Working with csv files in Python

Difficulty Level : Medium

Last Updated : 19 Feb, 2022

This article explains how to load and parse a CSV file in Python.

First of all, what is a CSV ?

CSV (Comma Separated Values) is a simple file format used to store tabular data, such as a spreadsheet or database. A CSV file stores tabular data (numbers and text) in plain text. Each line of the file is a data record. Each record consists of one or more fields, separated by commas. The use of the comma as a field separator is the source of the name for this file format.

For working CSV files in python, there is an inbuilt module called csv.

Reading a CSV file

# importing csv module

import csv

# csv file name

filename = "aapl.csv"

# initializing the titles and rows list

fields = []

rows = []

# reading csv file

with open(filename, 'r') as csvfile:

# creating a csv reader object

csvreader = csv.reader(csvfile)

# extracting field names through first row

fields = next(csvreader)

# extracting each data row one by one

for row in csvreader:

rows.append(row)

# get total number of rows

print("Total no. of rows: %d"%(csvreader.line\_num))

# printing the field names

print('Field names are:' + ', '.join(field for field in fields))

# printing first 5 rows

print('\nFirst 5 rows are:\n')

for row in rows[:5]:

# parsing each column of a row

for col in row:

print("%10s"%col,end=" "),

print('\n')

The output of the above program looks like this:

The above example uses a CSV file aapl.csv which can be downloaded from here.

Run this program with the aapl.csv file in the same directory.

Let us try to understand this piece of code.

with open(filename, 'r') as csvfile:

csvreader = csv.reader(csvfile)

Here, we first open the CSV file in READ mode. The file object is named as csvfile. The file object is converted to csv.reader object. We save the csv.reader object as csvreader.

fields = csvreader.next()

csvreader is an iterable object. Hence, .next() method returns the current row and advances the iterator to the next row. Since the first row of our csv file contains the headers (or field names), we save them in a list called fields.

for row in csvreader:

rows.append(row)

Now, we iterate through the remaining rows using a for loop. Each row is appended to a list called rows. If you try to print each row, one can find that a row is nothing but a list containing all the field values.

print("Total no. of rows: %d"%(csvreader.line\_num))

csvreader.line\_num is nothing but a counter which returns the number of rows that have been iterated.

Writing to a CSV file

# importing the csv module

import csv

# field names

fields = ['Name', 'Branch', 'Year', 'CGPA']

# data rows of csv file

rows = [ ['Nikhil', 'COE', '2', '9.0'],

['Sanchit', 'COE', '2', '9.1'],

['Aditya', 'IT', '2', '9.3'],

['Sagar', 'SE', '1', '9.5'],

['Prateek', 'MCE', '3', '7.8'],

['Sahil', 'EP', '2', '9.1']]

# name of csv file

filename = "university\_records.csv"

# writing to csv file

with open(filename, 'w') as csvfile:

# creating a csv writer object

csvwriter = csv.writer(csvfile)

# writing the fields

csvwriter.writerow(fields)

# writing the data rows

csvwriter.writerows(rows)

Let us try to understand the above code in pieces.

fields and rows have been already defined. fields is a list containing all the field names. rows is a list of lists. Each row is a list containing the field values of that row.

with open(filename, 'w') as csvfile:

csvwriter = csv.writer(csvfile)

Here, we first open the CSV file in WRITE mode. The file object is named as csvfile. The file object is converted to csv.writer object. We save the csv.writer object as csvwriter.

csvwriter.writerow(fields)

Now we use writerow method to write the first row which is nothing but the field names.

csvwriter.writerows(rows)

We use writerows method to write multiple rows at once.

Writing a dictionary to a CSV file

# importing the csv module

import csv

# my data rows as dictionary objects

mydict =[{'branch': 'COE', 'cgpa': '9.0', 'name': 'Nikhil', 'year': '2'},

{'branch': 'COE', 'cgpa': '9.1', 'name': 'Sanchit', 'year': '2'},

{'branch': 'IT', 'cgpa': '9.3', 'name': 'Aditya', 'year': '2'},

{'branch': 'SE', 'cgpa': '9.5', 'name': 'Sagar', 'year': '1'},

{'branch': 'MCE', 'cgpa': '7.8', 'name': 'Prateek', 'year': '3'},

{'branch': 'EP', 'cgpa': '9.1', 'name': 'Sahil', 'year': '2'}]

# field names

fields = ['name', 'branch', 'year', 'cgpa']

# name of csv file

filename = "university\_records.csv"

# writing to csv file

with open(filename, 'w') as csvfile:

# creating a csv dict writer object

writer = csv.DictWriter(csvfile, fieldnames = fields)

# writing headers (field names)

writer.writeheader()

# writing data rows

writer.writerows(mydict)

In this example, we write a dictionary mydict to a CSV file.

with open(filename, 'w') as csvfile:

writer = csv.DictWriter(csvfile, fieldnames = fields)

Here, the file object (csvfile) is converted to a DictWriter object.

Here, we specify the fieldnames as an argument.

writer.writeheader()

writeheader method simply writes the first row of your csv file using the pre-specified fieldnames.

writer.writerows(mydict)

writerows method simply writes all the rows but in each row, it writes only the values(not keys).

So, in the end, our CSV file looks like this:

Important Points:

In csv modules, an optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV format. By default, csv module uses excel dialect which makes them compatible with excel spreadsheets. You can define your own dialect using register\_dialect method.

Here is an example:

Now, while defining a csv.reader or csv.writer object, we can specify the dialect like

this:

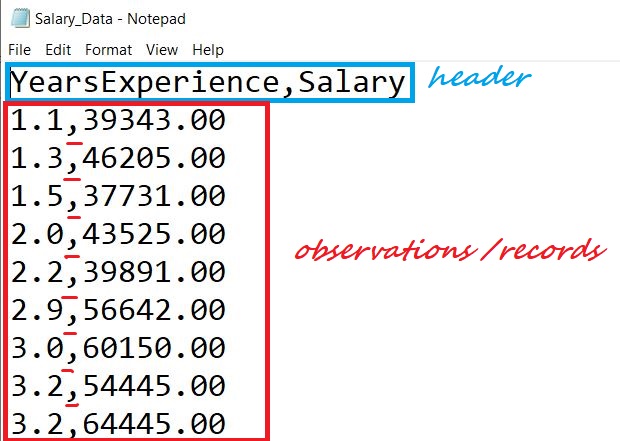
Now, consider that a CSV file looks like this in plain-text:

**Working with CSV file**

## **1. What is a CSV?**

CSV stands for “Comma Separated Values.” It is the simplest form of storing data in tabular form as plain text. It is important to know to work with CSV because we mostly rely on CSV data in our day-to-day lives as data scientists.

#### Structure of CSV:



We have a file named “Salary\_Data.csv.” The first line of a CSV file is the header and contains the names of the fields/features.

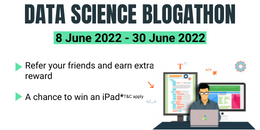
After the header, each line of the file is an observation/a record. The values of a record are separated by “comma.”

## **2. Reading a CSV**

CSV files can be handled in multiple ways in Python.

### **2.1 Using csv.reader**

Reading a CSV using Python’s inbuilt module called [**csv**](https://docs.python.org/2/library/csv.html) using **csv.reader** object.



##### Learn | Write | Earn

Assured INR 2000 ($26) for every published article![Register Now](https://datahack.analyticsvidhya.com/contest/data-science-blogathon-21/?utm_source=blog&utm_medium=banner_between_article&utm_campaign=blogathon21)

Steps to read a CSV file:

#### 1. Import the csv library

import csv

#### 2. Open the CSV file

The .**[open()](https://www.w3schools.com/python/ref_func_open.asp" \t "_blank)**method in python is used to open files and return a file object.

file = open('Salary\_Data.csv')

type(file)

The type of file is “**\_io.TextIOWrapper**” which is a file object that is returned by the **open()**method.

#### 3. Use the csv.reader object to read the CSV file

csvreader = csv.reader(file)

#### 4. Extract the field names

Create an empty list called header. Use the next() method to obtain the header.

The .next() method returns the current row and moves to the next row.

The first time you run next() it returns the header and the next time you run it returns the first record and so on.

header = []

header = next(csvreader)

header



#### 5. Extract the rows/records

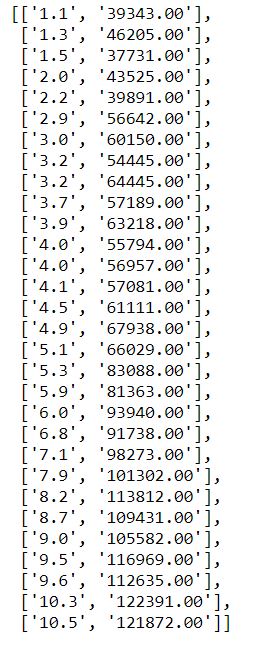
Create an empty list called rows and iterate through the csvreader object and append each row to the rows list.

rows = []

for row in csvreader:

rows.append(row)

rows



#### 6. Close the file

**.close()** method is used to close the opened file. Once it is closed, we cannot perform any operations on it.

file.close()

## Complete Code:

**Python Code:**

Naturally, we might forget to close an open file. To avoid that we can use the **with()**statement to automatically release the resources. In simple terms, there is no need to call the .close() method if we are using with() statement.

## Implementing the above code using with() statement:

**Syntax:**with open(filename, mode) as alias\_filename:

Modes:

‘r’ – to read an existing file,  
‘w’ – to create a new file if the given file doesn’t exist and write to it,  
‘a’ – to append to existing file content,  
‘+’ –  to create a new file for reading and writing

import csv

rows = []

with open("Salary\_Data.csv", 'r) as file:

csvreader = csv.reader(file)

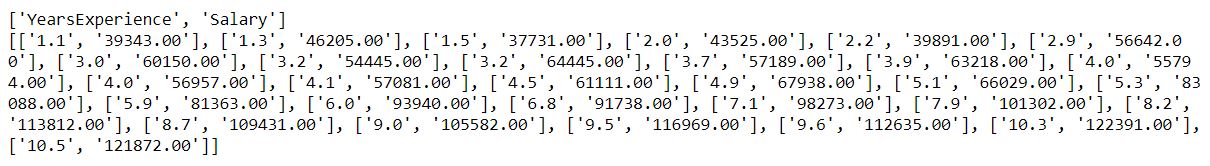
header = next(csvreader)

for row in csvreader:

rows.append(row)

print(header)

print(rows)



### **2.2 Using .readlines()**

Now the question is – “Is it possible to fetch the header, rows using only open() and with() statements and without the csv library?” Let’s see…

**.readlines()** method is the answer. It returns all the lines in a file as a list. Each item of the list is a row of our CSV file.

The first row of the file.readlines() is the header and the rest of them are the records.

with open('Salary\_Data.csv') as file:

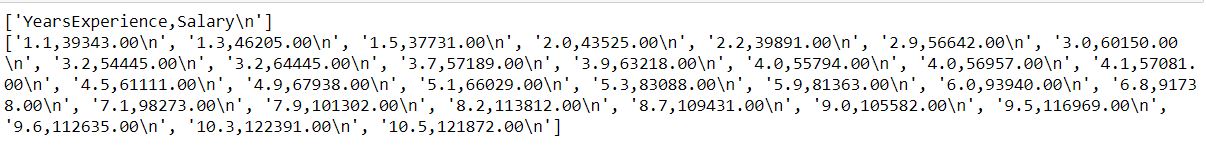
content = file.readlines()

header = content[:1]

rows = content[1:]

print(header)

print(rows)



\*\*The ‘n’ from the output can be removed using .strip() method.

What if we have a huge dataset with hundreds of features and thousands of records. Would it be possible to handle lists??

Here comes the pandas library into the picture.

### **2.3 Using pandas**

Steps of reading CSV files using pandas

**1. Import pandas library**

import pandas as pd

**2. Load CSV files to pandas using [read\_csv()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html" \t "_blank)**

**Basic Syntax:**pandas.read\_csv(filename, delimiter=’,’)

data= pd.read\_csv("Salary\_Data.csv")

data



**3. Extract the field names**

**.columns**is used to obtain the header/field names.

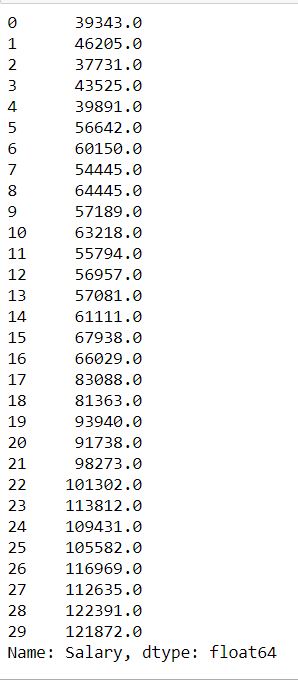
data.columns



**4. Extract the rows**

All the data of a data frame can be accessed using the field names.

data.Salary



## **3. Writing to a CSV file**

We can write to a CSV file in multiple ways.

### **3.1 Using csv.writer**

Let’s assume we are recording 3 Students data(Name, M1 Score, M2 Score)

header = ['Name', 'M1 Score', 'M2 Score']

data = [['Alex', 62, 80], ['Brad', 45, 56], ['Joey', 85, 98]]

Steps of writing to a CSV file:

**1. Import csv library**

import csv

**2. Define a filename and Open the file using open()**

**3. Create a csvwriter object using csv.writer()**

**4. Write the header**

**5. Write the rest of the data**

code for steps 2-5

filename = 'Students\_Data.csv'

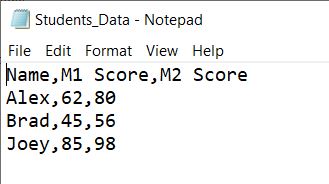
with open(filename, 'w', newline="") as file:

csvwriter = csv.writer(file) # 2. create a csvwriter object

csvwriter.writerow(header) # 4. write the header

csvwriter.writerows(data) # 5. write the rest of the data

Below is how our CSV file looks.



### **3.2 Using .writelines()**

Iterate through each list and convert the list elements to a string and write to the csv file.

header = ['Name', 'M1 Score', 'M2 Score']

data = [['Alex', 62, 80], ['Brad', 45, 56], ['Joey', 85, 98]]

filename = 'Student\_scores.csv'

with open(filename, 'w') as file:

for header in header:

file.write(str(header)+', ')

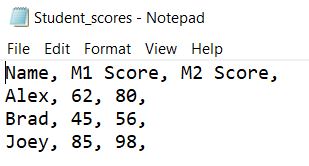
file.write('n')

for row in data:

for x in row:

file.write(str(x)+', ')

file.write('n')



### **3.3. Using pandas**

Steps to writing to a CSV using pandas

**1. Import pandas library**

import pandas as pd

**2. Create a pandas dataframe using [pd.DataFrame](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.html" \t "_blank)**

**Syntax:**pd.DataFrame(data, columns)

The data parameter takes the records/observations and the columns parameter takes the columns/field names.

header = ['Name', 'M1 Score', 'M2 Score']

data = [['Alex', 62, 80], ['Brad', 45, 56], ['Joey', 85, 98]]

data = pd.DataFrame(data, columns=header)

**3. Write to a CSV file using [to\_csv()](https://pandas.pydata.org/docs/reference/api/pandas.DataFrame.to_csv.html" \t "_blank)**

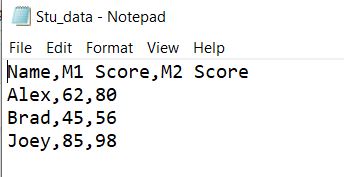
**Syntax:** DataFrame.to\_csv(filename, sep=’,’, index=False)

\*\*separator is ‘,’ by default.

index=False to remove the index numbers.

data.to\_csv('Stu\_data.csv', index=False)

Below is how our CSV looks like



### **End Notes:**

Thank you for reading till the conclusion. By the end of this article, we are familiar with different ways of handling CSV files in Python.

# CSV File Reading and Writing[¶](https://docs.python.org/3/library/csv.html#module-csv)

**Source code:** [Lib/csv.py](https://github.com/python/cpython/tree/3.10/Lib/csv.py)

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. CSV format was used for many years prior to attempts to describe the format in a standardized way in [**RFC 4180**](https://tools.ietf.org/html/rfc4180.html). The lack of a well-defined standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The [csv](https://docs.python.org/3/library/csv.html#module-csv) module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

The [csv](https://docs.python.org/3/library/csv.html#module-csv) module’s [reader](https://docs.python.org/3/library/csv.html#csv.reader) and [writer](https://docs.python.org/3/library/csv.html#csv.writer) objects read and write sequences. Programmers can also read and write data in dictionary form using the [DictReader](https://docs.python.org/3/library/csv.html" \l "csv.DictReader" \o "csv.DictReader) and [DictWriter](https://docs.python.org/3/library/csv.html" \l "csv.DictWriter" \o "csv.DictWriter) classes.

**See also**

[**PEP 305**](https://www.python.org/dev/peps/pep-0305)**- CSV File API**

The Python Enhancement Proposal which proposed this addition to Python.

## **Module Contents**

The [csv](https://docs.python.org/3/library/csv.html#module-csv) module defines the following functions:

csv.**reader**(*csvfile*, *dialect='excel'*, *\*\*fmtparams*)

Return a reader object which will iterate over lines in the given csvfile. csvfile can be any object which supports the [iterator](https://docs.python.org/3/glossary.html#term-iterator) protocol and returns a string each time its \_\_next\_\_() method is called — [file objects](https://docs.python.org/3/glossary.html#term-file-object) and list objects are both suitable. If csvfile is a file object, it should be opened with newline=''. [1](https://docs.python.org/3/library/csv.html#id3) An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the [Dialect](https://docs.python.org/3/library/csv.html#csv.Dialect) class or one of the strings returned by the [list\_dialects()](https://docs.python.org/3/library/csv.html" \l "csv.list_dialects" \o "csv.list_dialects) function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about the dialect and formatting parameters, see section [Dialects and Formatting Parameters](https://docs.python.org/3/library/csv.html#csv-fmt-params).

Each row read from the csv file is returned as a list of strings. No automatic data type conversion is performed unless the QUOTE\_NONNUMERIC format option is specified (in which case unquoted fields are transformed into floats).

A short usage example:

>>>

**>>> import** **csv**

**>>> with** open('eggs.csv', newline='') **as** csvfile:

**...**  spamreader = csv.reader(csvfile, delimiter=' ', quotechar='|')

**...**  **for** row **in** spamreader:

**...**  print(', '.join(row))

Spam, Spam, Spam, Spam, Spam, Baked Beans

Spam, Lovely Spam, Wonderful Spam

csv.**writer**(*csvfile*, *dialect='excel'*, *\*\*fmtparams*)

Return a writer object responsible for converting the user’s data into delimited strings on the given file-like object. csvfile can be any object with a write() method. If csvfile is a file object, it should be opened with newline='' [1](https://docs.python.org/3/library/csv.html#id3). An optional dialect parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. It may be an instance of a subclass of the [Dialect](https://docs.python.org/3/library/csv.html#csv.Dialect) class or one of the strings returned by the [list\_dialects()](https://docs.python.org/3/library/csv.html" \l "csv.list_dialects" \o "csv.list_dialects) function. The other optional fmtparams keyword arguments can be given to override individual formatting parameters in the current dialect. For full details about dialects and formatting parameters, see the [Dialects and Formatting Parameters](https://docs.python.org/3/library/csv.html#csv-fmt-params) section. To make it as easy as possible to interface with modules which implement the DB API, the value [None](https://docs.python.org/3/library/constants.html#None) is written as the empty string. While this isn’t a reversible transformation, it makes it easier to dump SQL NULL data values to CSV files without preprocessing the data returned from a cursor.fetch\* call. All other non-string data are stringified with [str()](https://docs.python.org/3/library/stdtypes.html#str) before being written.

A short usage example:

**import** **csv**

**with** open('eggs.csv', 'w', newline='') **as** csvfile:

spamwriter = csv.writer(csvfile, delimiter=' ',

quotechar='|', quoting=csv.QUOTE\_MINIMAL)

spamwriter.writerow(['Spam'] \* 5 + ['Baked Beans'])

spamwriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])

csv.**register\_dialect**(name[, dialect[, \*\*fmtparams]])

Associate dialect with name. name must be a string. The dialect can be specified either by passing a sub-class of [Dialect](https://docs.python.org/3/library/csv.html#csv.Dialect), or by fmtparams keyword arguments, or both, with keyword arguments overriding parameters of the dialect. For full details about dialects and formatting parameters, see section [Dialects and Formatting Parameters](https://docs.python.org/3/library/csv.html#csv-fmt-params).

csv.**unregister\_dialect**(*name*)

Delete the dialect associated with name from the dialect registry. An [Error](https://docs.python.org/3/library/csv.html#csv.Error) is raised if name is not a registered dialect name.

csv.**get\_dialect**(*name*)

Return the dialect associated with name. An [Error](https://docs.python.org/3/library/csv.html#csv.Error) is raised if name is not a registered dialect name. This function returns an immutable [Dialect](https://docs.python.org/3/library/csv.html#csv.Dialect).

csv.**list\_dialects**()

Return the names of all registered dialects.

csv.**field\_size\_limit**([new\_limit])

Returns the current maximum field size allowed by the parser. If new\_limit is given, this becomes the new limit.

The [csv](https://docs.python.org/3/library/csv.html#module-csv) module defines the following classes:

class csv.**DictReader**(*f*, *fieldnames=None*, *restkey=None*, *restval=None*, *dialect='excel'*, *\*args*, *\*\*kwds*)

Create an object that operates like a regular reader but maps the information in each row to a [dict](https://docs.python.org/3/library/stdtypes.html" \l "dict" \o "dict) whose keys are given by the optional fieldnames parameter.

The fieldnames parameter is a [sequence](https://docs.python.org/3/glossary.html#term-sequence). If fieldnames is omitted, the values in the first row of file f will be used as the fieldnames. Regardless of how the fieldnames are determined, the dictionary preserves their original ordering.

If a row has more fields than fieldnames, the remaining data is put in a list and stored with the fieldname specified by restkey (which defaults to None). If a non-blank row has fewer fields than fieldnames, the missing values are filled-in with the value of restval (which defaults to None).

All other optional or keyword arguments are passed to the underlying [reader](https://docs.python.org/3/library/csv.html#csv.reader) instance.

*Changed in version 3.6:*Returned rows are now of type OrderedDict.

*Changed in version 3.8:*Returned rows are now of type [dict](https://docs.python.org/3/library/stdtypes.html#dict).

A short usage example:

>>>

**>>> import** **csv**

**>>> with** open('names.csv', newline='') **as** csvfile:

**...**  reader = csv.DictReader(csvfile)

**...**  **for** row **in** reader:

**...**  print(row['first\_name'], row['last\_name'])

**...**

Eric Idle

John Cleese

**>>>** print(row)

{'first\_name': 'John', 'last\_name': 'Cleese'}

class csv.**DictWriter**(*f*, *fieldnames*, *restval=''*, *extrasaction='raise'*, *dialect='excel'*, *\*args*, *\*\*kwds*)

Create an object which operates like a regular writer but maps dictionaries onto output rows. The fieldnames parameter is a [sequence](https://docs.python.org/3/library/collections.abc.html#module-collections.abc) of keys that identify the order in which values in the dictionary passed to the writerow() method are written to file f. The optional restval parameter specifies the value to be written if the dictionary is missing a key in fieldnames. If the dictionary passed to the writerow() method contains a key not found in fieldnames, the optional extrasaction parameter indicates what action to take. If it is set to 'raise', the default value, a [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError" \o "ValueError) is raised. If it is set to 'ignore', extra values in the dictionary are ignored. Any other optional or keyword arguments are passed to the underlying [writer](https://docs.python.org/3/library/csv.html#csv.writer) instance.

Note that unlike the [DictReader](https://docs.python.org/3/library/csv.html" \l "csv.DictReader" \o "csv.DictReader) class, the fieldnames parameter of the [DictWriter](https://docs.python.org/3/library/csv.html" \l "csv.DictWriter" \o "csv.DictWriter) class is not optional.

A short usage example:

**import** **csv**

**with** open('names.csv', 'w', newline='') **as** csvfile:

fieldnames = ['first\_name', 'last\_name']

writer = csv.DictWriter(csvfile, fieldnames=fieldnames)

writer.writeheader()

writer.writerow({'first\_name': 'Baked', 'last\_name': 'Beans'})

writer.writerow({'first\_name': 'Lovely', 'last\_name': 'Spam'})

writer.writerow({'first\_name': 'Wonderful', 'last\_name': 'Spam'})

## **What is Pandas?**

Pandas is defined as an open-source library that provides high-performance data manipulation in Python. It is built on top of the NumPy package, which means **Numpy** is required for operating the Pandas. The name of Pandas is derived from the word **Panel Data**, which means **an Econometrics from Multidimensional data**. It is used for data analysis in Python and developed by **Wes McKinney in 2008**.

Before Pandas, Python was capable for data preparation, but it only provided limited support for data analysis. So, Pandas came into the picture and enhanced the capabilities of data analysis. It can perform five significant steps required for processing and analysis of data irrespective of the origin of the data, i.e., **load, manipulate, prepare, model, and analyze**.

## **What is NumPy?**

NumPy is mostly written in C language, and it is an extension module of Python. It is defined as a Python package used for performing the various numerical computations and processing of the multidimensional and single-dimensional array elements. The calculations using Numpy arrays are faster than the normal Python array.

The NumPy package is created by the **Travis Oliphant** in 2005 by adding the functionalities of the ancestor module Numeric into another module **Numarray**. It is also capable of handling a vast amount of data and convenient with Matrix multiplication and data reshaping.

## Installing Pandas

The code in this tutorial is executed with CPython 3.7.4 and Pandas 0.25.1. It would be beneficial to make sure you have the latest versions of Python and Pandas on your machine. You might want to create a new [virtual environment](https://realpython.com/python-virtual-environments-a-primer/) and install the [dependencies](https://realpython.com/courses/managing-python-dependencies/) for this tutorial.

First, you’ll need the Pandas library. You may already have it installed. If you don’t, then you can install it with [pip](https://realpython.com/what-is-pip/):

$ pip install pandas

Once the installation process completes, you should have Pandas installed and ready.

[**Anaconda**](https://www.anaconda.com/) is an excellent Python distribution that comes with Python, many useful packages like Pandas, and a package and environment manager called [Conda](https://docs.conda.io/en/latest/). To learn more about Anaconda, check out [Setting Up Python for Machine Learning on Windows](https://realpython.com/python-windows-machine-learning-setup/).

If you don’t have Pandas in your virtual environment, then you can install it with Conda:

$ conda install pandas

Conda is powerful as it manages the dependencies and their versions. To learn more about working with Conda, you can check out the [official documentation](https://docs.conda.io/projects/conda/en/latest/user-guide/tasks/manage-environments.html).

[Remove ads](https://realpython.com/account/join/)

## Preparing Data

In this tutorial, you’ll use the data related to 20 countries. Here’s an overview of the data and sources you’ll be working with:

* **Country** is denoted by the country name. Each country is in the top 10 list for either population, area, or gross domestic product (GDP). The row labels for the dataset are the three-letter country codes defined in [ISO 3166-1](https://en.wikipedia.org/wiki/ISO_3166-1_alpha-3). The column label for the dataset is COUNTRY.
* **Population** is expressed in millions. The data comes from a list of countries and dependencies by population on [Wikipedia](https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_population). The column label for the dataset is POP.
* **Area** is expressed in thousands of kilometers squared. The data comes from a list of countries and dependencies by area on [Wikipedia](https://en.wikipedia.org/wiki/List_of_countries_and_dependencies_by_area). The column label for the dataset is AREA.
* **Gross domestic product** is expressed in millions of U.S. dollars, according to the United Nations data for 2017. You can find this data in the list of countries by nominal GDP on [Wikipedia](https://en.wikipedia.org/wiki/List_of_countries_by_GDP_(nominal)). The column label for the dataset is GDP.
* **Continent** is either Africa, Asia, Oceania, Europe, North America, or South America. You can find this information on [Wikipedia](https://simple.wikipedia.org/wiki/List_of_countries_by_continents) as well. The column label for the dataset is CONT.
* **Independence day** is a date that commemorates a nation’s independence. The data comes from the list of national independence days on [Wikipedia](https://en.wikipedia.org/wiki/List_of_national_independence_days). The dates are shown in [ISO 8601](https://en.wikipedia.org/wiki/ISO_8601) format. The first four digits represent the year, the next two numbers are the month, and the last two are for the day of the month. The column label for the dataset is IND\_DAY.

This is how the data looks as a table:

|  | **COUNTRY** | **POP** | **AREA** | **GDP** | **CONT** | **IND\_DAY** |
| --- | --- | --- | --- | --- | --- | --- |
| **CHN** | China | 1398.72 | 9596.96 | 12234.78 | Asia |  |
| **IND** | India | 1351.16 | 3287.26 | 2575.67 | Asia | 1947-08-15 |
| **USA** | US | 329.74 | 9833.52 | 19485.39 | N.America | 1776-07-04 |
| **IDN** | Indonesia | 268.07 | 1910.93 | 1015.54 | Asia | 1945-08-17 |
| **BRA** | Brazil | 210.32 | 8515.77 | 2055.51 | S.America | 1822-09-07 |
| **PAK** | Pakistan | 205.71 | 881.91 | 302.14 | Asia | 1947-08-14 |
| **NGA** | Nigeria | 200.96 | 923.77 | 375.77 | Africa | 1960-10-01 |
| **BGD** | Bangladesh | 167.09 | 147.57 | 245.63 | Asia | 1971-03-26 |
| **RUS** | Russia | 146.79 | 17098.25 | 1530.75 |  | 1992-06-12 |
| **MEX** | Mexico | 126.58 | 1964.38 | 1158.23 | N.America | 1810-09-16 |
| **JPN** | Japan | 126.22 | 377.97 | 4872.42 | Asia |  |
| **DEU** | Germany | 83.02 | 357.11 | 3693.20 | Europe |  |
| **FRA** | France | 67.02 | 640.68 | 2582.49 | Europe | 1789-07-14 |
| **GBR** | UK | 66.44 | 242.50 | 2631.23 | Europe |  |
| **ITA** | Italy | 60.36 | 301.34 | 1943.84 | Europe |  |
| **ARG** | Argentina | 44.94 | 2780.40 | 637.49 | S.America | 1816-07-09 |
| **DZA** | Algeria | 43.38 | 2381.74 | 167.56 | Africa | 1962-07-05 |
| **CAN** | Canada | 37.59 | 9984.67 | 1647.12 | N.America | 1867-07-01 |
| **AUS** | Australia | 25.47 | 7692.02 | 1408.68 | Oceania |  |
| **KAZ** | Kazakhstan | 18.53 | 2724.90 | 159.41 | Asia | 1991-12-16 |

You may notice that some of the data is missing. For example, the continent for Russia is not specified because it spreads across both Europe and Asia. There are also several missing independence days because the [data source](https://en.wikipedia.org/wiki/List_of_national_independence_days) omits them.

You can organize this data in Python using a nested [dictionary](https://realpython.com/python-dicts/):

data = {

'CHN': {'COUNTRY': 'China', 'POP': 1\_398.72, 'AREA': 9\_596.96,

'GDP': 12\_234.78, 'CONT': 'Asia'},

'IND': {'COUNTRY': 'India', 'POP': 1\_351.16, 'AREA': 3\_287.26,

'GDP': 2\_575.67, 'CONT': 'Asia', 'IND\_DAY': '1947-08-15'},

'USA': {'COUNTRY': 'US', 'POP': 329.74, 'AREA': 9\_833.52,

'GDP': 19\_485.39, 'CONT': 'N.America',

'IND\_DAY': '1776-07-04'},

'IDN': {'COUNTRY': 'Indonesia', 'POP': 268.07, 'AREA': 1\_910.93,

'GDP': 1\_015.54, 'CONT': 'Asia', 'IND\_DAY': '1945-08-17'},

'BRA': {'COUNTRY': 'Brazil', 'POP': 210.32, 'AREA': 8\_515.77,

'GDP': 2\_055.51, 'CONT': 'S.America', 'IND\_DAY': '1822-09-07'},

'PAK': {'COUNTRY': 'Pakistan', 'POP': 205.71, 'AREA': 881.91,

'GDP': 302.14, 'CONT': 'Asia', 'IND\_DAY': '1947-08-14'},

'NGA': {'COUNTRY': 'Nigeria', 'POP': 200.96, 'AREA': 923.77,

'GDP': 375.77, 'CONT': 'Africa', 'IND\_DAY': '1960-10-01'},

'BGD': {'COUNTRY': 'Bangladesh', 'POP': 167.09, 'AREA': 147.57,

'GDP': 245.63, 'CONT': 'Asia', 'IND\_DAY': '1971-03-26'},

'RUS': {'COUNTRY': 'Russia', 'POP': 146.79, 'AREA': 17\_098.25,

'GDP': 1\_530.75, 'IND\_DAY': '1992-06-12'},

'MEX': {'COUNTRY': 'Mexico', 'POP': 126.58, 'AREA': 1\_964.38,

'GDP': 1\_158.23, 'CONT': 'N.America', 'IND\_DAY': '1810-09-16'},

'JPN': {'COUNTRY': 'Japan', 'POP': 126.22, 'AREA': 377.97,

'GDP': 4\_872.42, 'CONT': 'Asia'},

'DEU': {'COUNTRY': 'Germany', 'POP': 83.02, 'AREA': 357.11,

'GDP': 3\_693.20, 'CONT': 'Europe'},

'FRA': {'COUNTRY': 'France', 'POP': 67.02, 'AREA': 640.68,

'GDP': 2\_582.49, 'CONT': 'Europe', 'IND\_DAY': '1789-07-14'},

'GBR': {'COUNTRY': 'UK', 'POP': 66.44, 'AREA': 242.50,

'GDP': 2\_631.23, 'CONT': 'Europe'},

'ITA': {'COUNTRY': 'Italy', 'POP': 60.36, 'AREA': 301.34,

'GDP': 1\_943.84, 'CONT': 'Europe'},

'ARG': {'COUNTRY': 'Argentina', 'POP': 44.94, 'AREA': 2\_780.40,

'GDP': 637.49, 'CONT': 'S.America', 'IND\_DAY': '1816-07-09'},

'DZA': {'COUNTRY': 'Algeria', 'POP': 43.38, 'AREA': 2\_381.74,

'GDP': 167.56, 'CONT': 'Africa', 'IND\_DAY': '1962-07-05'},

'CAN': {'COUNTRY': 'Canada', 'POP': 37.59, 'AREA': 9\_984.67,

'GDP': 1\_647.12, 'CONT': 'N.America', 'IND\_DAY': '1867-07-01'},

'AUS': {'COUNTRY': 'Australia', 'POP': 25.47, 'AREA': 7\_692.02,

'GDP': 1\_408.68, 'CONT': 'Oceania'},

'KAZ': {'COUNTRY': 'Kazakhstan', 'POP': 18.53, 'AREA': 2\_724.90,

'GDP': 159.41, 'CONT': 'Asia', 'IND\_DAY': '1991-12-16'}

}

columns = ('COUNTRY', 'POP', 'AREA', 'GDP', 'CONT', 'IND\_DAY')

Each row of the table is written as an inner dictionary whose keys are the column names and values are the corresponding data. These dictionaries are then collected as the values in the outer data dictionary. The corresponding keys for data are the three-letter country codes.

You can use this data to create an instance of a Pandas [DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/frame.html). First, you need to import Pandas:

>>>

>>> import pandas as pd

Now that you have Pandas imported, you can use the [DataFrame constructor](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.html) and data to create a DataFrame object.

data is organized in such a way that the country codes correspond to columns. You can reverse the rows and columns of a DataFrame with the property [.T](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.T.html):

>>>

>>> df = pd.DataFrame(data=data).T

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.8 Asia NaN

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.4 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.2 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaN

DEU Germany 83.02 357.11 3693.2 Europe NaN

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.5 2631.23 Europe NaN

ITA Italy 60.36 301.34 1943.84 Europe NaN

ARG Argentina 44.94 2780.4 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaN

KAZ Kazakhstan 18.53 2724.9 159.41 Asia 1991-12-16

Now you have your DataFrame object populated with the data about each country.

**Note:** You can use [.transpose()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.transpose.html) instead of .T to reverse the rows and columns of your dataset. If you use .transpose(), then you can set the optional parameter copy to specify if you want to copy the underlying data. The default behavior is False.

Versions of Python older than 3.6 did not guarantee the order of keys in dictionaries. To ensure the order of columns is maintained for older versions of Python and Pandas, you can specify index=columns:

>>>

>>> df = pd.DataFrame(data=data, index=columns).T

Now that you’ve prepared your data, you’re ready to start working with files!

## Using the Pandas read\_csv() and .to\_csv() Functions

A [comma-separated values (CSV)](https://en.wikipedia.org/wiki/Comma-separated_values) file is a plaintext file with a .csv extension that holds tabular data. This is one of the most popular file formats for storing large amounts of data. Each row of the CSV file represents a single table row. The values in the same row are by default separated with commas, but you could change the separator to a semicolon, tab, space, or some other character.

[Remove ads](https://realpython.com/account/join/)

### **Write a CSV File**

You can save your Pandas DataFrame as a CSV file with [.to\_csv()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_csv.html):

>>>

>>> df.to\_csv('data.csv')

That’s it! You’ve created the file data.csv in your current working directory. You can expand the code block below to see how your CSV file should look:

data.csvShow/Hide

This text file contains the data separated with **commas**. The first column contains the row labels. In some cases, you’ll find them irrelevant. If you don’t want to keep them, then you can pass the argument index=False to .to\_csv().

### **Read a CSV File**

Once your data is saved in a CSV file, you’ll likely want to load and use it from time to time. You can do that with the Pandas [read\_csv()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_csv.html) function:

>>>

>>> df = pd.read\_csv('data.csv', index\_col=0)

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaN

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaN

DEU Germany 83.02 357.11 3693.20 Europe NaN

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaN

ITA Italy 60.36 301.34 1943.84 Europe NaN

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaN

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

In this case, the Pandas read\_csv() function returns a new DataFrame with the data and labels from the file data.csv, which you specified with the first argument. This string can be any valid path, including [URLs](https://en.wikipedia.org/wiki/URL).

The parameter index\_col specifies the column from the CSV file that contains the row labels. You assign a zero-based column index to this parameter. You should determine the value of index\_col when the CSV file contains the row labels to avoid loading them as data.

You’ll learn more about using Pandas with CSV files [later on in this tutorial](https://realpython.com/pandas-read-write-files/#csv-files). You can also check out [Reading and Writing CSV Files in Python](https://realpython.com/python-csv/) to see how to handle CSV files with the built-in Python library [csv](https://docs.python.org/3/library/csv.html) as well.

## Using Pandas to Write and Read Excel Files

[Microsoft Excel](https://products.office.com/en/excel) is probably the most widely-used spreadsheet software. While older versions used binary [.xls](https://en.wikipedia.org/wiki/Microsoft_Excel#File_formats) files, Excel 2007 introduced the new XML-based [.xlsx](https://en.wikipedia.org/wiki/Microsoft_Office_XML_formats) file. You can read and write Excel files in Pandas, similar to CSV files. However, you’ll need to install the following Python packages first:

* [xlwt](https://xlwt.readthedocs.io/en/latest/) to write to .xls files
* [openpyxl](https://openpyxl.readthedocs.io/en/stable/) or [XlsxWriter](https://xlsxwriter.readthedocs.io/) to write to .xlsx files
* [xlrd](https://xlrd.readthedocs.io/en/latest/) to read Excel files

You can install them using [pip](https://realpython.com/what-is-pip/) with a single command:

$ pip install xlwt openpyxl xlsxwriter xlrd

You can also use Conda:

$ conda install xlwt openpyxl xlsxwriter xlrd

Please note that you don’t have to install all these packages. For example, you don’t need both [openpyxl](https://realpython.com/openpyxl-excel-spreadsheets-python/) and XlsxWriter. If you’re going to work just with .xls files, then you don’t need any of them! However, if you intend to work only with .xlsx files, then you’re going to need at least one of them, but not xlwt. Take some time to decide which packages are right for your project.

[Remove ads](https://realpython.com/account/join/)

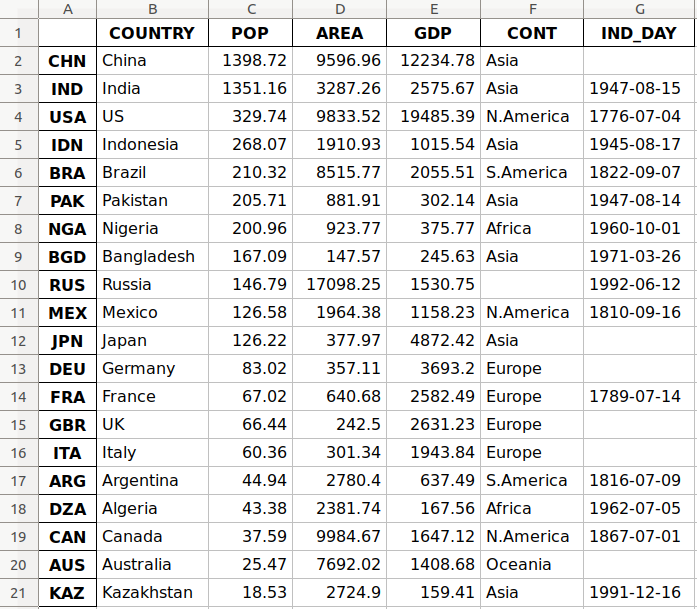
### **Write an Excel File**

Once you have those packages installed, you can save your DataFrame in an Excel file with [.to\_excel()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_excel.html):

>>>

>>> df.to\_excel('data.xlsx')

The argument 'data.xlsx' represents the target file and, optionally, its path. The above statement should create the file data.xlsx in your current working directory. That file should look like this:

[](https://files.realpython.com/media/excel.ca33ad30becb.png)

The first column of the file contains the labels of the rows, while the other columns store data.

### **Read an Excel File**

You can load data from Excel files with [read\_excel()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_excel.html):

>>>

>>> df = pd.read\_excel('data.xlsx', index\_col=0)

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaN

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaN

DEU Germany 83.02 357.11 3693.20 Europe NaN

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaN

ITA Italy 60.36 301.34 1943.84 Europe NaN

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaN

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

read\_excel() returns a new DataFrame that contains the values from data.xlsx. You can also use read\_excel() with [OpenDocument spreadsheets](http://www.opendocumentformat.org/aboutODF/), or .ods files.

You’ll learn more about working with Excel files [later on in this tutorial](https://realpython.com/pandas-read-write-files/#excel-files). You can also check out [Using Pandas to Read Large Excel Files in Python](https://realpython.com/working-with-large-excel-files-in-pandas/).

## Understanding the Pandas IO API

[**Pandas IO Tools**](https://pandas.pydata.org/pandas-docs/stable/user_guide/io.html) is the API that allows you to save the contents of Series and DataFrame objects to the clipboard, objects, or files of various types. It also enables loading data from the clipboard, objects, or files.

### **Write Files**

Series and DataFrame objects have methods that enable writing data and labels to the clipboard or files. They’re named with the pattern **.to\_<file-type>()**, where <file-type> is the type of the target file.

You’ve learned about .to\_csv() and .to\_excel(), but there are others, including:

* .to\_json()
* .to\_html()
* .to\_sql()
* .to\_pickle()

There are still more file types that you can write to, so this list is not exhaustive.

**Note:** To find similar methods, check the official documentation about serialization, IO, and conversion related to [Series](https://pandas.pydata.org/pandas-docs/stable/reference/series.html#serialization-io-conversion) and [DataFrame](https://pandas.pydata.org/pandas-docs/stable/reference/frame.html" \l "serialization-io-conversion) objects.

These methods have parameters specifying the target file path where you saved the data and labels. This is mandatory in some cases and optional in others. If this option is available and you choose to omit it, then the methods return the objects (like strings or iterables) with the contents of DataFrame instances.

The optional parameter compression decides how to compress the file with the data and labels. You’ll learn more about it [later on](https://realpython.com/pandas-read-write-files/#compress-and-decompress-files). There are a few other parameters, but they’re mostly specific to one or several methods. You won’t go into them in detail here.

[Remove ads](https://realpython.com/account/join/)

### **Read Files**

Pandas functions for reading the contents of files are named using the pattern **.read\_<file-type>()**, where <file-type> indicates the type of the file to read. You’ve already seen the Pandas read\_csv() and read\_excel() functions. Here are a few others:

* read\_json()
* read\_html()
* read\_sql()
* read\_pickle()

These functions have a parameter that specifies the target file path. It can be any valid string that represents the path, either on a local machine or in a URL. Other objects are also acceptable depending on the file type.

The optional parameter compression determines the type of decompression to use for the compressed files. You’ll learn about it [later on in this tutorial](https://realpython.com/pandas-read-write-files/#compress-and-decompress-files). There are other parameters, but they’re specific to one or several functions. You won’t go into them in detail here.

## Working With Different File Types

The Pandas library offers a wide range of possibilities for saving your data to files and loading data from files. In this section, you’ll learn more about working with CSV and Excel files. You’ll also see how to use other types of files, like JSON, web pages, databases, and Python pickle files.

### **CSV Files**

You’ve already learned [how to read and write CSV files](https://realpython.com/pandas-read-write-files/#using-the-pandas-read_csv-and-to_csv-functions). Now let’s dig a little deeper into the details. When you use .to\_csv() to save your DataFrame, you can provide an argument for the parameter path\_or\_buf to specify the path, name, and extension of the target file.

path\_or\_buf is the first argument .to\_csv() will get. It can be any string that represents a valid file path that includes the file name and its extension. You’ve seen this in a [previous example](https://realpython.com/pandas-read-write-files/#using-the-pandas-read_csv-and-to_csv-functions). However, if you omit path\_or\_buf, then .to\_csv() won’t create any files. Instead, it’ll return the corresponding string:

>>>

>>> df = pd.DataFrame(data=data).T

>>> s = df.to\_csv()

>>> print(s)

,COUNTRY,POP,AREA,GDP,CONT,IND\_DAY

CHN,China,1398.72,9596.96,12234.78,Asia,

IND,India,1351.16,3287.26,2575.67,Asia,1947-08-15

USA,US,329.74,9833.52,19485.39,N.America,1776-07-04

IDN,Indonesia,268.07,1910.93,1015.54,Asia,1945-08-17

BRA,Brazil,210.32,8515.77,2055.51,S.America,1822-09-07

PAK,Pakistan,205.71,881.91,302.14,Asia,1947-08-14

NGA,Nigeria,200.96,923.77,375.77,Africa,1960-10-01

BGD,Bangladesh,167.09,147.57,245.63,Asia,1971-03-26

RUS,Russia,146.79,17098.25,1530.75,,1992-06-12

MEX,Mexico,126.58,1964.38,1158.23,N.America,1810-09-16

JPN,Japan,126.22,377.97,4872.42,Asia,

DEU,Germany,83.02,357.11,3693.2,Europe,

FRA,France,67.02,640.68,2582.49,Europe,1789-07-14

GBR,UK,66.44,242.5,2631.23,Europe,

ITA,Italy,60.36,301.34,1943.84,Europe,

ARG,Argentina,44.94,2780.4,637.49,S.America,1816-07-09

DZA,Algeria,43.38,2381.74,167.56,Africa,1962-07-05

CAN,Canada,37.59,9984.67,1647.12,N.America,1867-07-01

AUS,Australia,25.47,7692.02,1408.68,Oceania,

KAZ,Kazakhstan,18.53,2724.9,159.41,Asia,1991-12-16

Now you have the string s instead of a CSV file. You also have some **missing values** in your DataFrame object. For example, the continent for Russia and the independence days for several countries (China, Japan, and so on) are not available. In data science and machine learning, you must handle missing values carefully. Pandas excels here! By default, Pandas uses the [NaN value](https://en.wikipedia.org/wiki/NaN) to replace the missing values.

**Note:** [nan](https://en.wikipedia.org/wiki/NaN), which stands for “not a number,” is a particular floating-point value in Python.

You can get a nan value with any of the following functions:

* [float('nan')](https://docs.python.org/3/library/functions.html#float)
* [math.nan](https://docs.python.org/3/library/math.html#math.nan)
* [numpy.nan](https://docs.scipy.org/doc/numpy/reference/constants.html#numpy.nan)

The continent that corresponds to Russia in df is nan:

>>>

>>> df.loc['RUS', 'CONT']

nan

This example uses [.loc[]](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.loc.html) to get data with the specified row and column names.

When you save your DataFrame to a CSV file, empty strings ('') will represent the missing data. You can see this both in your file data.csv and in the string s. If you want to change this behavior, then use the optional parameter na\_rep:

>>>

>>> df.to\_csv('new-data.csv', na\_rep='(missing)')

This code produces the file new-data.csv where the missing values are no longer empty strings. You can expand the code block below to see how this file should look:

new-data.csvShow/Hide

Now, the string '(missing)' in the file corresponds to the nan values from df.

When Pandas reads files, it considers the empty string ('') and a few others as missing values by default:

* 'nan'
* '-nan'
* 'NA'
* 'N/A'
* 'NaN'
* 'null'

If you don’t want this behavior, then you can pass keep\_default\_na=False to the Pandas read\_csv() function. To specify other labels for missing values, use the parameter na\_values:

>>>

>>> pd.read\_csv('new-data.csv', index\_col=0, na\_values='(missing)')

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaN

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaN

DEU Germany 83.02 357.11 3693.20 Europe NaN

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaN

ITA Italy 60.36 301.34 1943.84 Europe NaN

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaN

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

Here, you’ve marked the string '(missing)' as a new missing data label, and Pandas replaced it with nan when it read the file.

When you load data from a file, Pandas assigns the [data types](https://docs.scipy.org/doc/numpy/user/basics.types.html) to the values of each column by default. You can check these types with [.dtypes](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.dtypes.html):

>>>

>>> df = pd.read\_csv('data.csv', index\_col=0)

>>> df.dtypes

COUNTRY object

POP float64

AREA float64

GDP float64

CONT object

IND\_DAY object

dtype: object

The columns with strings and dates ('COUNTRY', 'CONT', and 'IND\_DAY') have the data type object. Meanwhile, the numeric columns contain 64-bit floating-point numbers (float64).

You can use the parameter dtype to specify the desired data types and parse\_dates to force use of [datetimes](https://docs.scipy.org/doc/numpy/reference/arrays.datetime.html):

>>>

>>> dtypes = {'POP': 'float32', 'AREA': 'float32', 'GDP': 'float32'}

>>> df = pd.read\_csv('data.csv', index\_col=0, dtype=dtypes,

... parse\_dates=['IND\_DAY'])

>>> df.dtypes

COUNTRY object

POP float32

AREA float32

GDP float32

CONT object

IND\_DAY datetime64[ns]

dtype: object

>>> df['IND\_DAY']

CHN NaT

IND 1947-08-15

USA 1776-07-04

IDN 1945-08-17

BRA 1822-09-07

PAK 1947-08-14

NGA 1960-10-01

BGD 1971-03-26

RUS 1992-06-12

MEX 1810-09-16

JPN NaT

DEU NaT

FRA 1789-07-14

GBR NaT

ITA NaT

ARG 1816-07-09

DZA 1962-07-05

CAN 1867-07-01

AUS NaT

KAZ 1991-12-16

Name: IND\_DAY, dtype: datetime64[ns]

Now, you have 32-bit floating-point numbers (float32) as specified with dtype. These differ slightly from the original 64-bit numbers because of smaller **precision**. The values in the last column are considered as dates and have the data type datetime64. That’s why the NaN values in this column are replaced with NaT.

Now that you have real dates, you can save them in the format you like:

>>>

>>> df = pd.read\_csv('data.csv', index\_col=0, parse\_dates=['IND\_DAY'])

>>> df.to\_csv('formatted-data.csv', date\_format='%B %d, %Y')

Here, you’ve specified the parameter date\_format to be '%B %d, %Y'. You can expand the code block below to see the resulting file:

formatted-data.csvShow/Hide

The format of the dates is different now. The format '%B %d, %Y' means the date will first display the full name of the month, then the day followed by a comma, and finally the full year.

There are several other optional parameters that you can use with .to\_csv():

* **sep** denotes a values separator.
* **decimal** indicates a decimal separator.
* **encoding** sets the file encoding.
* **header** specifies whether you want to write column labels in the file.

Here’s how you would pass arguments for sep and header:

>>>

>>> s = df.to\_csv(sep=';', header=False)

>>> print(s)

CHN;China;1398.72;9596.96;12234.78;Asia;

IND;India;1351.16;3287.26;2575.67;Asia;1947-08-15

USA;US;329.74;9833.52;19485.39;N.America;1776-07-04

IDN;Indonesia;268.07;1910.93;1015.54;Asia;1945-08-17

BRA;Brazil;210.32;8515.77;2055.51;S.America;1822-09-07

PAK;Pakistan;205.71;881.91;302.14;Asia;1947-08-14

NGA;Nigeria;200.96;923.77;375.77;Africa;1960-10-01

BGD;Bangladesh;167.09;147.57;245.63;Asia;1971-03-26

RUS;Russia;146.79;17098.25;1530.75;;1992-06-12

MEX;Mexico;126.58;1964.38;1158.23;N.America;1810-09-16

JPN;Japan;126.22;377.97;4872.42;Asia;

DEU;Germany;83.02;357.11;3693.2;Europe;

FRA;France;67.02;640.68;2582.49;Europe;1789-07-14

GBR;UK;66.44;242.5;2631.23;Europe;

ITA;Italy;60.36;301.34;1943.84;Europe;

ARG;Argentina;44.94;2780.4;637.49;S.America;1816-07-09

DZA;Algeria;43.38;2381.74;167.56;Africa;1962-07-05

CAN;Canada;37.59;9984.67;1647.12;N.America;1867-07-01

AUS;Australia;25.47;7692.02;1408.68;Oceania;

KAZ;Kazakhstan;18.53;2724.9;159.41;Asia;1991-12-16

The data is separated with a semicolon (';') because you’ve specified sep=';'. Also, since you passed header=False, you see your data without the header row of column names.

The Pandas read\_csv() function has many additional options for managing missing data, working with dates and times, quoting, encoding, handling errors, and more. For instance, if you have a file with one data column and want to get a Series object instead of a DataFrame, then you can pass squeeze=True to read\_csv(). You’ll learn [later on](https://realpython.com/pandas-read-write-files/#working-with-big-data) about data compression and decompression, as well as how to skip rows and columns.

[Remove ads](https://realpython.com/account/join/)

### **JSON Files**

[JSON](https://realpython.com/python-json/) stands for JavaScript object notation. JSON files are plaintext files used for data interchange, and humans can read them easily. They follow the [ISO/IEC 21778:2017](https://www.iso.org/standard/71616.html) and [ECMA-404](https://www.ecma-international.org/publications/standards/Ecma-404.htm) standards and use the .json extension. Python and Pandas work well with JSON files, as Python’s [json](https://docs.python.org/3/library/json.html) library offers built-in support for them.

You can save the data from your DataFrame to a JSON file with [.to\_json()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_json.html). Start by creating a DataFrame object again. Use the dictionary data that holds the data about countries and then apply .to\_json():

>>>

>>> df = pd.DataFrame(data=data).T

>>> df.to\_json('data-columns.json')

This code produces the file data-columns.json. You can expand the code block below to see how this file should look:

data-columns.jsonShow/Hide

data-columns.json has one large dictionary with the column labels as keys and the corresponding inner dictionaries as values.

You can get a different file structure if you pass an argument for the optional parameter orient:

>>>

>>> df.to\_json('data-index.json', orient='index')

The orient parameter defaults to 'columns'. Here, you’ve set it to index.

You should get a new file data-index.json. You can expand the code block below to see the changes:

data-index.jsonShow/Hide

data-index.json also has one large dictionary, but this time the row labels are the keys, and the inner dictionaries are the values.

There are few more options for orient. One of them is 'records':

>>>

>>> df.to\_json('data-records.json', orient='records')

This code should yield the file data-records.json. You can expand the code block below to see the content:

data-records.jsonShow/Hide

data-records.json holds a list with one dictionary for each row. The row labels are not written.

You can get another interesting file structure with orient='split':

>>>

>>> df.to\_json('data-split.json', orient='split')

The resulting file is data-split.json. You can expand the code block below to see how this file should look:

data-split.jsonShow/Hide

data-split.json contains one dictionary that holds the following lists:

* **The names** of the columns
* **The labels** of the rows
* **The inner lists** (two-dimensional sequence) that hold data values

If you don’t provide the value for the optional parameter path\_or\_buf that defines the file path, then .to\_json() will return a JSON string instead of writing the results to a file. This behavior is consistent with .to\_csv().

There are other optional parameters you can use. For instance, you can set index=False to forgo saving row labels. You can manipulate precision with double\_precision, and dates with date\_format and date\_unit. These last two parameters are particularly important when you have time series among your data:

>>>

>>> df = pd.DataFrame(data=data).T

>>> df['IND\_DAY'] = pd.to\_datetime(df['IND\_DAY'])

>>> df.dtypes

COUNTRY object

POP object

AREA object

GDP object

CONT object

IND\_DAY datetime64[ns]

dtype: object

>>> df.to\_json('data-time.json')

In this example, you’ve created the DataFrame from the dictionary data and used [to\_datetime()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.to_datetime.html) to convert the values in the last column to datetime64. You can expand the code block below to see the resulting file:

data-time.jsonShow/Hide

In this file, you have large integers instead of dates for the independence days. That’s because the default value of the optional parameter date\_format is 'epoch' whenever orient isn’t 'table'. This default behavior expresses dates as an [epoch](https://www.epochconverter.com/) in milliseconds relative to midnight on January 1, 1970.

However, if you pass date\_format='iso', then you’ll get the dates in the ISO 8601 format. In addition, date\_unit decides the units of time:

>>>

>>> df = pd.DataFrame(data=data).T

>>> df['IND\_DAY'] = pd.to\_datetime(df['IND\_DAY'])

>>> df.to\_json('new-data-time.json', date\_format='iso', date\_unit='s')

This code produces the following JSON file:

new-data-time.jsonShow/Hide

The dates in the resulting file are in the ISO 8601 format.

You can load the data from a JSON file with [read\_json()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_json.html):

>>>

>>> df = pd.read\_json('data-index.json', orient='index',

... convert\_dates=['IND\_DAY'])

The parameter convert\_dates has a similar purpose as parse\_dates when you use it to read CSV files. The optional parameter orient is very important because it specifies how Pandas understands the structure of the file.

There are other optional parameters you can use as well:

* **Set the encoding** with encoding.
* **Manipulate dates** with convert\_dates and keep\_default\_dates.
* **Impact precision** with dtype and precise\_float.
* **Decode numeric data** directly to [NumPy arrays](https://realpython.com/numpy-array-programming/) with numpy=True.

Note that you might lose the order of rows and columns when using the JSON format to store your data.

[Remove ads](https://realpython.com/account/join/)

### **HTML Files**

An [HTML](https://html.spec.whatwg.org/multipage/) is a plaintext file that uses hypertext markup language to help browsers render web pages. The extensions for HTML files are .html and .htm. You’ll need to install an HTML parser library like [lxml](https://lxml.de/) or [html5lib](https://github.com/html5lib/html5lib-python) to be able to work with HTML files:

$pip install lxml html5lib

You can also use Conda to install the same packages:

$ conda install lxml html5lib

Once you have these libraries, you can save the contents of your DataFrame as an HTML file with [.to\_html()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_html.html):

>>>

df = pd.DataFrame(data=data).T

df.to\_html('data.html')

This code generates a file data.html. You can expand the code block below to see how this file should look:

data.htmlShow/Hide

This file shows the DataFrame contents nicely. However, notice that you haven’t obtained an entire web page. You’ve just output the data that corresponds to df in the HTML format.

.to\_html() won’t create a file if you don’t provide the optional parameter buf, which denotes the buffer to write to. If you leave this parameter out, then your code will return a string as it did with .to\_csv() and .to\_json().

Here are some other optional parameters:

* **header** determines whether to save the column names.
* **index** determines whether to save the row labels.
* **classes** assigns [cascading style sheet (CSS)](https://developer.mozilla.org/en-US/docs/Web/CSS) classes.
* **render\_links** specifies whether to convert URLs to HTML links.
* **table\_id** assigns the CSS id to the table tag.
* **escape** decides whether to convert the characters <, >, and & to HTML-safe strings.

You use parameters like these to specify different aspects of the resulting files or strings.

You can create a DataFrame object from a suitable HTML file using [read\_html()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_html.html), which will return a DataFrame instance or a list of them:

>>>

>>> df = pd.read\_html('data.html', index\_col=0, parse\_dates=['IND\_DAY'])

This is very similar to what you did when reading CSV files. You also have parameters that help you work with dates, missing values, precision, encoding, HTML parsers, and more.

### **Excel Files**

You’ve already learned [how to read and write Excel files with Pandas](https://realpython.com/pandas-read-write-files/#using-pandas-to-write-and-read-excel-files). However, there are a few more options worth considering. For one, when you use .to\_excel(), you can specify the name of the target worksheet with the optional parameter sheet\_name:

>>>

>>> df = pd.DataFrame(data=data).T

>>> df.to\_excel('data.xlsx', sheet\_name='COUNTRIES')

Here, you create a file data.xlsx with a worksheet called COUNTRIES that stores the data. The string 'data.xlsx' is the argument for the parameter excel\_writer that defines the name of the Excel file or its path.

The optional parameters startrow and startcol both default to 0 and indicate the upper left-most cell where the data should start being written:

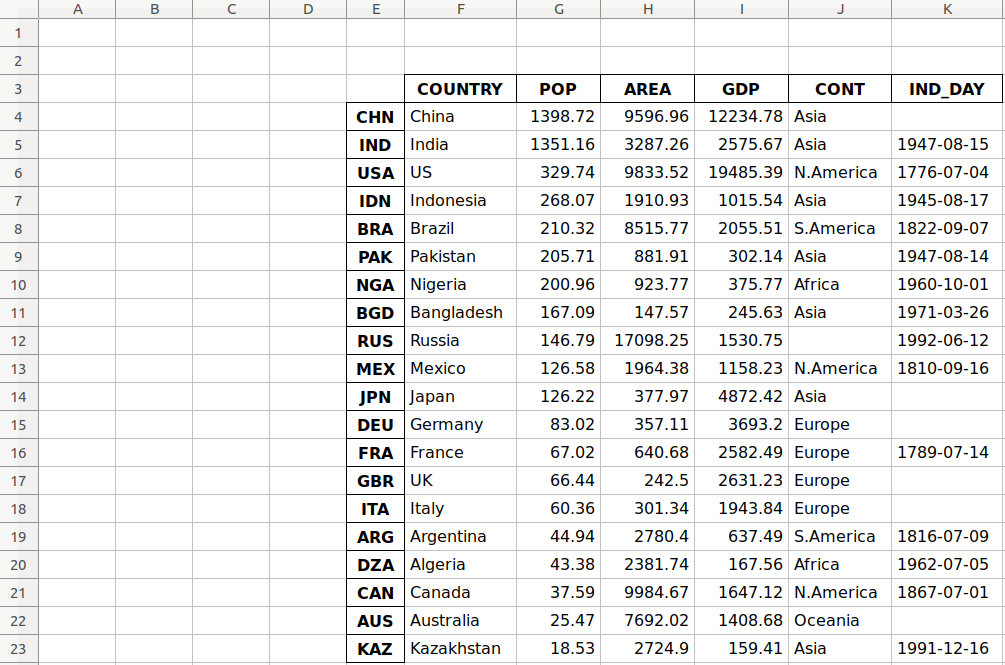
>>>

>>> df.to\_excel('data-shifted.xlsx', sheet\_name='COUNTRIES',

... startrow=2, startcol=4)

Here, you specify that the table should start in the third row and the fifth column. You also used zero-based indexing, so the third row is denoted by 2 and the fifth column by 4.

Now the resulting worksheet looks like this:

[](https://files.realpython.com/media/excel-shifted.404aac55d957.png)

As you can see, the table starts in the third row 2 and the fifth column E.

.read\_excel() also has the optional parameter sheet\_name that specifies which worksheets to read when loading data. It can take on one of the following values:

* **The zero-based index** of the worksheet
* **The name** of the worksheet
* **The list** of indices or names to read multiple sheets
* **The value None** to read all sheets

Here’s how you would use this parameter in your code:

>>>

>>> df = pd.read\_excel('data.xlsx', sheet\_name=0, index\_col=0,

... parse\_dates=['IND\_DAY'])

>>> df = pd.read\_excel('data.xlsx', sheet\_name='COUNTRIES', index\_col=0,

... parse\_dates=['IND\_DAY'])

Both statements above create the same DataFrame because the sheet\_name parameters have the same values. In both cases, sheet\_name=0 and sheet\_name='COUNTRIES' refer to the same worksheet. The argument parse\_dates=['IND\_DAY'] tells Pandas to try to consider the values in this column as dates or times.

There are other optional parameters you can use with .read\_excel() and .to\_excel() to determine the Excel engine, the encoding, the way to handle missing values and infinities, the method for writing column names and row labels, and so on.

[Remove ads](https://realpython.com/account/join/)

### **SQL Files**

Pandas IO tools can also read and write [databases](https://realpython.com/tutorials/databases/). In this next example, you’ll write your data to a database called data.db. To get started, you’ll need the [SQLAlchemy](https://www.sqlalchemy.org/) package. To learn more about it, you can read the [official ORM tutorial](https://docs.sqlalchemy.org/en/13/orm/tutorial.html). You’ll also need the database driver. Python has a built-in driver for [SQLite](https://www.sqlite.org/index.html).

You can install SQLAlchemy with pip:

$ pip install sqlalchemy

You can also install it with Conda:

$ conda install sqlalchemy

Once you have SQLAlchemy installed, import create\_engine() and create a database engine:

>>>

>>> from sqlalchemy import create\_engine

>>> engine = create\_engine('sqlite:///data.db', echo=False)

Now that you have everything set up, the next step is to create a DataFrame object. It’s convenient to specify the data types and apply [.to\_sql()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html).

>>>

>>> dtypes = {'POP': 'float64', 'AREA': 'float64', 'GDP': 'float64',

... 'IND\_DAY': 'datetime64'}

>>> df = pd.DataFrame(data=data).T.astype(dtype=dtypes)

>>> df.dtypes

COUNTRY object

POP float64

AREA float64

GDP float64

CONT object

IND\_DAY datetime64[ns]

dtype: object

[.astype()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.astype.html) is a very convenient method you can use to set multiple data types at once.

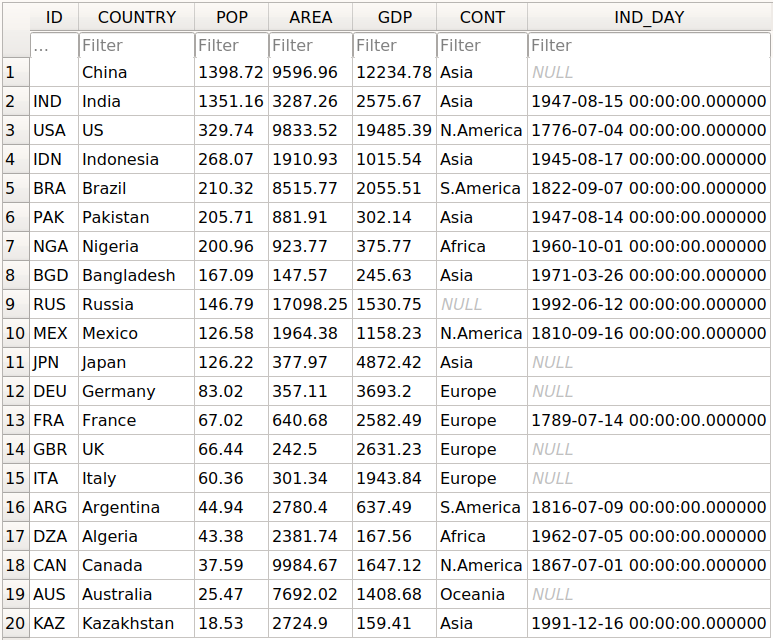
Once you’ve created your DataFrame, you can save it to the database with [.to\_sql()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_sql.html):

>>>

>>> df.to\_sql('data.db', con=engine, index\_label='ID')

The parameter con is used to specify the database connection or engine that you want to use. The optional parameter index\_label specifies how to call the database column with the row labels. You’ll often see it take on the value ID, Id, or id.

You should get the database data.db with a single table that looks like this:

[](https://files.realpython.com/media/db.8b0ba5d7a2f5.png)

The first column contains the row labels. To omit writing them into the database, pass index=False to .to\_sql(). The other columns correspond to the columns of the DataFrame.

There are a few more optional parameters. For example, you can use schema to specify the database schema and dtype to determine the types of the database columns. You can also use if\_exists, which says what to do if a database with the same name and path already exists:

* **if\_exists='fail'** raises a [ValueError](https://docs.python.org/3/library/exceptions.html" \l "ValueError) and is the default.
* **if\_exists='replace'** drops the table and inserts new values.
* **if\_exists='append'** inserts new values into the table.

You can load the data from the database with [read\_sql()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql.html):

>>>

>>> df = pd.read\_sql('data.db', con=engine, index\_col='ID')

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

ID

CHN China 1398.72 9596.96 12234.78 Asia NaT

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 None 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaT

DEU Germany 83.02 357.11 3693.20 Europe NaT

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaT

ITA Italy 60.36 301.34 1943.84 Europe NaT

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaT

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

The parameter index\_col specifies the name of the column with the row labels. Note that this inserts an extra row after the header that starts with ID. You can fix this behavior with the following line of code:

>>>

>>> df.index.name = None

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaT

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 None 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaT

DEU Germany 83.02 357.11 3693.20 Europe NaT

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaT

ITA Italy 60.36 301.34 1943.84 Europe NaT

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaT

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

Now you have the same DataFrame object as before.

Note that the continent for Russia is now None instead of nan. If you want to fill the missing values with nan, then you can use [.fillna()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.fillna.html):

>>>

>>> df.fillna(value=float('nan'), inplace=True)

.fillna() replaces all missing values with whatever you pass to value. Here, you passed float('nan'), which says to fill all missing values with nan.

Also note that you didn’t have to pass parse\_dates=['IND\_DAY'] to read\_sql(). That’s because your database was able to detect that the last column contains dates. However, you can pass parse\_dates if you’d like. You’ll get the same results.

There are other functions that you can use to read databases, like [read\_sql\_table()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql_table.html) and [read\_sql\_query()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_sql_query.html). Feel free to try them out!

[Remove ads](https://realpython.com/account/join/)

### **Pickle Files**

[Pickling](https://realpython.com/python-pickle-module/) is the act of converting Python objects into [byte streams](https://en.wikipedia.org/wiki/Bitstream). Unpickling is the inverse process. [Python pickle files](https://docs.python.org/3/library/pickle.html) are the binary files that keep the data and hierarchy of Python objects. They usually have the extension .pickle or .pkl.

You can save your DataFrame in a pickle file with [.to\_pickle()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_pickle.html):

>>>

>>> dtypes = {'POP': 'float64', 'AREA': 'float64', 'GDP': 'float64',

... 'IND\_DAY': 'datetime64'}

>>> df = pd.DataFrame(data=data).T.astype(dtype=dtypes)

>>> df.to\_pickle('data.pickle')

Like you did with databases, it can be convenient first to specify the data types. Then, you create a file data.pickle to contain your data. You could also pass an integer value to the optional parameter protocol, which specifies the [protocol](https://docs.python.org/3/library/pickle.html#module-interface) of the pickler.

You can get the data from a pickle file with [read\_pickle()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.read_pickle.html):

>>>

>>> df = pd.read\_pickle('data.pickle')

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaT

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaT

DEU Germany 83.02 357.11 3693.20 Europe NaT

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaT

ITA Italy 60.36 301.34 1943.84 Europe NaT

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaT

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

read\_pickle() returns the DataFrame with the stored data. You can also check the data types:

>>>

>>> df.dtypes

COUNTRY object

POP float64

AREA float64

GDP float64

CONT object

IND\_DAY datetime64[ns]

dtype: object

These are the same ones that you specified before using .to\_pickle().

As a word of caution, you should always beware of loading pickles from untrusted sources. **This can be dangerous!** When you unpickle an untrustworthy file, it could execute arbitrary code on your machine, gain remote access to your computer, or otherwise [exploit your device](https://blog.nelhage.com/2011/03/exploiting-pickle/) in other ways.

## Working With Big Data

If your files are too large for saving or processing, then there are several approaches you can take to reduce the required disk space:

* **Compress** your files
* **Choose** only the columns you want
* **Omit** the rows you don’t need
* **Force** the use of less precise data types
* **Split** the data into chunks

You’ll take a look at each of these techniques in turn.

### **Compress and Decompress Files**

You can create an [archive file](https://en.wikipedia.org/wiki/Archive_file) like you would a regular one, with the addition of a suffix that corresponds to the desired compression type:

* '.gz'
* '.bz2'
* '.zip'
* '.xz'

Pandas can deduce the compression type by itself:

>>>

>>> df = pd.DataFrame(data=data).T

>>> df.to\_csv('data.csv.zip')

Here, you create a compressed .csv file as an [archive](https://realpython.com/working-with-files-in-python/#archiving). The size of the regular .csv file is 1048 bytes, while the compressed file only has 766 bytes.

You can open this compressed file as usual with the Pandas read\_csv() function:

>>>

>>> df = pd.read\_csv('data.csv.zip', index\_col=0,

... parse\_dates=['IND\_DAY'])

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaT

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

RUS Russia 146.79 17098.25 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaT

DEU Germany 83.02 357.11 3693.20 Europe NaT

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaT

ITA Italy 60.36 301.34 1943.84 Europe NaT

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaT

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

read\_csv() decompresses the file before reading it into a DataFrame.

You can specify the type of compression with the optional parameter compression, which can take on any of the following values:

* 'infer'
* 'gzip'
* 'bz2'
* 'zip'
* 'xz'
* None

The default value compression='infer' indicates that Pandas should deduce the compression type from the file extension.

Here’s how you would compress a pickle file:

>>>

>>> df = pd.DataFrame(data=data).T

>>> df.to\_pickle('data.pickle.compress', compression='gzip')

You should get the file data.pickle.compress that you can later decompress and read:

>>>

>>> df = pd.read\_pickle('data.pickle.compress', compression='gzip')

df again corresponds to the DataFrame with the same data as before.

You can give the other compression methods a try, as well. If you’re using pickle files, then keep in mind that the .zip format supports reading only.

[Remove ads](https://realpython.com/account/join/)

### **Choose Columns**

The Pandas read\_csv() and read\_excel() functions have the optional parameter usecols that you can use to specify the columns you want to load from the file. You can pass the list of column names as the corresponding argument:

>>>

>>> df = pd.read\_csv('data.csv', usecols=['COUNTRY', 'AREA'])

>>> df

COUNTRY AREA

0 China 9596.96

1 India 3287.26

2 US 9833.52

3 Indonesia 1910.93

4 Brazil 8515.77

5 Pakistan 881.91

6 Nigeria 923.77

7 Bangladesh 147.57

8 Russia 17098.25

9 Mexico 1964.38

10 Japan 377.97

11 Germany 357.11

12 France 640.68

13 UK 242.50

14 Italy 301.34

15 Argentina 2780.40

16 Algeria 2381.74

17 Canada 9984.67

18 Australia 7692.02

19 Kazakhstan 2724.90

Now you have a DataFrame that contains less data than before. Here, there are only the names of the countries and their areas.

Instead of the column names, you can also pass their indices:

>>>

>>> df = pd.read\_csv('data.csv',index\_col=0, usecols=[0, 1, 3])

>>> df

COUNTRY AREA

CHN China 9596.96

IND India 3287.26

USA US 9833.52

IDN Indonesia 1910.93

BRA Brazil 8515.77

PAK Pakistan 881.91

NGA Nigeria 923.77

BGD Bangladesh 147.57

RUS Russia 17098.25

MEX Mexico 1964.38

JPN Japan 377.97

DEU Germany 357.11

FRA France 640.68

GBR UK 242.50

ITA Italy 301.34

ARG Argentina 2780.40

DZA Algeria 2381.74

CAN Canada 9984.67

AUS Australia 7692.02

KAZ Kazakhstan 2724.90

Expand the code block below to compare these results with the file 'data.csv':

data.csvShow/Hide

You can see the following columns:

* The column at **index 0** contains the row labels.
* The column at **index 1** contains the country names.
* The column at **index 3** contains the areas.

Simlarly, read\_sql() has the optional parameter columns that takes a list of column names to read:

>>>

>>> df = pd.read\_sql('data.db', con=engine, index\_col='ID',

... columns=['COUNTRY', 'AREA'])

>>> df.index.name = None

>>> df

COUNTRY AREA

CHN China 9596.96

IND India 3287.26

USA US 9833.52

IDN Indonesia 1910.93

BRA Brazil 8515.77

PAK Pakistan 881.91

NGA Nigeria 923.77

BGD Bangladesh 147.57

RUS Russia 17098.25

MEX Mexico 1964.38

JPN Japan 377.97

DEU Germany 357.11

FRA France 640.68

GBR UK 242.50

ITA Italy 301.34

ARG Argentina 2780.40

DZA Algeria 2381.74

CAN Canada 9984.67

AUS Australia 7692.02

KAZ Kazakhstan 2724.90

Again, the DataFrame only contains the columns with the names of the countries and areas. If columns is None or omitted, then all of the columns will be read, as [you saw before](https://realpython.com/pandas-read-write-files/#sql-files). The default behavior is columns=None.

### **Omit Rows**

When you test an algorithm for data processing or machine learning, you often don’t need the entire dataset. It’s convenient to load only a subset of the data to speed up the process. The Pandas read\_csv() and read\_excel() functions have some optional parameters that allow you to select which rows you want to load:

* **skiprows:** either the number of rows to skip at the beginning of the file if it’s an integer, or the zero-based indices of the rows to skip if it’s a list-like object
* **skipfooter:** the number of rows to skip at the end of the file
* **nrows:** the number of rows to read

Here’s how you would skip rows with odd zero-based indices, keeping the even ones:

>>>

>>> df = pd.read\_csv('data.csv', index\_col=0, skiprows=range(1, 20, 2))

>>> df

COUNTRY POP AREA GDP CONT IND\_DAY

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

DEU Germany 83.02 357.11 3693.20 Europe NaN

GBR UK 66.44 242.50 2631.23 Europe NaN

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

In this example, skiprows is range(1, 20, 2) and corresponds to the values 1, 3, …, 19. The instances of the Python built-in class [range](https://docs.python.org/3/library/stdtypes.html#range) behave like sequences. The first row of the file data.csv is the header row. It has the index 0, so Pandas loads it in. The second row with index 1 corresponds to the label CHN, and Pandas skips it. The third row with the index 2 and label IND is loaded, and so on.

If you want to choose rows randomly, then skiprows can be a list or NumPy array with [pseudo-random](https://realpython.com/python-random/) numbers, obtained either with [pure Python](https://docs.python.org/3/library/random.html) or with [NumPy](https://docs.scipy.org/doc/numpy/reference/random/index.html).

[Remove ads](https://realpython.com/account/join/)

### **Force Less Precise Data Types**

If you’re okay with less precise data types, then you can potentially save a significant amount of memory! First, get the data types with .dtypes again:

>>>

>>> df = pd.read\_csv('data.csv', index\_col=0, parse\_dates=['IND\_DAY'])

>>> df.dtypes

COUNTRY object

POP float64

AREA float64

GDP float64

CONT object

IND\_DAY datetime64[ns]

dtype: object

The columns with the floating-point numbers are 64-bit floats. Each number of this type float64 consumes 64 bits or 8 bytes. Each column has 20 numbers and requires 160 bytes. You can verify this with [.memory\_usage()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.memory_usage.html):

>>>

>>> df.memory\_usage()

Index 160

COUNTRY 160

POP 160

AREA 160

GDP 160

CONT 160

IND\_DAY 160

dtype: int64

.memory\_usage() returns an instance of Series with the memory usage of each column in bytes. You can conveniently combine it with .loc[] and [.sum()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.Series.sum.html) to get the memory for a group of columns:

>>>

>>> df.loc[:, ['POP', 'AREA', 'GDP']].memory\_usage(index=False).sum()

480

This example shows how you can combine the numeric columns 'POP', 'AREA', and 'GDP' to get their total memory requirement. The argument index=False excludes data for row labels from the resulting Series object. For these three columns, you’ll need 480 bytes.

You can also extract the data values in the form of a NumPy array with [.to\_numpy()](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.to_numpy.html#pandas.DataFrame.to_numpy) or [.values](https://pandas.pydata.org/pandas-docs/stable/reference/api/pandas.DataFrame.values.html). Then, use the [.nbytes](https://docs.scipy.org/doc/numpy/reference/generated/numpy.ndarray.nbytes.html) attribute to get the total bytes consumed by the items of the array:

>>>

>>> df.loc[:, ['POP', 'AREA', 'GDP']].to\_numpy().nbytes

480

The result is the same 480 bytes. So, how do you save memory?

In this case, you can specify that your numeric columns 'POP', 'AREA', and 'GDP' should have the type float32. Use the optional parameter dtype to do this:

>>>

>>> dtypes = {'POP': 'float32', 'AREA': 'float32', 'GDP': 'float32'}

>>> df = pd.read\_csv('data.csv', index\_col=0, dtype=dtypes,

... parse\_dates=['IND\_DAY'])

The dictionary dtypes specifies the desired data types for each column. It’s passed to the Pandas read\_csv() function as the argument that corresponds to the parameter dtype.

Now you can verify that each numeric column needs 80 bytes, or 4 bytes per item:

>>>

>>> df.dtypes

COUNTRY object

POP float32

AREA float32

GDP float32

CONT object

IND\_DAY datetime64[ns]

dtype: object

>>> df.memory\_usage()

Index 160

COUNTRY 160

POP 80

AREA 80

GDP 80

CONT 160

IND\_DAY 160

dtype: int64

>>> df.loc[:, ['POP', 'AREA', 'GDP']].memory\_usage(index=False).sum()

240

>>> df.loc[:, ['POP', 'AREA', 'GDP']].to\_numpy().nbytes

240

Each value is a floating-point number of 32 bits or 4 bytes. The three numeric columns contain 20 items each. In total, you’ll need 240 bytes of memory when you work with the type float32. This is half the size of the 480 bytes you’d need to work with float64.

In addition to saving memory, you can significantly reduce the time required to process data by using float32 instead of float64 in some cases.

### **Use Chunks to Iterate Through Files**

Another way to deal with very large datasets is to split the data into smaller **chunks** and process one chunk at a time. If you use read\_csv(), read\_json() or read\_sql(), then you can specify the optional parameter chunksize:

>>>

>>> data\_chunk = pd.read\_csv('data.csv', index\_col=0, chunksize=8)

>>> type(data\_chunk)

<class 'pandas.io.parsers.TextFileReader'>

>>> hasattr(data\_chunk, '\_\_iter\_\_')

True

>>> hasattr(data\_chunk, '\_\_next\_\_')

True

chunksize defaults to None and can take on an integer value that indicates the number of items in a single chunk. When chunksize is an integer, read\_csv() returns an iterable that you can use in a [for loop](https://realpython.com/courses/python-for-loop/) to get and process only a fragment of the dataset in each iteration:

>>>

>>> for df\_chunk in pd.read\_csv('data.csv', index\_col=0, chunksize=8):

... print(df\_chunk, end='\n\n')

... print('memory:', df\_chunk.memory\_usage().sum(), 'bytes',

... end='\n\n\n')

...

COUNTRY POP AREA GDP CONT IND\_DAY

CHN China 1398.72 9596.96 12234.78 Asia NaN

IND India 1351.16 3287.26 2575.67 Asia 1947-08-15

USA US 329.74 9833.52 19485.39 N.America 1776-07-04

IDN Indonesia 268.07 1910.93 1015.54 Asia 1945-08-17

BRA Brazil 210.32 8515.77 2055.51 S.America 1822-09-07

PAK Pakistan 205.71 881.91 302.14 Asia 1947-08-14

NGA Nigeria 200.96 923.77 375.77 Africa 1960-10-01

BGD Bangladesh 167.09 147.57 245.63 Asia 1971-03-26

memory: 448 bytes

COUNTRY POP AREA GDP CONT IND\_DAY

RUS Russia 146.79 17098.25 1530.75 NaN 1992-06-12

MEX Mexico 126.58 1964.38 1158.23 N.America 1810-09-16

JPN Japan 126.22 377.97 4872.42 Asia NaN

DEU Germany 83.02 357.11 3693.20 Europe NaN

FRA France 67.02 640.68 2582.49 Europe 1789-07-14

GBR UK 66.44 242.50 2631.23 Europe NaN

ITA Italy 60.36 301.34 1943.84 Europe NaN

ARG Argentina 44.94 2780.40 637.49 S.America 1816-07-09

memory: 448 bytes

COUNTRY POP AREA GDP CONT IND\_DAY

DZA Algeria 43.38 2381.74 167.56 Africa 1962-07-05

CAN Canada 37.59 9984.67 1647.12 N.America 1867-07-01

AUS Australia 25.47 7692.02 1408.68 Oceania NaN

KAZ Kazakhstan 18.53 2724.90 159.41 Asia 1991-12-16

memory: 224 bytes

In this example, the chunksize is 8. The first iteration of the for loop returns a DataFrame with the first eight rows of the dataset only. The second iteration returns another DataFrame with the next eight rows. The third and last iteration returns the remaining four rows.

# Select Rows from Pandas DataFrame

May 29, 2021

Need to select rows from Pandas DataFrame?

If so, you’ll see the full steps to select rows from Pandas DataFrame based on the conditions specified.

## Steps to Select Rows from Pandas DataFrame

### Step 1: Gather your data

Firstly, you’ll need to gather your data. Here is an example of a data gathered about boxes:

|  |  |  |
| --- | --- | --- |
| **Color** | **Shape** | **Price** |
| Green | Rectangle | 10 |
| Green | Rectangle | 15 |
| Green | Square | 5 |
| Blue | Rectangle | 5 |
| Blue | Square | 10 |
| Red | Square | 15 |
| Red | Square | 15 |
| Red | Rectangle | 5 |

### Step 2: Create a DataFrame

Once you have your data ready, you’ll need to [create a DataFrame](https://datatofish.com/create-pandas-dataframe/) to capture that data in Python.

For our example, you may use the code below to create a DataFrame:

import pandas as pd

boxes = {'Color': ['Green','Green','Green','Blue','Blue','Red','Red','Red'],

'Shape': ['Rectangle','Rectangle','Square','Rectangle','Square','Square','Square','Rectangle'],

'Price': [10,15,5,5,10,15,15,5]

}

df = pd.DataFrame(boxes, columns= ['Color','Shape','Price'])

print (df)

Run the code in [Python](https://datatofish.com/python-tutorials/) and you’ll see this DataFrame:

Color Shape Price

0 Green Rectangle 10

1 Green Rectangle 15

2 Green Square 5

3 Blue Rectangle 5

4 Blue Square 10

5 Red Square 15

6 Red Square 15

7 Red Rectangle 5

### Step 3: Select Rows from Pandas DataFrame

You can use the following logic to select rows from Pandas DataFrame based on specified conditions:

**df.loc[df[‘column name’] condition]**

For example, if you want to get the rows where the color is green, then you’ll need to apply:

**df.loc[df[‘Color’] == ‘Green’]**

Where:

* **Color** is the column name
* **Green** is the condition

And here is the full Python code for our example:

import pandas as pd

boxes = {'Color': ['Green','Green','Green','Blue','Blue','Red','Red','Red'],

'Shape': ['Rectangle','Rectangle','Square','Rectangle','Square','Square','Square','Rectangle'],

'Price': [10,15,5,5,10,15,15,5]

}

df = pd.DataFrame(boxes, columns= ['Color','Shape','Price'])

select\_color = df.loc[df['Color'] == 'Green']

print (select\_color)

Once you run the code, you’ll get the rows where the color is green:

Color Shape Price

0 Green Rectangle 10

1 Green Rectangle 15

2 Green Square 5

## Additional Examples of Selecting Rows from Pandas DataFrame

Let’s now review additional examples to get a better sense of selecting rows from Pandas DataFrame.

### Example 1: Select rows where the price is equal or greater than 10

To get all the rows where the price is equal or greater than 10, you’ll need to apply this condition:

**df.loc[df[‘Price’] >= 10]**

And this is the complete Python code:

import pandas as pd

boxes = {'Color': ['Green','Green','Green','Blue','Blue','Red','Red','Red'],

'Shape': ['Rectangle','Rectangle','Square','Rectangle','Square','Square','Square','Rectangle'],

'Price': [10,15,5,5,10,15,15,5]

}

df = pd.DataFrame(boxes, columns= ['Color','Shape','Price'])

select\_price = df.loc[df['Price'] >= 10]

print (select\_price)

Run the code, and you’ll get all the rows where the price is equal or greater than 10:

Color Shape Price

0 Green Rectangle 10

1 Green Rectangle 15

4 Blue Square 10

5 Red Square 15

6 Red Square 15

### Example 2: Select rows where the color is green AND the shape is rectangle

Now the goal is to select rows based on two conditions:

* Color is green; and
* Shape is rectangle

You may then use the **&** symbol to apply multiple conditions. In our example, the code would look like this:

**df.loc[(df[‘Color’] == ‘Green’) & (df[‘Shape’] == ‘Rectangle’)]**

Putting everything together:

import pandas as pd

boxes = {'Color': ['Green','Green','Green','Blue','Blue','Red','Red','Red'],

'Shape': ['Rectangle','Rectangle','Square','Rectangle','Square','Square','Square','Rectangle'],

'Price': [10,15,5,5,10,15,15,5]

}

df = pd.DataFrame(boxes, columns= ['Color','Shape','Price'])

color\_and\_shape = df.loc[(df['Color'] == 'Green') & (df['Shape'] == 'Rectangle')]

print (color\_and\_shape)

Run the code and you’ll get the rows with the green color and rectangle shape:

Color Shape Price

0 Green Rectangle 10

1 Green Rectangle 15

### Example 3: Select rows where the color is green OR the shape is rectangle

You can also select the rows based on one condition or another. For instance, you can select the rows if the color is green or the shape is rectangle.

To achieve this goal, you can use the | symbol as follows:

**df.loc[(df[‘Color’] == ‘Green’) | (df[‘Shape’] == ‘Rectangle’)]**

And here is the complete Python code:

import pandas as pd

boxes = {'Color': ['Green','Green','Green','Blue','Blue','Red','Red','Red'],

'Shape': ['Rectangle','Rectangle','Square','Rectangle','Square','Square','Square','Rectangle'],

'Price': [10,15,5,5,10,15,15,5]

}

df = pd.DataFrame(boxes, columns= ['Color','Shape','Price'])

color\_or\_shape = df.loc[(df['Color'] == 'Green') | (df['Shape'] == 'Rectangle')]

print (color\_or\_shape)

Here is the result, where the color is green or the shape is rectangle:

Color Shape Price

0 Green Rectangle 10

1 Green Rectangle 15

2 Green Square 5

3 Blue Rectangle 5

7 Red Rectangle 5

### Example 4: Select rows where the price is not equal to 15

You can use the combination of symbols **!=** to select the rows where the price is not equal to 15:

**df.loc[df[‘Price’] != 15]**

import pandas as pd

boxes = {'Color': ['Green','Green','Green','Blue','Blue','Red','Red','Red'],

'Shape': ['Rectangle','Rectangle','Square','Rectangle','Square','Square','Square','Rectangle'],

'Price': [10,15,5,5,10,15,15,5]

}

df = pd.DataFrame(boxes, columns= ['Color','Shape','Price'])

not\_eqaul\_to = df.loc[df['Price'] != 15]

print (not\_eqaul\_to)

Once you run the code, you’ll get all the rows where the price is not equal to 15:

Color Shape Price

0 Green Rectangle 10

2 Green Square 5

3 Blue Rectangle 5

4 Blue Square 10

7 Red Rectangle 5

**Statistical Language - Measures of Central Tendency**

Calculate the average, variance and standard deviation in Python using NumPy

Difficulty Level : Basic

Last Updated : 08 Oct, 2021

Numpy in Python is a general-purpose array-processing package. It provides a high-performance multidimensional array object and tools for working with these arrays. It is the fundamental package for scientific computing with Python. Numpy provides very easy methods to calculate the average, variance, and standard deviation.

Average

Average a number expressing the central or typical value in a set of data, in particular the mode, median, or (most commonly) the mean, which is calculated by dividing the sum of the values in the set by their number. The basic formula for the average of n numbers x1, x2, ……xn is

A = (x\_1 + x\_2 ........ + x\_n)/ n

Example:

Suppose there are 8 data points,

2,\ 4,\ 4,\ 4,\ 5,\ 5,\ 7,\ 9

The average of these 8 data points is,

A = \frac{2 + 4 + 4 + 4 + 5 + 5 + 7 + 9}{8} = 5

Average in Python Using Numpy:

One can calculate the average by using numpy.average() function in python.

Syntax:

numpy.average(a, axis=None, weights=None, returned=False)

Parameters:

a: Array containing data to be averaged

axis: Axis or axes along which to average a

weights: An array of weights associated with the values in a

returned: Default is False. If True, the tuple is returned, otherwise only the average is returned

Example 1:

# Python program to get average of a list

# Importing the NumPy module

import numpy as np

# Taking a list of elements

list = [2, 4, 4, 4, 5, 5, 7, 9]

# Calculating average using average()

print(np.average(list))

Output:

5.0

Example 2:

# Python program to get average of a list

# Importing the NumPy module

import numpy as np

# Taking a list of elements

list = [2, 40, 2, 502, 177, 7, 9]

# Calculating average using average()

print(np.average(list))

Output:

105.57142857142857

Variance

Variance is the sum of squares of differences between all numbers and means. The mathematical formula for variance is as follows,

Formula: \sigma^{2}= \frac { \sum\_{i=1}^{N} (x\_{i}-\mu)^{2}}{N}

Where,

? is Mean,

N is the total number of elements or frequency of distribution.

Example:

Let’s consider the same dataset that we have taken in average. First, calculate the deviations of each data point from the mean, and square the result of each,

\begin{array}{lll} (2-5)^2 = (-3)^2 = 9 && (5-5)^2 = 0^2 = 0 \\ (4-5)^2 = (-1)^2 = 1 && (5-5)^2 = 0^2 = 0 \\ (4-5)^2 = (-1)^2 = 1 && (7-5)^2 = 2^2 = 4 \\ (4-5)^2 = (-1)^2 = 1 && (9-5)^2 = 4^2 = 16. \\ \end{array}

variance = \frac{9 + 1 + 1 + 1 + 0 + 0 + 4 + 16}{8} = 4

Variance in Python Using Numpy:

One can calculate the variance by using numpy.var() function in python.

Syntax:

numpy.var(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>)

Parameters:

a: Array containing data to be averaged

axis: Axis or axes along which to average a

dtype: Type to use in computing the variance.

out: Alternate output array in which to place the result.

ddof: Delta Degrees of Freedom

keepdims: If this is set to True, the axes which are reduced are left in the result as dimensions with size one

Example 1:

# Python program to get variance of a list

# Importing the NumPy module

import numpy as np

# Taking a list of elements

list = [2, 4, 4, 4, 5, 5, 7, 9]

# Calculating variance using var()

print(np.var(list))

Output:

4.0

Example 2:

# Python program to get variance of a list

# Importing the NumPy module

import numpy as np

# Taking a list of elements

list = [212, 231, 234, 564, 235]

# Calculating variance using var()

print(np.var(list))

Output:

18133.359999999997

Standard Deviation

Standard Deviation is the square root of variance. It is a measure of the extent to which data varies from the mean. The mathematical formula for calculating standard deviation is as follows,

Standard Deviation = \sqrt{ variance }

Example:

Standard Deviation for the above data,

Standard Deviation = \sqrt{ 4 } = 2

Standard Deviation in Python Using Numpy:

One can calculate the standard deviation by using numpy.std() function in python.

Syntax:

numpy.std(a, axis=None, dtype=None, out=None, ddof=0, keepdims=<no value>)

Parameters:

a: Array containing data to be averaged

axis: Axis or axes along which to average a

dtype: Type to use in computing the variance.

out: Alternate output array in which to place the result.

ddof: Delta Degrees of Freedom

keepdims: If this is set to True, the axes which are reduced are left in the result as dimensions with size one

Example 1:

# Python program to get

# standard deviation of a list

# Importing the NumPy module

import numpy as np

# Taking a list of elements

list = [2, 4, 4, 4, 5, 5, 7, 9]

# Calculating standard

# deviation using var()

print(np.std(list))

Output:

2.0

Example 2:

# Python program to get

# standard deviation of a list

# Importing the NumPy module

import numpy as np

# Taking a list of elements

list = [290, 124, 127, 899]

# Calculating standard

# deviation using var()

print(np.std(list))

Output:

318.35750344541907

## **Pandas head**

**Pandas** **DataFrame** **head()** method returns the top n rows of a **DataFrame** or **Series** where n is a user input value. The **head()** function is used to get the first n rows. It is helpful for quickly testing if your object has the right type of data in it. For negative values of n, the head() function returns all rows except the last n rows, equivalent to df[:-n].

### **Syntax**

DataFrame.head(n=5) (n=5 is default we can set any value)

### **Parameters**

The head() method in python contains only one parameter, n. It is an optional parameter. By setting it, we fix the number of rows we want from the **[DataFrame](https://appdividend.com/2019/01/17/python-pandas-dataframe-tutorial-data-structure-in-pandas/" \t "_blank)**.

### **Return Value**

The head() function returns n rows from the **DataFrame**.

### **Example**

**Write a program to show the working of the head().**

PLAY

UNMUTE

Loaded: 1.01%

FULLSCREEN

**import** pandas **as** pd

**import** numpy **as** np

data\_set = pd.DataFrame({'Name': ['Rohit', 'Mohit', 'Shubh', 'Pranav', 'Shivam', 'Prince'],

'Class': ['10', '09', '11', '12', '05', '07']})

print(data\_set.head(5))

#### **Output**

Name Class

0 Rohit 10

1 Mohit 09

2 Shubh 11

3 Pranav 12

4 Shivam 05

Here we can see that we have created a [DataFrame](https://appdividend.com/2019/01/17/python-pandas-dataframe-tutorial-data-structure-in-pandas/" \t "_blank) data\_set, which holds the values as names of 6 students and their respective classes in which they study.

Suppose we want to extract the data of only the top 5 students and not all the students. When this problem arises, we can use the head() method, defined in the Pandas library, to extract the top n rows of a dataset.

#### **Write a program to use the head() function when the DataFrame consists of 5 columns.**

import pandas as pd

import numpy as np

data\_frame = pd.DataFrame({'Name': ['Rohit', 'Mohit', 'Shubh', 'Pranav', 'Shivam', 'Prince'],

'Class': ['10', '09', '11', '12', '05', '07'], 'Roll no': ['25', '37', '48', '47', '46', '35'], 'Fav Subject': ['C++', 'Python', 'Kotlin', 'C', 'Java', 'C#'], 'Favourite Sports': ['Football', 'Basketball', 'Hockey', 'Cricket', 'Handball', 'Soccer']})

**print**("DataFrame::\n")

**print**(data\_frame)

**print**("\n")

**print**("Top 3 students::")

**print**("\n")

**print**(data\_frame.head(3))

#### **Output**

DataFrame::

Name Class Roll no Fav Subject Favourite Sports

0 Rohit 10 25 C++ Football

1 Mohit 09 37 Python Basketball

2 Shubh 11 48 Kotlin Hockey

3 Pranav 12 47 C Cricket

4 Shivam 05 46 Java Handball

5 Prince 07 35 C*# Soccer*

Top 3 students::

Name Class Roll no Fav Subject Favourite Sports

0 Rohit 10 25 C++ Football

1 Mohit 09 37 Python Basketball

2 Shubh 11 48 Kotlin Hockey

Here we can see five columns in the **DataFrame,** and with the help of the head function, we are showing the data of the top 3 students.

## **Passing no arguments to head() Function**

If you don’t pass any argument to the **DataFrame head()** function, you will get the default first five rows in return.

import pandas as pd

import numpy as np

data\_frame = pd.DataFrame({'Name': ['Rohit', 'Mohit', 'Shubh', 'Pranav', 'Shivam', 'Prince'],

'Class': ['10', '09', '11', '12', '05', '07'], 'Roll no': ['25', '37', '48', '47', '46', '35'], 'Fav Subject': ['C++', 'Python', 'Kotlin', 'C', 'Java', 'C#'], 'Favourite Sports': ['Football', 'Basketball', 'Hockey', 'Cricket', 'Handball', 'Soccer']})

**print**("DataFrame::\n")

**print**(data\_frame)

**print**("\n")

**print**(data\_frame.head())

#### **Output**

DataFrame::

Name Class Roll no Fav Subject Favourite Sports

0 Rohit 10 25 C++ Football

1 Mohit 09 37 Python Basketball

2 Shubh 11 48 Kotlin Hockey

3 Pranav 12 47 C Cricket

4 Shivam 05 46 Java Handball

5 Prince 07 35 C*# Soccer*

Name Class Roll no Fav Subject Favourite Sports

0 Rohit 10 25 C++ Football

1 Mohit 09 37 Python Basketball

2 Shubh 11 48 Kotlin Hockey

3 Pranav 12 47 C Cricket

4 Shivam 05 46 Java Handball

## **Passing negative arguments to head() Function**

Let’s pass the negative arguments to the head() function and see the result.

import pandas as pd

import numpy as np

data\_frame = pd.DataFrame({'Name': ['Rohit', 'Mohit', 'Shubh', 'Pranav', 'Shivam', 'Prince'],

'Class': ['10', '09', '11', '12', '05', '07'], 'Roll no': ['25', '37', '48', '47', '46', '35'], 'Fav Subject': ['C++', 'Python', 'Kotlin', 'C', 'Java', 'C#'], 'Favourite Sports': ['Football', 'Basketball', 'Hockey', 'Cricket', 'Handball', 'Soccer']})

**print**("DataFrame::\n")

**print**(data\_frame)

**print**("\n")

**print**(data\_frame.head(-4))

#### **Output**

DataFrame::

Name Class Roll no Fav Subject Favourite Sports

0 Rohit 10 25 C++ Football

1 Mohit 09 37 Python Basketball

2 Shubh 11 48 Kotlin Hockey

3 Pranav 12 47 C Cricket

4 Shivam 05 46 Java Handball

5 Prince 07 35 C*# Soccer*

Name Class Roll no Fav Subject Favourite Sports

0 Rohit 10 25 C++ Football

1 Mohit 09 37 Python Basketball

From the output, you can see that it returns the first two rows. That means it won’t count the last four rows. So, if you have passed **-5,** it won’t count the last five rows and returns the first row.

## **Pandas Series head()**

**Pandas Series head()** method is called on [series](https://appdividend.com/2019/01/16/python-pandas-series-tutorial-data-structure-example/) with custom input of n parameter to return the top n rows of the series.

**import** pandas **as** pd

**import** numpy **as** np

df = pd.DataFrame({'Name': ['Rohit', 'Mohit', 'Shubh', 'Pranav', 'Shivam', 'Prince'],

'Class': ['10', '09', '11', '12', '05', '07'],

'Roll no': ['25', '37', '48', '47', '46', '35'],

'Fav Subject': ['C++', 'Python', 'Kotlin', 'C', 'Java', 'C#'],

'Favourite Sports': ['Football', 'Basketball', 'Hockey', 'Cricket', 'Handball', 'Soccer']})

series = df['Favourite Sports']

top5 = series.head()

print(top5)

#### **Output**

0 **Football**

1 **Basketball**

2 **Hockey**

3 **Cricket**

4 **Handball**

**Name**: **Favourite** **Sports**, d**type**: object

## **DataFrame - tail() function**

The tail() function is used to get the last n rows.

This function returns last n rows from the object based on position. It is useful for quickly verifying data, for example, after sorting or appending rows.

**Syntax:**

DataFrame.tail(self, n=5)

**Parameters:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Name** | **Description** | **Type/Default Value** | **Required / Optional** |
| n | Number of rows to select. | int Default Value: 5 | Required |

**Returns:**type of caller  
The last n rows of the caller object.

**Example:**

## **DataFrame - loc property**

The loc property is used to access a group of rows and columns by label(s) or a boolean array.

.loc[] is primarily label based, but may also be used with a boolean array.

Allowed inputs are:

* A single label, e.g. 5 or 'a', (note that 5 is interpreted as a label of the index, and never as an integer position along the index).
* A list or array of labels, e.g. ['a', 'b', 'c'].
* A slice object with labels, e.g. 'a':'f'.
* A boolean array of the same length as the axis being sliced, e.g. [True, False, True].
* A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above)

**Syntax:**

DataFrame.loc

**Raises:**KeyError  
when any items are not found

**Example:**

Dataframe.iloc[] method is used when the index label of a data frame is something other than numeric series of 0, 1, 2, 3….n or in case the user doesn’t know the index label. Rows can be extracted using an imaginary index position which isn’t visible in the data frame.

Syntax: pandas.DataFrame.iloc[]

Parameters:

Index Position: Index position of rows in integer or list of integer.

Return type: Data frame or Series depending on parameters

To download the CSV used in code, click here.

Example #1: Extracting single row and comparing with .loc[] In this example, same index number row is extracted by both .iloc[] and.loc[] method and compared. Since the index column by default is numeric, hence the index label will also be integers.

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("nba.csv")

# retrieving rows by loc method

row1 = data.loc[3]

# retrieving rows by iloc method

row2 = data.iloc[3]

# checking if values are equal

row1 == row2

Output:

As shown in the output image, the results returned by both methods are the same.

Example #2: Extracting multiple rows with index In this example, multiple rows are extracted, first by passing a list and then by passing integers to extract rows between that range. After that, both the values are compared.

# importing pandas package

import pandas as pd

# making data frame from csv file

data = pd.read\_csv("nba.csv")

# retrieving rows by loc method

row1 = data.iloc[[4, 5, 6, 7]]

# retrieving rows by loc method

row2 = data.iloc[4:8]

# comparing values

row1 == row2

Output:

As shown in the output image, the results returned by both methods are the same. All values are True except values in the college column since those were NaN values.

## **DataFrame - values property**

The values property is used to get a Numpy representation of the DataFrame.

Only the values in the DataFrame will be returned, the axes labels will be removed.

**Syntax:**

DataFrame.values

**Returns:**numpy.ndarray  
The values of the DataFrame.

**Example:**

pandas.DataFrame.to\_numpy

**DataFrame.to\_numpy(*dtype=None*, *copy=False*, *na\_value=NoDefault.no\_default*)**[**[source]**](https://github.com/pandas-dev/pandas/blob/v1.4.2/pandas/core/frame.py#L1693-L1760)

Convert the DataFrame to a NumPy array.

By default, the dtype of the returned array will be the common NumPy dtype of all types in the DataFrame. For example, if the dtypes are float16 and float32, the results dtype will be float32. This may require copying data and coercing values, which may be expensive.

**Parameters**

**dtype*str or numpy.dtype, optional***

The dtype to pass to **numpy.asarray()**.

**copy*bool, default False***

Whether to ensure that the returned value is not a view on another array. Note that copy=False does not *ensure* that to\_numpy() is no-copy. Rather, copy=True ensure that a copy is made, even if not strictly necessary.

**na\_value*Any, optional***

The value to use for missing values. The default value depends on *dtype* and the dtypes of the DataFrame columns.

***New in version 1.1.0.***

**Returns**

**numpy.ndarray**

**See also**

[**Series.to\_numpy**](https://pandas.pydata.org/docs/reference/api/pandas.Series.to_numpy.html#pandas.Series.to_numpy)

Similar method for Series.

**Examples**

**>>>** pd.DataFrame({"A": [1, 2], "B": [3, 4]}).to\_numpy()

array([[1, 3],

[2, 4]])

With heterogeneous data, the lowest common type will have to be used.

**>>>** df = pd.DataFrame({"A": [1, 2], "B": [3.0, 4.5]})

**>>>** df.to\_numpy()

array([[1. , 3. ],

[2. , 4.5]])

For a mix of numeric and non-numeric types, the output array will have object dtype.

**>>>** df['C'] = pd.date\_range('2000', periods=2)

**>>>** df.to\_numpy()

array([[1, 3.0, Timestamp('2000-01-01 00:00:00')],

[2, 4.5, Timestamp('2000-01-02 00:00:00')]], dtype=object)

# Pandas DataFrame describe() Method

[❮ DataFrame Reference](https://www.w3schools.com/python/pandas/pandas_ref_dataframe.asp)

### **Example**

Multiply the values for each row with the values from the previous row:

import pandas as pd  
  
data = [[10, 18, 11], [13, 15, 8], [9, 20, 3]]  
  
df = pd.DataFrame(data)  
  
print(df.describe())

[Try it Yourself »](https://www.w3schools.com/python/pandas/trypandas.asp?filename=demo_ref_df_describe)

## **Definition and Usage**

The describe() method returns description of the data in the DataFrame.

If the DataFrame contains numerical data, the description contains these information for each column:

count - The number of not-empty values.  
mean - The average (mean) value.  
std - The standard deviation.  
min - the minimum value.  
25% - The 25% percentile\*.  
50% - The 50% percentile\*.  
75% - The 75% percentile\*.  
max - the maximum value.

\*Percentile meaning: how many of the values are less than the given percentile. Read more about percentiles in our [Machine Learning Percentile](https://www.w3schools.com/python/python_ml_percentile.asp) chapter.

## **Syntax**

dataframe.describe(percentiles, include, exclude, datetime\_is\_numeric)

## **Parameters**

The percentile, include, exclude, datetime\_is\_numeric parameters are [keyword arguments](https://www.w3schools.com/python/gloss_python_function_keyword_arguments.asp).

|  |  |  |
| --- | --- | --- |
| **Parameter** | **Value** | **Description** |
| percentile | numbers between:0 and 1 | Optional, a list of percentiles to include in the result, default is : [.25, .50, .75]. |
| include | None 'all' datatypes | Optional, a list of the data types to allow in the result |
| exclude | None 'all' datatypes | Optional, a list of the data types to disallow in the result |
| datetime\_is\_numeric | True False | Optional, default False. Set to True to treat datetime data as numeric |

## **Return Value**

A [DataFrame](https://www.w3schools.com/python/pandas/pandas_dataframes.asp) object with statistics for each row.