Different Ways of Loading Data using Python

Prepared By
K.Lakshmi Kanth
Assistant Professor
RGUKT Nuzvid

Loading Data in Python Using 5 different methods

1.Manually loading a file:

This is the first, most **popular,** and **least recommended way to** load data **frame as it requires many code parts to** read one tuple from the data frame. This way comes into the picture when the dataset **doesn't have any particular pattern to identify** or a **specific pattern.**

1.np. load txt:

One of the **NumPy methods** for loading different types of data though it is only supported when we have data in a specific format, i.e., **pattern recognizable,** unlike the manual way of reading the dataset.

3. Using np.GenFromTxt:

This is another NumPy way to read the data, but this time it is much better than the **np. load txt()** method recognizes the column header's presence on its own, which the previous one cannot follow. Along with that, it can also detect the **right data type** for each column.

4. Using PD.read_csv:

Here is the most recommended and widely used method for **reading, writing, and manipulating** the dataset. It only deals with **CSV formatted** data, but the support of various parameters makes **it a gold mine for data analysts** to work with different sorts of data (they should have a specific format).

5.Using pickle:

We will also use **a pickle** to read the dataset present in the **binary format**. So far, we chose pickle only to save the model.

Importing the Required Python Libraries

The first thing for using the python supported libraries we first need to import them to establish the environment and configuration. Here are the following modules that we will import.

- **Numpy:** Numpy is mainly used to perform mathematical calculations though here we will use it to access the methods for our needs. They are **np. load txt()** and **np. GenFromTxt()**.

- **Pandas:** For **converting** the unsorted data into the **correctly formatted** dataset, we can later manipulate and draw some insights.

- **Pickle:** We need to import the same independently for using the pickle module.

```
import numpy as np
import pickle
import pandas as pd
file_dir = “load_dataset_blog.csv”
```

So we have imported the modules we might need and stored the dataset’s path in a variable so that we don’t have to write it again and again.

Importing Files

To access any information stored in the files, you must first make it import it into Python. Otherwise, Python would not recognize and allow us to use undefined items. It is often a reality that you will have to work to work with different file formats.

Importing Files (Reading Text Files)

Text files are one of the most common file formats to store data. In Python, you can use the open() function to read the .txt files.

Notice that the open() function takes two input parameters: file path (or file name if the file is in the current working directory) and the file access mode. There are many modes for opening a file:

- **open('path','r')**: opens a file in read mode
- **open('path',w')**: opens or creates a text file in write mode
- **open('path',a')**: opens a file in append mode
- **open('path','r+')**: opens a file in both read and write mode
- **open('path',w+')**: opens a file in both read and write mode
- **open('path',a+')**: opens a file in both read and write mode

```
text_file = open('path-to-file','r')    # Full path to the txt file, or simply the file name
                                        # if the file is in your current working directory
```

After opening the file with the **read mode**, you can also use the following function to access or examine the Information stored in the file:

- **.read(<number>)**: This function reads the complete information from the file unless a number is specified. Otherwise, it will read the first n bytes from the text files.
- **.readline(<number>)**: This function reads the information from the file but not more than one line of information unless a number is specified. Otherwise, it will read the first n bytes from the text files. It is usually used in loops
- **.readlines()** – This function reads the complete information in the file and prints them as well in a list format

```
text_file.read()    # Read complete information from the file
```

np.loadtxt

This method is widely used for simple data arrays requiring very minimal formatting, i.e., for simple values where such changes are unnecessary.

We can also use the np. save_txt function to save the data.

```
import numpy as np
d1 = np.loadtxt(filename, skiprows=1, delimiter=",")
```

- `np.loadtxt()` loads the data from a text file into a NumPy array.
- `filename`: The name of the file from which data is being read.
- `skiprows=1`: Skips the first row of the file (often used when there is a header).
- `delimiter=","`: Specifies that the values in the file are **comma-separated** (CSV format).

```
print(d1.dtype)  # Prints the data type of the loaded array
print(d1[:5, :])  # Prints the first 5 rows of the array
```

- `d1.dtype`: Displays the data type of the array elements.
- `d1[:5, :]`: Slices the array to **show the first 5 rows**, with all columns.

```
np.savetxt("output.csv", d1, delimiter=",")
```

- Saves `d1` to **output.csv** using a **comma** as the separator.

Purpose of `np.genfromtxt`

`numpy.genfromtxt()` is a more flexible function compared to `numpy.loadtxt()`.

It allows for:

- Handling missing values (which `loadtxt()` does not support).
- Auto-detection of data types (whereas `loadtxt()` assumes a single type).
- Named columns (so you can access data using column names).

```
import numpy as np
d2 = np.genfromtxt(filename, delimiter=",", names=True,
dtype=None)
```

`np.genfromtxt()`: Loads data from a text file into a structured NumPy array.

`filename`: The name of the file (path to the dataset).

`delimiter=","`: Specifies that values are **comma-separated** (CSV format).

`names=True`: This tells NumPy to **use the first row as column names**, allowing for structured data access.

`dtype=None`: NumPy will **automatically detect** the data type for each column.

```
print(d2.dtype)  # Prints the detected data types of columns
print(d2[:5])    # Prints the first 5 rows of the loaded
data
```

- `d2.dtype`: Shows the **structured data types** (e.g., integers, floats, strings).
- `d2[:5]`: Displays **the first 5 rows** of the dataset.

Importing Files (Reading CVS Files)

The CSV (or Comma Separated Value) files are also one of the most common file formats that data scientists work with. The name "Comma Separated" means that those files use a "," as a delimiter to separate the values.

We usually use the Pandas library to read CSV files:

```
import pandas as pd

dataframe = pd.read_csv('path-to-file')    # Full path to the txt file, or simply the file name
                                           # if the file is in your current working directory
```

Sometimes, you may have the dataset in .csv file format but has the data items separated by a delimiter other than a comma. Possible delimiters include semicolons, colons, tab spaces, vertical bars, etc. In such cases, we need to use the 'sep' parameter inside the read.csv() function to read the values properly

```
dataframe = pd.read_csv('path-to-file', sep = ';')
```

Notice that we use '|' to represent the vertical-bar separator, and we use 't' to represent the tab-separator.

Importing Files (Reading Excel Files)

Pandas library also has a function, read_excel(), to read Excel files:

```
dataframe = pd.read_excel('path-to-file')    # Full path to the txt file, or simply the file name
                                             # if the file is in your current working directory
```

It is often the case where an Excel file can contain multiple sheets. You can read data from any sheet by providing its name in the sheet_name parameter in the read_excel() function:

```
dataframe = pd.read_excel('path-to-file', sheet_name='sheet-name')
```

You can get the list of dataframe headers using the columns property of the dataframe object.

```
print(dataframe.columns.ravel())
```

Sometimes, you might want to use one column of data for Analysis. To do this, you can get the column data and convert it into a list of values.

```
print(dataframe['column-name'].tolist())
```

Importing Files (Extracting from Zip Files)

To open a ZIP folder, you will first need to import the zip file library in Python, which is in the standard library as well, so no additional installation is needed.

```
from zipfile import ZipFile    # Import zipfile



file = 'path-to-file'          # Full path to the zip file, or simply the file name
                               # if the file is in your current working directory
zip_file = ZipFile(file, 'r')  # Read zipfile and extract contents
zip_file.printdir()            # List files in the zip files
zip_file.extractall()          # Extract files in the zip file
```

Once you run the above code, you can see the extracted files are in the same folder as your script

Importing Files (Working with JSON Files)

JSON (JavaScript Object Notation) files store data within square brackets {}, which is similar to how a dictionary stores information in Python. The major benefit of JSON files is that they are language-independent, meaning they can be used with any programming language.

Python has a built-in json module to read JSON files. The read function is json.load() function, which takes a JSON file and returns a JSON dictionary.

```
import json


file = open('path-to-file')     # Full path to the json file, or simply the file name
                                # if the firle is in your current working directory
json_file = json.load(file)     # Returns JSON object as a dictionary
```

Once you have executed the code above, you can convert it into a Pandas dataframe using the pandas.DataFrame() function:

```
dataframe = pd.DataFrame(json_file)
```

You can also load the JSON file directly into a dataframe using the pandas.read_json() function:

```
dataframe = pd.read_json('path-to-file')
```

Now, the JSON file can be manipulated just like a Pandas Dataframe

Importing Files (Working with Pickle Files)

Pickle files store the serialized form of Python objects, including lists, dictionaries, tuples, etc. They are converted to a character stream before being stored in the Pickle files. Storing those Python objects as Pickle files allows you to continue working with the objects later on. Pickle files are particularly useful when you are training your machine learning model and want to save them for making predictions later.

To open a Pickle file, you must de-serialize them first. Fortunately, there is a built-in Python package, pickle, to help us with that, so you can use the pickle.load() function in the pickle module to load a Pickle file.

Notice that, to read a binary file, you need to provide an 'rb' parameter when you open the pickle file with Python's open() function.

```
import pickle

file = open('path-to-file','rb')     # Full path to the Pickle file, or simply the file name
                                     # if the file is in your current working directory
pickle_file = pickle.load(file)      # Load Pickle file
```

Once you have executed the code above, you can also convert it into a Pandas dataframe using the pandas.DataFrame() function:

```
dataframe = pd.DataFrame(pickle_file)
```

Importing Files (Web Scraping)

Web Scraping refers to the task of extracting data from a website. Python contains powerful modules to retrieve data from websites for future analysis.

The get() function in the requests package takes a URL as its parameter and returns the HTML response as its output. Then, Python packages the scraping request from the website, sends the request to the server, receives the HTML response, and stores it in a Python object.

```
import requests


url = "path-to-website"
response = requests.get(url)     # Generate response object
text = response.text             # Return the HTML of webpage as string
print(text)
```