



# INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

**SRM** CSS101J – Programming for Problem

Solving Unit 5





# INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.

**SRM**

## LEARNING RESOURCES

S. No	TEXT BOOKS
1.	Python Datascience Handbook, Oreilly, Jake VanderPlas, 2017. [Chapters 2 & 3]
2.	Python For Beginners, Timothy C. Needham, 2019. [Chapters 1 to 4]
3.	<a href="https://www.tutorialspoint.com/python/index.htm">https://www.tutorialspoint.com/python/index.htm</a>
4.	<a href="https://www.w3schools.com/python/">https://www.w3schools.com/python/</a>



# **INSTITUTE OF SCIENCE AND TECHNOLOGY, CHENNAI.**

**SRM**

## **UNIT V**

### **(TOPICS COVERED)**

Creating NumPy Array - Numpy Indexing - Numpy Array attributes - Slicing using Numpy - Descriptive Statistics in Numpy: Percentile - Variance in Numpy - Introduction to Pandas - Creating Series Objects, Data Frame Objects – Simple Operations with Data frames - Querying from Data Frames - Applying Functions to Data frames - Comparison between Numpy and Pandas - Speed Testing between Numpy and Pandas - Other Python Libraries

**21CSS101J**

# **PROGRAMMING FOR PROBLEM SOLVING**

## **UNIT-5 Numpy (Numerical Python)**

# NumPy

Stands for Numerical Python

Is the fundamental package required for high performance computing and data analysis

NumPy is so important for numerical computations in Python is because it is designed for efficiency on large arrays of data.

It provides

- ndarray for creating multiple dimensional arrays

- Internally stores data in a contiguous block of memory, independent of other built-in Python objects, use much less memory than built-in Python sequences.

- Standard math functions for fast operations on entire arrays of data without having to write loops

- NumPy Arrays are important because they enable you to express batch operations on data without writing any *for* loops. We call this *vectorization*.

## NumPy ndarray vs list

One of the key features of NumPy is its N-dimensional array object, or ndarray, which is a fast, flexible container for large datasets in Python.

Whenever you see “array,” “NumPy array,” or “ndarray” in the text, with few exceptions they all refer to the same thing: the ndarray object.

NumPy-based algorithms are generally 10 to 100 times faster (or more) than their pure Python counterparts and use significantly less memory.

```
import numpy as np
my_arr = np.arange(1000000)
my_list = list(range(1000000))
```

## ndarray

ndarray is used for storage of homogeneous data

i.e., all elements the same type

Every array must have a shape and a dtype

Supports convenient slicing, indexing and efficient vectorized computation

```
import numpy as np
data1 = [6, 7.5, 8, 0, 1]
arr1 = np.array(data1)
print(arr1)
print(arr1.dtype)
print(arr1.shape)
print(arr1.ndim)
```

## Numpy

- Numerical Python
- Fast Computation with n-dimensional arrays
- Based around one data structure
- ndarray
- n-dimensional array
- import with *import numpy as np*
- Usage is *np.command(xxx)*



## ndarrays

1d: 5,67,43,76,2,21

```
a=np.array([5,67,43,76,2,21])
```

2d: 4,5,8,4

6,3,2,1

8,6,4,3

```
a=np.array([4,5,8,4],[6,3,2,1],[8,6,4,3])
```

## Creating ndarrays

- `data1 = [6, 7.5, 8, 0, 1]`
- `arr1 = np.array(data1)`
- `print arr1`

- **Output**

- `[ 6. 7.5 8. 0. 1.]`

## Multidimensional arrays

```
data2 = [[1, 2, 3, 4], [5, 6, 7, 8]]
```

```
arr2 = np.array(data2)
```

```
print arr2
```

```
print arr2.ndim
```

```
print arr2.shape
```

**OUTPUT**

```
[[1 2 3 4]
```

```
 [5 6 7 8]]
```

```
2
```

```
(2L, 4L)
```

- `print arr2`
- `print arr2.ndim`
- `print type(arr2.ndim)`
- `print arr2.shape`
- `print type(arr2.shape)`
- `print arr2.shape[0]`
- `print arr2.shape[1]`

## OUTPUT

```
[[1 2 3 4]
 [5 6 7 8]]
```

```
2
```

```
<type 'int'>
```

```
(2L, 4L)
```

```
<type 'tuple'>
```

```
2
```

```
4
```

## Operations between arrays and scalars

• <code>arr = np.array</code> <code>([1., 2., 3.])</code>	output [ 1.  2.  3.]
• <code>print arr</code>	[ 1.  4.  9.]
• <code>print arr * arr</code>	[ 0.  0.  0.]
• <code>print arr - arr</code>	
• <code>print 1 / arr</code>	[ 1.    0.5   0.3333]
• <code>print arr ** 0.5</code>	[ 1.    1.4142  1.7321]

## Array creation functions

Function	Description
<code>array</code>	Convert input data (list, tuple, array, or other sequence type) to an ndarray either by inferring a dtype or explicitly specifying a dtype. Copies the input data by default.
<code>asarray</code>	Convert input to ndarray, but do not copy if the input is already an ndarray
<code>arange</code>	Like the built-in <code>range</code> but returns an ndarray instead of a list.
<code>ones</code> , <code>ones_like</code>	Produce an array of all 1's with the given shape and dtype. <code>ones_like</code> takes another array and produces a ones array of the same shape and dtype.
<code>zeros</code> , <code>zeros_like</code>	Like <code>ones</code> and <code>ones_like</code> but producing arrays of 0's instead

## astype

[ 3.7 -1.2 -2.6]

- `arr = np.array([`
- `3.7, -1.2, -2.6])` [ 3 -1 -2]
- `print arr`
- `print arr.astype(np.int32)`
- note that the data has been **truncated**.



## Astype – string to float

- `numeric_strings =  
np.array(['1.25', '-9.6', '42'],  
dtype=np.string_)`
- `print numeric_strings`
- `print  
numeric_strings.astype(float)`

```
['1.25' '-9.6' '42']  
[ 1.25 -9.6  42. ]
```



## Basic indexing and slicing (broadcasting)

```
arr = np.arange(10)
print arr           [0 1 2 3 4 5 6 7 8 9]
print arr[5]        5
print arr[5:8]       [5 6 7]
arr[5:8] = 12        [0 1 2 3 4 12 12 12 8 9]
print arr
```

**The original array has changed**

```
arr_slice = arr[5:8]
```

```
arr_slice[1] = 12345
```

```
print arr
```

```
arr_slice[:] = 64
```

```
print arr
```

```
[ 0  1  2  3  4 12 12345 12  8  9]
```

```
[0 1 2 3 4 64 64 64 8 9]
```

## Numpy Indexing

Contents of ndarray object can be accessed and modified by indexing or slicing, just like Python's in-built container objects.

items in ndarray object follows zero-based index. Three types of indexing methods are available – **field access**, **basic slicing** and **advanced indexing**.

Basic slicing is an extension of Python's basic concept of slicing to n dimensions. A Python slice object is constructed by giving **start**, **stop**, and **step** parameters to the built-in **slice** function. This slice object is passed to the array to extract a part of array.

## Example :

```
import numpy as np
a = np.arange(10)
s = slice(2,7,2)
print a[s]
```

```
import numpy as np
a = np.arange(10)
b = a[2:7:2]
print b
```

Output:

[2 4 6]

**ndarray** object is prepared by **arange()** function. Then a slice object is defined with start, stop, and step values 2, 7, and 2 respectively. When this slice object is passed to the ndarray, a part of it starting with index 2 up to 7 with a step of 2 is sliced.

If a **:** is inserted in front of it, all items from that index onwards will be extracted.

## Slicing Numpy:

```
import numpy as np
a = np.array([[1,2,3],[3,4,5],[4,5,6]])
print a
# slice items starting from index
print 'Now we will slice the array from the index a[1:]'
print a[1:]
```

OutPut:

```
[[1 2 3]
 [3 4 5]
 [4 5 6]]
```

Now we will slice the array from the index a[1:]

```
[[3 4 5]
 [4 5 6]]
```

Slicing can also include ellipsis (...) to make a selection tuple of the same length as the dimension of an array.

# Slicing arrays

Slicing in python means taking elements from one given index to another given index.

We pass slice instead of index like this: `[start:end]`.

We can also define the step, like this: `[start:end:step]`.

If we don't pass start its considered 0

If we don't pass end its considered length of array in that dimension

If we don't pass step its considered 1

## Example:

Slice elements from index 1 to index 5 from the following array:

```
import numpy as np  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
print(arr[1:5])
```

Slice elements from index 4 to the end of the array:

```
import numpy as np  
arr = np.array([1, 2, 3, 4, 5, 6, 7])  
print(arr[4:])
```

## Descriptive Statistics in Numpy:

Descriptive statistics allow us to summarise data sets quickly with just a couple of numbers, and are in general easy to explain to others.

Descriptive statistics fall into two general categories:

- 1) **Measures of central tendency** which describe a ‘typical’ or common value (e.g. mean, median, and mode); and,
- 2) **Measures of spread** which describe how far apart values are (e.g. percentiles, variance, and standard deviation).



## Percentile:

**numpy.percentile()** function used to compute the nth percentile of the given data (array elements) along the specified axis.

**Syntax :** `numpy.percentile(arr, n, axis=None, out=None)`

**Parameters :**

**arr** :input array.

**n** : percentile value.

**axis** : axis along which we want to calculate the percentile value.

Otherwise, it will consider arr to be flattened(works on all the axis).

axis = 0 means along the column and axis = 1 means working along the row.

**out** :Different array in which we want to place the result. The array must have same dimensions as expected output.

**Return** :nth Percentile of the array (a scalar value if axis is none)or array with percentile values along specified axis.

## Example:

```
# Python Program illustrating  
# numpy.percentile() method
```

```
import numpy as np
```

```
# 1D array
```

```
arr = [20, 2, 7, 1, 34]
```

```
print("arr : ", arr)
```

```
print("50th percentile of arr : ",  
      np.percentile(arr, 50))
```

```
print("25th percentile of arr : ",  
      np.percentile(arr, 25))
```

```
print("75th percentile of arr : ",  
      np.percentile(arr, 75))
```

## Output :

arr : [20, 2, 7, 1, 34]

50th percentile of arr : 7.0

25th percentile of arr : 2.0

75th percentile of arr : 20.0

## Example:

**Numpy:** Has two related functions, percentile and quantile. The percentile function uses  $q$  in range  $[0,100]$  e.g. for 90th percentile use 90, whereas the quantile function uses  $q$  in range  $[0,1]$ , so the equivalent  $q$  would be 0.9. They can be used interchangeably.

```
p25 = np.percentile(data_sample_even, q=25, interpolation='linear')
p75 = np.percentile(data_sample_even, q=75, interpolation='linear')
iqr = p75 - p25
```

## Variance in Numpy -

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

$$\text{Formula : } \sigma^2 = \frac{\sum_{i=1}^N (x_i - \mu)^2}{N}$$

**Where,**

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

calculate the variance by using **numpy.var()** function

## *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:

# 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

## **Introduction to Pandas -**

### **What is Pandas?**

Pandas is a Python library used for working with data sets. It has functions for analyzing, cleaning, exploring, and manipulating data.

The name "Pandas" has a reference to both "Panel Data", and "Python Data Analysis" and was created by Wes McKinney in 2008.

### **Why Use Pandas?**

Pandas allows us to analyze big data and make conclusions based on statistical theories.

Pandas can clean messy data sets, and make them readable and relevant.

Relevant data is very important in data science.

## Installation of pandas:

```
C:\Users\Your Name>pip install pandas
```

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

```
import pandas
```

Example:

```
import pandas
mydataset = {
    'cars': ["BMW", "Volvo", "Ford"],
    'passings': [3, 7, 2]
}
myvar = pandas.DataFrame(mydataset)
print(myvar)
```



# Pandas as pd

```
import pandas as pd
```

```
mydataset = {  
    'cars': ["BMW", "Volvo", "Ford"],  
    'passings': [3, 7, 2]  
}
```

```
myvar = pd.DataFrame(mydataset)
```

```
print(myvar)
```

## Creating Series Objects

What is a Series?

A Pandas Series is like a column in a table.

It is a one-dimensional array holding data of any type.

Example

Create a simple Pandas Series from a list:

```
import pandas as pd
```

```
a = [1, 7, 2]
```

```
myvar = pd.Series(a)
```

```
print(myvar)
```

```
0    1
1    7
2    2
dtype: int64
```

# Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc. This label can be used to access a specified value.

## Example

Return the first value of the Series:

```
print(myvar[0])
```

## Create Labels

With the index argument, you can name your own labels.

### Example

Create your own labels:

```
import pandas as pd
```

```
a = [1, 7, 2]
```

```
myvar = pd.Series(a, index = ["x", "y", "z"])
```

```
print(myvar)
```

# Key/Value Objects as Series

You can also use a key/value object, like a dictionary, when creating a Series.

## Example

Create a simple Pandas Series from a dictionary:

```
import pandas as pd
```

```
calories = {"day1": 420, "day2": 380, "day3": 390}
```

```
myvar = pd.Series(calories)
```

```
print(myvar)
```

```
day1    420  
day2    380  
day3    390  
dtype: int64
```

To select only some of the items in the dictionary, use the index argument and specify only the items you want to include in the Series.

### Example

Create a Series using only data from "day1" and "day2":

```
import pandas as pd
```

```
calories = {"day1": 420, "day2": 380, "day3": 390}
```

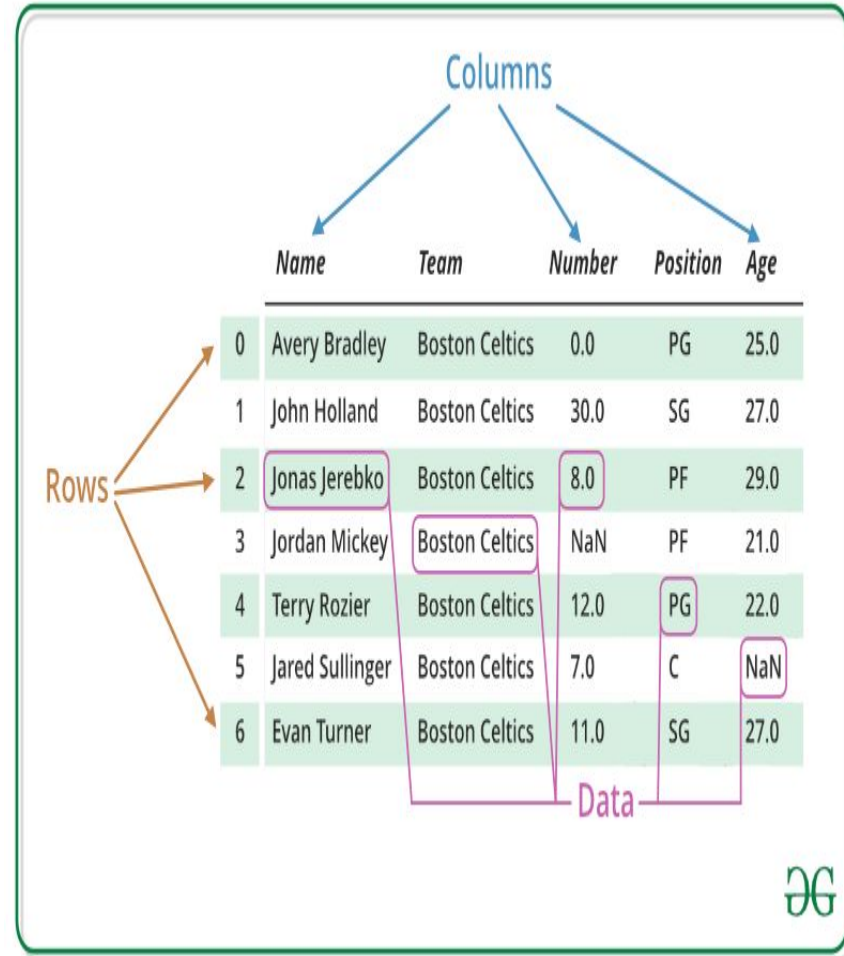
```
myvar = pd.Series(calories, index = ["day1", "day2"])
```

```
print(myvar)
```

```
day1    420  
day2    380  
dtype: int64
```


# Pandas DataFrame

It is two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the **data**, **rows**, and **columns**.



The diagram illustrates the structure of a Pandas DataFrame. At the top, the word "Columns" is written in blue. Below it, five arrows point to the column headers: "Name", "Team", "Number", "Position", and "Age". To the left of the table, the word "Rows" is written in orange. Four orange arrows point to the row indices 0, 1, 2, and 3. A purple box labeled "Data" is positioned at the bottom right, with a purple line connecting it to the data cells of the table. The table itself has a light green background and contains the following data:

	Name	Team	Number	Position	Age
0	Avery Bradley	Boston Celtics	0.0	PG	25.0
1	John Holland	Boston Celtics	30.0	SG	27.0
2	Jonas Jerebko	Boston Celtics	8.0	PF	29.0
3	Jordan Mickey	Boston Celtics	NaN	PF	21.0
4	Terry Rozier	Boston Celtics	12.0	PG	22.0
5	Jared Sullinger	Boston Celtics	7.0	C	NaN
6	Evan Turner	Boston Celtics	11.0	SG	27.0



## Data Frame Objects

Data sets in Pandas are usually multi-dimensional tables, called DataFrames.

Series is like a column, a DataFrame is the whole table.

Example

Create a DataFrame from two Series:

```
import pandas as pd
data = {
    "calories": [420, 380, 390],
    "duration": [50, 40, 45]
}
myvar = pd.DataFrame(data)
print(myvar)
```

	calories	duration
0	420	50
1	380	40
2	390	45



## What is a DataFrame?

A Pandas DataFrame is a 2 dimensional data structure, like a 2 dimensional array, or a table with rows and columns.

### Example

Create a simple Pandas DataFrame:

```
import pandas as pd
```

```
data = {  
    "calories": [420, 380, 390],  
    "duration": [50, 40, 45]  
}
```

	calories	duration
0	420	50
1	380	40
2	390	45

#load data into a DataFrame object:

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

```
print(df)
```

## Locate Row

As you can see from the result above, the DataFrame is like a table with rows and columns.

Pandas use the loc attribute to return one or more specified row(s)

## Example

Return row 0:

#refer to the row index:  
`print(df.loc[0])`

```
calories    420
duration     50
Name: 0, dtype: int64
```

Return row 0 and 1:

#use a list of indexes:  
`print(df.loc[[0, 1]])`

```
   calories  duration
0       420        50
1       380        40
```

## Named Indexes

With the index argument, you can name your own indexes.

### Example

Add a list of names to give each row a name:

```
import pandas as pd
```

```
data = {  
    "calories": [420, 380, 390],  
    "duration": [50, 40, 45]  
}
```

	calories	duration
day1	420	50
day2	380	40
day3	390	45

```
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])
```

```
print(df)
```

## Locate Named Indexes

Use the named index in the loc attribute to return the specified row(s).

### Example

Return "day2":

#refer to the named index:  
`print(df.loc["day2"])`

```
calories    380
duration     40
Name: 0, dtype: int64
```

## Load Files Into a DataFrame

If your data sets are stored in a file, Pandas can load them into a DataFrame.

### Example

Load a comma separated file (CSV file) into a DataFrame:

```
import pandas as pd
```

```
df = pd.read_csv('data.csv')
```

```
print(df)
```

## Output:

	Duration	Pulse	Maxpulse	Calories
0	60	110	130	409.1
1	60	117	145	479.0
2	60	103	135	340.0
3	45	109	175	282.4
4	45	117	148	406.0
..	...	...	...	...
164	60	105	140	290.8
165	60	110	145	300.4
166	60	115	145	310.2
167	75	120	150	320.4
168	75	125	150	330.4

[169 rows x 4 columns]

## Simple Operations with Data frames

Basic operation which can be performed on Pandas DataFrame :

Creating a DataFrame

Dealing with Rows and Columns

Indexing and Selecting Data

Working with Missing Data

Iterating over rows and columns

# Create a Pandas DataFrame from Lists

DataFrame can be created using a single list or a list of lists.

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

```
# list of strings  
lst = ['Geeks', 'For', 'Geeks', 'is',  
       'portal', 'for', 'Geeks']
```

```
# Calling DataFrame constructor  
on list  
df = pd.DataFrame(lst)  
print(df)
```

**Output:**

	0
0	Geeks
1	For
2	Geeks
3	is
4	portal
5	for
6	Geeks



# Dealing with Rows and Columns

A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. We can perform basic operations on rows/columns like selecting, deleting, adding, and renaming.

**Column Selection:** In Order to select a column in Pandas DataFrame, we can either access the columns by calling them by their columns name.

```
# Import pandas package
import pandas as pd
```

```
# Define a dictionary containing employee
data
```

```
data = {'Name':['Jai', 'Princi', 'Gaurav', 'Anuj'],
        'Age':[27, 24, 22, 32],
        'Address':['Delhi', 'Kanpur', 'Allahabad',
                    'Kannauj'],
        'Qualification':['Msc', 'MA', 'MCA',
                          'Phd']}
```

	Name	Qualification
0	Jai	Msc
1	Princi	MA
2	Gaurav	MCA
3	Anuj	Phd

```
# Convert the dictionary into DataFrame
df = pd.DataFrame(data)
```

```
# select two columns
print(df[['Name', 'Qualification']])
```

**Row Selection:** Pandas provide a unique method to retrieve rows from a Data frame. `DataFrame.loc[]` method is used to retrieve rows from Pandas DataFrame. Rows can also be selected by passing integer location to an `iloc[]` function.

```
# importing pandas package
import pandas as pd
# making data frame from csv file
data = pd.read_csv("nba.csv",
index_col="Name")
```

```
# retrieving row by loc method
first = data.loc["Avery Bradley"]
second = data.loc["R.J. Hunter"]
print(first, "\n\n", second)
```

```
print(first, "\n\n", second)
```

Team	Boston Celtics
Number	0
Position	PG
Age	25
Height	6-2
Weight	180
College	Texas
Salary	7.73034e+06

Name: Avery Bradley, dtype: object

Team	Boston Celtics
Number	28
Position	SG
Age	22
Height	6-5
Weight	185
College	Georgia State
Salary	1.14864e+06

Name: R.J. Hunter, dtype: object

# Indexing and Selecting Data

Indexing in pandas means simply selecting particular rows and columns of data from a DataFrame. Indexing could mean selecting all the rows and some of the columns, some of the rows and all of the columns, or some of each of the rows and columns. Indexing can also be known as Subset Selection.

Indexing a Dataframe using indexing operator [] :

Indexing operator is used to refer to the square brackets following an object. The .loc and .iloc indexers also use the indexing operator to make selections. In this indexing operator to refer to `df[]`.

## Selecting a single columns

In order to select a single column, we simply put the name of the column in-between the brackets

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

```
# making data frame from csv file  
data = pd.read_csv("nba.csv", index_col="Name")
```

```
# retrieving columns by indexing operator  
first = data["Age"]
```

```
print(first)
```

## Working with Missing Data

Checking for missing values using `isnull()` and `notnull()` :

In order to check missing values in Pandas DataFrame, we use a function `isnull()` and `notnull()`. Both function help in checking whether a value is NaN or not.

These function can also be used in Pandas Series in order to find null values in a series.

```
# importing pandas as pd
import pandas as pd
# importing numpy as np
import numpy as np
# dictionary of lists
dict = {'First Score':[100, 90, np.nan, 95],
        'Second Score': [30, 45, 56, np.nan],
        'Third Score':[np.nan, 40, 80, 98]}
# creating a dataframe from list
df = pd.DataFrame(dict)
# using isnull() function
df.isnull()
```

	First Score	Second Score	Third Score
0	False	False	True
1	False	False	False
2	True	False	False
3	False	True	False

## Querying from Data Frames

The `query()` method allows you to query the DataFrame. The `query()` method takes a query expression as a string parameter, which has to evaluate to either True or False. It returns the DataFrame where the result is True according to the query expression.

Syntax

```
dataframe.query(expr, inplace)
```

Parameter	Values	Description
<i>expr</i>		Required. A string that represents a query expression.
<i>inplace</i>	True False	Optional. A boolean value that specifies if the <code>query()</code> method should leave the original DataFrame untouched and return a copy ( <code>inplace = False</code> ). This is Default. Or: Make the changes in the original DataFrame ( <code>inplace = True</code> )

## Example:

Return the rows where age is over 35:

```
import pandas as pd
```

```
data = {  
    "name": ["Sally", "Mary", "John"],  
    "age": [50, 40, 30]  
}
```

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

```
print(df.query('age > 35'))
```

	name	age
0	Sally	50
1	Mary	40



## Applying Functions to Data frames

The `apply()` function is used to apply a function along an axis of the `DataFrame`.

Objects passed to the function are `Series` objects whose index is either the `DataFrame`'s index (`axis=0`) or the `DataFrame`'s columns (`axis=1`).

By default (`result_type=None`), the final return type is inferred from the return type of the applied function. Otherwise, it depends on the `result_type` argument.

Syntax:

```
DataFrame.apply(self, func, axis=0, broadcast=None, raw=False,  
reduce=None, result_type=None, args=(), **kwds)
```

## Parameters:

Name	Description	Type/Default Value	Required / Optional
func	Function to apply to each column or row.	function	Required
axis	Axis along which the function is applied: <ul style="list-style-type: none"> <li>0 or 'index': apply function to each column.</li> <li>1 or 'columns': apply function to each row.</li> </ul>	{0 or 'index', 1 or 'columns'} Default Value: 0	Required
raw	<ul style="list-style-type: none"> <li>False : passes each row or column as a Series to the function.</li> <li>True : the passed function will receive ndarray objects instead. If you are just applying a NumPy reduction function this will achieve much better performance.</li> </ul>	bool Default Value: False	Required
result_type	These only act when axis=1 (columns): <ul style="list-style-type: none"> <li>'expand' : list-like results will be turned into columns.</li> <li>'reduce' : returns a Series if possible rather than expanding list-like results. This is the opposite of 'expand'.</li> <li>'broadcast' : results will be broadcast to the original shape of the DataFrame, the original index and columns will be retained.</li> </ul> The default behaviour (None) depends on the return value of the applied function: list-like results will be returned as a Series of those. However if the apply function returns a Series these are expanded to columns.	{'expand', 'reduce', 'broadcast', None} Default Value: None	Required
args	Positional arguments to pass to func in addition to the array/series.	tuple	Required
**kwargs	Additional keyword arguments to pass as keywords arguments to func.		Required

## Returns: Series or DataFrame

Result of applying func along the given axis of the DataFrame.

## Example:

```
In [1]: import numpy as np  
import pandas as pd
```

```
In [2]: df = pd.DataFrame([[9, 25]] * 3, columns=['P', 'Q'])  
df
```

Out[2]:

	P	Q
0	9	25
1	9	25
2	9	25

`df = pd.DataFrame ([[ 9, 25]] * 3, columns = ['P', 'Q'])`

↓

DataFrame

df	P	Q
0	9	25
1	9	25
2	9	25

```
In [3]: df.apply(np.sqrt)
```

```
Out[3]:
```

	P	Q
0	3.0	5.0
1	3.0	5.0
2	3.0	5.0

`df.apply ( np.sqrt )`



DataFrame

df	P	Q
0	9	25
1	9	25
2	9	25



DataFrame

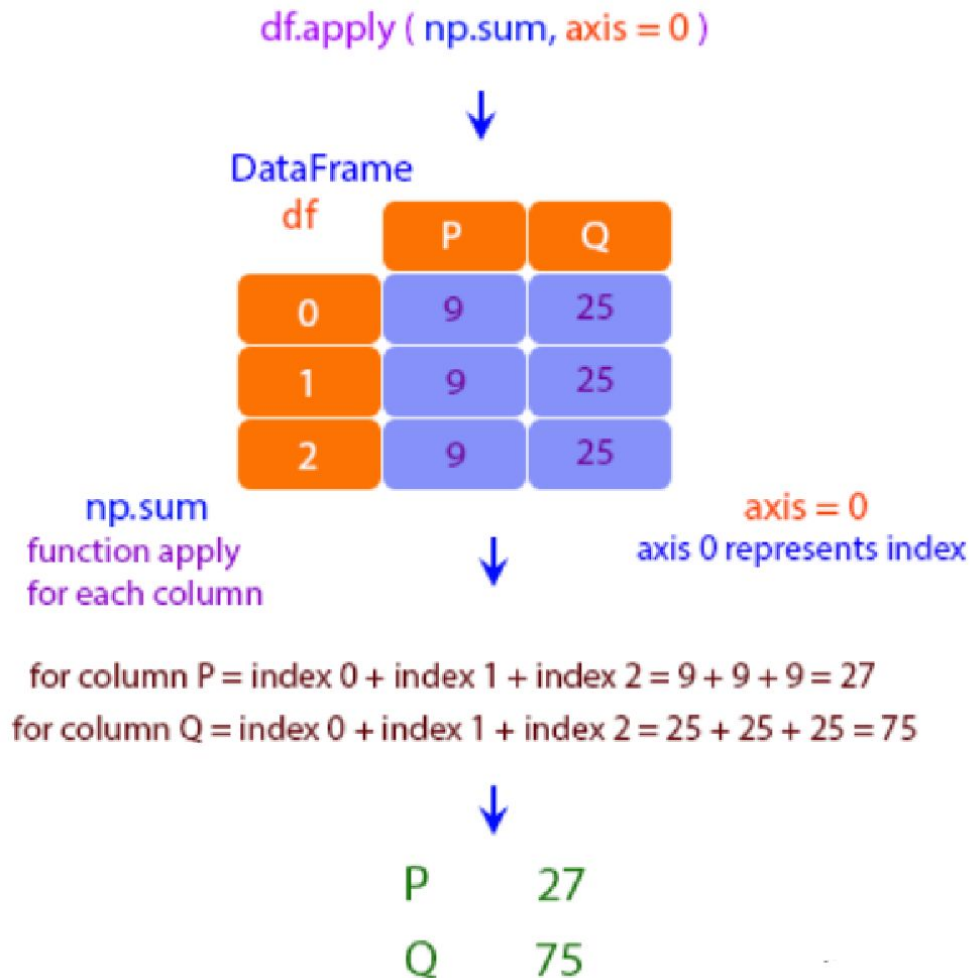
df	P	Q
0	3.0	5.0
1	3.0	5.0
2	3.0	5.0

square root  
of each element



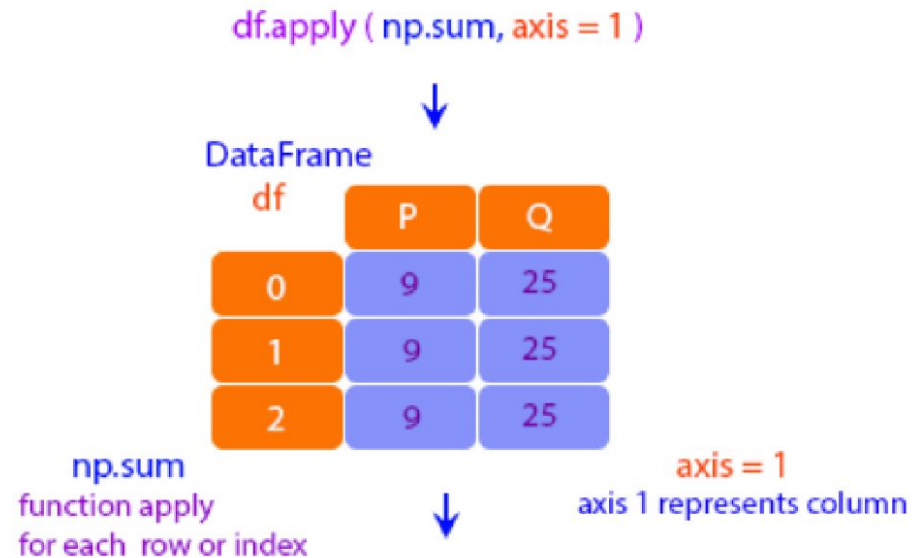
```
In [4]: df.apply(np.sum, axis=0)
```

```
Out[4]: P      27  
        Q      75  
        dtype: int64
```



```
In [5]: df.apply(np.sum, axis=1)
```

```
Out[5]: 0    34  
        1    34  
        2    34  
        dtype: int64
```



for index 0 = column P + column Q = 9 + 25 = 34

for index 1 = column P + column Q = 9 + 25 = 34

for index 2 = column P + column Q = 9 + 25 = 34

↓

0	34
1	34
2	34

Returning a list-like will result in a Series:

```
In [6]: df.apply(lambda x: [1, 2], axis=1)
```

```
Out[6]: 0    [1, 2]
        1    [1, 2]
        2    [1, 2]
        dtype: object
```

Passing result\_type='expand' will expand list-like results to columns of a Dataframe:

```
In [7]: df.apply(lambda x: [1, 2], axis=1, result_type='expand')
```

```
Out[7]:
```

	0	1
0	1	2
1	1	2
2	1	2

Returning a Series inside the function is similar to passing result\_type='expand'.  
The resulting column names will be the Series index.

```
In [8]: df.apply(lambda x: pd.Series([1, 2], index=['f1', 'b1']), axis=1)
```

Out[8]:

	f1	b1
0	1	2
1	1	2
2	1	2

Passing `result_type='broadcast'` will ensure the same shape result, whether list-like or scalar is returned by the function, and broadcast it along the axis. The resulting column names will be the originals.

```
In [9]: df.apply(lambda x: [1, 2], axis=1, result_type='broadcast')
```

Out[9]:

	P	Q
0	1	2
1	1	2
2	1	2



## Comparison between Numpy and Pandas

PANDAS	NUMPY
When we have to work on <b>Tabular data</b> , we prefer the <i>pandas</i> module.	When we have to work on <b>Numerical data</b> , we prefer the <i>numpy</i> module.
The powerful tools of pandas are <b>Data frame and Series</b> .	Whereas the powerful tool of <i>numpy</i> is <b>Arrays</b> .
<i>Pandas</i> consume <b>more memory</b> .	<i>Numpy</i> is <b>memory efficient</b> .
<i>Pandas</i> has a better performance when a number of rows is <b>500K or more</b> .	<i>Numpy</i> has a better performance when number of rows is <b>50K or less</b> .
Indexing of the <i>pandas</i> series is <b>very slow</b> as compared to <i>numpy</i> arrays.	Indexing of <i>numpy</i> Arrays is <b>very fast</b> .

## PANDAS

*Pandas* offer a have2d table object called **DataFrame**.

It was developed by Wes McKinney and was released in 2008.

It is used in a lot of organizations like Kaidee, Trivago, Abeja Inc. , and a lot more.

It has a higher industry application.

## NUMPY

*Numpy* is capable of providing **multi-dimensional arrays**.

It was developed by Travis Oliphant and was released in 2005.

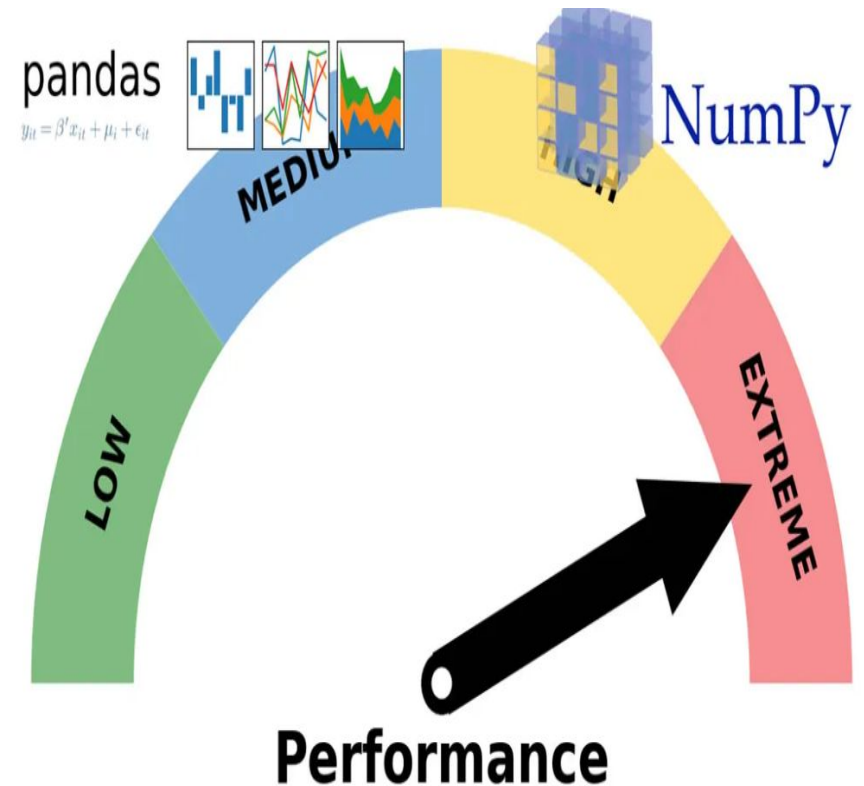
It is being used in organizations like Walmart Tokopedia, Instacart, and many more.

It has a lower industry application.

## Speed Testing between Numpy and Pandas

For Data Scientists, Pandas and Numpy are both essential tools in Python.

Numpy runs vector and matrix operations very efficiently, while Pandas provides the R-like data frames allowing intuitive tabular data analysis. A consensus is that Numpy is more optimized for arithmetic computations.



Ref:

<https://towardsdatascience.com/speed-testing-pandas-vs-numpy-ffbf80070ee7>

## Other Python Libraries

A Python library is a collection of related modules. It contains bundles of code that can be used repeatedly in different programs. It makes Python Programming simpler and convenient for the programmer. As we don't need to write the same code again and again for different programs. Python libraries play a very vital role in fields of Machine Learning, Data Science, Data Visualization, etc.

- 1.TensorFlow:** This library was developed by Google in collaboration with the Brain Team. It is an open-source library used for high-level computations. It is also used in machine learning and deep learning algorithms. It contains a large number of tensor operations. Researchers also use this Python library to solve complex computations in Mathematics and Physics.
- 2.Matplotlib:** This library is responsible for plotting numerical data. And that's why it is used in data analysis. It is also an open-source library and plots high-defined figures like pie charts, histograms, scatterplots, graphs, etc.
- 3.Pandas:** Pandas are an important library for data scientists. It is an open-source machine learning library that provides flexible high-level data structures and a variety of analysis tools. It eases data analysis, data manipulation, and cleaning of data. Pandas support operations like Sorting, Re-indexing, Iteration, Concatenation, Conversion of data, Visualizations, Aggregations, etc.

**4. Numpy:** The name “Numpy” stands for “Numerical Python”. It is the commonly used library. It is a popular machine learning library that supports large matrices and multi-dimensional data. It consists of in-built mathematical functions for easy computations. Even libraries like TensorFlow use Numpy internally to perform several operations on tensors. Array Interface is one of the key features of this library.

**5. SciPy:** The name “SciPy” stands for “Scientific Python”. It is an open-source library used for high-level scientific computations. This library is built over an extension of Numpy. It works with Numpy to handle complex computations. While Numpy allows sorting and indexing of array data, the numerical data code is stored in SciPy. It is also widely used by application developers and engineers.

**6. Scrappy:** It is an open-source library that is used for extracting data from websites. It provides very fast web crawling and high-level screen scraping. It can also be used for data mining and automated testing of data.

**7. Scikit-learn:** It is a famous Python library to work with complex data. Scikit-learn is an open-source library that supports machine learning. It supports variously supervised and unsupervised algorithms like linear regression, classification, clustering, etc. This library works in association with Numpy and SciPy.

**8. PyGame:** This library provides an easy interface to the Standard Directmedia Library (SDL) platform-independent graphics, audio, and input libraries. It is used for developing video games using computer graphics and audio libraries along with Python programming language.



**9. PyTorch:** PyTorch is the largest machine learning library that optimizes tensor computations. It has rich APIs to perform tensor computations with strong GPU acceleration. It also helps to solve application issues related to neural networks.

**10. PyBrain:** The name “PyBrain” stands for Python Based Reinforcement Learning, Artificial Intelligence, and Neural Networks library. It is an open-source library built for beginners in the field of Machine Learning. It provides fast and easy-to-use algorithms for machine learning tasks. It is so flexible and easily understandable and that’s why is really helpful for developers that are new in research fields.



# Use of Libraries in Python Program

Python libraries are used to create applications and models in a variety of fields, for instance, machine learning, data science, data visualization, image and data manipulation, and many more.

## Creating NumPy Array

NumPy is used to work with arrays.  
The array object in NumPy is  
called `ndarray`

## Activity

Consider the two-dimensional array, arr2d.

```
arr2d = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
```

Write a code to slice this array to display the last column,  
[[3] [6] [9]]

Write a code to slice this array to display the last 2 elements of  
middle array,  
[5 6]