NumPy Introduction

NumPy stands for '**Numerical Python'** or '**Numeric Python'**.
It is an open source module of Python which provides fast mathematical computation on arrays and matrices. Since, arrays and matrices are an essential part of the Machine Learning ecosystem, NumPy along with Machine Learning modules like Scikit-learn, Pandas, Matplotlib, TensorFlow, etc. complete the Python Machine Learning Ecosystem.

What is NumPy?

NumPy is a Python library used for working with arrays.
It also has functions for working in domain of linear algebra, fourier transform, and matrices.
NumPy was created in 2005 by Travis Oliphant.
It is an open source project and you can use it freely.
NumPy stands for Numerical Python.

Why Use NumPy?

In Python we have lists that serve the purpose of arrays, but they are slow to process.
NumPy aims to provide an array object that is up to 50x faster than traditional Python lists.
The array object in NumPy is called ndarray, it provides a lot of supporting functions that make working with ndarray very easy.
Arrays are very frequently used in data science, where speed and resources are very important.
Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

Why is NumPy Faster Than Lists?

NumPy arrays are stored at one continuous place in memory unlike lists, so processes can access and manipulate them very efficiently.
This behavior is called locality of reference in computer science.
This is the main reason why NumPy is faster than lists. Also it is optimized to work with latest CPU architectures.

Which Language is NumPy written in?

NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.

Import NumPy

Once NumPy is installed, import it in your applications by adding the import keyword:

```
import numpy
```

Example:

```
import numpy
arr=numpy.array([1, 2, 3, 4, 5])
print(arr)
file name : simple_numpy.py
```

NumPy as np

NumPy is usually imported under the np alias.
alias: In Python alias are an alternate name for referring to the same thing.
Create an alias with the as keyword while importing:

```
import numpy as np
```

Now the NumPy package can be referred to as np instead of numpy.

Example

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5])
print(arr)
```

file name : simple_numpy.py

Checking NumPy Version

The version string is stored under __version__ attribute.

Example

```
import numpy as np
print(np.__version__)
```

file name : simple_numpy.py

NumPy Creating Arrays

Create a NumPy ndarray Object

NumPy is used to work with arrays.
The array object in NumPy is called ndarray.
We can create a NumPy ndarray object by using the array() function.

Example:

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5])
print(arr)
print(type(arr))
```

file name : simple_numpy.py

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

To create an ndarray, we can pass a list, tuple or any array-like object into the array() method, and it will be converted into an ndarray:

Example:Use a tuple to create a NumPy array:

```
import numpy as np
arr=np.array((1, 2, 3, 4, 5))
print(arr)
```

Dimensions in Arrays

A dimension in arrays is one level of array depth (nested arrays).

nested array: are arrays that have arrays as their elements.

0-D Arrays

0-D arrays, or Scalars, are the elements in an array. Each value in an array is a 0-D array.

Example: Create a 0-D array with value 42

```
import numpy as np
arr=np.array(42)
print(arr)
```

file name : array_type.py

1-D Arrays

An array that has 0-D arrays as its elements is called uni-dimensional or 1-D array.
These are the most common and basic arrays.

Example:Create a 1-D array containing the values 1,2,3,4,5:

```
import numpy as np
arr=np.array([1, 2, 3, 4, 5])
print(arr)
```

file name : array_type.py

2-D Arrays

An array that has 1-D arrays as its elements is called a 2-D array.
These are often used to represent matrix or 2nd order tensors.
NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

Example:Create a 2-D array containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr)
Or
arr = np.array([[1, 2, 3], ['a', 'b', 'c']])
>>> print(arr)
```

output :

```
[['1' '2' '3']
 ['a' 'b' 'c']
```

file name : array_type.py

3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.
These are often used to represent a 3rd order tensor.

Example:Create a 3-D array with two 2-D arrays, both containing two arrays with the values 1,2,3 and 4,5,6:

```
import numpy as np
arr=np.array([[[1, 2, 3],[4, 5, 6]],[[1, 2, 3],[4, 5, 6]]])
print(arr)
```

Check Number of Dimensions?

NumPy Arrays provides the ndim attribute that returns an integer that tells us how many dimensions the array have.

Example:Check how many dimensions the arrays have:

```
import numpy as np

a = np.array(42)
b = np.array([1, 2, 3, 4, 5])
c = np.array([[1, 2, 3], [4, 5, 6]])
d = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])
```

```
print(a.ndim)
print(b.ndim)
print(c.ndim)
print(d.ndim)
```

file name : check_dimension.py

NumPy **Array Indexing**

Access Array Elements

Array indexing is the same as accessing an array element.
You can access an array element by referring to its index number.
The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

Example: Get the first element from the following array:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[0])
```

Example: Get the second element from the following array.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[1])
```

Example : Get third and fourth elements from the following array and add them.

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr[2] + arr[3])
```

Access 2-D Arrays

To access elements from 2-D arrays we can use comma separated integers representing the dimension and the index of the element.

Example:Access the 2nd element on 1st dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('2nd element on 1st dim: ', arr[0, 1])
```

Example:Access the 5th element on 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('5th element on 2nd dim: ', arr[1, 4])
```

Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

Example:Access the third element of the second array of the first array:

```
import numpy as np
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr[0, 1, 2])
```

Negative Indexing

Use negative indexing to access an array from the end.

Example:Print the last element from the 2nd dim:

```
import numpy as np
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])
print('Last element from 2nd dim: ', arr[1, -1])
```

NumPy Data Types

Data Types in Python

By default Python have these data types:

- strings - used to represent text data, the text is given under quote marks. e.g. "ABCD"
- integer - used to represent integer numbers. e.g. -1, -2, -3
- float - used to represent real numbers. e.g. 1.2, 42.42
- boolean - used to represent True or False.
- complex - used to represent complex numbers. e.g. 1.0 + 2.0j, 1.5 + 2.5j

Data Types in NumPy

NumPy has some extra data types, and refer to data types with one character, like i for integers, u for unsigned integers etc.

Below is a list of all data types in NumPy and the characters used to represent them.

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string
- U - unicode string
- V - fixed chunk of memory for other type (void)

Checking the Data Type of an Array

The NumPy array object has a property called dtype that returns the data type of the array:

Example:Get the data type of an array object:

```
import numpy as np
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

Example:Get the data type of an array containing strings:

```
import numpy as np
arr = np.array(['apple', 'banana', 'cherry'])
print(arr.dtype)
```

Creating Arrays With a Defined Data Type

We use the array() function to create arrays, this function can take an optional argument: dtype that allows us to define the expected data type of the array elements:

Example:Create an array with data type string:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='S')
print(arr)
print(arr.dtype)
```

For i, u, f, S and U we can define size as well.

Example:Create an array with data type 4 bytes integer:

```
import numpy as np
arr = np.array([1, 2, 3, 4], dtype='i4')
```

```
print(arr)
print(arr.dtype)
```

Note:

[What if a Value Can Not Be Converted?

If a type is given in which elements can't be casted then NumPy will raise a ValueError.

ValueError: In Python ValueError is raised when the type of passed argument to a function is unexpected/incorrect.

Example:**A non integer string like 'a' can not be converted to integer (will raise an error):**

```
import numpy as np
arr = np.array(['a', '2', '3'], dtype='i')   ]
```

Converting Data Type on Existing Arrays

The best way to change the data type of an existing array, is to make a copy of the array with the astype() method.

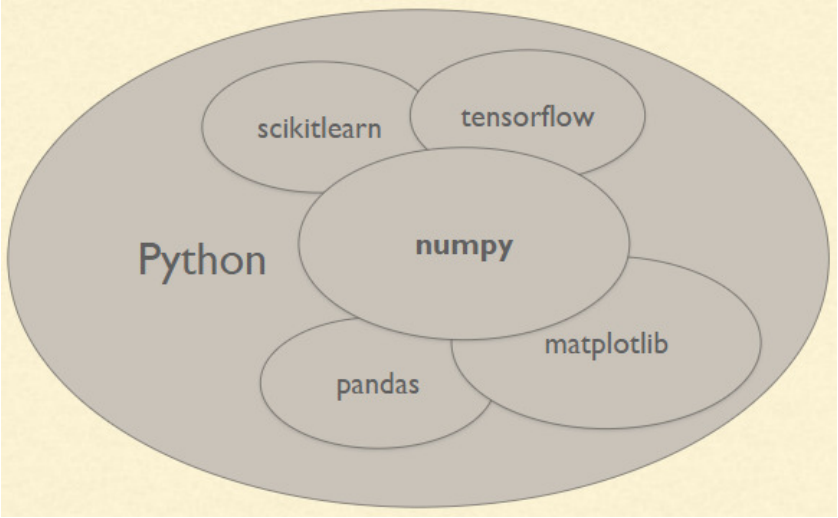
The astype() function creates a copy of the array, and allows you to specify the data type as a parameter.

The data type can be specified using a string, like 'f' for float, 'i' for integer etc. or you can use the data type directly like float for float and int for integer.

Example:Change data type from float to integer by using 'i' as parameter value:

```
import numpy as np
arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')
print(newarr)
print(newarr.dtype)
```

4.3 Dataframe Handling using Panda and Numpy:



4.3.1 csv and excel file extract and write using Dataframe

Reading and Writing CSV Files using Pandas

Pandas is a very powerful and popular framework for data analysis and manipulation.

One of the most striking features of Pandas is its ability to read and write various types of files including **CSV** and **Excel**.

You can effectively and easily manipulate CSV files in Pandas using functions like **read_csv() and to_csv().**

Installing Pandas

We have to install Pandas before using it. Let's use pip:

```
$ pip install pandas
```

Reading CSV Files with read_csv()

Let's import the Titan Dataset,

```
import pandas as pd

titan_data = pd.read_csv('titan.csv')
```

Pandas will search for this file in the directory of the script, naturally, and we just supply the filepath to the file we'd like to parse as the one and only required argument of this method.
Let's take a look at the head() of this dataset to make sure it's imported correctly:

```
titan_data.head()
```

Alternatively, you can also read CSV files from online resources,simply by passing in the URL of the resource to the **read_csv()** function.
Let's read this same CSV file from the GitHub repository, without downloading it to our local machine first:

```
import pandas as pd
titanic_data=                                                    =
        pd.read_csv(r'https://raw.githubusercontent.com/datasciencedojo/datas
ets/    master/titanic.csv')
print(titanic_data.head())
```

Customizing Headers

By default, the read_csv() method uses the first row of the CSV file as the column headers.
Sometimes, these headers might have odd names, and you might want to use your own headers. Y
ou can set headers either after reading the file, simply by assigning the columns field of the DataFrame instance another list, or you can set the headers while reading the CSV in the first place.

```
import pandas as pd
col_names = ['Id',
        'Survived',
        'Passenger Class',
        'Full Name']
titanic_data = pd.read_csv(r'E:\Datasets\titanic.csv', names=col_names)
print(titanic_data.head())
```

Removing Headers

You can also decide to remove the header completely, which would result in a DataFrame that simply has 0...n header columns, by setting the header argument to None:

```
titanic_data = pd.read_csv(r'E:\Datasets\titanic.csv', header=None)
```

Specifying Delimiters

As stated earlier, you'll eventually probably encounter a CSV file that doesn't actually use commas to separate data.
In such cases, you can use the sep argument to specify other delimiters:

```
titanic_data = pd.read_csv(r'E:\Datasets\titanic.csv', sep=';')
```

code :

readcsv.py

```
import pandas as pd
#df = pd.read_csv('a1.csv',header=None)
df = pd.read_csv(r'a1.csv')
col_names = ['studno','studname']
df = pd.read_csv(r'a1.csv',names=col_names)
print(df)
#print(df.head())
```

```
import pandas as pd
df = pd.read_csv('a1.csv',header=None)
df.to_csv('a1_update.csv')
print(df)
'''
to_csv is a method to an object which is a df (DataFrame);
while pd is Panda module.
Hence, your code was not working and throwing this Error
" AttributeError: module 'pandas' has no attribute 'to_csv'"
'''
```

Writing CSV Files with to_csv()

Again, DataFrames are tabular.

Turning a DataFrame into a CSV file is as simple as turning a CSV file into a DataFrame - we call the write_csv() function on the DataFrame instance.

When writing a DataFrame to a CSV file, you can also change the column names, using the columns argument, or specify a delimiter via the sep argument.

If you don't specify either of these, you'll end up with a standard Comma-Separated Value file.

```
import pandas as pd
cities = pd.DataFrame([['Sacramento', 'California'], ['Miami', 'Florida']], columns=['City',
    'State'])
cities.to_csv('cities.csv')
```

Here, we've made a simple DataFrame with two cities and their respective states. Then, we've gone ahead and saved that data into a CSV file using to_csv() and providing the filename.

code :

Tocsv.py

```
import pandas as pd
df = pd.read_csv('a1.csv',header=None)
df.to_csv('a1_update.csv')
print(df)

'''  to_csv is a method to an object which is a df (DataFrame);
     while pd is Panda module.
     Hence, your code was not working and throwing this Error
     " AttributeError: module 'pandas' has no attribute 'to_csv'"
```

some Examples:

Example #1: Save csv to working directory.

```
# importing pandas as pd
import pandas as pd
```

```
# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
dict = {'name': nme, 'degree': deg, 'score': scr}
df = pd.DataFrame(dict)
# saving the dataframe
df.to_csv('file1.csv')
```

Example #2: Saving CSV without *headers* and *index*.

```
# importing pandas as pd
import pandas as pd

# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
dict = {'name': nme, 'degree': deg, 'score': scr}
df = pd.DataFrame(dict)

# saving the dataframe
df.to_csv('file2.csv', header=False, index=False)
```

Example #3: Save csv file to a specified location.

```
# importing pandas as pd
import pandas as pd

# list of name, degree, score
nme = ["aparna", "pankaj", "sudhir", "Geeku"]
deg = ["MBA", "BCA", "M.Tech", "MBA"]
scr = [90, 40, 80, 98]

# dictionary of lists
dict = {'name': nme, 'degree': deg, 'score': scr}
df = pd.DataFrame(dict)

# saving the dataframe
df.to_csv(r'C:\Users\Admin\Desktop\file3.csv', index=False)
```

Reading and Writing EXCEL Files using Pandas

Just like with all other types of files, you can use the Pandas library to read and write Excel files using Python as well.

Reading and Writing Excel Files in Python with Pandas

Naturally, to use Pandas, we first have to install it. The easiest method to install it is via pip.
If you're running Windows:

```
$ python pip install pandas
```

If you're using Linux or MacOS:

```
$ pip install pandas
```

Note that you may get a ModuleNotFoundError or ImportError error when running the code in this article.

For example:

ModuleNotFoundError: No module named 'openpyxl'

If this is the case, then you'll need to install the missing module(s):

```
$ pip install openpyxl xlsxwriter xlrd
```

Writing Excel Files Using Pandas

We'll be storing the information we'd like to write to an Excel file in a DataFrame. Using the built-in to_excel() function, we can extract this information into an Excel file.

First, let's import the Pandas module:

```
import pandas as pd
    #Now, let's use a dictionary to populate a DataFrame:
df = pd.DataFrame
({'States':['California', 'Florida', 'Montana', 'Colorodo', 'Washington', 'Virginia'],
  'Capitals':['Sacramento', 'Tallahassee', 'Helena', 'Denver', 'Olympia', 'Richmond'],
  'Population':['508529', '193551', '32315', '619968', '52555', '227032']})
```

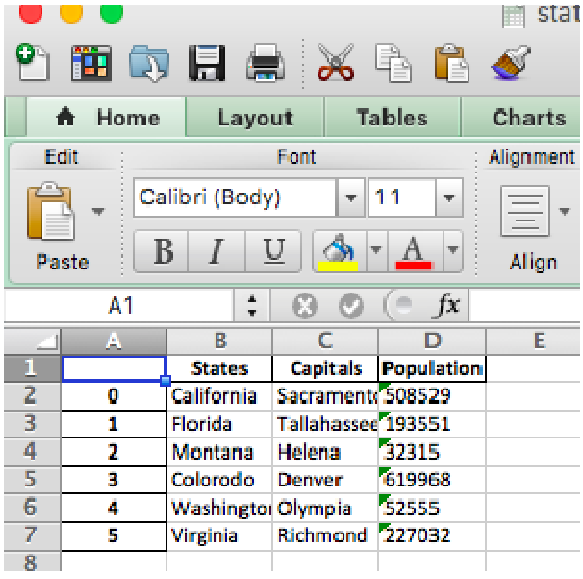
file name : readexcel.py and write_excel.py

The keys in our dictionary will serve as column names.
Similarly, the values become the rows containing the information.

Now, we can use the to_excel() function to write the contents to a file.
The only argument is the file path:

```
df.to_excel('./states.xlsx')
```

Here's the Excel file that was created:



These numbers are the indices for each row, coming straight from the Pandas DataFrame.

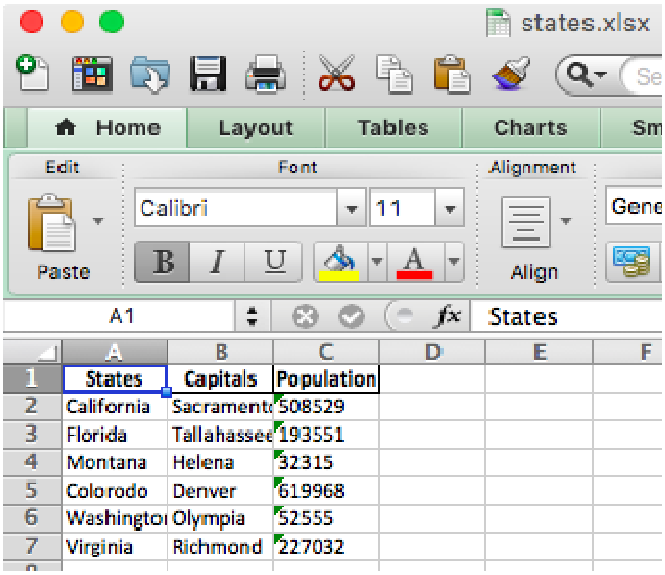
We can change the name of our sheet by adding the sheet_name parameter to our to_excel() call:

```
df.to_excel('./states.xlsx', sheet_name='States')
```

Similarly, adding the index parameter and setting it to False will remove the index column from the output:

```
df.to_excel('./states.xlsx', sheet_name='States', index=False)
```

Now, the Excel file looks like this:



Writing Multiple DataFrames to an Excel File

It is also possible to write multiple dataframes to an Excel file. If you'd like to, you can set a different sheet for each dataframe as well:

```
income1 = pd.DataFrame
            ({'Names': ['Stephen', 'Camilla', 'Tom'], 'Salary':[100000, 70000, 60000]})

income2 = pd.DataFrame
            ({'Names': ['Pete', 'April', 'Marty'], 'Salary':[120000, 110000, 50000]})

income3 = pd.DataFrame
            ({'Names': ['Victor', 'Victoria', 'Jennifer'],'Salary':[75000, 90000, 40000]})

income_sheets = {'Group1': income1, 'Group2': income2, 'Group3': income3}

writer = pd.ExcelWriter('./income.xlsx', engine='xlsxwriter')

for sheet_name in income_sheets.keys():

  income_sheets[sheet_name].to_excel(writer, sheet_name=sheet_name, index=False)
writer.save()
```

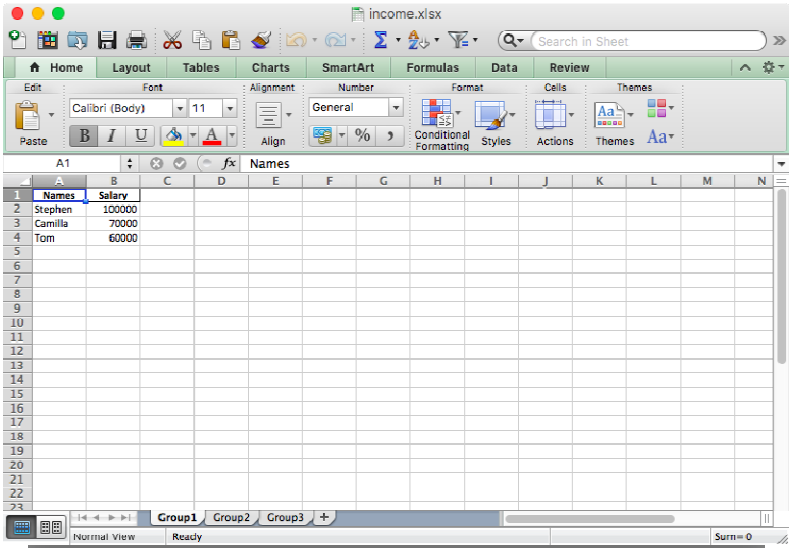
file name : multi_dataframe.py

Here, we've created 3 different dataframes containing various names of employees and their salaries as data. Each of these dataframes is populated by its respective dictionary.

We've combined these three within the income_sheets variable, where each key is the sheet name, and each value is the DataFrame object.

Finally, we've used the xlsxwriter engine to create a writer object. This object is passed to the to_excel() function call.

Before we even write anything, we loop through the keys of income and for each key, write the content to the respective sheet name.
Here is the generated file:



You can see that the Excel file has three different sheets named Group1, Group2, and Group3. Each of these sheets contains names of employees and their salaries with respect to the date in the three different dataframes in our code.

The engine parameter in the to_excel() function is used to specify which underlying module is used by the Pandas library to create the Excel file.

In our case, the xlsxwriter module is used as the engine for the ExcelWriter class.

Different engines can be specified depending on their respective features.

Depending upon the Python modules installed on your system, the other options
for the engine attribute are: openpyxl (for xlsx and xlsm), and xlwt (for xls).

Further details of using the xlsxwriter module with Pandas library are available at the official documentation.

Last but not least, in the code above we have to explicitly save the file using writer.save(), otherwise it won't be persisted on the disk.

4.3.3 Central Tendency measures :

Mathematically central tendency means measuring the center or distribution of location of values of a data set.
It gives an idea of the average value of the data in the data set and also an indication of how widely the values are spread in the data set.
That in turn helps in evaluating the chances of a new input fitting into the existing data set and hence probability of success.
There are three main measures of central tendency which can be calculated using the methods in pandas python library.

- **Mean** - It is the Average value of the data which is a division of sum of the values with the number of values.

- **Median -** It is the middle value in distribution when the values are arranged in ascending or descending order.
- **Mode -** It is the most commonly occurring value in a distribution.

Calculating Mean and Median

If we have a sample of numeric values, then its *mean* or the *average* is the total sum of the values (or observations) divided by the number of values.

Say we have the sample [4, 8, 6, 5, 3, 2, 8, 9, 2, 5].

We can calculate its mean by performing the operation:
(4 + 8 + 6 + 5 + 3 + 2 + 8 + 9 + 2 + 5) / 10 = 5.2

code :

```
# Python program to demonstrate mean()function from the statistics module
# Importing the statistics module

import statistics
        # list of positive integer numbers
list1 = [1, 3, 4, 5, 7, 9, 2]
x = statistics.mean(list1)
        # Printing the mean
print("Mean is :", x)
```

file name : mean_function.py and mean_function2.py

Python statistics.median() Method

Example: Calculate the median (middle value) of the given data

```
        # Import statistics Library
import statistics

        # Calculate middle values
print(statistics.median([1, 3, 5, 7, 9, 11, 13]))
print(statistics.median([1, 3, 5, 7, 9, 11]))
print(statistics.median([-11, 5.5, -3.4, 7.1, -9, 22]))
```

file name : median_function.py

The statistics.median() method calculates the median (middle value) of the given data set. This method also sorts the data in ascending order before calculating the median.
Tip: The mathematical formula for Median is: **Median = {(n + 1) / 2}**th value, where n is the number of values in a set of data.
In order to calculate the median, the data must first be **sorted in ascending order**. The median is the number in the middle.

Note: If the number of data values is odd, it returns the exact middle value. If the number of data values is even, it returns the average of the two middle values.

Syntax : **statistics.median(*data*)**

Parameter	Description
data	Required. The data values to be used (can be any sequence, list or iterator)

Note: If *data* is empty, it returns a StatisticsError.

Finding the Median of a Sample

The *median* of a sample of numeric data is the value that lies in the middle when we sort the data. The data may be sorted in ascending or descending order, the median remains the same.

To find the median, we need to:

1. *Sort* the sample
2. *Locate* the value in the middle of the sorted sample

When locating the number in the middle of a sorted sample, we can face two kinds of situations:

1. If the sample has an **odd number of observations**, then the middle value in the sorted sample is the median
2. If the sample has an **even number of observations**, then we'll need to calculate the mean of the two middle values in the sorted sample

If we have the sample `[3, 5, 1, 4, 2]` and want to find its median, then we first sort the sample to `[1, 2, 3, 4, 5]`. The median would be `3` since that's the value in the middle.

On the other hand, if we have the sample `[1, 2, 3, 4, 5, 6]`, then its median will be `(3 + 4) / 2 = 3.5`.

Calculating Mode

Mode may or may not be available in a distribution depending on whether the data is continuous or whether there are values which has maximum frquency.

The mode is the statistical term that refers to the most frequently occurring number found in a set of numbers. The mode is detected by collecting and organizing data to count the frequency of each result.

Example: Calculate the mode (central tendency) of the given data:

```
        # Import statistics Library
import statistics
        # Calculate the mode
print(statistics.mode([1, 3, 3, 3, 5, 7, 7 9, 11]))
print(statistics.mode([1, 1, 3, -5, 7, -9, 11]))
print(statistics.mode(['red', 'green', 'blue', 'red']))

output:
        3
        1
        red
```

Example :

```
>>> import statistics
>>> statistics.mode([4, 1, 2, 2, 3, 5])
2
```

```
>>> statistics.mode([4, 1, 2, 2, 3, 5, 4])
4
>>> st.mode(["few", "few", "many", "some", "many"])
'few'
```

Python - Measuring Variance

In statistics, variance is a measure of how far a value in a data set lies from the mean value.
In other words, it indicates how dispersed the values are.
It is measured by using standard deviation.
The other method commonly used is skewness.
Both of these are calculated by using functions available in pandas library.

Measuring Standard Deviation

Standard deviation is square root of variance. variance is the average of squared difference of values in a data set from the mean value. In python we calculate this value by using the function std() from pandas library.

```
import pandas as pd
        #Create a Dictionary of series
d = {'Name':pd.Series(['Tom','James','Ricky','Vin','Steve','Smith','Jack',
          'Lee','Chanchal','Gasper','Naviya','Andres']),
  'Age':pd.Series([25,26,25,23,30,25,23,34,40,30,25,46]),
  'Rating':pd.Series([4.23,3.24,3.98,2.56,3.20,4.6,3.8,3.78,2.98,4.80,4.10,3.65])}

        #Create a DataFrame
df = pd.DataFrame(d)
        # Calculate the standard deviation
print df.std()
```

Its output is as follows −

```
Age      7.265527
Rating   0.661628
dtype: float64
```

note:

Range:

Difference between the largest and smallest values.
It takes largest and smallest value from the data set.
It uses only the data less than 6 in a data set.

Variance:

Variance measures how far a data set is spread out.
The average squared deviation of values from the mean.

Standard Deviation

It takes all data from the data set.
We can use standard deviation, if the data in a data set are more than 6.
Standard Deviation is the square root of variance.

Dataframe functions: head, tail, loc, iloc, value, to_numpy(), describe()

pandas.DataFrame.head()

This method is used to get the first n rows of the DataFrame.

This method is convenient when we want to check which type of data our object has in it.

If we give **negative** values for 'n', this method returns all the rows except the last n rows which is equivalent to df[:-n].

Syntax : **DataFrame.head(n)**

Parameter = (n) : **It represents the int, and the default value is 5 that is the number of rows to select.**

```
        #importing pandas as pd
import pandas as pd
        #creating the DataFrame
df= pd.DataFrame({'Language': [English,'Hindi', Gujarati, 'Tamil',
'Malyalam','Marathi','Konkani']})
print("First 5 rows of the DataFrame is:")
print(df.head())
```

Example: Specify the number of rows in the DataFrame.head() Method

We can specify the number of lines to view and here, in this example, we give n value as 2.

The DataFrame.head() method returns the first 2 lines.

```
        #importing pandas as pd
import pandas as pd
        #creating the DataFrame
df= pd.DataFrame({'Language': ['Kannada','Hindi', 'Telugu', 'Tamil',
'Malyalam','Marathi','Konkani','Tulu']})
print("----First n rows of the DataFrame is-----")
print(df.head(n=2))
```

Example: Negative value to 'n' in the DataFrame.head() Method

In this example, we give a negative value to n. For negative values of n, this method returns all rows except the last n rows, equivalent to df[:-n].

```
#importing pandas as pd
import pandas as pd
#creating the DataFrame
df= pd.DataFrame({'Language': ['Kannada','Hindi', 'Telugu', 'Tamil',
'Malyalam','Marathi','Konkani','Tulu']})
print("----First n rows of the DataFrame is-----")
print(df.head(-2))
```

file : C:/notes/303_sqlite_python/numpy/head.py

pandas.DataFrame.tail()

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(n)**

Examples

```
import numpy as np
import pandas as pd
df = pd.DataFrame({'animal':['snake', 'bat', 'tiger', 'lion', 'fox', 'eagle', 'shark', 'dog', 'deer']})
df.tail()
df.tail(4)
```

DataFrame.loc

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)

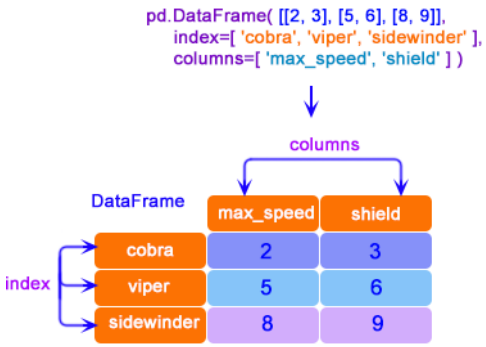
Syntax: DataFrame.loc
Raises: KeyError when any items are not found

Examples

```
import numpy as np
import pandas as pd

df = pd.DataFrame([[2, 3], [5, 6], [8, 9]],index=['cobra', 'viper', 'sidewinder'],columns=['max_speed', 'shield'])
```

	max_speed	shield
cobra	2	3
viper	5	6
sidewinder	8	9

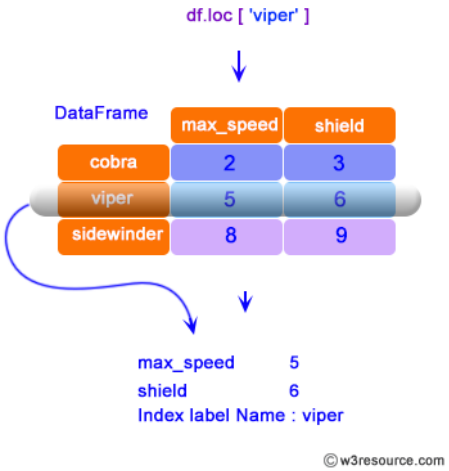


Single label:

```
f.loc['viper']
```

output :

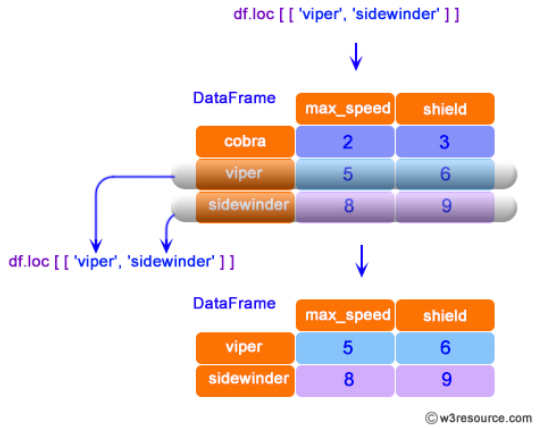
max_speed 5
shield 6
Name: viper, dtype: int64



List of labels:

df.loc[['viper', 'sidewinder']]

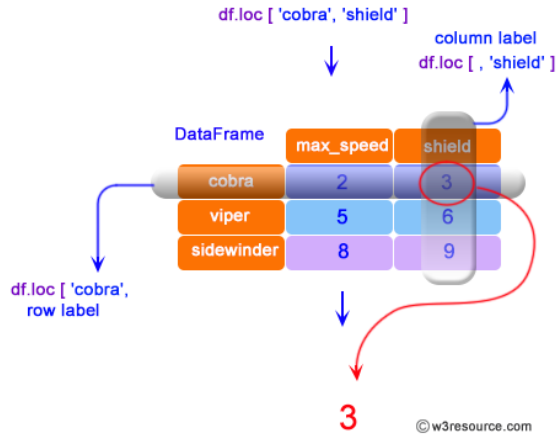
output:	max_speed	shield
viper	**5**	**6**
sidewinder	**8**	**9**



Single label for row and column:

df.loc['cobra', 'shield']

output : 3



DataFrame - iloc property

The iloc property returns purely integer-location based indexing for selection by position. .iloc[] is primarily integer position based (from 0 to length-1 of the axis), but may also be used with a boolean array.

Allowed inputs are:

- An integer, e.g. 5.
- A list or array of integers, e.g. [4, 3, 0].
- A slice object with ints, e.g. 1:7.
- A boolean array.
- A callable function with one argument (the calling Series or DataFrame) and that returns valid output for indexing (one of the above).
- This is useful in method chains, when you don't have a reference to the calling object, but would like to base your selection on some value.

.iloc will raise IndexError if a requested indexer is out-of-bounds, except slice indexers which allow out-of-bounds indexing (this conforms with python/numpy slice semantics).

Syntax: DataFrame.iloc
Example:

```
#display only
import numpy as np
import pandas as pd
mydict = [{'p': 2, 'q': 3, 'r': 4, 's': 5},
        {'p': 200, 'q': 300, 'r': 400, 's': 500},
        {'p': 2000, 'q': 3000, 'r': 4000, 's': 5000 }]
df = pd.DataFrame(mydict)
print(df)
```

Output :

	p	q	r	s
0	2	3	4	5
1	200	300	400	500
2	2000	3000	4000	5000

Indexing just the rows With a scalar integer.

```
type(df.iloc[0])
pandas.core.series.Series
df.iloc[0]
```

Output :
p 2
q 3
r 4
s 5
Name: 0, dtype: int64

With a list of integers.

```
df.iloc[[0]]
```

output:

	p	q	r	s
0	2	3	4	5

pandas.DataFrame.to_numpy

Syntax :
DataFrame.to_numpy(dtype=None,copy=False, na_value=<no_default>**)[source]**

It 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 values, which may be expensive.

Parameters:

- dtypestr or numpy.dtype, optional : The dtype to pass to numpy.asarray().
- copybool, 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_valueAny, optional: The value to use for missing values. The default value depends on dtype and the dtypes of the DataFrame columns.
- Returns :numpy.ndarray

file : C:/notes/303_sqlite_python/numpy/toNumpy.py

pandas.DataFrame.describe

The describe() function is used to generate descriptive statistics that summarize the central tendency, dispersion and shape of a dataset's distribution, excluding **NaN values**.

Syntax:
DataFrame.describe(percentiles=None, include=None, exclude=None)

Parameters:

Name	Description	Type/Default Value	Required / Optional
percentiles	The percentiles to include in the output. All should fall between 0 and 1. The default is [.25, .5, .75], which returns the 25th, 50th, and 75th percentiles.	list-like of numbers	Optional
include	A white list of data types to include in the result. Ignored for Series. Here are the options: • 'all' : All columns of the input will be included in the output. • A list-like of dtypes : Limits the results to the provided data types. To limit the result to numeric types submit numpy.number. To limit it instead to object columns submit the numpy.object data type. Strings can also be used in the style of select_dtypes (e.g. df.describe(include=['O'])). To select pandas categorical columns, use 'category' • None (default) : The result will include all numeric columns.	'all', list-like of dtypes or None (default)	Optional
exclude	A black list of data types to omit from the result. Ignored for Series. Here are the options: • A list-like of dtypes : Excludes the provided data types from the result. To exclude numeric types submit numpy.number. To exclude object columns submit the data type numpy.object. Strings can also be used in the style of select_dtypes (e.g. df.describe(include=['O'])). To exclude pandas categorical columns, use 'category' • None (default) : The result will exclude nothing.	list-like of dtypes or None (default)	Optional

Returns: Series or DataFrame

Example : Describing a DataFrame using the DataFrame.describe() Method

```
import pandas as pd
df= pd.DataFrame
([['Abhishek',100,'Science',90], ['Anurag',101,'Science',85],['Chetan',103,'Maths',75]], columns=[
'Name', 'Roll No', 'Subject', 'Marks'])
print(df.describe())
```

file name : python/numpy/describe.py

Example 2: Describing all columns of a DataFrame using the DataFrame.describe() Method

```
import pandas as pd
df= pd.DataFrame([['Abhishek',100,'Science',90], ['Anurag',101,'Science',85],['Chetan',103,'Mat
hs',75]], columns=['Name', 'Roll No', 'Subject', 'Marks'])
print(df.describe(include='all'))
```

file name : python/numpy/describe.py