Beginner's Tutorial on the Pandas Python Library

r 03, 2018 •

2 e-pandas-python

ıar goofs, and more. Try

Pa ge that provides
nu ackage comes
wi used for many
dif has a variety of
me alysis, which
co science and
ma

ıtages of the Pandas

ıt is suitable for data

aFrame data

methods for

to perform

amless manner. It

formats such as

es not come with

party module, you

must install it.

The nice thing about Python is that it comes bundled with a tool called *pip* that can be used for the installation of Pandas. The do the installation, you need to run the following command:

```
$ pip install pandas
```

If you have installed Anaconda (https://anaconda.org/anaconda/python) on your system, just run the following command to install Pandas:

```
$ conda install pandas
```

It is highly recommended that you install the latest version of the Pandas package. However, if you want to install an older version you can specify it by running the `conda install` command as follows:

```
$ conda install pandas=0.23.4
```

Pandas Data Structures

Pandas has two main data structures for data storage:

1. Series
2. DataFrame

Series

A series is similar to a one-dimensional array. It can store data of any type. The values of a Pandas `Series` are mutable but the size of a `Series` is immutable and cannot be changed.

The first element in the series is assigned the index `0`, while the last element is at index `N-1`, where `N` is the total number of elements in the series.

To create a Pandas `Series`, we must first import the Pandas package via the Python's `import` command:

```
import pandas as pd
```

To create the `Series`, we invoke the `pd.Series()` method and pass an array, as shown below:


```
series1 = pd.Series([1,2,3,4])
```

Next, run the `print` statement to display the contents of the `Series`:

```
print(series1)
```

Output:

```
0    1
1    2
2    3
3    4
dtype: int64
```

You can see that we have two columns, the first one with numbers starting from index `0` and the second one with the elements that were added to the series.

The first column denotes the indexes for the elements.

However, you may get an error when you try to display the `Series` . The major cause of this error is that Pandas looks for the amount of information to display, therefore you should provide sys output information.

You can solve the error by executing the code as follows:

```
import pandas as pd
import sys

sys.__stdout__ = sys.stdout

series1 = pd.Series([1,2,3,4])
print(series1)
```

A `Series` may also be created from a numpy (https://www.numpy.org/) array. Let us create a numpy array then convert it into a Pandas `Series` :

```
import pandas as pd
import numpy as np
import sys

sys.__stdout__ = sys.stdout

fruits = np.array(['apple','orange','mango','pear'])
series2 = pd.Series(fruits)
print(series2)
```

Output:

```
0     apple
1    orange
2     mango
3      pear
dtype: object
```

We start by importing the necessary libraries, including `numpy` . Next, we called the numpy's `array()` function to create an array of fruits. We then use Pandas `Series()` function and pass it the array that we want to convert into a series. Finally, we call the `print()` function to display the `Series` .

DataFrame

The Pandas DataFrame can be seen as a table. It organizes data into rows and columns, making it a two-dimensional data structure. Potentially, the columns are of a different type and the size of the DataFrame is mutable, and hence can be modified.

To create a DataFrame, you can choose to start from scratch or convert other data structures like Numpy arrays into a DataFrame. Here is how you can create a DataFrame from scratch:

```
import pandas as pd
df = pd.DataFrame({
    "Column1": [1, 4, 8, 7, 9],
    "Column2": ['a', 'column', 'with', 'a', 'string'],
    "Column3": [1.23, 23.5, 45.6, 32.1234, 89.453],
    "Column4": [True, False, True, False, True]
})
print(df)
```

Output:

```
   Column1 Column2  Column3  Column4
0        1       a   1.2300     True
1        4  column  23.5000    False
2        8    with  45.6000     True
3        7       a  32.1234    False
4        9  string  89.4530     True
```


In this example we have created a DataFrame named `df` . The first column of the DataFrame has integer values. The second column has a string, the third column has floating point values, while the fourth column has boolean values.

The statement `print(df)` will display the contents of the DataFrame to us via the console, allowing us to inspect and verify its contents.

However, when displaying the DataFrame, you may have noticed that there is an additional column at the start of the table, with its elements beginning at 0. This column is created automatically and it marks the indexes of the rows.

To create a DataFrame, we must invoke the `pd.DataFrame()` method as shown in the above example.

It is possible for us to create a DataFrame from a list or even a set of lists. We only have to call the `pd.DataFrame()` method and then pass it the list variable as its only argument.

Consider the following example:

```
import pandas as pd
mylist = [4, 8, 12, 16, 20]
df = pd.DataFrame(mylist)
print(df)
```

Output:

```
   0
0  4
1  8
2  12
3  16
4  20
```

in this example we created a list named `mylist` with a sequence of 5 integers. We then called the `DataFrame()` method and passed the name of the list to it as the argument. This is where the conversion of the list to a DataFrame happened.

We have then printed out the contents of the DataFrame. The DataFrame has a default column showing indexes, with the first element being at index 0 and the last one at index `N-1`, where N is the total number of elements in the DataFrame.

Here is another example:

```
import pandas as pd
items = [['Phone', 2000], ['TV', 1500], ['Radio', 800]]
df = pd.DataFrame(items, columns=['Item', 'Price'], dtype=flo
at)
print(df)
```

Output:


```
   Item   Price
0  Phone  2000.0
1     TV  1500.0
2  Radio   800.0
```

Here we have created a list named `items` with a set of 3 items. For each item, we have a name and price. The list is then passed to the `DataFrame()` method in order to convert it into a `DataFrame` object.

In this example the names of the columns for the DataFrame have been specified as well. The numeric values have also been converted into floating point values since we specified the `dtype` argument as "float".

To get a summary of this item's data, we can call the `describe()` function on the DataFrame variable, that is, `df`:

```
df.describe()
```

Output:


```
       Price
count      3.000000
mean    1433.333333
std      602.771377
min      800.000000
25%     1150.000000
50%     1500.000000
75%     1750.000000
max     2000.000000
```

The `describe()` function returns some common statistical details of the data, including the mean, standard deviation, minimum element, maximum element, and some other details. This is a great way to get a snapshot of the data you're working with if the dataset is relatively unknown to you. It could also be a good way to quickly compare two separate datasets of similar data.

Importing Data

Often times you'll need to use Pandas to analyze data that is stored in an Excel file or in a CSV file. This requires you to open and import the data from such sources into Pandas.

Luckily, Pandas provides us with numerous methods that we can use to load the data from such sources into a Pandas DataFrame.

Importing CSV Data

A CSV file, which stands for *comma separated value*, is simply a text file with values separated by a comma (,). Since this is a very well-known and often-used standard, we can use Pandas to read CSV files either in whole or in part.

For this example we will create a CSV file named *cars.csv*. The file should have the following data:

```
Number,Type,Capacity
SSD,Premio,1800
KCN,Fielder,1500
USG,Benz,2200
TCH,BMW,2000
KBQ,Range,3500
TBD,Premio,1800
KCP,Benz,2200
USD,Fielder,1500
UGB,BMW,2000
TBG,Range,3200
```


You can copy the data and paste in a text editor like Notepad, and then save it with the name *cars.csv* in the same directory as your Python scripts.

Pandas provides us with a method named `read_csv` that can be used for reading CSV values into a Pandas DataFrame. The method takes the path to the CSV file as the argument.

The following code is what we'll use to help us read the *cars.csv* file:

```
import pandas as pd
data = pd.read_csv('cars.csv')
print(data)
```


Output:

```
 Number     Type  Capacity
0    SSD   Premio      1800
1    KCN  Fielder      1500
2    USG     Benz      2200
3    TCH      BMW      2000
4    KBQ    Range      3500
5    TBD   Premio      1800
6    KCP     Benz      2200
7    USD  Fielder      1500
8    UGB      BMW      2000
9    TBG    Range      3200
```

In my case, I saved the CSV file in the same directory as the Python script, hence I simply passed the name of the file to the `read_csv` method and it knew to check the current working directory.

If you have saved your file in a different path, ensure you pass the correct path as the argument to the method. This can either be a relative path, like "../cars.csv", or an absolute path like "/Users/nicholas/data/cars.csv".

in some cases, you may have thousands of rows in your dataset. In such a case, it would be more helpful to you to print only the first few rows on the console rather than printing all the rows.

This can be done by calling the `head()` method on the DataFrame as shown below:


```
data.head()
```

For our data above, the above command returns only the first 5 rows of the dataset, allowing you to inspect a small sample of the data. This is shown below:

Output:

```
  Number     Type  Capacity
0    SSD   Premio      1800
1    KCN  Fielder      1500
2    USG     Benz      2200
3    TCH      BMW      2000
4    KBQ    Range      3500
```

The `loc()` method is a nice utility that helps us read only certain rows of a specific column in the dataset, as demonstrated in the following example:

```
import pandas as pd
data = pd.read_csv('cars.csv')

print (data.loc[[0, 4, 7], ['Type']])
```

Output:

```
  Type
0  Premio
4   Range
7  Fielder
```


Here we used the `loc()` method to only read the elements at indexes 0, 4, and 7 of the *Type* column.

At times Wwe may need to only read certain columns and not others. This can be done using the `loc()` method as well, shown below in this example:

```
import pandas as pd
data = pd.read_csv('cars.csv')

print (data.loc[:, ['Type', 'Capacity']])
```

Output:

```
  Type  Capacity
0   Premio      1800
1  Fielder      1500
2     Benz      2200
3      BMW      2000
4    Range      3500
5   Premio      1800
6     Benz      2200
7  Fielder      1500
8      BMW      2000
9    Range      3200
```


Here we used the `loc()` method to read all rows (the `:` part) of only two of our columns from the dataset, that is, the *Type* and *Capacity* columns, as specified in the argument.

Importing Excel Data

In addition to the `read_csv` method, Pandas also has the `read_excel` function that can be used for reading Excel data into a Pandas DataFrame. In this example, we will use an Excel file named *workers.xlsx* with details of workers in a company.

The following code can be used to load the contents of the Excel file into a Pandas DataFrame:

```
import pandas as pd
data = pd.read_excel('workers.xlsx')
print (data)
```

Output:

```
   ID    Name       Dept  Salary
0   1     John        ICT    3000
1   2     Kate    Finance    2500
2   3   Joseph         HR    3500
3   4   George        ICT    2500
4   5     Lucy      Legal    3200
5   6    David    Library    2000
6   7    James         HR    2000
7   8    Alice   Security    1500
8   9    Bosco    Kitchen    1000
9  10     Mike        ICT    3300
```

After calling the `read_excel` function we then passed the name of the file as the argument, which `read_excel` used to open/load the file and then parse the data. The `print()` function then helps us display the contents of the DataFrame, as we've done in past examples.

And just like with our CSV example, this function can be combined with the `loc()` method to help us read specific rows and columns from the Excel file.

For example:

```
import pandas as pd
data = pd.read_excel('workers.xlsx')

print (data.loc[[1,4,7],['Name','Salary']])
```

Output:

```
  Name  Salary
1  Kate    2500
4  Lucy    3200
7 Alice    1500
```


We have used the `loc()` method to retrieve the *Name* and *Salary* values of the elements at indexes 1, 4, and 7.

Pandas also allows us to read from two Excel sheets simultaneously. Suppose our previous data is in Sheet1, and we have some other data in Sheet2 of the same Excel file. The following code shows how we can read from the two sheets simultaneously:

```
import pandas as pd
with pd.ExcelFile('workers.xlsx') as x:
    s1 = pd.read_excel(x, 'Sheet1')
    s2 = pd.read_excel(x, 'Sheet2')

print("Sheet 1:")
print (s1)
print("")
print("Sheet 2:")
print (s2)
```

Output:

```
Sheet 1:
   ID    Name       Dept  Salary
0   1    John        ICT    3000
1   2    Kate    Finance    2500
2   3  Joseph         HR    3500
3   4  George        ICT    2500
4   5    Lucy      Legal    3200
5   6   David    Library    2000
6   7   James         HR    2000
7   8   Alice   Security    1500
8   9   Bosco    Kitchen    1000
9  10    Mike        ICT    3300

Sheet 2:
   ID    Name  Age  Retire
0   1    John   55    2023
1   2    Kate   45    2033
2   3  Joseph   55    2023
3   4  George   35    2043
4   5    Lucy   42    2036
5   6   David   50    2028
6   7   James   30    2048
7   8   Alice   24    2054
8   9   Bosco   33    2045
9  10    Mike   35    2043
```

What happened is that we combined the `read_excel()` function with the `ExcelFile` wrapper class. The variable `x` was created when calling the wrapper class and `with` Python keyword, which we use to temporarily open the file.

From the `ExcelFile` variable `x`, we have created two more variables, `s1` and `s2` to represent the contents that were read from the different sheets.

We then used `print` statements to view the contents of the two sheets in the console. The blank `print` statement, `print("")`, is only used to print a blank line between our sheet data.

Data Wrangling

Data wrangling is the process of processing data to prepare it for use in the next step. Examples of data wrangling processes include merging, grouping, and concatenation. This kind of manipulation is often needed in data science to get your data in to a form that works well with whatever analysis or algorithms that you're going to put it through.

Merging

The Pandas library allows us to join DataFrame objects via the `merge()` function. Let us create two DataFrames and demonstrate how to merge them.

Here is the first DataFrame, `df1`:

```
import pandas as pd

d = {
    'subject_id': ['1', '2', '3', '4', '5'],
    'student_name': ['John', 'Emily', 'Kate', 'Joseph', 'Denn
is']
}
df1 = pd.DataFrame(d, columns=['subject_id', 'student_name'])
print(df1)
```

Output:

```
subject_id student_name
0          1         John
1          2        Emily
2          3         Kate
3          4       Joseph
4          5       Dennis
```

Here is the code to create the second DataFrame, `df2`:

```
import pandas as pd

data = {
    'subject_id': ['4', '5', '6', '7', '8'],
    'student_name': ['Brian', 'William', 'Lilian', 'Grace',
'Caleb']
}
df2 = pd.DataFrame(data, columns=['subject_id', 'student_nam
e'])
print(df2)
```


Output:

```
subject_id student_name
0          4        Brian
1          5      William
2          6       Lilian
3          7        Grace
4          8        Caleb
```

We now need to merge the two DataFrames, that is, `df1` and `df2` along the values of `subject_id`. We simply call the `merge()` function as shown below:

```
pd.merge(df1, df2, on='subject_id')
```

Output:

```
  subject_id student_name_x student_name_y
0          4          Joseph           Brian
1          5          Dennis         William
```


What merging does is that it returns the rows from both DataFrames with the same value for the column you are using for the merge.

There are many other ways to use the `pd.merge` function that we won't be covering in this article, such as what data should be merged, how it should be merged, if it should be sorted, etc. For more information, check out the official documentation on the merge function (https://pandas.pydata.org/pandas-docs/stable/generated/pandas.DataFrame.merge.html).

Grouping

Grouping is the process of putting data into various categories. Here is a simple example:

```
# import pandas library
import pandas as pd
```



```
raw = {
    'Name': ['John', 'John', 'Grace', 'Grace', 'Benjamin', 'B
enjamin', 'Benjamin',
        'Benjamin', 'John', 'Alex', 'Alex', 'Alex'],
    'Position': [2, 1, 1, 4, 2, 4, 3, 1, 3, 2, 4, 3],
    'Year': [2009, 2010, 2009, 2010, 2010, 2010, 2011, 2012,
2011, 2013, 2013, 2012],
    'Marks':[408, 398, 422, 376, 401, 380, 396, 388, 356, 402
, 368, 378]
}
df = pd.DataFrame(raw)

group = df.groupby('Year')
print(group.get_group(2010))
```

Output:

```
   Marks      Name  Position  Year
1    398      John         1  2010
3    376     Grace         4  2010
5    380  Benjamin         4  2010
```

In this simple example, we have grouped the data by year, which in this case was 2010. We could have also grouped by any of the other columns, like "Name", "Position", etc.

Concatenation

Concatenation of data, which basically means to add one set of data to another, can be done by calling the `concat()` function.

Let us demonstrate how to concatenate DataFrames using our two previous Dataframes, that is, `df1` and `df2`, each with two columns, "subject_id" and "student_name":

```
print(pd.concat([df1, df2]))
```

Output:

```
subject_id student_name
0          1         John
1          2        Emily
2          3         Kate
3          4       Joseph
4          5       Dennis
0          4        Brian
1          5      William
2          6       Lilian
3          7        Grace
4          8        Caleb
```

Descriptive Statistics

As I briefly showed earlier, when we use the `describe()` function we get the descriptive statistics for numerical columns, but the character columns are excluded.

Let's first create a DataFrame showing student names and their scores in Math and English:

```
import pandas as pd

data = {
    'Name': ['John', 'Alice', 'Joseph', 'Alex'],
    'English': [64, 78, 68, 58],
    'Maths': [76, 54, 72, 64]
}

df = pd.DataFrame(data)
print(df)
```

Output:

```
  English  Maths    Name
0      64     76    John
1      78     54   Alice
2      68     72  Joseph
3      58     64    Alex
```


We only have to call the `describe()` function on the DataFrame and get the various measures like the mean, standard deviation, median, maximum element, minimum element, etc:

```
df.describe()
```

Output:

```
   English      Maths
count   4.000000   4.000000
mean   67.000000  66.500000
std     8.406347   9.712535
min    58.000000  54.000000
25%    62.500000  61.500000
50%    66.000000  68.000000
75%    70.500000  73.000000
max    78.000000  76.000000
```

As you can see, the `describe()` method completely ignored the "Name" column since it is not numberical, which is what we want. This simplifies things for the caller since you don't need to worry about removing non-numerical columns before calculating the numerical stats you want.

Conclusion

Pandas is an extremely useful Python library, particularly for data science. Various Pandas functionalities make data preprocessing extremely simple. This article provides a brief introduction to the main functionalities of the library. In this article, we saw working examples of all the major utilities of Pandas library. To get the most out of Pandas, I would suggest you practice the examples in this article and also test the library with your own datasets. Happy Coding!

---

python (/tag/python/), pandas (/tag/pandas/)

‹ Previous Post (/introduction-to-the-elk-stack/)

Next Post › (/text-summarization-with-nltk-in-python/)

Follow Us

(https://twitter.com/Stackabuse) (https://www.facebook.com/stackabuse) (https://stackabuse.com/rs

Newsletter

Subscribe to our newsletter! Get occassional tutorials, guides, and reviews in your inbox.

Enter your email...

Subscribe

No spam ever. Unsubscribe at any time.

Ad

Recent Posts

Guide to JPA with Hibernate: Basic Mapping (/guide-to-jpa-with-hibernate-basic-mapping/)

Default Arguments in Python Functions (/default-arguments-in-python-functions/)

The Observer Design Pattern in Java (/the-observer-design-pattern-in-java/)

Tags

algorithms (/tag/algorithms/) amqp (/tag/amqp/)
angular (/tag/angular/)
announcements (/tag/announcements/)
apache (/tag/apache/) api (/tag/api/)
arduino (/tag/arduino/)
artificial intelligence (/tag/artificial-intelligence/)
asynchronous (/tag/asynchronous/) aws (/tag/aws/)

Follow Us

(https://twitter.com/Stackabuse) (https://www.facebook.com/stackabuse) (https://stackabuse.com/rs