowledge to the others. This reading will focus on the read_csv() function, since .csv files are one of the most common forms of data storage and you will work with them frequently.

In most cases, these functions will work automatically: you supply the path to a file, run the function, and you get a tibble that displays the data in the file. Behind the scenes, readr parses

the overall file and specifies how each column should be converted from a character vector to the most appropriate data type.

Reading a .csv file with readr

The readr package comes with some sample files from built-in datasets that you can use for example code. To list the sample files, you can run the readr_example() function with no arguments.

```
readr_example()
```

```
[1] "challenge.csv"     "epa78.txt"         "example.log"
[4] "fwf-sample.txt"    "massey-rating.txt" "mtcars.csv"
[7] "mtcars.csv.bz2"    "mtcars.csv.zip"
```

The “mtcars.csv” file refers to the *mtcars* dataset that was mentioned earlier. Let’s use the **read_csv()** function to read the “mtcars.csv” file, as an example. In the parentheses, you need to supply the path to the file. In this case, it’s “readr_example(“mtcars.csv”).

```
read_csv(readr_example("mtcars.csv"))
```

When you run the function, R prints out a column specification that gives the name and type of each column.

```
── Column specification ──────────────────────────────────────────────
cols(
  mpg = col_double(),
  cyl = col_double(),
  disp = col_double(),
  hp = col_double(),
  drat = col_double(),
  wt = col_double(),
  qsec = col_double(),
  vs = col_double(),
  am = col_double(),
  gear = col_double(),
  carb = col_double()
)
```

R also prints a tibble.

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# … with 22 more rows
```

Optional: the readxl package

To import spreadsheet data into R, you can use the readxl package. The readxl package makes it easy to transfer data from Excel into R. Readxl supports both the legacy .xls file format and the modern xml-based .xlsx file format.

The readxl package is part of the tidyverse but is not a *core* tidyverse package, so you need to load readxl in R by using the library() function.

`library(readxl)`

Reading an .xlsx file with readxl

Like the readr package, readxl comes with some sample files from built-in datasets that you can use for practice. You can run the code `readxl_example()` to see the list.

You can use the **read_excel()** function to read a spreadsheet file just like you used read_csv() function to read a .csv file. The code for reading the example file `“type-me.xlsx”` includes the path to the file in the parentheses of the function.

`read_excel(readxl_example("type-me.xlsx"))`

You can use the [excel_sheets()](#)

Opens in a new tab

function to list the names of the individual sheets.

```
excel_sheets(readxl_example("type-me.xlsx"))
```

```
[1] "logical_coercion" "numeric_coercion" "date_coercion" "text_coercion"
```

You can also specify a sheet by name or number. Just type "sheet =" followed by the name or number of the sheet. For example, you can use the sheet named "numeric_coercion" from the list above.

```
read_excel(readxl_example("type-me.xlsx"), sheet = "numeric_coercion")
```

When you run the function, R returns a tibble of the sheet.

```
# A tibble: 7 x 2
  `maybe numeric?` explanation
  <chr>            <chr>
1 NA               "empty"
2 TRUE             "boolean true"
3 FALSE            "boolean false"
4 40534            "datetime"
5 123456           "the string \"123456\""
6 123456           "the number 123456"
7 cabbage          "\"cabbage\""
```

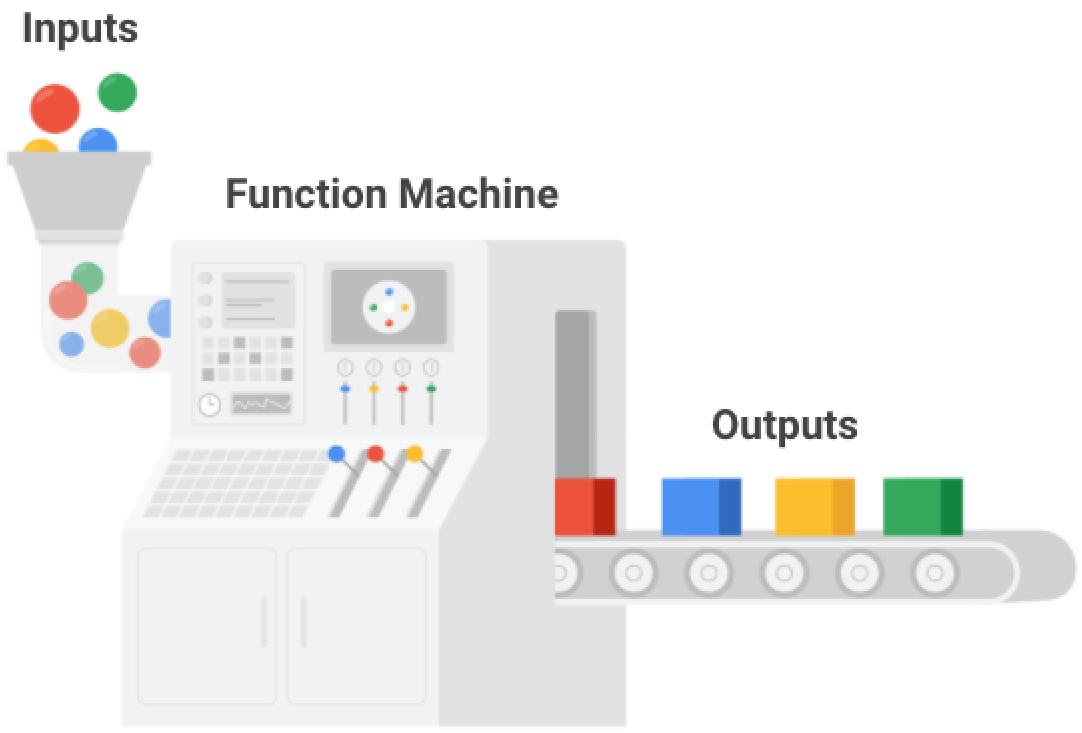
Additional resources

- If you want to learn how to use readr functions to work with more complex files, check out the Data Import chapter
  Opens in a new tab
  of the R for Data Science book. It explores some of the common issues you might encounter when reading files, and how to use readr to manage those issues.
- The readxl
  Opens in a new tab
  entry in the tidyverse documentation gives a good overview of the basic functions in readxl, provides a detailed explanation of how the package operates and the coding concepts behind them, and offers links to other useful resources.
- The R "datasets" package contains lots of useful preloaded datasets. Check out The R Datasets Package
  Opens in a new tab
  for a list. The list includes links to detailed descriptions of each dataset.

Data Import

In this reading, you will learn the basics of importing data into R. It comes with built-in datasets that are great tools for learning how to use R and practice analyzing data. You will explore the data() function and learn how to load sample datasets into RStudio. Then you will go through how to use two tidyverse packages—readr and readxl—to import files from other sources into R. You will learn how to use readr to read a .csv file, and how to use readxl to read a .xlsx file.

The data() function

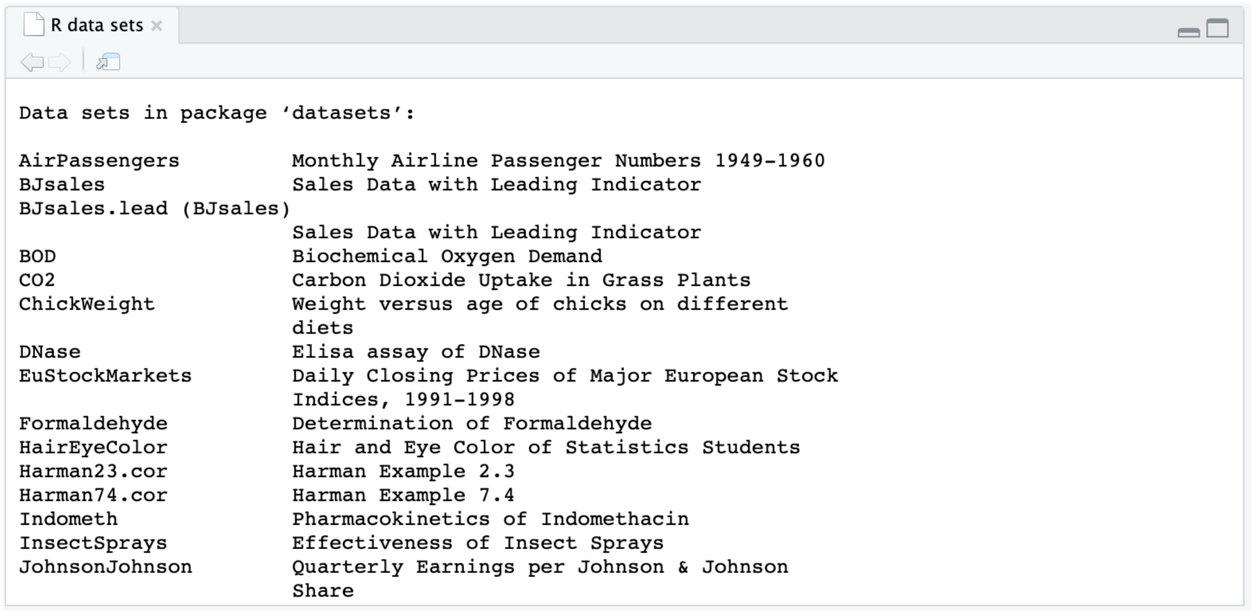


The default installation of R comes with a number of preloaded datasets that you can practice with. This is a great way to develop your R skills and learn about some important data analysis functions. Plus, many online resources and tutorials use these sample datasets to teach coding concepts in R.

You can use the **data()** function to load these datasets in R. If you run the data function without an argument, R will display a list of the available datasets.

```
data()
```

This includes the list of preloaded datasets from the *datasets* package.



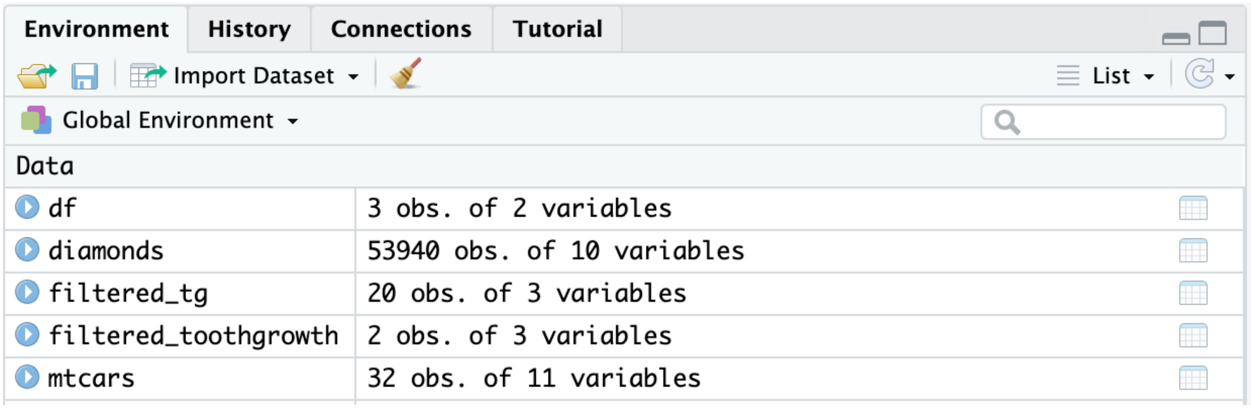
```
Data sets in package 'datasets':

AirPassengers          Monthly Airline Passenger Numbers 1949-1960
BJsales                Sales Data with Leading Indicator
BJsales.lead (BJsales)
                       Sales Data with Leading Indicator
BOD                    Biochemical Oxygen Demand
CO2                    Carbon Dioxide Uptake in Grass Plants
ChickWeight            Weight versus age of chicks on different
                       diets
DNase                  Elisa assay of DNase
EuStockMarkets         Daily Closing Prices of Major European Stock
                       Indices, 1991-1998
Formaldehyde           Determination of Formaldehyde
HairEyeColor           Hair and Eye Color of Statistics Students
Harman23.cor           Harman Example 2.3
Harman74.cor           Harman Example 7.4
Indometh               Pharmacokinetics of Indomethacin
InsectSprays           Effectiveness of Insect Sprays
JohnsonJohnson         Quarterly Earnings per Johnson & Johnson
                       Share
```

If you want to load a specific dataset, just enter its name in the parentheses of the data() function. For example, let's load the *mtcars* dataset, which has information about cars that have been featured in past issues of *Motor Trend* magazine.

```
data(mtcars)
```

When you run the function, R will load the dataset. The dataset will also appear in the Environment pane of your RStudio. The Environment pane displays the names of the data objects, such as data frames and variables, that you have in your current workspace. In this image, *mtcars* appears in the fifth row of the pane. R tells us that it contains 32 observations and 11 variables.



df	3 obs. of 2 variables
diamonds	53940 obs. of 10 variables
filtered_tg	20 obs. of 3 variables
filtered_toothgrowth	2 obs. of 3 variables
mtcars	32 obs. of 11 variables

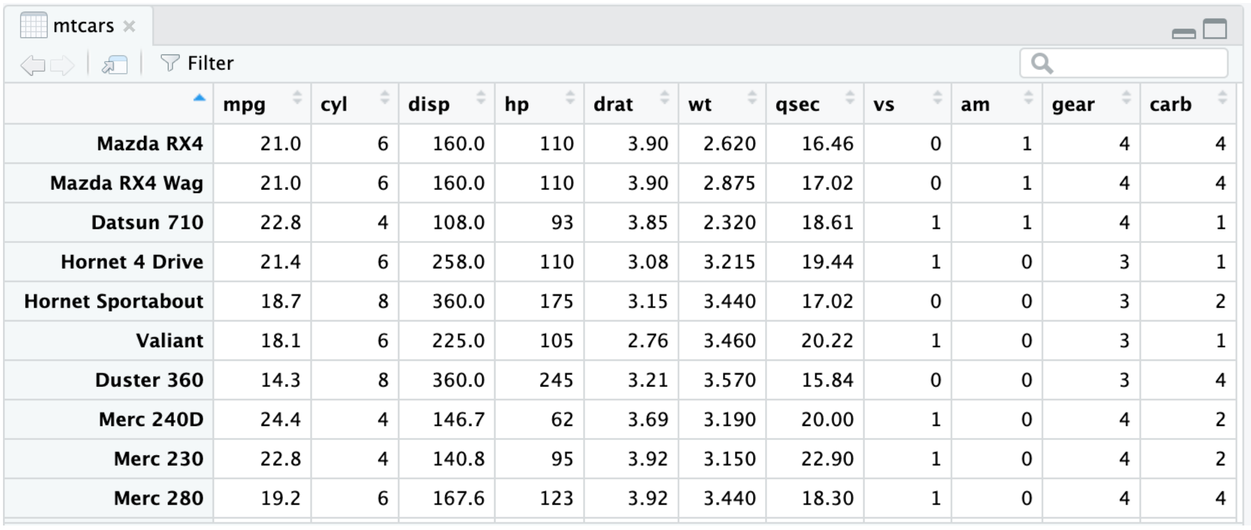
Now that the dataset is loaded, you can get a preview of it in the R console pane. Just type its name...

```
mtcars
```

...and then press ctrl (or cmnd) and enter.

```
                   mpg cyl  disp  hp drat    wt  qsec vs am gear carb
Mazda RX4         21.0   6 160.0 110 3.90 2.620 16.46  0  1    4    4
Mazda RX4 Wag     21.0   6 160.0 110 3.90 2.875 17.02  0  1    4    4
Datsun 710        22.8   4 108.0  93 3.85 2.320 18.61  1  1    4    1
Hornet 4 Drive    21.4   6 258.0 110 3.08 3.215 19.44  1  0    3    1
Hornet Sportabout 18.7   8 360.0 175 3.15 3.440 17.02  0  0    3    2
Valiant           18.1   6 225.0 105 2.76 3.460 20.22  1  0    3    1
Duster 360        14.3   8 360.0 245 3.21 3.570 15.84  0  0    3    4
Merc 240D         24.4   4 146.7  62 3.69 3.190 20.00  1  0    4    2
Merc 230          22.8   4 140.8  95 3.92 3.150 22.90  1  0    4    2
Merc 280          19.2   6 167.6 123 3.92 3.440 18.30  1  0    4    4
```

You can also display the dataset by clicking directly on the name of the dataset in the Environment pane. So, if you click on **mtcars** in the Environment pane, R automatically runs the View() function and displays the dataset in the RStudio data viewer.



	mpg	cyl	disp	hp	drat	wt	qsec	vs	am	gear	carb
Mazda RX4	21.0	6	160.0	110	3.90	2.620	16.46	0	1	4	4
Mazda RX4 Wag	21.0	6	160.0	110	3.90	2.875	17.02	0	1	4	4
Datsun 710	22.8	4	108.0	93	3.85	2.320	18.61	1	1	4	1
Hornet 4 Drive	21.4	6	258.0	110	3.08	3.215	19.44	1	0	3	1
Hornet Sportabout	18.7	8	360.0	175	3.15	3.440	17.02	0	0	3	2
Valiant	18.1	6	225.0	105	2.76	3.460	20.22	1	0	3	1
Duster 360	14.3	8	360.0	245	3.21	3.570	15.84	0	0	3	4
Merc 240D	24.4	4	146.7	62	3.69	3.190	20.00	1	0	4	2
Merc 230	22.8	4	140.8	95	3.92	3.150	22.90	1	0	4	2
Merc 280	19.2	6	167.6	123	3.92	3.440	18.30	1	0	4	4

Try experimenting with other datasets in the list if you want some more practice.

The readr package

In addition to using R's built-in datasets, it is also helpful to import data from other sources to use for practice or analysis. The readr package in R is a great tool for reading rectangular data. Rectangular data is data that fits nicely inside a rectangle of rows and columns, with each column referring to a single variable and each row referring to a single observation.

Here are some examples of file types that store rectangular data:

- **.csv (comma separated values)**: a .csv file is a plain text file that contains a list of data. They mostly use commas to separate (or delimit) data, but sometimes they use other characters, like semicolons.
- **.tsv (tab separated values)**: a .tsv file stores a data table in which the columns of data are separated by tabs. For example, a database table or spreadsheet data.
- **.fwf (fixed width files)**: a .fwf file has a specific format that allows for the saving of textual data in an organized fashion.
- **.log:** a .log file is a computer-generated file that records events from operating systems and other software programs.

Base R also has functions for reading files, but the equivalent functions in readr are typically *much* faster. They also produce tibbles, which are easy to use and read.

The readr package is part of the core tidyverse. So, if you've already installed the tidyverse, you have what you need to start working with readr. If not, you can install the tidyverse now.

readr functions

The goal of readr is to provide a fast and friendly way to read rectangular data. readr supports several read_ functions. Each function refers to a specific file format.

- `read_csv()`
  : comma-separated values (.csv) files
- `read_tsv()`
  : tab-separated values files
- `read_delim()`
  : general delimited files
- `read_fwf()`
  : fixed-width files
- `read_table()`
  : tabular files where columns are separated by white-space
- `read_log()`
  : web log files

These functions all have similar syntax, so once you learn how to use one of them, you can apply your knowledge to the others. This reading will focus on the read_csv() function, since .csv files are one of the most common forms of data storage and you will work with them frequently.

In most cases, these functions will work automatically: you supply the path to a file, run the function, and you get a tibble that displays the data in the file. Behind the scenes, readr parses

the overall file and specifies how each column should be converted from a character vector to the most appropriate data type.

Reading a .csv file with readr

The readr package comes with some sample files from built-in datasets that you can use for example code. To list the sample files, you can run the readr_example() function with no arguments.

```
readr_example()
```

```
[1] "challenge.csv"     "epa78.txt"         "example.log"
[4] "fwf-sample.txt"    "massey-rating.txt" "mtcars.csv"
[7] "mtcars.csv.bz2"    "mtcars.csv.zip"
```

The “mtcars.csv” file refers to the *mtcars* dataset that was mentioned earlier. Let’s use the **read_csv()** function to read the “mtcars.csv” file, as an example. In the parentheses, you need to supply the path to the file. In this case, it’s “readr_example(“mtcars.csv”).

```
read_csv(readr_example("mtcars.csv"))
```

When you run the function, R prints out a column specification that gives the name and type of each column.

```
── Column specification ──────────────────────────────────────────────
cols(
  mpg = col_double(),
  cyl = col_double(),
  disp = col_double(),
  hp = col_double(),
  drat = col_double(),
  wt = col_double(),
  qsec = col_double(),
  vs = col_double(),
  am = col_double(),
  gear = col_double(),
  carb = col_double()
)
```

R also prints a tibble.

```
# A tibble: 32 x 11
     mpg   cyl  disp    hp  drat    wt  qsec    vs    am  gear  carb
   <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl> <dbl>
 1  21       6  160    110  3.9   2.62  16.5     0     1     4     4
 2  21       6  160    110  3.9   2.88  17.0     0     1     4     4
 3  22.8     4  108     93  3.85  2.32  18.6     1     1     4     1
 4  21.4     6  258    110  3.08  3.22  19.4     1     0     3     1
 5  18.7     8  360    175  3.15  3.44  17.0     0     0     3     2
 6  18.1     6  225    105  2.76  3.46  20.2     1     0     3     1
 7  14.3     8  360    245  3.21  3.57  15.8     0     0     3     4
 8  24.4     4  147.    62  3.69  3.19  20       1     0     4     2
 9  22.8     4  141.    95  3.92  3.15  22.9     1     0     4     2
10  19.2     6  168.   123  3.92  3.44  18.3     1     0     4     4
# … with 22 more rows
```

Optional: the readxl package

To import spreadsheet data into R, you can use the readxl package. The readxl package makes it easy to transfer data from Excel into R. Readxl supports both the legacy .xls file format and the modern xml-based .xlsx file format.

The readxl package is part of the tidyverse but is not a *core* tidyverse package, so you need to load readxl in R by using the library() function.

```
library(readxl)
```

Reading an .xlsx file with readxl

Like the readr package, readxl comes with some sample files from built-in datasets that you can use for practice. You can run the code `readxl_example()` to see the list.

You can use the **read_excel()** function to read a spreadsheet file just like you used read_csv() function to read a .csv file. The code for reading the example file `“type-me.xlsx”` includes the path to the file in the parentheses of the function.

```
read_excel(readxl_example("type-me.xlsx"))
```

You can use the excel_sheets()

Opens in a new tab
function to list the names of the individual sheets.

excel_sheets(readxl_example("type-me.xlsx"))

[1] "logical_coercion" "numeric_coercion" "date_coercion" "text_coercion"

You can also specify a sheet by name or number. Just type "sheet =" followed by the name or number of the sheet. For example, you can use the sheet named "numeric_coercion" from the list above.

```
read_excel(readxl_example("type-me.xlsx"), sheet = "numeric_coercion")
```

When you run the function, R returns a tibble of the sheet.

```
# A tibble: 7 x 2
  `maybe numeric?` explanation
  <chr>            <chr>
1 NA               "empty"
2 TRUE             "boolean true"
3 FALSE            "boolean false"
4 40534            "datetime"
5 123456           "the string \"123456\""
6 123456           "the number 123456"
7 cabbage          "\"cabbage\""
```

Additional resources

- If you want to learn how to use readr functions to work with more complex files, check out the Data Import chapter
  Opens in a new tab
  of the R for Data Science book. It explores some of the common issues you might encounter when reading files, and how to use readr to manage those issues.
- The readxl
  Opens in a new tab
  entry in the tidyverse documentation gives a good overview of the basic functions in readxl, provides a detailed explanation of how the package operates and the coding concepts behind them, and offers links to other useful resources.
- The R "datasets" package contains lots of useful preloaded datasets. Check out The R Datasets Package
  Opens in a new tab
  for a list. The list includes links to detailed descriptions of each dataset.

Keeping Your Code Readable

When writing R code (or any other programming language), it is important to use a clear and consistent style that is free from errors. This helps make your code easier to read and understand. In this reading, you will learn some best practices to follow when writing R code. You will also go through some tips for identifying and fixing errors in R code, also known as debugging.

Style

Using a clear and consistent coding style generally makes your code easier for others to read. There's no official coding style guide that is mandatory for all R users. But over the years, the wider community of R users has developed a coding style based on shared conventions and preferences. You can think of these conventions as the unwritten rules of R style.

There are two main reasons for using a consistent coding style:

- If you are working with collaborators or teammates, using a consistent style is important so that everyone can easily read, share, edit, and work on each other's code.
- If you are working alone, using a consistent style is important because it makes it much easier and faster to review your code later on and fix errors or make revisions.

Let's go over a few of the most widely accepted stylistic conventions for writing R code.

Naming

	Guidance	**Examples of best practice**	**Examples to avoid**

Files	File names should be meaningful and end in `.R`. Avoid using special characters in file	`# Good`
		`explore_penguins.R`
		`annual_sales.R`
			`# Bad`
			`Untitled.r`
			`stuff.r`

	names—stick with numbers, letters, dashes, and underscores.		

Object names	Variable and function names should be lowercase. Use an underscore `_` to separate words within a name. Try to create names that are clear, concise, and meaningful.
	
	Generally, variable names should be nouns.
		`# Good`
		`day_one`
			`# Bad`
			`DayOne`

	Function names should be verbs.	`# Good`
		`add ()`
			`# Bad`
			`addition ()`

Syntax

	Guidance	**Examples of best practice**	**Examples to avoid**

Spacing	Most operators (`==`, `+`, `-`, `<-`, etc.) should be surrounded by spaces.	`# Good`
		`x == y`
		`a <- 3 * 2`
			`# Bad`
			`x==y`
			`a<-3*2`

	Always put a space *after* a comma (never before).	`# Good`
		`y[, 2]`
			`# Bad`
			`y[,2]`
			`y[ ,2]`

	Do not place spaces around code in parentheses or square brackets (unless there’s a comma, in which case see above).	`# Good`
		`if (debug) do(x)`
		`species[“dolphin”, ]`
			`# Bad`
			`if ( debug ) do(x)`
			`species[ “dolphin” ,]`

	Place a space before left parentheses, except in a function call.	`# Good`
		`sum(1:5)`
		`plot(x, y)`
			`# Bad`
			`sum (1:5)`
			`plot (x, y)`

Curly braces	An opening curly brace should never go on its own line and should always be followed by a new line. A closing curly brace should always go on its own line (unless it’s followed by an `else` statement). Always indent the code inside curly braces.	`# Good`
		`x <- 7`
		`if (x > 0) {`
		` print("x is a positive number")`
		`} else {`
		` print ("x is either a negative number or zero")`
		`}`
			`# Bad`
			`x <- 7`
			`if (x > 0)`
			`{`
			` print("x is a positive number")`
			`}`
			`else {`
			` print ("x is either a negative number or zero")`
			`}`

Indentation	When indenting your code, use two spaces. Do not use tabs or mix tabs and spaces.	`-`	`-`
Line length	Try to limit your code to 80 characters per line. This fits nicely on a printed page with a reasonably sized font.	`-`	`-`

	Note that many style guides mention to never let a line go past 80 (or 120) characters. If you're using RStudio, there's a helpful setting for this. Go to Tools -> Global Options -> Code -> Display, and select the option Show margin, and set margin column to 80 (or 120).		

Assignment	Use `<-` , not `=` , for assignment.	`# Good`
		`z <- 4`
			`# Bad`
			`Z = 4`

Organization

	Guidance	**Examples of best practice**	**Examples to avoid**

Commenting	Entire commented lines should begin with the comment symbol and a single space: `#`.	`# Good`
		`# Load data`
			`# Bad`
			`Loaddata`

Resources

- Check out this tidyverse style guide to get a more comprehensive breakdown of the most important stylistic conventions for writing R code (and working with the tidyverse).

- The styler package is an automatic styling tool that follows the tidyverse formatting rules. Check out the styler webpage to learn more about the basic features of this tool.

Debugging

Successfully debugging any R code begins with correctly diagnosing the problem. The first step in diagnosing the problem in your code is to understand what you expected to occur. Then, you can identify what actually occurred, and how it differed from your expectations.

For example, imagine you want to run the **glimpse()** function to get a summary view of the *penguins* dataset. You write the following code:

```
Glimpse(penguins)
```

When you run the function, you get the following result:

```
Error in Glimpse(penguins) : could not find function "Glimpse"
```

You were expecting a display of the dataset. Instead, you got an error message. What went wrong? In this case, the problem can be diagnosed as a stylistic error: you wrote `Glimpse` with a capital "G," but the code is case sensitive and requires a lowercase "g." If you run the code `glimpse(penguins)` you'll get the result you expected.

When diagnosing the problem, it is more likely that you–and anyone else who might help debug your code–will understand the problem if you ask the following questions:

- What was your input?
- What were you expecting?
- What did you get?
- How does the result differ from your original expectations?
- Were your expectations correct in the first place?

Some bugs are difficult to discover, and finding the cause of the problem can be challenging. If you come across error messages or you need help with a bug, start by searching online for information about it. You might discover that it's actually a common error with a quick solution.

Resources

- For more information on the technical aspects of debugging R code, check out Debugging with RStudio on the RStudio Support website. RStudio Support is a great place to find answers to your questions about RStudio. This article will take you through the R debugging tools built into RStudio, and show you how to use them to help debug R code.

- To learn more about problem-solving strategies for debugging R code, check out the chapter on Debugging in Advanced R. Advanced R is a great resource if you want to explore the finer details of an R topic and take your knowledge to the next level.

Logical operators and conditional statements

Earlier, you learned that an **operator** is a symbol that identifies the type of operation or calculation to be performed in a formula. In this reading, you'll learn about the main types of logical operators in R, and how they can be used to create conditional statements in R code.

Logical operators

Logical operators return a logical data type such as TRUE or FALSE.

There are three primary types of logical operators:

- AND (sometimes represented as & or && in R)
- OR (sometimes represented as | or || in R)
- NOT (!)

This table summarizes the logical operators:

AND operator “&”	**OR operator “|”**	**NOT operator “!”**
The AND operator takes two logical values. It returns TRUE only if *both* individual values are TRUE. This means that TRUE & TRUE evaluates to `TRUE.` However, FALSE & TRUE, TRUE & FALSE, and FALSE & FALSE all evaluate to `FALSE`.	The OR operator (|) works in a similar way to the AND operator (&). The main difference is that at least one of the values of the OR operation must be TRUE for the entire OR operation to evaluate to `TRUE`. This means that TRUE | TRUE, TRUE | FALSE, and FALSE | TRUE all evaluate to `TRUE`. When both values are FALSE, the result is `FALSE`.	The NOT operator (!) simply negates the logical value it applies to. In other words, !TRUE evaluates to FALSE, and !FALSE evaluates to `TRUE`.

If you run the the corresponding code in R, you get the following results:
 	
`> TRUE & TRUE`
`[1] TRUE`
`> TRUE & FALSE`
`[1] FALSE`
`> FALSE & TRUE`
`[1] FALSE`
`> FALSE & FALSE`
`[1] FALSE`
	If you write out the code, you get the following results:
	
	`> TRUE | TRUE`
	`[1] TRUE`
	`> TRUE | FALSE`
	`[1] TRUE`
	`> FALSE | TRUE`
	`[1] TRUE`
	`> FALSE | FALSE`
	`[1] FALSE`
		When you run the code, you get the following results:
		
		`> !TRUE`
		`[1] FALSE`
		`> !FALSE`
		`[1] TRUE`
		
		Just like the OR and AND operators, you can use the NOT operator in combination with logical operators. Zero is considered FALSE and non-zero numbers are taken

You can illustrate this using the results of our comparisons. Imagine you create a variable *x* that is equal to 10.

```
x <- 10
```

To check if “x” is greater than 3 but less than 12, you can use x > 3 and x < 12 as the values of an “AND” expression.

```
x > 3 & x < 12
```

When you run the function, R returns the result TRUE.

```
[1] TRUE
```

The first part, x > 3 will evaluate to TRUE since 10 is greater than 3. The second part, x < 12 will also evaluate to TRUE since 10 is less than 12. So, since *both* values are TRUE, the result of the AND expression is TRUE. The number 10 lies between the numbers 3 and 12.

However, if you make “x” equal to 20, the expression x > 3 & x < 12 will return a difierent result.

For example, suppose you create a variable *y* equal to 7. To check if *y* is less than 8 or greater than 16, you can use the following expression:

```
y <- 7
y < 8 | y > 16
```

The comparison result is TRUE (7 is less than 8) | FALSE (7 is not greater than 16). Since only one value of an OR expression needs to be TRUE for the entire expression to be TRUE, R returns a result of `TRUE`.

```
[1] TRUE
```

Now, suppose y is 12. The expression y < 8 | y > 16 now evaluates to FALSE (12 < 8) | FALSE (12 > 16). Both comparisons are FALSE, so the result is FALSE.

```
y <- 12
y < 8 | y > 16
[1] FALSE
```

as TRUE. The NOT operator evaluates to the opposite logical value.

Let's imagine you have a variable "x" that equals 2:

```
x <- 2
```

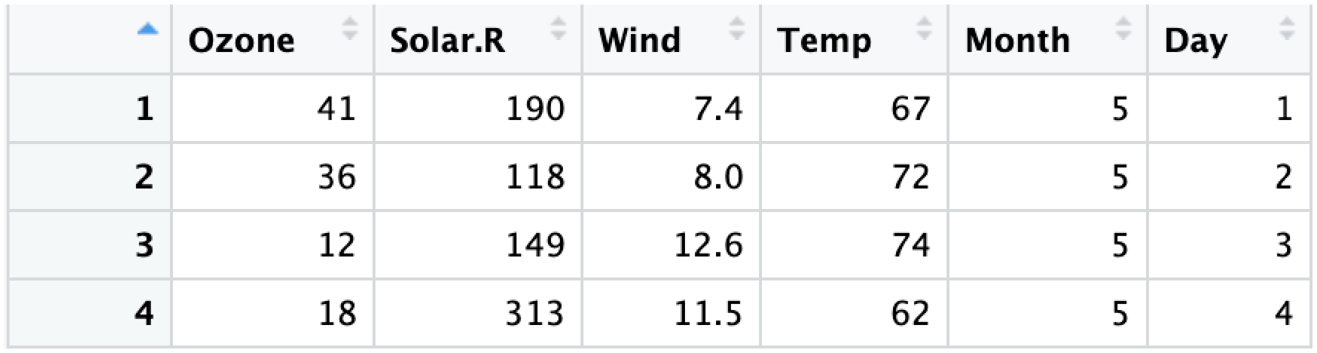
The NOT operation evaluates to `FALSE` because it takes the opposite logical value of a non-zero number (TRUE).

```
> !x
[1] FALSE
```

`x <- 20`
`x > 3 & x < 12`
`[1] FALSE`
 	
Although x > 3 is TRUE (20 > 3), x < 12 is FALSE (20 < 12). If one part of an AND expression is FALSE, the entire expression is FALSE (TRUE & FALSE = FALSE). So, R returns the result FALSE.		

Let’s check out an example of how you might use logical operators to analyze data. Imagine you are working with the “airquality” dataset that is preloaded in RStudio. It contains data on daily air quality measurements in New York from May to September of 1973.

The data frame has six columns: Ozone (the ozone measurement), Solar.R (the solar measurement), Wind (the wind measurement), Temp (the temperature in Fahrenheit), and the Month and Day of these measurements (each row represents a specific month and day combination).



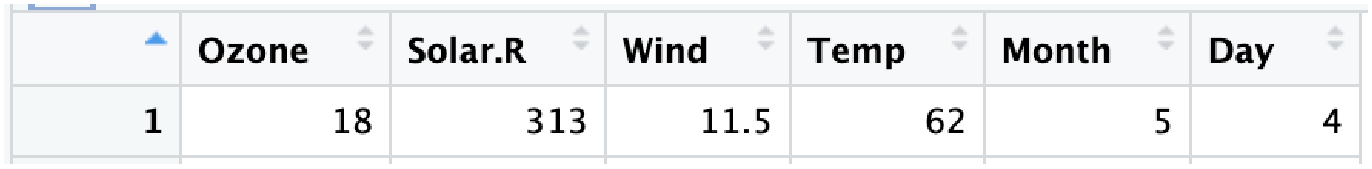
	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	36	118	8.0	72	5	2
3	12	149	12.6	74	5	3
4	18	313	11.5	62	5	4

Let’s go through how the AND, OR, and NOT operators might be helpful in this situation.

AND example
Imagine you want to specify rows that are extremely sunny and windy, which you define as having a Solar measurement of over 150 *and* a Wind measurement of over 10.

In R, you can express this logical statement as Solar.R > 150 & Wind > 10.

Only the rows where *both* of these conditions are true fulfill the criteria:

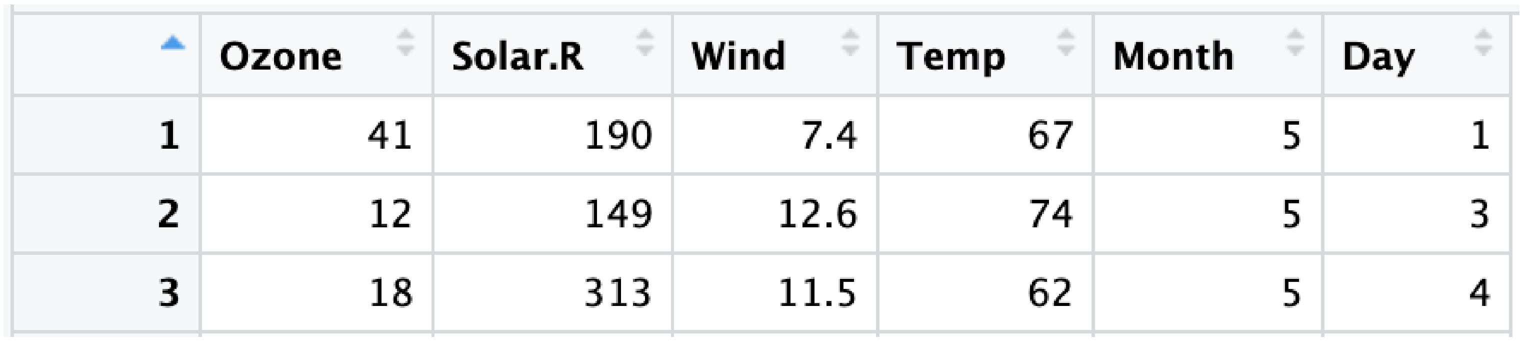


	Ozone	Solar.R	Wind	Temp	Month	Day
1	18	313	11.5	62	5	4

OR example
Next, imagine you want to specify rows where it's extremely sunny or it's extremely windy, which you define as having a Solar measurement of over 150 *or* a Wind measurement of over 10.

In R, you can express this logical statement as Solar.R > 150 | Wind > 10.

All the rows where *either* of these conditions are true fulfill the criteria:

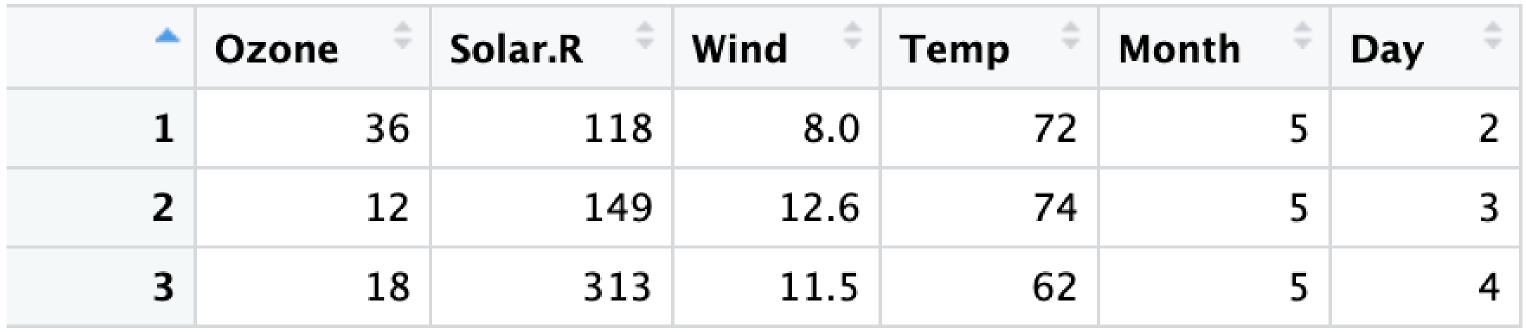


	Ozone	Solar.R	Wind	Temp	Month	Day
1	41	190	7.4	67	5	1
2	12	149	12.6	74	5	3
3	18	313	11.5	62	5	4

NOT example
Now, imagine you just want to focus on the weather measurements for days that are *not* the first day of the month.

In R, you can express this logical statement as Day != 1.

The rows where this condition is true fulfill the criteria:

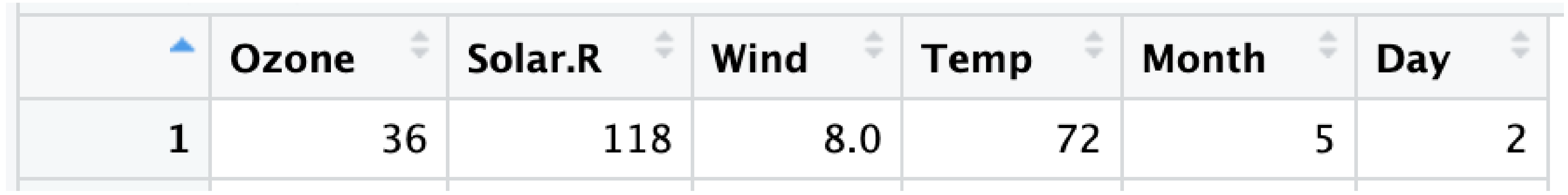


	Ozone	Solar.R	Wind	Temp	Month	Day
1	36	118	8.0	72	5	2
2	12	149	12.6	74	5	3
3	18	313	11.5	62	5	4

Finally, imagine you want to focus on scenarios that are not extremely sunny and not extremely windy, based on your previous definitions of extremely sunny and extremely windy. In other words, the following statement should *not* be true: either a Solar measurement greater than 150 *or* a Wind measurement greater than 10.

Notice that this statement is the opposite of the OR statement used above. To express this statement in R, you can put an exclamation point (!) in front of the previous OR statement: !(Solar.R > 150 | Wind > 10). R will apply the NOT operator to everything within the parentheses.

In this case, only one row fulfills the criteria:



	Ozone	Solar.R	Wind	Temp	Month	Day
1	36	118	8.0	72	5	2

Conditional statements

A **conditional statement** is a declaration that if a certain condition holds, then a certain event must take place. For example, "*If* the temperature is above freezing, *then* I will go outside for a walk." If the first condition is true (the temperature is above freezing), then the second condition will occur (I will go for a walk). Conditional statements in R code have a similar logic.

Let's discuss how to create conditional statements in R using three related statements:

- **if()**
- **else()**
- **else if()**

if statement

The **if** statement sets a condition, and if the condition evaluates to TRUE, the R code associated with the if statement is executed.

In R, you place the code for the condition inside the parentheses of the if statement. The code that has to be executed if the condition is TRUE follows in curly braces ("expr"). Note that, in this case, the second curly brace is placed on its own line of code and identifies the end of the code that you want to execute.

```
if (condition) {
 expr
}
```

For example, let's create a variable "x" equal to 4.

```
x <- 4
```

Next, let’s create a conditional statement: if x is greater than 0, then R will print out the string “x is a positive number.”

```
if (x > 0) {
  print("x is a positive number")
}
```

Since x = 4, the condition is true (4 > 0). Therefore, when you run the code, R prints out the string “x is a positive number.”

```
[1] "x is a positive number"
```

But if you change x to a negative number, like -4, then the condition will be FALSE (-4 > 0). If you run the code, R will not execute the print statement. Instead, a blank line will appear as the result.

else statement

The **else** statement is used in combination with an **if** statement. This is how the code is structured in R:

```
if (condition) {
 expr1
} else {
 expr2
}
```

The code associated with the else statement gets executed whenever the condition of the if statement is *not* TRUE. In other words, if the condition is TRUE, then R will execute the code in the if statement (“expr1”); if the condition is *not* TRUE, then R will execute the code in the else statement (“expr2”).

Let’s try an example. First, create a variable “x” equal to 7.

```
x <- 7
```

Next, let’s set up the following conditions:

- If x is greater than 0, R will print “x is a positive number.”
- If x is less than or equal to 0, R will print “x is either a negative number or zero.”

In our code, the first condition (x > 0) will be part of the if statement. The second condition of x less than or equal to 0 is implied in the else statement. If x > 0, then R will print “x is a positive number.” Otherwise, R will print “x is either a negative number or zero.”

```
x <- 7
if (x > 0) {
 print("x is a positive number")
} else {
 print ("x is either a negative number or zero")
}
```

Since 7 is greater than 0, the condition of the if statement is true. So, when you run the code, R prints out “x is a positive number.”

```
[1] "x is a positive number"
```

But if you make x equal to -7, the condition of the if statement is *not* true (-7 is not greater than 0). Therefore, R will execute the code in the else statement. When you run the code, R prints out “x is either a positive number or zero.”

```
x <- -7
if (x > 0) {
 print("x is a positive number")
} else {
 print ("x is either a negative number or zero")
}
[1] "x is either a negative number or zero"
```

else if statement

In some cases, you might want to customize your conditional statement even further by adding the **else if** statement. The else if statement comes in between the if statement and the else statement.

This is the code structure:

```
if (condition1) {
 expr1
} else if (condition2) {
 expr2
} else {
 expr3
}
```

If the if condition (“condition1”) is met, then R executes the code in the first expression (“expr1”). If the if condition is not met, and the else if condition (“condition2”) is met, then R executes the code in the second expression (“expr2”). If neither of the two conditions are met, R executes the code in the third expression (“expr3”).

In our previous example, using only the if and else statements, R can only print “x is either a negative number or zero” if x equals 0 or x is less than zero. Imagine you want R to print the string “x is zero” if x equals 0. You need to add another condition using the else if statement.

Let’s try an example. First, create a variable “x” equal to negative 1 (“-1”).

```
x <- -1
```

Now, you want to set up the following conditions:

- If x is less than 0, print “x is a negative number.”
- If x equals 0, print “x is zero.”
- Otherwise, print “x is a positive number.”

In our code, the first condition will be part of the if statement, the second condition will be part of the else if statement, and the third condition will be part of the else statement. If x < 0, then R will print “x is a positive number.” If x = 0, then R will print “x is zero.” Otherwise, R will print “x is a positive number.”

```
x <- -1
if (x < 0) {
 print("x is a negative number")
} else if (x == 0) {
 print("x is zero")
} else {
 print("x is a positive number")
}
```

Since -1 is less than 0, the condition for the if statement evaluates to TRUE, and R prints “x is a negative number.”

```
[1] "x is a negative number"
```

If you make x equal to 0, R will first check the if condition (x < 0), and determine that it is FALSE. Then, R will evaluate the else if condition. This condition, x==0, is TRUE. So, in this case, R prints “x is zero.”

If you make x equal to 1, both the if condition and the else if condition evaluate to FALSE. So, R will execute the else statement and print “x is a positive number.”

As soon as R discovers a condition that evaluates to TRUE, R executes the corresponding code and ignores the rest.

Resources

To learn more about logical operators and conditional statements, check out the tutorial on “Conditionals and Control Flow in R” on the DataCamp website. DataCamp is a popular resource for people learning about computer programming. The tutorial is filled with useful examples of coding applications for logical operators and conditional statements (and relational operators), and offers a helpful overview of each topic and the connections between them.

Top 50 Python Interview Questions

CREATE BY - ATUL KUMAR (LINKEDIN)

1. How will you improve the performance of a program in Python?
2. What are the benefits of using Python?
3. How will you specify source code encoding in a Python source file?
4. What is the use of PEP 8 in Python?
5. What is Pickling in Python?
6. How does memory management work in Python?
7. How will you perform Static Analysis on a Python Script?
8. What is the difference between a Tuple and List in Python?
9. What is a Python Decorator?
10. How are arguments passed in a Python method? By value or by reference?
11. What is the difference between List and Dictionary data types in Python?
12. What are the different built-in data types available in Python?
13. What is a Namespace in Python?
14. How will you concatenate multiple strings together in Python?
15. What is the use of Pass statement in Python?
16. What is the use of Slicing in Python?
17. What is the difference between Docstring in Python and Javadoc in Java?
18. How do you perform unit testing for Python code?
19. What is the difference between an Iterator and Iterable in Python?
20. What is the use of Generator in Python?

21. What is the significance of functions that start and end with _ symbol in Python?

22. What is the difference between xrange and range in Python?

23. What is lambda expression in Python?

24. How will you copy an object in Python?

25. What are the main benefits of using Python?

26. What is a metaclass in Python?

27. What is the use of frozenset in Python?

28. What is Python Flask?

29. What is None in Python?

30. What is the use of zip() function in Python?

31. What is the use of // operator in Python?

32. What is a Module in Python?

33. How can we create a dictionary with ordered set of keys in Python?

34. Python is an Object Oriented programming language or a functional programming language?

35. How can we retrieve data from a MySQL database in a Python script?

36. What is the difference between append() and extend() functions of a list in Python?

37. How will you handle an error condition in Python code?

38. What is the difference between split() and slicing in Python?

39. How will you check in Python, if a class is subclass of another class?

40. How will you debug a piece of code in Python?

41. How do you profile a Python script?

42. What is the difference between ‘is’ and ‘==’ in Python?

43. How will you share variables across modules in Python?

44. How can we do Functional programming in Python?

45. What is the improvement in enumerate() function of Python?

46. How will you execute a Python script in Unix?

47. What are the popular Python libraries used in Data analysis?

48. What is the output of following code in Python?

49. What is the output of following code in Python?

50. If you have data with name of customers and their location, which data type will you use to store it in Python?

ACKNOWLEDGMENTS

We thank our readers who constantly send feedback and reviews to motivate us in creating these useful books with the latest information!

INTRODUCTION

This book contains basic to expert level Python interview questions that an interviewer asks. Each question is accompanied with an answer so that you can prepare for job interview in short time.

We have compiled this list after attending dozens of technical interviews in top-notch companies like- Google, Facebook, Netflix, Amazon etc.

Often, these questions and concepts are used in our daily programming work. But these are most helpful when an Interviewer is trying to test your deep knowledge of Python.

The difficulty rating on these Questions varies from a Fresher level software programmer to a Senior software programmer.

Once you go through them in the first pass, mark the questions that you could not answer by yourself. Then, in second pass go through only the difficult questions.

After going through this book 2-3 times, you will be well prepared to face a technical interview on Python for an experienced programmer.

Python Interview Questions

1. How will you improve the performance of a program in Python?

There are many ways to improve the performance of a Python program. Some of these are as follows:

i. **Data Structure**: We have to select the right data structure for our purpose in a Python program.

ii. **Standard Library**: Wherever possible, we should use methods from standard library. Methods implemented in standard library have much better performance than user implementation.

iii. **Abstraction**: At times, a lot of abstraction and indirection can cause slow performance of a program. We should remove the redundant abstraction in code.

iv. **Algorithm**: Use of right algorithm can make a big difference in a program. We have to find and select the suitable algorithm to solve our problem with high performance.

2. What are the benefits of using Python?

Python is strong that even Google uses it. Some of the benefits of using Python are as follows:

i. **Efficient**: Python is very efficient in memory management. For a large data set like Big Data, it is much easier to program in Python.

ii. **Faster**: Though Python code is interpreted, still Python has very fast performance.

iii. **Wide usage**: Python is widely used among different organizations for different projects. Due to this wide usage, there are thousands of add-ons available for use with Python.

iv. **Easy to learn**: Python is quite easy to learn. This is the biggest benefit of using Python. Complex tasks can be very easily implemented in Python.

3. How will you specify source code encoding in a Python source file?

By default, every source code file in Python is in UTF-8 encoding. But we can also specify our own encoding for source files. This can be done by adding following line after #! line in the source file.

```
# -*- coding: encoding -*-
```

In the above line we can replace encoding with the encoding that we want to use.

4. What is the use of PEP 8 in Python?

PEP 8 is a style guide for Python code. This document provides the coding conventions for writing code in Python. Coding conventions are about indentation, formatting, tabs, maximum line length, imports organization, line spacing etc. We use PEP 8 to bring consistency in our code. We consistency it is easier for other developers to read the code.

5. What is Pickling in Python?

Pickling is a process by which a Python object hierarchy can be converted into a byte stream. The reverse operation of Pickling is Unpickling.

Python has a module named pickle. This module has the implementation of a powerful algorithm for serialization and de-serialization of Python object structure.

Some people also call Pickling as Serialization or Marshalling.

With Serialization we can transfer Python objects over the network. It is also used in persisting the state of a Python object. We can write it to a file or a database.

6. How does memory management work in Python?

There is a private heap space in Python that contains all the Python objects and data structures. In CPython there is a memory manager responsible for managing the heap space.

There are different components in Python memory manager that handle segmentation, sharing, caching, memory pre-allocation etc.

Python memory manager also takes care of garbage collection by using Reference counting algorithm.

7. How will you perform Static Analysis on a Python Script?

We can use Static Analysis tool called PyChecker for this purpose. PyChecker can detect errors in Python code.

PyChecker also gives warnings for any style issues.

Some other tools to find bugs in Python code are pylint and pyflakes.

8. What is the difference between a Tuple and List in Python?

In Python, Tuple and List are built-in data structures.

Some of the differences between Tuple and List are as follows:

I. **Syntax**: A Tuple is enclosed in parentheses:
   E.g. myTuple = (10, 20, “apple”);
   A List is enclosed in brackets:
   E.g. myList = [10, 20, 30];

II. **Mutable**: Tuple is an immutable data structure. Whereas, a List is a mutable data structure.

III. **Size**: A Tuple takes much lesser space than a List in Python.

IV. **Performance**: Tuple is faster than a List in Python. So it gives us good performance.

V. **Use case**: Since Tuple is immutable, we can use it in cases like Dictionary creation. Whereas, a List is preferred in the use case where data can alter.

9. What is a Python Decorator?

A Python Decorator is a mechanism to wrap a Python function and modify its behavior by adding more functionality to it. We can use @ symbol to call a Python Decorator function.

10. How are arguments passed in a Python method? By value or by reference?

Every argument in a Python method is an Object. All the variables in Python have reference to an Object. Therefore arguments in Python method are passed by Reference.

Since some of the objects passed as reference are mutable, we can change those objects in a method. But for an Immutable object like String, any change done within a method is not reflected outside.

11. What is the difference between List and Dictionary data types in Python?

Main differences between List and Dictionary data types in Python are as follows:

I. **Syntax**: In a List we store objects in a sequence. In a Dictionary we store objects in key-value pairs.

II. **Reference**: In List we access objects by index number. It starts from 0 index. In a Dictionary we access objects by key specified at the time of Dictionary creation.

III. **Ordering**: In a List objects are stored in an ordered sequence. In a Dictionary objects are not stored in an ordered sequence.

IV. **Hashing**: In a Dictionary, keys have to be hashable. In a List there is no need for hashing.

12. What are the different built-in data types available in Python?

Some of the built-in data types available in Python are as follows:

Numeric types: These are the data types used to represent numbers in Python.

int: It is used for Integers

long: It is used for very large integers of non-limited length.

float: It is used for decimal numbers.

complex: This one is for representing complex numbers

Sequence types: These data types are used to represent sequence of characters or objects.

str: This is similar to String in Java. It can represent a sequence of characters.

bytes: This is a sequence of integers in the range of 0-255.

byte array: like bytes, but mutable (see below); only available in Python 3.x

list: This is a sequence of objects.

tuple: This is a sequence of immutable objects.

Sets: These are unordered collections.

set: This is a collection of unique objects.

frozen set: This is a collection of unique immutable objects.

Mappings: This is similar to a Map in Java.

dict: This is also called hashmap. It has key value pair to store information by using hashing.

13. What is a Namespace in Python?

A Namespace in Python is a mapping between a name and an object. It is currently implemented as Python dictionary.

E.g. the set of built-in exception names, the set of built-in names, local names in a function

At different moments in Python, different Namespaces are created. Each Namespace in Python can have a different lifetime.

For the list of built-in names, Namespace is created when Python interpreter starts.

When Python interpreter reads the definition of a module, it creates global namespace for that module.

When Python interpreter calls a function, it creates local namespace for that function.

14. How will you concatenate multiple strings together in Python?

We can use following ways to concatenate multiple string together in Python:

I. use + operator:

E.g.

```
>>> fname="John"
>>> lname="Ray"
>>> print fname+lname
JohnRay
```

II. use join function:

E.g.

```
>>> ''.join(['John','Ray'])
'JohnRay'
```

15. What is the use of Pass statement in Python?

The use of Pass statement is to do nothing. It is just a placeholder for a statement that is required for syntax purpose. It does not execute any code or command.

Some of the use cases for pass statement are as follows:

I. **Syntax purpose:**

```
>>> while True:

... pass # Wait till user input is received
```

II. **Minimal Class:** It can be used for creating minimal classes:

```
>>> class MyMinimalClass:

... pass
```

III. **Place-holder for TODO work:**

We can also use it as a placeholder for TODO work on a function or code that needs to be implemented at a later point of time.

```
>>> def initialization():

... pass # TODO
```

16. What is the use of Slicing in Python?

We can use Slicing in Python to get a substring from a String.

The syntax of Slicing is very convenient to use.

E.g. In following example we are getting a substring out of the name John.

```
>>> name="John"

>>> name[1:3]

'oh'
```

In Slicing we can give two indices in the String to create a Substring. If we do not give first index, then it defaults to 0.

E.g.

```
>>> name="John"

>>> name[:2]

'Jo'
```

If we do not give second index, then it defaults to the size of the String.

```
>>> name="John"

>>> name[3:]

'n'
```

17. What is the difference between Docstring in Python and Javadoc in Java?

A Docstring in Python is a string used for adding comments or summarizing a piece of code in Python.

The main difference between Javadoc and Docstring is that docstring is available during runtime as well. Whereas, Javadoc is removed from the Bytecode and it is not present in .class file.

We can even use Docstring comments at run time as an interactive help manual.

In Python, we have to specify docstring as the first statement of a code object, just after the def or class statement.

The docstring for a code object can be accessed from the '__doc__' attribute of that object.

18. How do you perform unit testing for Python code?

We can use the unit testing modules **unittest** or **unittest2** to create and run unit tests for Python code.

We can even do automation of tests with these modules. Some of the main components of unittest are as follows:

I. **Test fixture**: We use test fixture to create preparation methods required to run a test. It can even perform post-test cleanup.

II. **Test case**: This is main unit test that we run on a piece of code. We can use **Testcase** base class to create new test cases.

III. **Test suite**: We can aggregate our unit test cases in a Test suite.

IV. **Test runner**: We use test runner to execute unit tests and produce reports of the test run.

19. What is the difference between an Iterator and Iterable in Python?

An Iterable is an object that can be iterated by an Iterator.

In Python, Iterator object provides _iter_() and next() methods.

In Python, an Iterable object has _iter_ function that returns an Iterator object.

When we work on a map or a for loop in Python, we can use next() method to get an Iterable item from the Iterator.

20. What is the use of Generator in Python?

We can use Generator to create Iterators in Python. A Generator is written like a regular function. It can make use yield statement to return data during the function call. In this way we can write complex logic that works as an Iterator.

A Generator is more compact than an Iterator due to the fact that _iter_() and next() functions are automatically created in a Generator.

Also within a Generator code, local variables and execution state are saved between multiple calls. Therefore, there is no need to add extra variables like self.index etc to keep track of iteration.

Generator also increases the readability of the code written in Python. It is a very simple implementation of an Iterator.

21. What is the significance of functions that start and end with _ symbol in Python?

Python provides many built-in functions that are surrounded by _ symbol at the start and end of the function name. As per Python documentation, double _ symbol is used for reserved names of functions.

These are also known as System-defined names.

Some of the important functions are:

Object._new_

Object._init_

Object._del_

22. What is the difference between xrange and range in Python?

In Python, we use range(0,10) to create a list in memory for 10 numbers.

Python provides another function xrange() that is similar to range() but xrange() returns a sequence object instead of list object. In xrange() all the values are not stored simultaneously in memory. It is a lazy loading based function.

But as per Python documentation, the benefit of xrange() over range() is very minimal in regular scenarios.

As of version 3.1, xrange is deprecated.

23. What is lambda expression in Python?

A lambda expression in Python is used for creating an anonymous function.

Wherever we need a function, we can also use a lambda expression.

We have to use lambda keyword for creating a lambda expression. Syntax of lambda function is as follows:

lambda argumentList: expression

E.g. lambda a,b: a+b

The above mentioned lambda expression takes two arguments and returns their sum.

We can use lambda expression to return a function.

A lambda expression can be used to pass a function as an argument in another function.

24. How will you copy an object in Python?

In Python we have two options to copy an object. It is similar to cloning an object in Java.

I. **Shallow Copy**: To create a shallow copy we call copy.copy(x). In a shallow copy, Python creates a new compound object based on the original object. And it tries to put references from the original object into copy object.

II. **Deep Copy**: To create a deep copy, we call copy.deepcopy(x). In a deep copy, Python creates a new object and recursively creates and inserts copies of the objects from original object into copy object. In a deep copy, we may face the issue of recursive loop due to infinite recursion.

25. What are the main benefits of using Python?

Some of the main benefits of using Python are as follows:

I. **Easy to learn**: Python is simple language. It is easy to learn for a new programmer.

II. **Large library**: There is a large library for utilities in Python that can be used for different kinds of applications.

III. **Readability**: Python has a variety of statements and expressions that are quite readable and very explicit in their use. It increases the readability of overall code.

IV. **Memory management**: In Python, memory management is built into the Interpreter. So a developer does not have to spend effort on managing memory among objects.

V. **Complex built-in Data types**: Python has built-in Complex data types like list, set, dict etc. These data types give very good performance as well as save time in coding new features.

26. What is a metaclass in Python?

A metaclass in Python is also known as class of a class. A class defines the behavior of an instance. A metaclass defines the behavior of a class.

One of the most common metaclass in Python is type. We can subclass type to create our own metaclass.

We can use metaclass as a class-factory to create different types of classes.

27. What is the use of frozenset in Python?

A frozenset is a collection of unique values in Python. In addition to all the properties of set, a frozenset is immutable and hashable.

Once we have set the values in a frozenset, we cannot change. So we cannot use and update methods from set on frozenset.

Being hashable, we can use the objects in frozenset as keys in a Dictionary.

28. What is Python Flask?

Python Flask is a micro-framework based on Python to develop a web application.

It is a very simple application framework that has many extensions to build an enterprise level application.

Flask does not provide a data abstraction layer or form validation by default. We can use external libraries on top of Flask to perform such tasks.

29. What is None in Python?

None is a reserved keyword used in Python for null objects. It is neither a null value nor a null pointer. It is an actual object in Python. But there is only one instance of None in a Python environment.

We can use None as a default argument in a function.

During comparison we have to use “is” operator instead of “==” for None.

30. What is the use of zip() function in Python?

In Python, we have a built-in function zip() that can be used to aggregate all the Iterable objects of an Iterator.

We can use it to aggregate Iterable objects from two iterators as well.

E.g.

```
list_1 = ['a', 'b', 'c']

list_2 = ['1', '2', '3']

for a, b in zip(list_1, list_2):

   print a, b
```

Output:

```
a1

b2

c3
```

By using zip() function we can divide our input data from different sources into fixed number of sets.

31. What is the use of // operator in Python?

Python provides // operator to perform floor division of a number by another. The result of // operator is a whole number (without decimal part) quotient that we get by dividing left number with right number.

It can also be used floordiv(a,b).

E.g.

10// 4 = 2

-10//4 = -3

32. What is a Module in Python?

A Module is a script written in Python with import statements, classes, functions etc. We can use a module in another Python script by importing it or by giving the complete namespace.

With Modules, we can divide the functionality of our application in smaller chunks that can be easily managed.

33. How can we create a dictionary with ordered set of keys in Python?

In a normal dictionary in Python, there is no order maintained between keys. To solve this problem, we can use OrderDict class in Python. This class is available for use since version 2.7.

It is similar to a dictionary in Python, but it maintains the insertion order of keys in the dictionary collection.

34. Python is an Object Oriented programming language or a functional programming language?

Python uses most of the Object Oriented programming concepts. But we can also do functional programming in Python. As per the opinion of experts, Python is a multi-paradigm programming language.

We can do functional, procedural, object-oriented and imperative programming with the help of Python.

35. How can we retrieve data from a MySQL database in a Python script?

To retrieve data from a database we have to make use of the module available for that database. For MySQL database, we import MySQLdb module in our Python script.

We have to first connect to a specific database by passing URL, username, password and the name of database.

Once we establish the connection, we can open a cursor with cursor() function. On an open cursor, we can run fetch() function to execute queries and retrieve data from the database tables.

36. What is the difference between append() and extend() functions of a list in Python?

In Python, we get a built-in sequence called list. We can call standard functions like append() and extend() on a list.

We call append() method to add an item to the end of a list.

We call extend() method to add another list to the end of a list.

In append() we have to add items one by one. But in extend() multiple items from another list can be added at the same time.

37. How will you handle an error condition in Python code?

We can implement exception handling to handle error conditions in Python code. If we are expecting an error condition that we cannot handle, we can raise an error with appropriate message.

E.g.

```
>>> if student_score < 0: raise ValueError("Score can not be negative")
```

If we do not want to stop the program, we can just catch the error condition, print a message and continue with our program.

E.g. In following code snippet we are catching the error and continuing with the default value of age.

```
#!/usr/bin/python
try:
   age=18+'duration'
except:
   print("duration has to be a number")
age=18
print(age)
```

38. What is the difference between split() and slicing in Python?

Both split() function and slicing work on a String object. By using split() function, we can get the list of words from a String.

E.g. 'a b c '.split() returns [‘a’, ‘b’, ‘c’]

Slicing is a way of getting substring from a String. It returns another String.

E.g. >>> 'a b c'[2:3] returns b

39. How will you check in Python, if a class is subclass of another class?

Python provides a useful method issubclass(a,b) to check whether class a is a subclass of b.

E.g. int is not a subclass of long

```
>>> issubclass(int,long)
False
```

bool is a subclass of int

```
>>> issubclass(bool,int)
True
```

40. How will you debug a piece of code in Python?

In Python, we can use the debugger pdb for debugging the code. To start debugging we have to enter following lines on the top of a Python script.

```
import pdb

pdb.set_trace()
```

After adding these lines, our code runs in debug mode. Now we can use commands like breakpoint, step through, step into etc for debugging.

41. How do you profile a Python script?

Python provides a profiler called cProfile that can be used for profiling Python code.

We can call it from our code as well as from the interpreter.

It gives use the number of function calls as well as the total time taken to run the script.

We can even write the profile results to a file instead of standard out.

42. What is the difference between ‘is’ and ‘==’ in Python?

We use ‘is’ to check an object against its identity.

We use ‘==’ to check equality of two objects.

E.g.

```
>>> lst = [10,20, 20]
>>> lst == lst[:]
True
>>> lst is lst[:]
False
```

43. How will you share variables across modules in Python?

We can create a common module with variables that we want to share.

This common module can be imported in all the modules in which we want to share the variables.

In this way, all the shared variables will be in one module and available for sharing with any new module as well.

44. How can we do Functional programming in Python?

In Functional Programming, we decompose a program into functions. These functions take input and after processing give an output. The function does not maintain any state.

Python provides built-in functions that can be used for Functional programming. Some of these functions are:

I. Map()
II. reduce()
III. filter()

Event iterators and generators can be used for Functional programming in Python.

45. What is the improvement in enumerate() function of Python?

In Python, enumerate() function is an improvement over regular iteration. The enumerate() function returns an iterator that gives (0, item[0]).

E.g.

```
>>> thelist=['a','b']
>>> for i,j in enumerate(thelist):
... print i,j
...
0 a
1 b
```

46. How will you execute a Python script in Unix?

To execute a Python script in Unix, we need to have Python executor in Unix environment.

In addition to that we have to add following line as the first line in a Python script file.

```
#!/usr/local/bin/python
```

This will tell Unix to use Python interpreter to execute the script.

47. What are the popular Python libraries used in Data analysis?

Some of the popular libraries of Python used for Data analysis are:

I. **Pandas**: Powerful Python Data Analysis Toolkit
II. **SciKit**: This is a machine learning library in Python.
III. **Seaborn**: This is a statistical data visualization library in Python.
IV. **SciPy**: This is an open source system for science, mathematics and engineering implemented in Python.

48. What is the output of following code in Python?

```
>>> thelist=['a','b']
>>> print thelist[3:]
```

Ans: The output of this code is following:

```
[]
```

Even though the list has only 2 elements, the call to thelist with index 3 does not give any index error.

49. What is the output of following code in Python?

```
>>>name=’John Smith’

>>>print name[:5] + name[5:]
```

Ans: Output of this will be

```
John Smith
```

This is an example of Slicing. Since we are slicing at the same index, the first name[:5] gives the substring name upto 5th location excluding 5th location. The name[5:] gives the rest of the substring of name from the 5th location. So we get the full name as output.

50. If you have data with name of customers and their location, which data type will you use to store it in Python?

In Python, we can use dict data type to store key value pairs. In this example, customer name can be the key and their location can be the value in a dict data type.

Dictionary is an efficient way to store data that can be looked up based on a key.