UNIT-IV

Overview of the Unit

In this unit, we'll cover the following:

- **Advanced Python**
- **NumPy Library**
- **Pandas Library**
- **Data Visualization**
- **GUI Programming**

List Comprehensions in Python

List comprehension is an elegant way to define and create a list in python. We can create lists just like mathematical statements and in one line only. The syntax of list comprehension is easier to grasp. A list comprehension generally consists of these parts :

- Output expression,
- Input sequence,
- A variable representing a member of the input sequence and
- An optional predicate part.

General syntax is:

```
Lst = [expression(i) for i in another_list if filter(i)]
```

List Comprehension

Example -1:

Lst = [x**2   for x in range(1,11)   if x% 2 == 1]

In the above example,

- x ** 2 is the expression.
- range (1, 11) is input sequence or another list.
- x is the variable.
- if x % 2 == 1 is predicate part.

This is the power of list comprehension. It can identify when it receives a string or a tuple and work on it like a list.

Nested IF with List Comprehension

```
num_list = [y    for y in range(100)    if y%2==0    if y%5==0]
print(num_list)
```

List Comprehension

Example -2 :

Suppose, we want to separate the letters of the word 'human' and add the letters as items of a list. The first thing that comes in mind would be using for loop.

```
h_letters = []
for letter in 'human':
          h_letters.append(letter)
print(h_letters)
```

Python has an easier way to solve this issue using List Comprehension.

```
h_letters = [letter for letter in 'human' ]
print(h_letters)
```

List Comprehension

In **example-2**, we can see that 'human' is a string, not a list. This is the power of list comprehension. It can identify when it receives a string or a tuple and work on it like a list.

You can do that using loops.

However, not every loop can be rewritten as list comprehension. But as you learn and get comfortable with list comprehensions, you will find yourself replacing more and more loops with this elegant syntax.

Advantages of List Comprehension

Advantages of List Comprehension

- More time efficient and space efficient than loops.
- Require fewer lines of code.
- Transforms iterative statement into a formula.

Key Points to Remember

- we should avoid writing very long list comprehensions in one line to ensure that code is user-friendly.
- Every list comprehension can be rewritten in for loop, but every for loop can't be rewritten in the form of list comprehension.

map() Function

The advantage of the lambda function can be seen when it is used in combination with the map() function. map() is a function which takes two arguments:

r = map(func, seq)

- The first argument func is the name of a function and the second a sequence (e.g. a list) seq. map() applies the function func to all the elements of the sequence seq.
- Before Python3, map() used to return a list, where each element of the result list was the result of the function func applied on the corresponding element of the list or tuple "seq". With Python 3, map() returns an iterator.

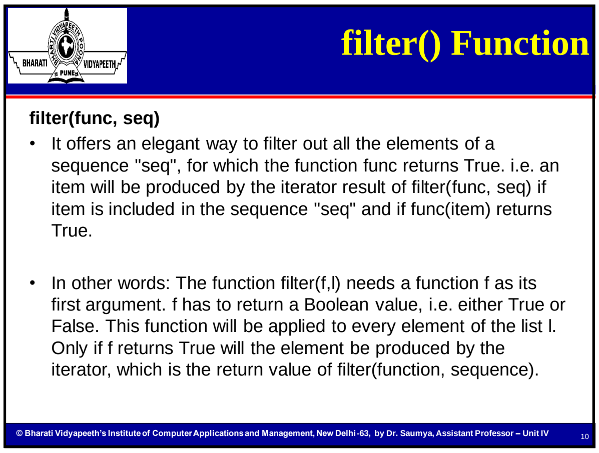
map() Function

- The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Example

Calculate the length of each word in the tuple:

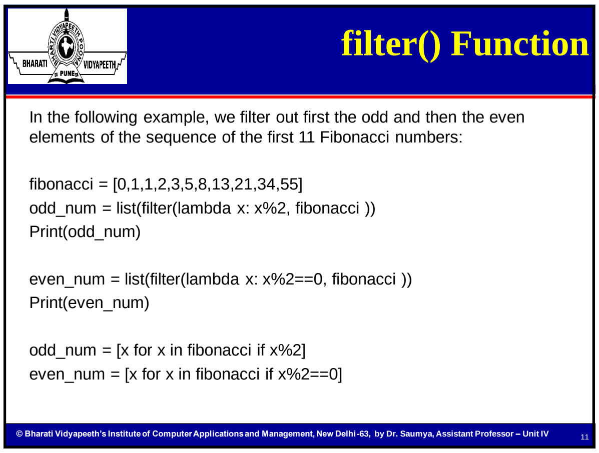
```
Def myFunc(s):
    return(len(s))
X = map(myFunc,('Apple','Banana','kiwi'))
Print(list(X))

X = list(map(lambda s: len(s) ,('Apple','Banana','kiwi')))
print(X)

X=[len(s)  for s in ('Apple','Banana','kiwi')]
print(X)
```



filter() Function

filter(func, seq)

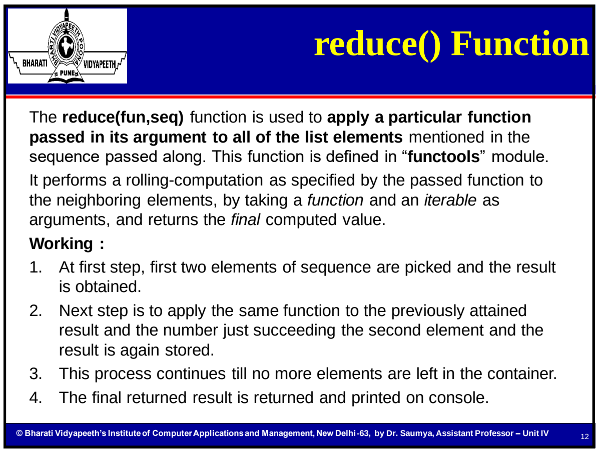
- It offers an elegant way to filter out all the elements of a sequence "seq", for which the function func returns True. i.e. an item will be produced by the iterator result of filter(func, seq) if item is included in the sequence "seq" and if func(item) returns True.
- In other words: The function filter(f,l) needs a function f as its first argument. f has to return a Boolean value, i.e. either True or False. This function will be applied to every element of the list l. Only if f returns True will the element be produced by the iterator, which is the return value of filter(function, sequence).



filter() Function

In the following example, we filter out first the odd and then the even elements of the sequence of the first 11 Fibonacci numbers:

```
fibonacci = [0,1,1,2,3,5,8,13,21,34,55]
odd_num = list(filter(lambda x: x%2, fibonacci ))
Print(odd_num)

even_num = list(filter(lambda x: x%2==0, fibonacci ))
Print(even_num)

odd_num = [x for x in fibonacci if x%2]
even_num = [x for x in fibonacci if x%2==0]
```



reduce() Function

The **reduce(fun,seq)** function is used to **apply a particular function passed in its argument to all of the list elements** mentioned in the sequence passed along. This function is defined in "**functools**" module.

It performs a rolling-computation as specified by the passed function to the neighboring elements, by taking a *function* and an *iterable* as arguments, and returns the *final* computed value.

Working :

1. At first step, first two elements of sequence are picked and the result is obtained.
2. Next step is to apply the same function to the previously attained result and the number just succeeding the second element and the result is again stored.
3. This process continues till no more elements are left in the container.
4. The final returned result is returned and printed on console.

reduce() Function

Example:

```
from functools import reduce
# Returns the sum of two elements
def sumTwo(a,b):
   return a+b

result = reduce(sumTwo, [1, 2, 3, 4])
print(result)

from functools import reduce
# Returns the sum of all the elements using `reduce`
result = reduce((lambda a, b: a + b), [1, 2, 3, 4])
print(result)
```

Comparison of Lambda and List Comprehension

- List Comprehension is used to create lists, Lambdas are functions that can process like other functions and thus return values or list.
- Lambda function process is the same as other functions and returns the value of the list. The **Lambda** function itself cannot be used to iterate through a list. It return a list with the help of **map()** and **list()** functions.

  list(map(lambda argument: manipulate(argument), iterable))
- List comprehension performance is better than lambda because filter() in lambda is slower than list comprehension.

NumPy Library

- NumPy, which stands for Numerical Python, is a library consisting of multidimensional array objects and a collection of routines for processing those arrays.
- NumPy is a general-purpose array-processing package. It provides a high-performance multidimensional array object, and tools for working with these arrays : the **n-dimensional array**. This is simple yet powerful data structure.
- 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.

NumPy Library continued ….

- It is the fundamental package for scientific computing with Python. It contains various features including these important ones:
- A powerful N-dimensional array object
- Sophisticated (broadcasting) functions
- Tools for integrating C/C++ and Fortran code
- Useful linear algebra, Fourier transform, and random number capabilities

Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.

Arbitrary data-types can be defined using Numpy which allows NumPy to seamlessly and speedily integrate with a wide variety of databases.

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.

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++.

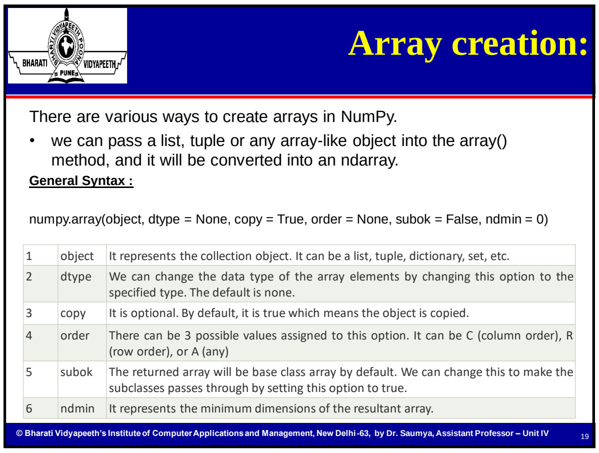
Arrays in NumPy

NumPy's main object is the homogeneous multidimensional array.

- It is a table of elements (usually numbers), all of the same type, indexed by a tuple of positive integers.
- In NumPy dimensions are called *axes*. The number of axes is *rank*.
- NumPy's array class is called **ndarray**. It is also known by the alias **array**.
- Items in the collection can be accessed using a zero-based index.
- Every item in an ndarray takes the same size of block in the memory.

Example:

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



Array creation:

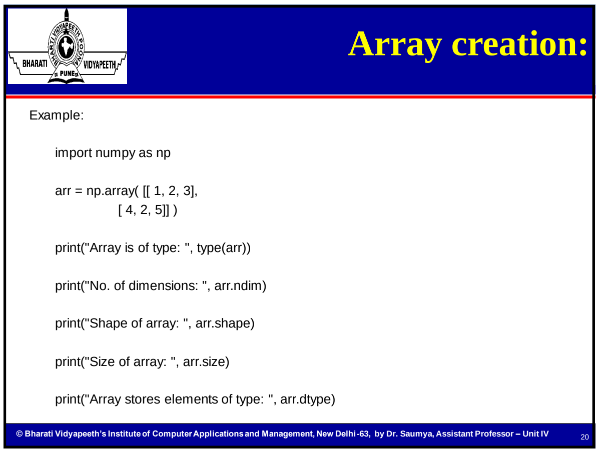
There are various ways to create arrays in NumPy.

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

General Syntax :

```
numpy.array(object, dtype = None, copy = True, order = None, subok = False, ndmin = 0)
```

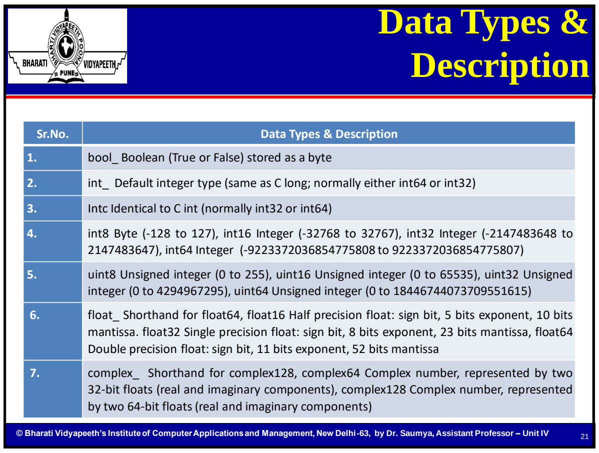
1	object	It represents the collection object. It can be a list, tuple, dictionary, set, etc.
2	dtype	We can change the data type of the array elements by changing this option to the specified type. The default is none.
3	copy	It is optional. By default, it is true which means the object is copied.
4	order	There can be 3 possible values assigned to this option. It can be C (column order), R (row order), or A (any)
5	subok	The returned array will be base class array by default. We can change this to make the subclasses passes through by setting this option to true.
6	ndmin	It represents the minimum dimensions of the resultant array.



Array creation:

Example:

```
import numpy as np

arr = np.array( [[ 1, 2, 3],
                [ 4, 2, 5]] )

print("Array is of type: ", type(arr))

print("No. of dimensions: ", arr.ndim)

print("Shape of array: ", arr.shape)

print("Size of array: ", arr.size)

print("Array stores elements of type: ", arr.dtype)
```



Data Types & Description

Sr.No.	Data Types & Description
1.	bool_ Boolean (True or False) stored as a byte
2.	int_ Default integer type (same as C long; normally either int64 or int32)
3.	Intc Identical to C int (normally int32 or int64)
4.	int8 Byte (-128 to 127), int16 Integer (-32768 to 32767), int32 Integer (-2147483648 to 2147483647), int64 Integer (-9223372036854775808 to 9223372036854775807)
5.	uint8 Unsigned integer (0 to 255), uint16 Unsigned integer (0 to 65535), uint32 Unsigned integer (0 to 4294967295), uint64 Unsigned integer (0 to 18446744073709551615)
6.	float_ Shorthand for float64, float16 Half precision float: sign bit, 5 bits exponent, 10 bits mantissa. float32 Single precision float: sign bit, 8 bits exponent, 23 bits mantissa, float64 Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
7.	complex_ Shorthand for complex128, complex64 Complex number, represented by two 32-bit floats (real and imaginary components), complex128 Complex number, represented by two 64-bit floats (real and imaginary components)

Data Types & Description

- Each built-in data type has a character code that uniquely identifies it.
- **'b'** − boolean
- **'i'** − (signed) integer
- **'u'** − unsigned integer
- **'f'** − floating-point
- **'c'** − complex-floating point
- **'m'** − timedelta
- **'M'** − datetime
- **'O'** − (Python) objects
- **'S', 'a'** − (byte-)string
- **'U'** − Unicode
- **'V'** − raw data (void)

Try This :

```
import numpy as np
student = np.dtype([('name','S20'), ('age', 'i1'), ('marks', 'f4')])
a = np.array([('abc', 21, 50),('xyz', 18, 75)], dtype = student)
print a
```

Dimensions in Arrays

0-D Arrays

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

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

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.

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

Dimensions in Arrays

2-D Arrays

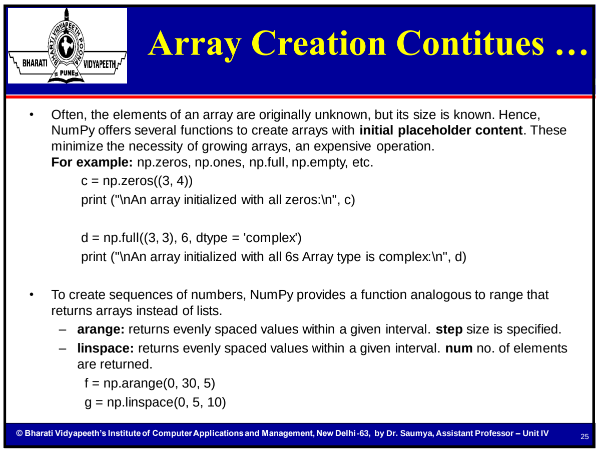
- An array that has 1-D arrays as its elements is called a 2-D array.
- These are often used to represent matrix or 2nd order tensors.

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

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.

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



Array Creation Contitues …

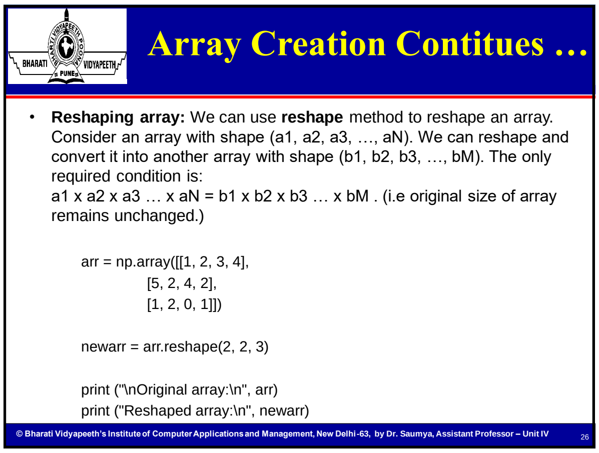
- Often, the elements of an array are originally unknown, but its size is known. Hence, NumPy offers several functions to create arrays with **initial placeholder content**. These minimize the necessity of growing arrays, an expensive operation.
  For example: np.zeros, np.ones, np.full, np.empty, etc.

```
c = np.zeros((3, 4))
print ("\nAn array initialized with all zeros:\n", c)

d = np.full((3, 3), 6, dtype = 'complex')
print ("\nAn array initialized with all 6s Array type is complex:\n", d)
```

- To create sequences of numbers, NumPy provides a function analogous to range that returns arrays instead of lists.
  - **arange:** returns evenly spaced values within a given interval. **step** size is specified.
  - **linspace:** returns evenly spaced values within a given interval. **num** no. of elements are returned.

```
f = np.arange(0, 30, 5)
g = np.linspace(0, 5, 10)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 25

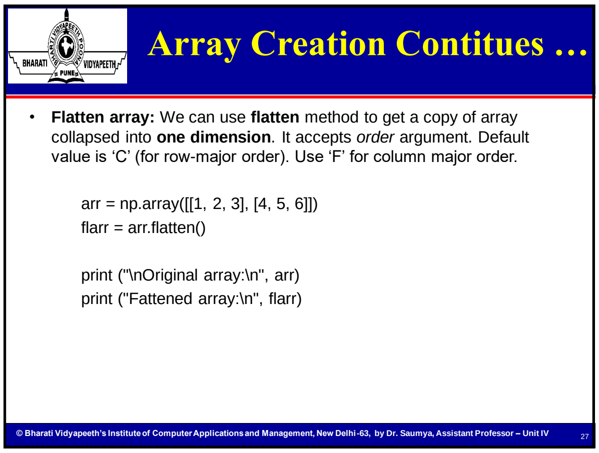


Array Creation Contitues …

- **Reshaping array:** We can use **reshape** method to reshape an array. Consider an array with shape (a1, a2, a3, …, aN). We can reshape and convert it into another array with shape (b1, b2, b3, …, bM). The only required condition is:
  a1 x a2 x a3 … x aN = b1 x b2 x b3 … x bM . (i.e original size of array remains unchanged.)

```
arr = np.array([[1, 2, 3, 4],
                [5, 2, 4, 2],
                [1, 2, 0, 1]])

newarr = arr.reshape(2, 2, 3)

print ("\nOriginal array:\n", arr)
print ("Reshaped array:\n", newarr)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 26

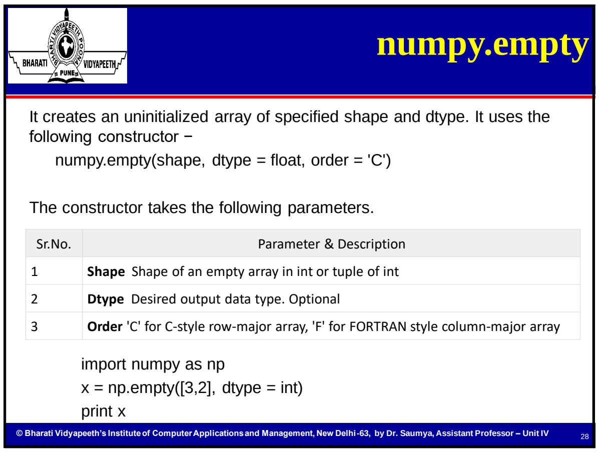


Array Creation Contitues …

- **Flatten array:** We can use **flatten** method to get a copy of array collapsed into **one dimension**. It accepts *order* argument. Default value is 'C' (for row-major order). Use 'F' for column major order.

```
arr = np.array([[1, 2, 3], [4, 5, 6]])
flarr = arr.flatten()

print ("\nOriginal array:\n", arr)
print ("Fattened array:\n", flarr)
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 27



numpy.empty

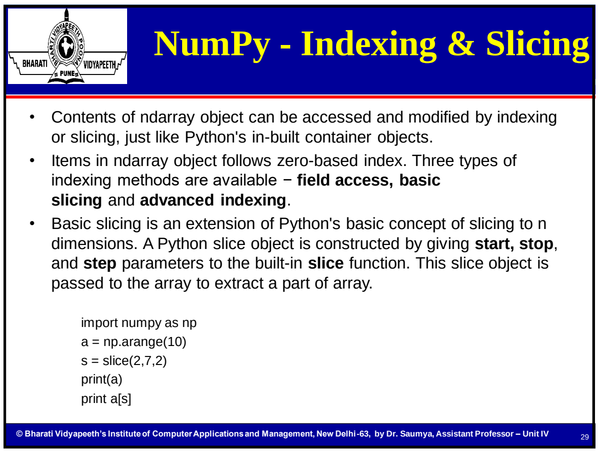
It creates an uninitialized array of specified shape and dtype. It uses the following constructor −

```
numpy.empty(shape, dtype = float, order = 'C')
```

The constructor takes the following parameters.

Sr.No.	Parameter & Description
1	**Shape** Shape of an empty array in int or tuple of int
2	**Dtype** Desired output data type. Optional
3	**Order** 'C' for C-style row-major array, 'F' for FORTRAN style column-major array

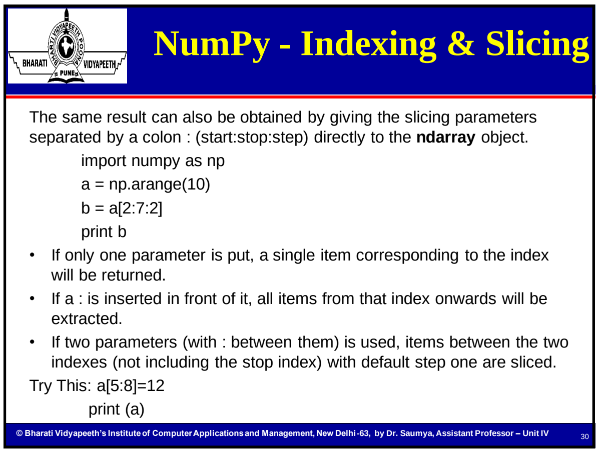
```
import numpy as np
x = np.empty([3,2], dtype = int)
print x
```



NumPy - Indexing & Slicing

- 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.

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



NumPy - Indexing & Slicing

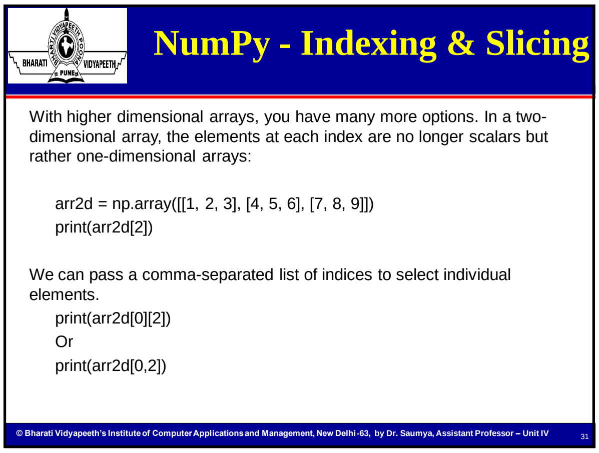
The same result can also be obtained by giving the slicing parameters separated by a colon : (start:stop:step) directly to the **ndarray** object.

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

- If only one parameter is put, a single item corresponding to the index will be returned.
- If a : is inserted in front of it, all items from that index onwards will be extracted.
- If two parameters (with : between them) is used, items between the two indexes (not including the stop index) with default step one are sliced.

Try This: a[5:8]=12

```
print (a)
```



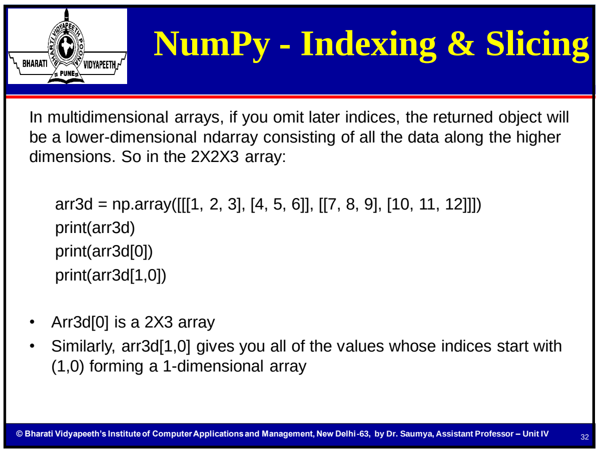
NumPy - Indexing & Slicing

With higher dimensional arrays, you have many more options. In a two-dimensional array, the elements at each index are no longer scalars but rather one-dimensional arrays:

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

We can pass a comma-separated list of indices to select individual elements.

```
print(arr2d[0][2])
Or
print(arr2d[0,2])
```

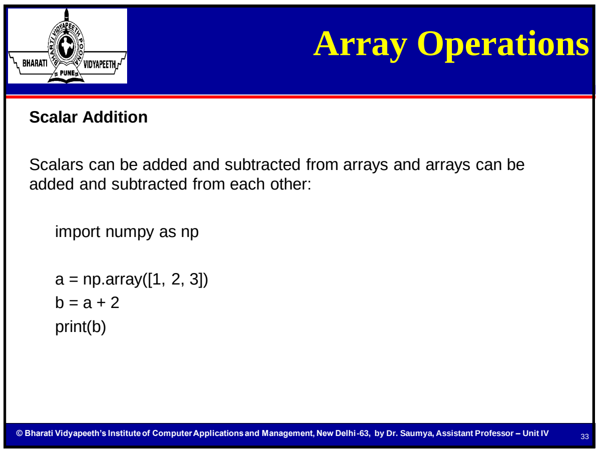


NumPy - Indexing & Slicing

In multidimensional arrays, if you omit later indices, the returned object will be a lower-dimensional ndarray consisting of all the data along the higher dimensions. So in the 2X2X3 array:

```
arr3d = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])
print(arr3d)
print(arr3d[0])
print(arr3d[1,0])
```

- Arr3d[0] is a 2X3 array
- Similarly, arr3d[1,0] gives you all of the values whose indices start with (1,0) forming a 1-dimensional array

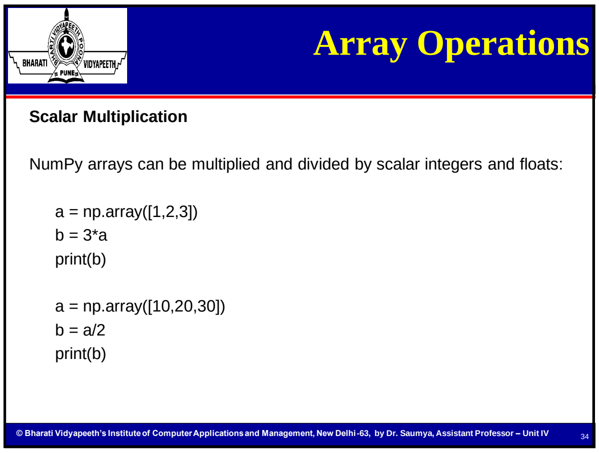


Array Operations

Scalar Addition

Scalars can be added and subtracted from arrays and arrays can be added and subtracted from each other:

```
import numpy as np

a = np.array([1, 2, 3])
b = a + 2
print(b)
```

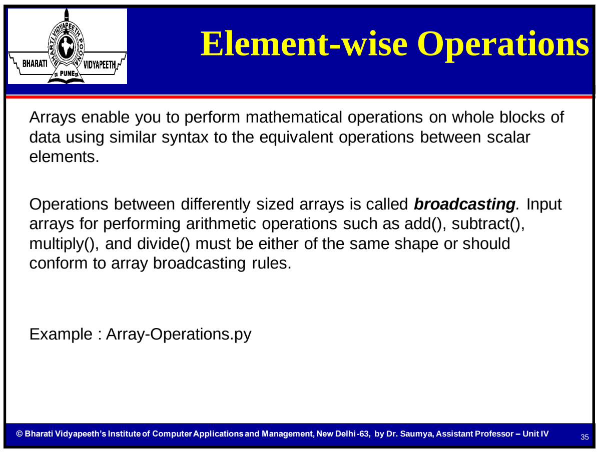


Array Operations

Scalar Multiplication

NumPy arrays can be multiplied and divided by scalar integers and floats:

```
a = np.array([1,2,3])
b = 3*a
print(b)

a = np.array([10,20,30])
b = a/2
print(b)
```

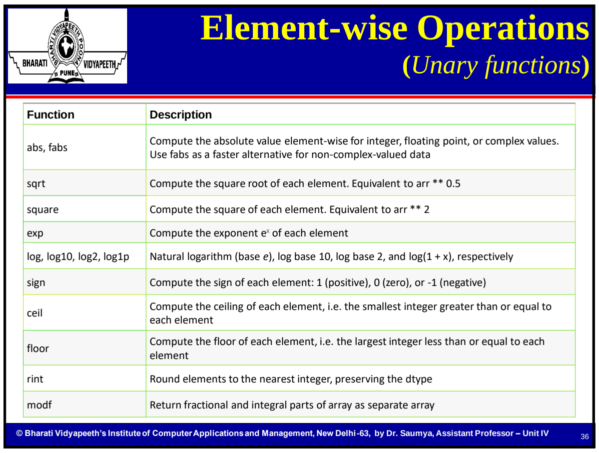


Element-wise Operations

Arrays enable you to perform mathematical operations on whole blocks of data using similar syntax to the equivalent operations between scalar elements.

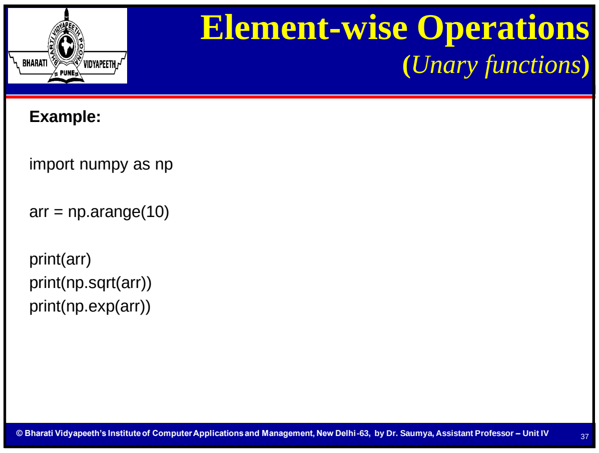
Operations between differently sized arrays is called ***broadcasting***. Input arrays for performing arithmetic operations such as add(), subtract(), multiply(), and divide() must be either of the same shape or should conform to array broadcasting rules.

Example : Array-Operations.py



Element-wise Operations (*Unary functions*)

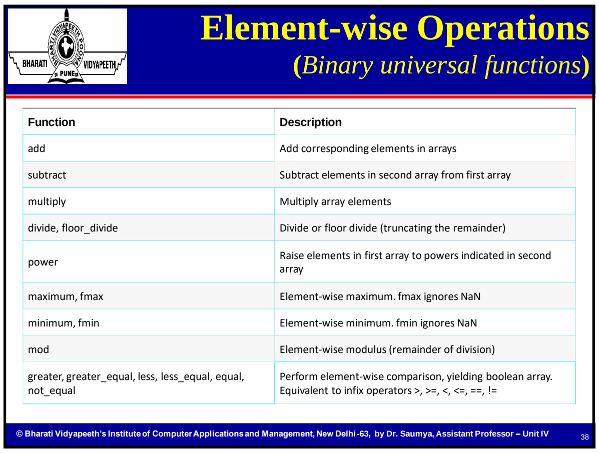
Function	Description
abs, fabs	Compute the absolute value element-wise for integer, floating point, or complex values. Use fabs as a faster alternative for non-complex-valued data
sqrt	Compute the square root of each element. Equivalent to arr ** 0.5
square	Compute the square of each element. Equivalent to arr ** 2
exp	Compute the exponent e^x of each element
log, log10, log2, log1p	Natural logarithm (base *e*), log base 10, log base 2, and log(1 + x), respectively
sign	Compute the sign of each element: 1 (positive), 0 (zero), or -1 (negative)
ceil	Compute the ceiling of each element, i.e. the smallest integer greater than or equal to each element
floor	Compute the floor of each element, i.e. the largest integer less than or equal to each element
rint	Round elements to the nearest integer, preserving the dtype
modf	Return fractional and integral parts of array as separate array



Element-wise Operations
(Unary functions)

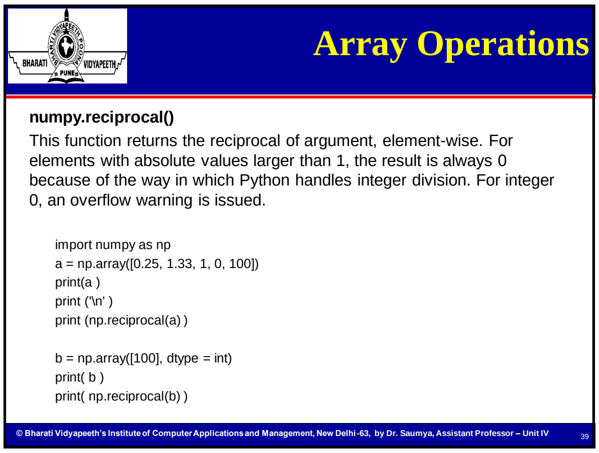
Example:

```
import numpy as np

arr = np.arange(10)

print(arr)
print(np.sqrt(arr))
print(np.exp(arr))
```



Element-wise Operations
(Binary universal functions)

Function	Description
add	Add corresponding elements in arrays
subtract	Subtract elements in second array from first array
multiply	Multiply array elements
divide, floor_divide	Divide or floor divide (truncating the remainder)
power	Raise elements in first array to powers indicated in second array
maximum, fmax	Element-wise maximum. fmax ignores NaN
minimum, fmin	Element-wise minimum. fmin ignores NaN
mod	Element-wise modulus (remainder of division)
greater, greater_equal, less, less_equal, equal, not_equal	Perform element-wise comparison, yielding boolean array. Equivalent to infix operators >, >=, <, <=, ==, !=

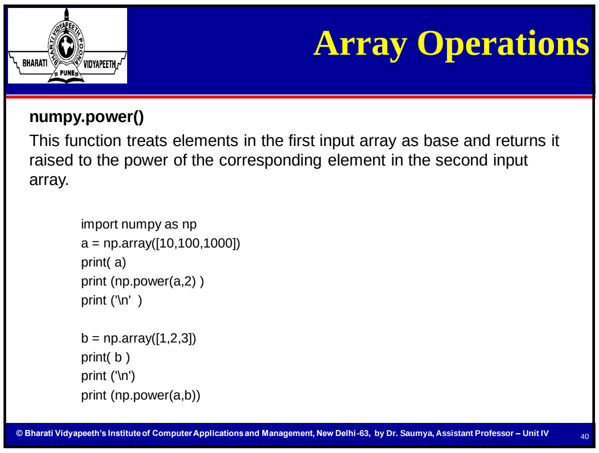


Array Operations

numpy.reciprocal()

This function returns the reciprocal of argument, element-wise. For elements with absolute values larger than 1, the result is always 0 because of the way in which Python handles integer division. For integer 0, an overflow warning is issued.

```
import numpy as np
a = np.array([0.25, 1.33, 1, 0, 100])
print(a )
print ('\n' )
print (np.reciprocal(a) )

b = np.array([100], dtype = int)
print( b )
print( np.reciprocal(b) )
```

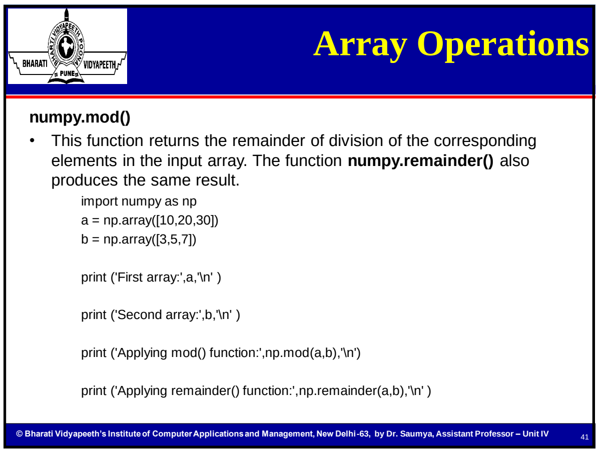


Array Operations

numpy.power()

This function treats elements in the first input array as base and returns it raised to the power of the corresponding element in the second input array.

```
import numpy as np
a = np.array([10,100,1000])
print( a)
print (np.power(a,2) )
print ('\n'  )

b = np.array([1,2,3])
print( b )
print ('\n')
print (np.power(a,b))
```

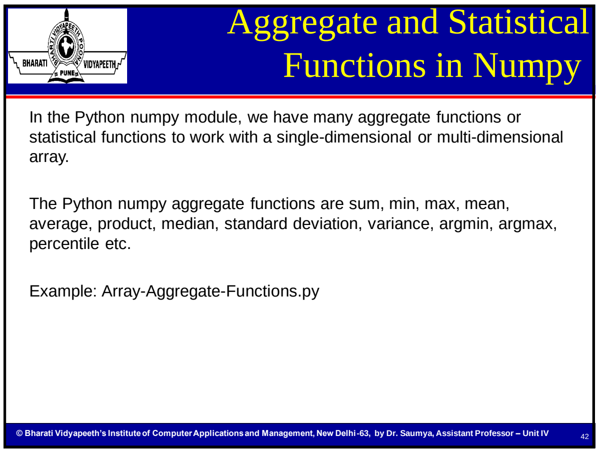


Array Operations

numpy.mod()

- This function returns the remainder of division of the corresponding elements in the input array. The function **numpy.remainder()** also produces the same result.

```
import numpy as np
a = np.array([10,20,30])
b = np.array([3,5,7])

print ('First array:',a,'\n' )

print ('Second array:',b,'\n' )

print ('Applying mod() function:',np.mod(a,b),'\n')

print ('Applying remainder() function:',np.remainder(a,b),'\n' )
```

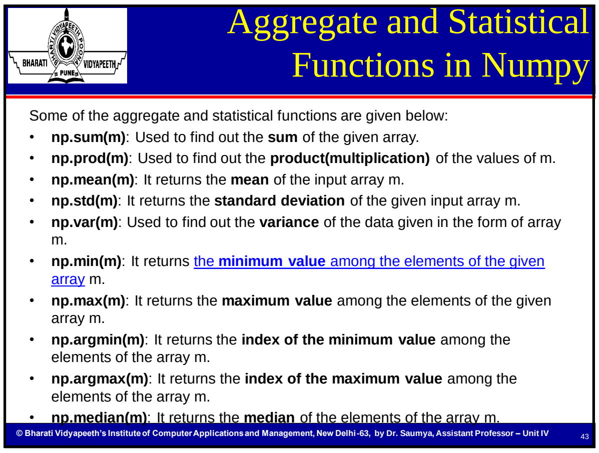


Aggregate and Statistical Functions in Numpy

In the Python numpy module, we have many aggregate functions or statistical functions to work with a single-dimensional or multi-dimensional array.

The Python numpy aggregate functions are sum, min, max, mean, average, product, median, standard deviation, variance, argmin, argmax, percentile etc.

Example: Array-Aggregate-Functions.py

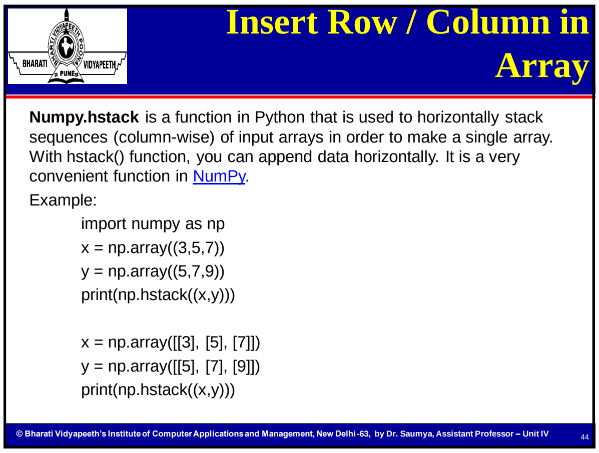


Aggregate and Statistical Functions in Numpy

Some of the aggregate and statistical functions are given below:

- **np.sum(m)**: Used to find out the **sum** of the given array.
- **np.prod(m)**: Used to find out the **product(multiplication)** of the values of m.
- **np.mean(m)**: It returns the **mean** of the input array m.
- **np.std(m)**: It returns the **standard deviation** of the given input array m.
- **np.var(m)**: Used to find out the **variance** of the data given in the form of array m.
- **np.min(m)**: It returns the **minimum value** among the elements of the given array m.
- **np.max(m)**: It returns the **maximum value** among the elements of the given array m.
- **np.argmin(m)**: It returns the **index of the minimum value** among the elements of the array m.
- **np.argmax(m)**: It returns the **index of the maximum value** among the elements of the array m.
- **np.median(m)**: It returns the **median** of the elements of the array m.

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 43



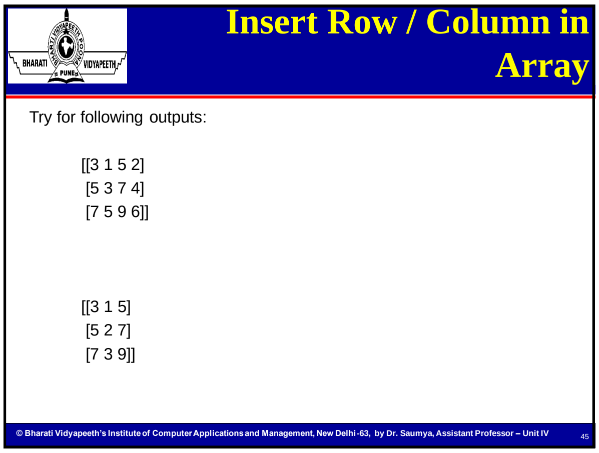
Insert Row / Column in Array

Numpy.hstack is a function in Python that is used to horizontally stack sequences (column-wise) of input arrays in order to make a single array. With hstack() function, you can append data horizontally. It is a very convenient function in NumPy.

Example:

```
import numpy as np
x = np.array((3,5,7))
y = np.array((5,7,9))
print(np.hstack((x,y)))

x = np.array([[3], [5], [7]])
y = np.array([[5], [7], [9]])
print(np.hstack((x,y)))
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 44



Insert Row / Column in Array

Try for following outputs:

```
[[3 1 5 2]
 [5 3 7 4]
 [7 5 9 6]]



[[3 1 5]
 [5 2 7]
 [7 3 9]]
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 45

Insert Row / Column in Array

The vstack() function is used to stack arrays in sequence vertically (row wise).

Example :

```
x = np.array([3, 5, 7])
y = np.array([5, 7, 9])
print(np.vstack((x,y)))

x = np.array([[3], [5], [7]])
y = np.array([[5], [7], [9]])
print(np.vstack((x,y)))
```

Append Row / Column in Array

You can add a NumPy array element by using the append() method of the NumPy module.

The syntax of append is as follows:

```
numpy.append(array, value, axis)
```

- The values will be appended at the end of the array and a new ndarray will be returned with new and old values.
- The axis is an optional integer along which define how the array is going to be displayed. If the axis is not specified, the array structure will be flattened as you will see later.

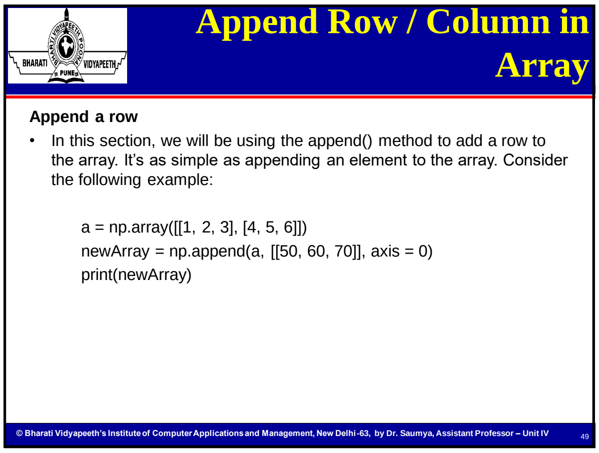
```
a = np.array([1, 2, 3])
newArray = np.append (a, [10, 11, 12])
print(newArray)
```

Append Row / Column in Array

Add a column

- We can use the append() method of NumPy to insert a column.
- Consider the example below where we created a 2-dimensional array and inserted two columns:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
b = np.array([[400], [800]])
newArray = np.append(a, b, axis=1)
print(newArray)
```

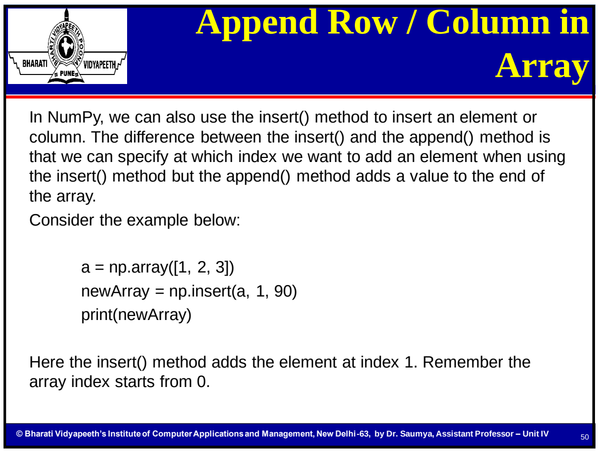


Append Row / Column in Array

Append a row

- In this section, we will be using the append() method to add a row to the array. It's as simple as appending an element to the array. Consider the following example:

```
a = np.array([[1, 2, 3], [4, 5, 6]])
newArray = np.append(a, [[50, 60, 70]], axis = 0)
print(newArray)
```



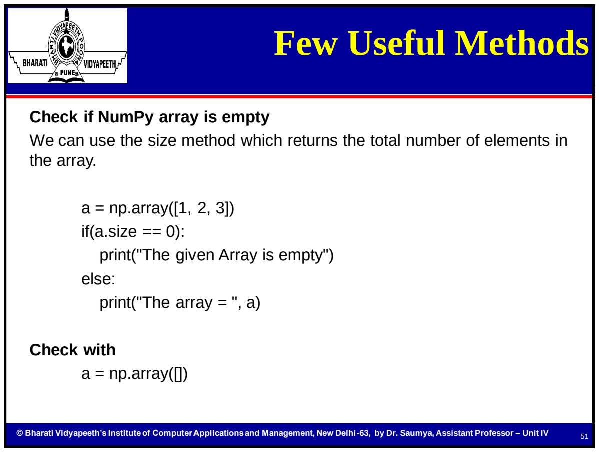
Append Row / Column in Array

In NumPy, we can also use the insert() method to insert an element or column. The difference between the insert() and the append() method is that we can specify at which index we want to add an element when using the insert() method but the append() method adds a value to the end of the array.

Consider the example below:

```
a = np.array([1, 2, 3])
newArray = np.insert(a, 1, 90)
print(newArray)
```

Here the insert() method adds the element at index 1. Remember the array index starts from 0.



Few Useful Methods

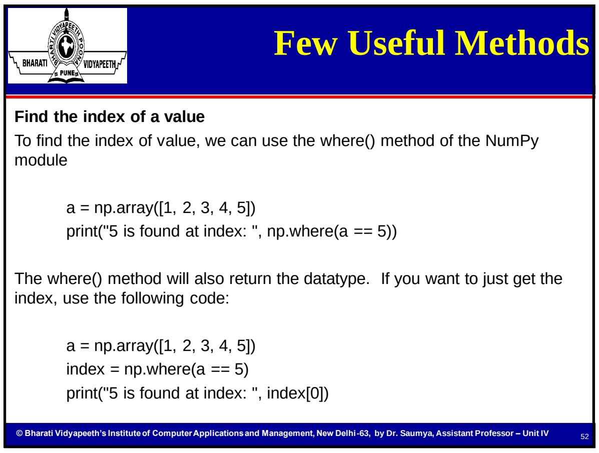
Check if NumPy array is empty

We can use the size method which returns the total number of elements in the array.

```
a = np.array([1, 2, 3])
if(a.size == 0):
    print("The given Array is empty")
else:
    print("The array = ", a)
```

Check with

```
a = np.array([])
```



Few Useful Methods

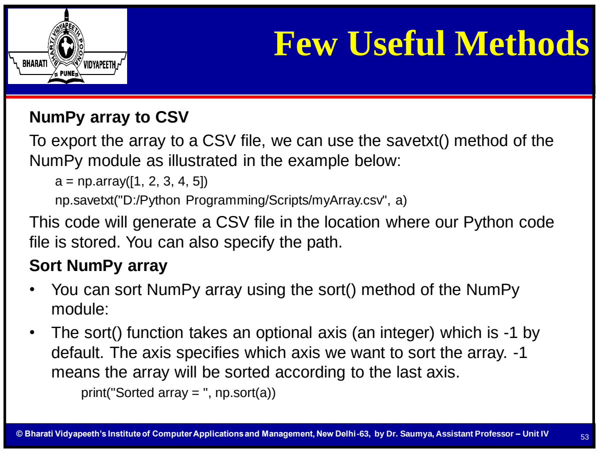
Find the index of a value

To find the index of value, we can use the where() method of the NumPy module

```
a = np.array([1, 2, 3, 4, 5])
print("5 is found at index: ", np.where(a == 5))
```

The where() method will also return the datatype. If you want to just get the index, use the following code:

```
a = np.array([1, 2, 3, 4, 5])
index = np.where(a == 5)
print("5 is found at index: ", index[0])
```



Few Useful Methods

NumPy array to CSV

To export the array to a CSV file, we can use the savetxt() method of the NumPy module as illustrated in the example below:

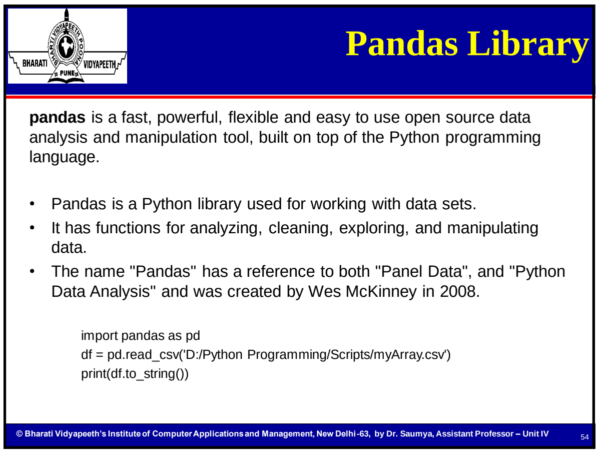
```
a = np.array([1, 2, 3, 4, 5])
np.savetxt("D:/Python Programming/Scripts/myArray.csv", a)
```

This code will generate a CSV file in the location where our Python code file is stored. You can also specify the path.

Sort NumPy array

- You can sort NumPy array using the sort() method of the NumPy module:
- The sort() function takes an optional axis (an integer) which is -1 by default. The axis specifies which axis we want to sort the array. -1 means the array will be sorted according to the last axis.

```
print("Sorted array = ", np.sort(a))
```



Pandas Library

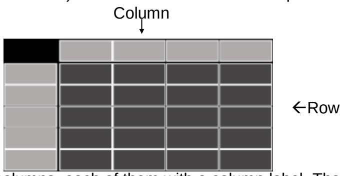
pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language.

- 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.

```
import pandas as pd
df = pd.read_csv('D:/Python Programming/Scripts/myArray.csv')
print(df.to_string())
```

Data -Frames

A **DataFrame** is a 2-dimensional data structure that can store data of different types (including characters, integers, floating point values, categorical data and more) in columns. It is similar to a spreadsheet, a SQL table



The table has 3 columns, each of them with a column label. The column labels are respectively Name, Age and Gender.

The column Name consists of textual data with each value a string, the column Age are numbers and the column Gender is textual data.

Data -Frames

```
df = pd.DataFrame(
{
   "Name": [
   "Braund, Mr. Owen Harris",
   "Allen, Mr. William Henry",
   "Bonnell, Miss. Elizabeth",
   ],
   "Age": [22, 35, 58],
   “Gender": ["male", "male", "female"],
}
)
print(df)
```

Data -Frames

Locate Row

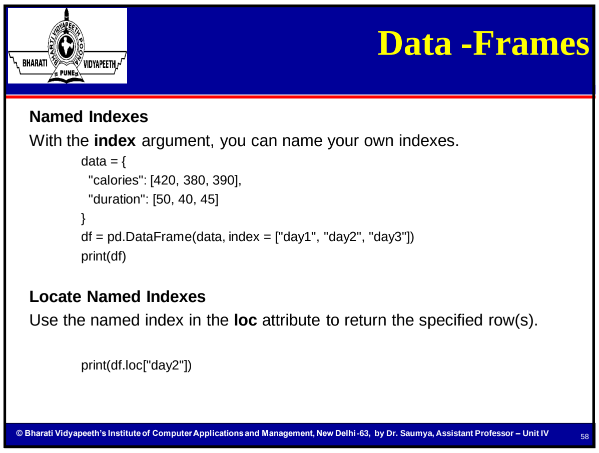
Pandas use the **loc** attribute to return one or more specified row(s)

```
print(df.loc[0])
```

This returns a Pandas **Series**.

```
print(df.loc[[0, 1]])
```

When using [ ], the result is a Pandas **DataFrame**.



Data -Frames

Named Indexes

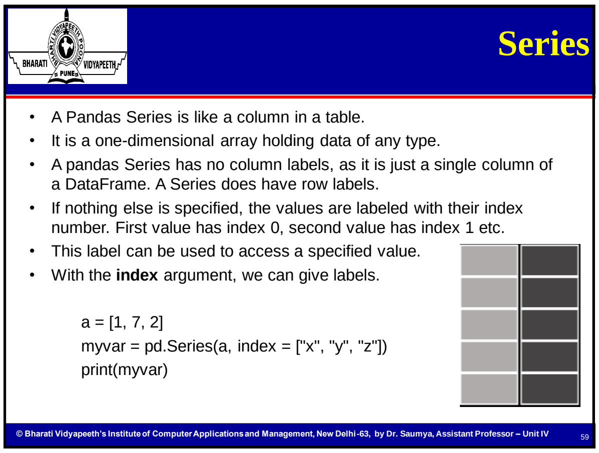
With the **index** argument, you can name your own indexes.

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

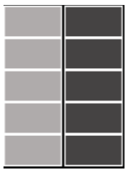
```
print(df.loc["day2"])
```

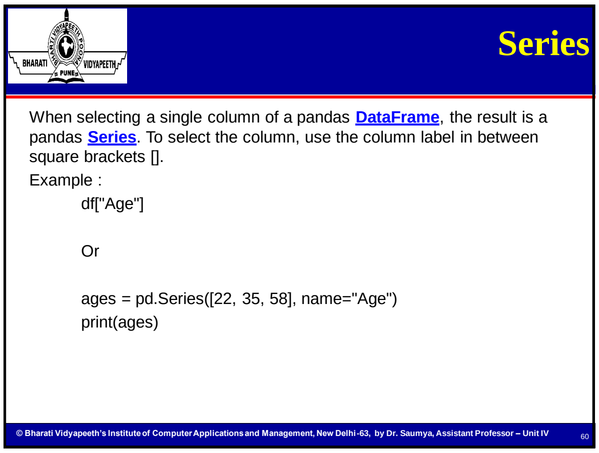


Series

- A Pandas Series is like a column in a table.
- It is a one-dimensional array holding data of any type.
- A pandas Series has no column labels, as it is just a single column of a DataFrame. A Series does have row 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.
- With the **index** argument, we can give labels.

```
a = [1, 7, 2]
myvar = pd.Series(a, index = ["x", "y", "z"])
print(myvar)
```





Series

When selecting a single column of a pandas DataFrame, the result is a pandas Series. To select the column, use the column label in between square brackets [].

Example :

```
df["Age"]

Or

ages = pd.Series([22, 35, 58], name="Age")
print(ages)
```

Series

Key/Value Objects as Series

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

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories)
print(myvar)
```

The keys of the dictionary become the labels.

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.

```
import pandas as pd
calories = {"day1": 420, "day2": 380, "day3": 390}
myvar = pd.Series(calories, index = ["day1", "day2"])
print(myvar)
```

Data Preparation and Pre-Processing

We can equate data preparation with the framework of the KDD Process -- specifically the first 3 major steps -- which re **selection**, **preprocessing**, and **transformation**.

1. **Loading data**

The first step for data preparation is to get some data. If you have a .csv file, you can easily load it up in your system using the **read_csv()** function in pandas. We can work with Data-frames and Series as well.

```
df = pd.read_csv('D:/Python Programming/Scripts/myArray.csv')
```

Data Preparation and Pre-Processing

2. **Missing Data**

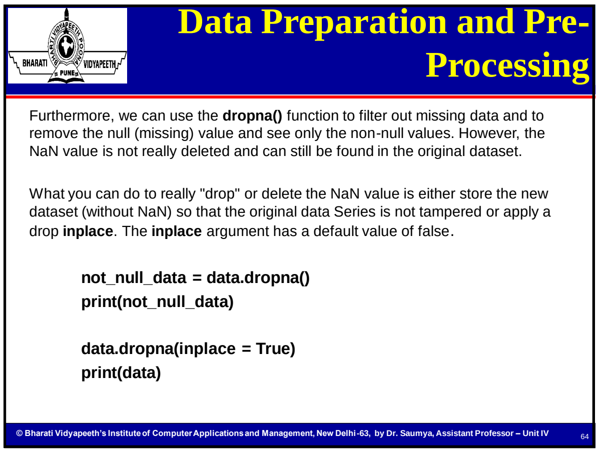
   – **Handling Missing Data**

Missing data can arise in the dataset due to multiple reasons: the data for the specific field was not added by the user/data collection application, data was lost while transferring manually, a programming error, etc.

For numerical data, pandas uses a floating point value **NaN** (Not a Number) to represent missing data. It is a unique value defined under the library **Numpy** so we will need to import it as well. NaN is the default missing value marker for reasons of computational speed and convenience. This is a sentinel value, in the sense that it is a dummy data or flag value that can be easily detected and worked with using functions in pandas.

```
data = pd.Series([0, 1, 2, 3, 4, 5, np.nan, 6, 7, 8])
data.isnull()
```

We used the function **isnull()** which returns a boolean true or false value. True, when the data at that particular index is actually missing or NaN. The opposite of this is the **notnull()** function.

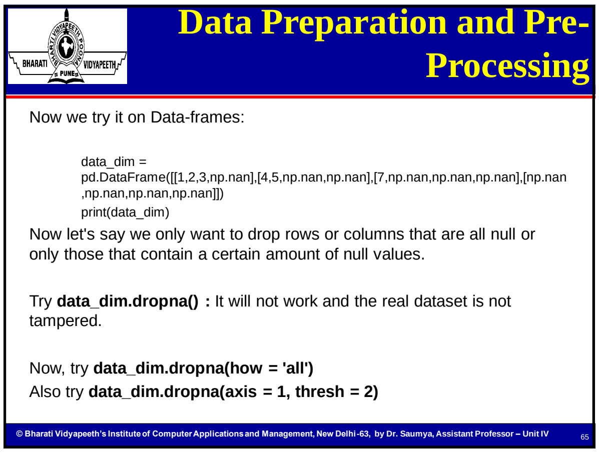


Data Preparation and Pre-Processing

Furthermore, we can use the **dropna()** function to filter out missing data and to remove the null (missing) value and see only the non-null values. However, the NaN value is not really deleted and can still be found in the original dataset.

What you can do to really "drop" or delete the NaN value is either store the new dataset (without NaN) so that the original data Series is not tampered or apply a drop **inplace**. The **inplace** argument has a default value of false.

```
not_null_data = data.dropna()
print(not_null_data)

data.dropna(inplace = True)
print(data)
```



Data Preparation and Pre-Processing

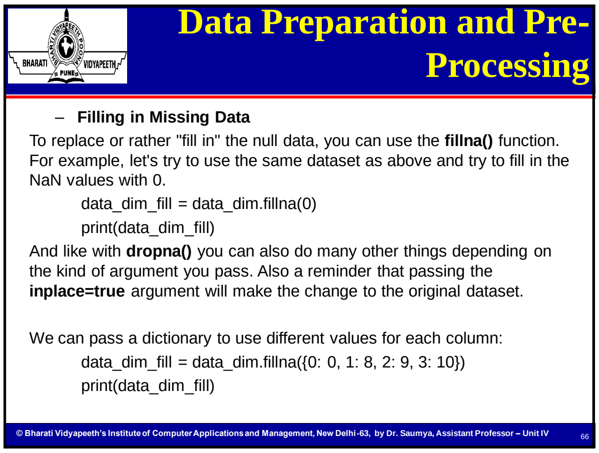
Now we try it on Data-frames:

```
data_dim =
pd.DataFrame([[1,2,3,np.nan],[4,5,np.nan,np.nan],[7,np.nan,np.nan,np.nan],[np.nan
,np.nan,np.nan,np.nan]])
print(data_dim)
```

Now let's say we only want to drop rows or columns that are all null or only those that contain a certain amount of null values.

Try **data_dim.dropna() :** It will not work and the real dataset is not tampered.

Now, try **data_dim.dropna(how = 'all')**
Also try **data_dim.dropna(axis = 1, thresh = 2)**



Data Preparation and Pre-Processing

– **Filling in Missing Data**

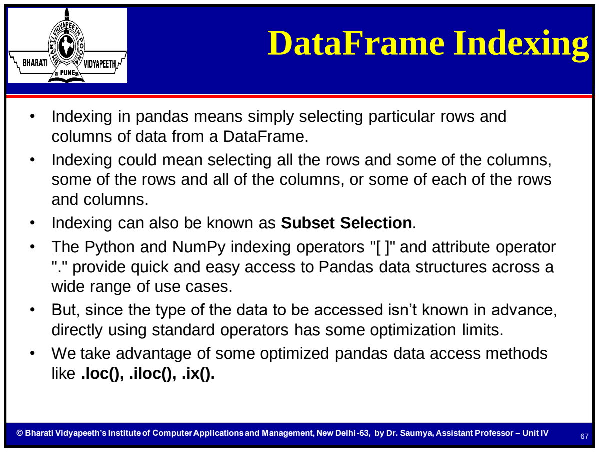
To replace or rather "fill in" the null data, you can use the **fillna()** function. For example, let's try to use the same dataset as above and try to fill in the NaN values with 0.

```
data_dim_fill = data_dim.fillna(0)
print(data_dim_fill)
```

And like with **dropna()** you can also do many other things depending on the kind of argument you pass. Also a reminder that passing the **inplace=true** argument will make the change to the original dataset.

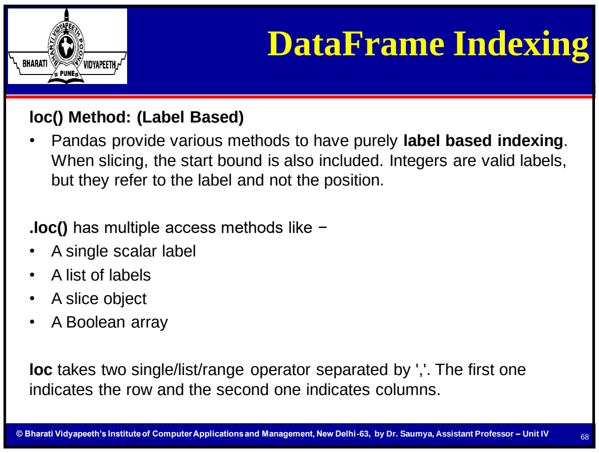
We can pass a dictionary to use different values for each column:

```
data_dim_fill = data_dim.fillna({0: 0, 1: 8, 2: 9, 3: 10})
print(data_dim_fill)
```



DataFrame Indexing

- 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**.
- The Python and NumPy indexing operators "[ ]" and attribute operator "." provide quick and easy access to Pandas data structures across a wide range of use cases.
- But, since the type of the data to be accessed isn't known in advance, directly using standard operators has some optimization limits.
- We take advantage of some optimized pandas data access methods like **.loc(), .iloc(), .ix().**



DataFrame Indexing

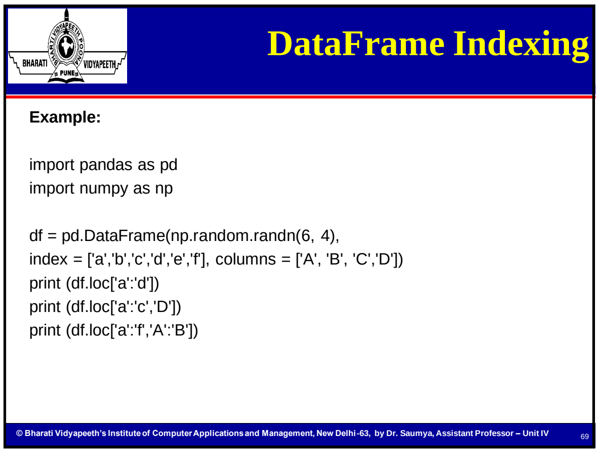
loc() Method: (Label Based)

- Pandas provide various methods to have purely **label based indexing**. When slicing, the start bound is also included. Integers are valid labels, but they refer to the label and not the position.

.loc() has multiple access methods like −

- A single scalar label
- A list of labels
- A slice object
- A Boolean array

loc takes two single/list/range operator separated by ','. The first one indicates the row and the second one indicates columns.



DataFrame Indexing

Example:

```
import pandas as pd
import numpy as np

df = pd.DataFrame(np.random.randn(6, 4),
index = ['a','b','c','d','e','f'], columns = ['A', 'B', 'C','D'])
print (df.loc['a':'d'])
print (df.loc['a':'c','D'])
print (df.loc['a':'f','A':'B'])
```

DataFrame Indexing

.iloc() method:

- Pandas provide various methods in order to get purely integer based indexing. Like python and numpy, these are **0-based** indexing.
- It is primarily integer position based from 0 to length – 1 of the axis

The various access methods are as follows −

- An Integer
- A list of integers
- A range of values

DataFrame Indexing

Example:

```
import pandas as pd
import numpy as np

df1 = pd.DataFrame(np.random.randn(8, 3),columns = ['A', 'B', 'C'])
print (df1.iloc[:8])
print (df1.iloc[:4])
print (df1.iloc[2:4, 1:3])
```

DataFrame Indexing

.ix() method:

- Besides pure label based and integer based, Pandas provides a hybrid method for selections and subsetting the object using the .ix() operator.
- The .ix indexer is deprecated in all the version after 0.20.0, in favor of the more strict .iloc and .loc indexers.

```
df = pd.DataFrame(np.random.randn(8, 4), columns = ['A', 'B', 'C', 'D'])
print (df.ix[:,'A'])
```

DataFrame Indexing

Select Data Using Columns

In addition to location-based and label-based indexing, you can also select data from **pandas** dataframes by selecting entire columns using the column names.

dataframe[“column”]

Above command provides the data from the column as a **pandas** series, which is a one-dimensional array. A **pandas** series is useful for selecting columns for plotting using **matplotlib**.

You can also specify that you want an output that is also a **pandas** dataframe.

dataframe[[“column”]]

which includes a second set of brackets [ ], to indicate that the output should be a **pandas** dataframe.

DataFrame Indexing

You can also select all data from multiple columns in a **pandas** dataframe using:

dataframe[[“column1”, “column2”]]

Since the results of your selection are also a **pandas** dataframe, you can assign the results to a new **pandas** dataframe.

Try This:

Use avg-precip-months.csv

create a new **pandas** dataframe that only contains the ***months*** and ***seasons*** column effectively dropping the ***precip*** values.

DataFrame Indexing

Filter Data Using Specific Values

In addition to location-based and label-based indexing, you can select or filter data based on specific values within a column using:

dataframe[dataframe[“column1”]==value]

This will return all rows containing that value within the specified column.

Again, you can also save the output to a new dataframe by setting it equal to the output of the filter.

You can also filter using a **comparison operator** on numeric values.

Try This :

Select all rows that have a season value of summer.

The query() Method

The query() Method

- Python is a great language for doing data analysis, primarily because of the fantastic ecosystem of data-centric Python packages. ***Pandas*** is one of those packages that makes importing and analyzing data much easier.
- Analyzing data requires a lot of filtering operations. Pandas provide many methods to filter a Data frame and dataframe.query() is one of them.
- DataFrame objects have a query() method that allows selection using an expression. You can get the value of the frame where column b has values between the values of columns a and c.

The query() Method

- ***Syntax:*** *DataFrame.query(expr, inplace=False, **kwargs)*
- ***Parameters:***
  expr: *Expression in string form to filter data.*
  inplace: *Make changes in the original data frame if True*
  kwargs: *Other keyword arguments.*
- ***Return type:*** *Filtered Data frame*

Note: dataframe.query() method only works if the column name doesn't have any empty spaces. So before applying the method, spaces in column names are replaced with '_'

The query() Method

Example #1: Single condition filtering (Employee.csv)

In this example, the data is filtered on the basis of single condition. Before applying the query() method, the spaces in column names have been replaced with '_'.

```
import pandas as pd
import numpy as np
# making data frame from csv file
data1 = pd.read_csv('D:/Python Programming/Scripts/employees.csv')
 # replacing blank spaces with '_'
data1.columns =[column.replace(" ", "_") for column in data1.columns]
 # filtering with query method
data1.query('Senior_Management == True', inplace = True)

# display
print(data1)
```

The query() Method

Example #2: Multiple condition filtering

```
# making data frame from csv file
data = pd.read_csv('D:/Python Programming/Scripts/employees.csv')

# replacing blank spaces with '_'
data.columns =[column.replace(" ", "_") for column in data.columns]

# filtering with query method
data.query('Senior_Management == True and Gender =="Male" and
Team =="Marketing" and First_Name =="Johnny"', inplace = True)

# display
print(data)
```

Data Visualization

Data visualization is the discipline of trying to understand data by placing it in a visual context so that patterns, trends and correlations that might not otherwise be detected can be exposed.

Data Visualization is the presentation of data in graphical format. It helps people understand the significance of data by summarizing and presenting huge amount of data in a simple and easy-to-understand format and helps communicate information clearly and effectively.

Data visualization in python is perhaps one of the most utilized features for data science with python in today's day and age. The libraries in python come with lots of different features that enable users to make highly customized, elegant, and interactive plots.

Data Visualization

Useful packages for visualizations in python

- **Matplotlib**
- Matplotlib is a visualization library in Python for 2D plots of arrays. Matplotlib is written in Python and makes use of the NumPy library. Matplotlib comes with a wide variety of plots like line, bar, scatter, histogram, etc. which can help us, deep-dive, into understanding trends, patterns, correlations. It was introduced by John Hunter in 2002.
- **Seaborn**
- Seaborn is a dataset-oriented library for making statistical representations in Python. It is developed atop matplotlib and to create different visualizations. It is integrated with pandas data structures. The library internally performs the required mapping and aggregation to create informative visuals It is recommended to use a Jupyter/IPython interface in matplotlib mode.

Data Visualization

- **Bokeh**
- Bokeh is an interactive visualization library for modern web browsers. It is suitable for large or streaming data assets and can be used to develop interactive plots and dashboards. There is a wide array of intuitive graphs in the library which can be leveraged to develop solutions. It works closely with PyData tools. The library is well-suited for creating customized visuals according to required use-cases.
- **Altair**
- Altair is a declarative statistical visualization library for Python. Altair's API is user-friendly and consistent and built atop Vega-Lite JSON specification. Declarative library indicates that while creating any visuals, we need to define the links between the data columns to the channels (x-axis, y-axis, size, color). With the help of Altair, it is possible to create informative visuals with minimal code.

Data Visualization

- **plotly**
- plotly.py is an interactive, open-source, high-level, declarative, and browser-based visualization library for Python. It holds an array of useful visualization which includes scientific charts, 3D graphs, statistical charts, financial charts among others. Plotly graphs can be viewed in Jupyter notebooks, standalone HTML files, or hosted online.
- **ggplot**
- ggplot is a Python implementation of the grammar of graphics. The Grammar of Graphics refers to the mapping of data to aesthetic attributes (colour, shape, size) and geometric objects (points, lines, bars). The basic building blocks according to the grammar of graphics are data, geom (geometric objects), stats (statistical transformations), scale, coordinate system, and facet.

Types of Charts for Analyzing & Presenting Data

- **Histogram**
  The histogram represents the frequency of occurrence of specific phenomena which lie within a specific range of values and arranged in consecutive and fixed intervals.

- **Bar Charts**
  A bar chart can be created using the bar method. The bar-chart isn't automatically calculating the frequency of a category so we are going to use pandas value_counts function to do this. The bar-chart is useful for categorical data that doesn't have a lot of different categories (less than 30) because else it can get quite messy.

Example: Test-plot.py

Types of Charts for Analyzing & Presenting Data

- **Pie Chart :**
  A pie chart shows a static number and how categories represent part of a whole the composition of something. A pie chart represents numbers in percentages, and the total sum of all segments needs to equal 100%.

- **Scatter plot :**
- A scatter chart shows the relationship between two different variables and it can reveal the distribution trends. It should be used when there are many different data points, and you want to highlight similarities in the data set. This is useful when looking for outliers and for understanding the distribution of your data.

Example: Piechart.py

Creating Pie Charts

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])

plt.pie(y)
plt.show()
```

- As you can see the pie chart draws one piece (called a wedge) for each value in the array (in this case [35, 25, 25, 15]).
- By default the plotting of the first wedge starts from the x-axis and move *counterclockwise*:

Creating Pie Charts

Labels

```
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
plt.pie(y, labels = mylabels)
plt.show()
```

Explode

- Maybe you want one of the wedges to stand out? The explode() parameter allows you to do that.
- The explode() parameter, if specified, and not None, must be an array with one value for each wedge.
- Each value represents how far from the center each wedge is displayed:

```
myexplode = [0.2, 0, 0, 0]
plt.pie(y, labels = mylabels, explode = myexplode, shadow=true)
plt.show()
```

Colors and legends

Color

You can set the color of each wedge with the colors parameter.

The colors parameter, if specified, must be an array with one value for each wedge:

```
mycolors = ["black", "hotpink", "b", "#4CAF50"]
plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```

Legend

To add a list of explanation for each wedge, use the legend() function:

```
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend()
plt.show()
```

Legend With Header

Legend With Header

To add a header to the legend, add the title parameter to the legend():

```
y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend(title = "Four Fruits:")
plt.show()
```

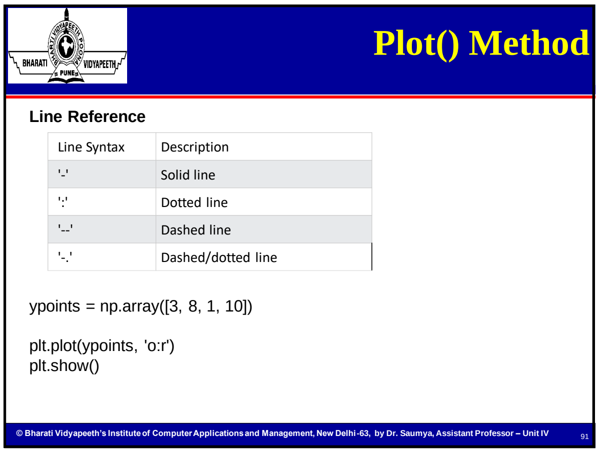
Change Figure Size in Matplotlib

```
plt.figure(figsize=(width, height))

plt.figure(figsize=(3, 3))
```

Plot() Method

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([0, 6])
ypoints = np.array([0, 250])

plt.plot(xpoints, ypoints)
plt.plot(xpoints, ypoints, marker= 'o')
plt.show()
```

Other possible markers are - *, . , x, X, +, P, s (Square), D (Diamond), v (Triangle Down), ^ (Triangle up), < (Triangle left), > (Triangle right), _ (Hline), | (Vline)

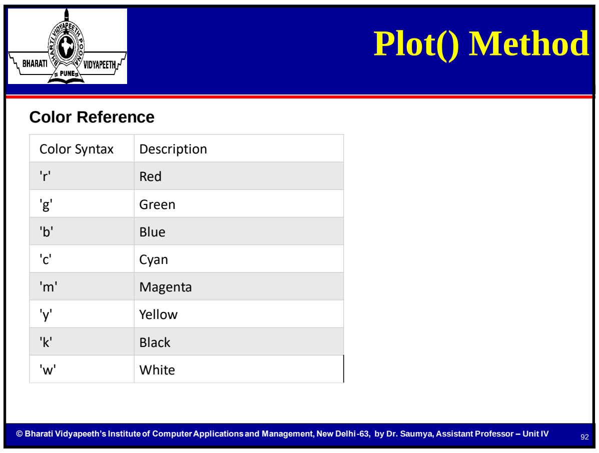


Plot() Method

Line Reference

Line Syntax	Description
'-'	Solid line
':'	Dotted line
'--'	Dashed line
'-.'	Dashed/dotted line

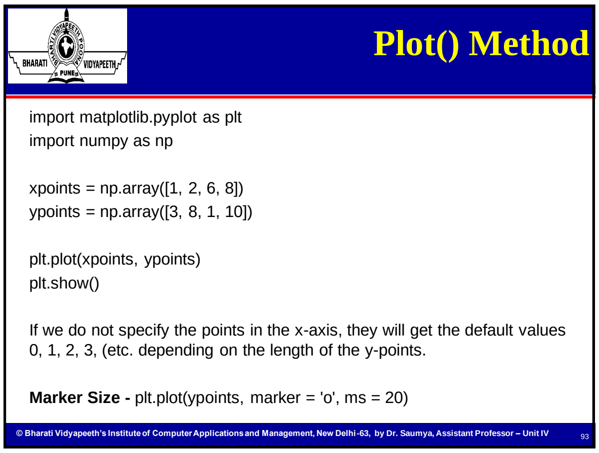
```
ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, 'o:r')
plt.show()
```



Plot() Method

Color Reference

Color Syntax	Description
'r'	Red
'g'	Green
'b'	Blue
'c'	Cyan
'm'	Magenta
'y'	Yellow
'k'	Black
'w'	White

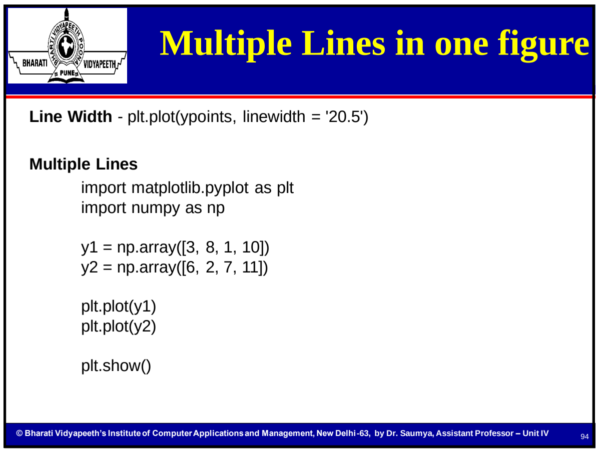


Plot() Method

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points.

Marker Size - plt.plot(ypoints, marker = 'o', ms = 20)



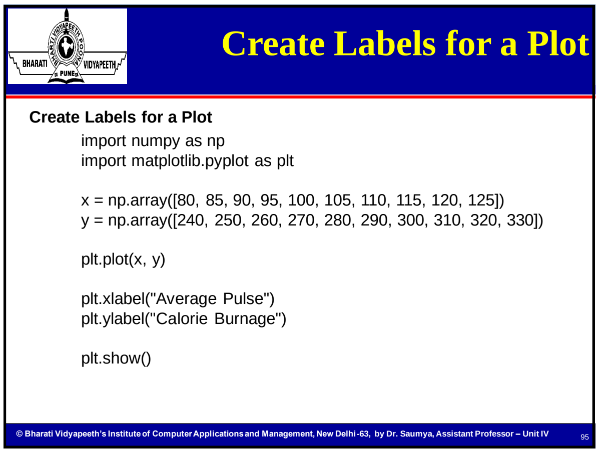
Multiple Lines in one figure

Line Width - plt.plot(ypoints, linewidth = '20.5')

Multiple Lines

```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 94

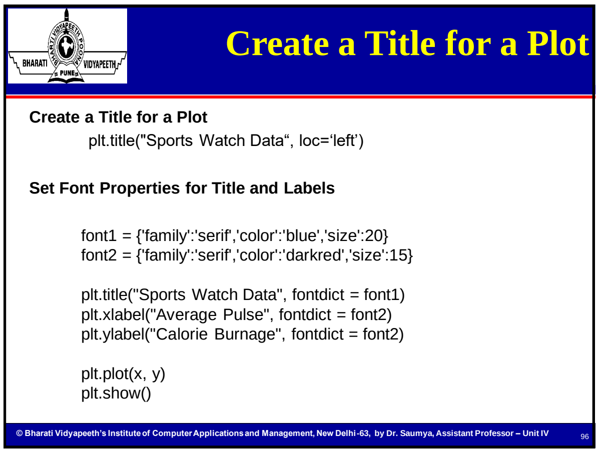


Create Labels for a Plot

Create Labels for a Plot

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 95



Create a Title for a Plot

Create a Title for a Plot

```
plt.title("Sports Watch Data“, loc=‘left’)
```

Set Font Properties for Title and Labels

```
font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y)
plt.show()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 96

Display Multiple Plots

Display Multiple Plots

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)

plt.show()
```

Display subplots

The subplots() Function

- The subplot() function takes three arguments that describes the layout of the figure.
- The layout is organized in rows and columns, which are represented by the *first* and *second* argument.
- The third argument represents the index of the current plot.

plt.subplot(1, 2, 1)
#the figuree has 1 row, 2 columns, and this plot is the *first* plot.

plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the *second* plot.

Try This: If we want a figure with 2 rows an 1 column

Example:

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.show()
```

GUI Programming

Most of the programs we have done till now are text-based programming. But many applications need GUI (Graphical User Interface).

Python provides several different options for writing GUI based programs. These are listed below:

- **Tkinter**: It is easiest to start with. Tkinter is Python's standard GUI (graphical user interface) package. It is the most commonly used toolkit for GUI programming in Python.
- **JPython**: It is the Python platform for Java that is providing Python scripts seamless access o Java class Libraries for the local machine.
- **wxPython**: It is an open-source, cross-platform GUI toolkit written in C++. It is one of the alternatives to Tkinter, which is bundled with Python.

There are many other interfaces available for GUI. But these are the most commonly used ones.

Tkinter

It is the standard GUI toolkit for Python. Fredrik Lundh wrote it. For modern Tk binding, Tkinter is implemented as a Python wrapper for the Tcl Interpreter embedded within the interpreter of Python. Tk provides the following widgets:

• Button	canvas
• combo-box	frame
• Level	check-button
• Entry	level-frame
• Menu	list - box
• menu button	message
• tk_optoinMenu	progress-bar
• radio button	scroll bar
• Separator	tree-view, and many more.

Tkinter

Steps to create GUI program using Tkinter:

- Import the module Tkinter
- Build a GUI application (as a window)
- Add those widgets that are discussed above
- Enter the primary, i.e., the main event's loop for taking action when the user triggered the event.

- Importing tkinter is same as importing any other module in the Python code. Note that the name of the module in Python 2.x is 'Tkinter' and in Python 3.x it is 'tkinter'.

Main Methods of GUI Application

There are two main methods used which the user needs to remember while creating the Python application with GUI.

1. **Tk(screenName=None, baseName=None, className='Tk', useTk=1):** To create a main window, tkinter offers a method 'Tk(screenName=None, baseName=None, className='Tk', useTk=1)'. To change the name of the window, you can change the className to the desired one.
2. **mainloop():** There is a method known by the name mainloop() is used when your application is ready to run. mainloop() is an infinite loop used to run the application, wait for an event to occur and process the event as long as the window is not closed.

```
import tkinter
m = tkinter.Tk()
m.mainloop()
```

Geometric Manager Classes

tkinter also offers access to the geometric configuration of the widgets which can organize the widgets in the parent windows. There are mainly three geometry manager classes class.

- **pack() method:**It organizes the widgets in blocks before placing in the parent widget.
- **grid() method:**It organizes the widgets in grid (table-like structure) before placing in the parent widget.
- **place() method:**It organizes the widgets by placing them on specific positions directed by the programmer.

There are a number of widgets which you can put in your tkinter application.

Tkinter grid() Method

This geometry manager organizes widgets in a table-like structure in the parent widget.

```
widget.grid( grid_options )
```

Here is the list of possible options −

- **column** − The column to put widget in; default 0 (leftmost column).
- **columnspan** − How many columns widgetoccupies; default 1.
- **ipadx, ipady** − How many pixels to pad widget, horizontally and vertically, inside widget's borders.
- **padx, pady** − How many pixels to pad widget, horizontally and vertically, outside v's borders.
- **row** − The row to put widget in; default the first row that is still empty.
- **rowspan** − How many rowswidget occupies; default 1.
- **sticky** − What to do if the cell is larger than widget. By default, with sticky='', widget is centered in its cell. sticky may be the string concatenation of zero or more of N, E, S, W, NE, NW, SE, and SW, compass directions indicating the sides and corners of the cell to which widget sticks.

Button

To add a button in your application, this widget is used.
The general syntax is:

```
w=Button(master, option=value)
```

master is the parameter used to represent the parent window.
There are number of options which are used to change the format of the Buttons. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **activebackground**: to set the background color when button is under the cursor.
- **activeforeground**: to set the foreground color when button is under the cursor.
- **bg**: to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the button.
- **width**: to set the width of the button.
- **height**: to set the height of the button.

Button (Example)

```
import tkinter as tk
r = tk.Tk()
r.title('Counting Seconds')
button = tk.Button(r, text='Stop', width=25, command=r.destroy)
button.pack()
r.mainloop()
```

Canvas

It is used to draw pictures and other complex layout like graphics, text and widgets. The general syntax is:

```
w = Canvas(master, option=value)
```

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used in the canvas.
- **highlightcolor**: to set the color shown in the focus highlight.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

Canvas (Example)

```
import tkinter as tk
master = tk.Tk()
w = tk.Canvas(master, width=40, height=60)
w.pack()
canvas_height=20
canvas_width=200
y = int(canvas_height / 2)
w.create_line(0, y, canvas_width, y )
w.mainloop()
```

CheckButton

To select any number of options by displaying a number of options to a user as toggle buttons.

```
import tkinter
from tkinter import *
top = tkinter.Tk()
CheckVar1 = IntVar()
CheckVar2 = IntVar()
tkinter.Checkbutton(top, text = "Machine Learning",variable = CheckVar1,onvalue = 1, offvalue=0).grid(row=0,sticky=W)
tkinter.Checkbutton(top, text = "Deep Learning", variable = CheckVar2, onvalue = 0, offvalue =1).grid(row=1,sticky=W)
top.mainloop()
```

Entry

Entry: It is used to input the single line text entry from the user.. For multi-line text input, Text widget is used.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used.
- **command**: to call a function.
- **highlightcolor**: to set the color shown in the focus highlight.
- **width**: to set the width of the button.
- **height**: to set the height of the button.

Entry (Example)

```
from tkinter import *
master = Tk()
Label(master, text='First Name').grid(row=0)
Label(master, text='Last Name').grid(row=1)
e1 = Entry(master)
e2 = Entry(master)
e1.grid(row=0, column=1)
e2.grid(row=1, column=1)
mainloop()
```

Frame

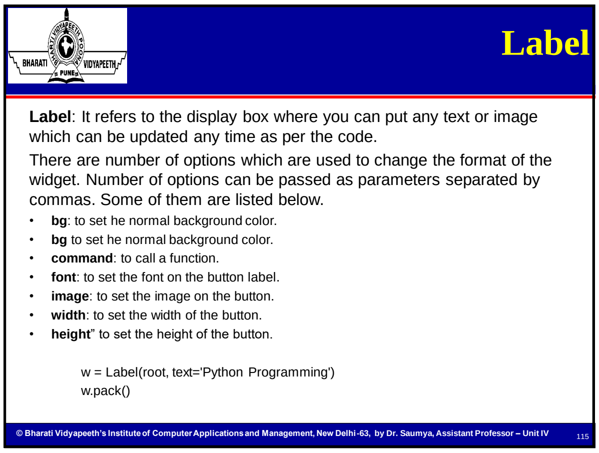
Frame: It acts as a container to hold the widgets. It is used for grouping and organizing the widgets.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.
- **bd**: to set the border width in pixels.
- **bg**: to set the normal background color.
- **cursor**: to set the cursor used.
- **width**: to set the width of the widget.
- **height**: to set the height of the widget.

Frame(Example)

```
from tkinter import *
root = Tk()
frame = Frame(root)
frame.pack()
bottomframe = Frame(root)
bottomframe.pack( side = BOTTOM )
redbutton = Button(frame, text = 'Red', fg ='red')
redbutton.pack( side = LEFT)
greenbutton = Button(frame, text = 'Brown', fg='brown')
greenbutton.pack( side = LEFT )
bluebutton = Button(frame, text ='Blue', fg ='blue')
bluebutton.pack( side = LEFT )
blackbutton = Button(bottomframe, text ='Black', fg ='black')
blackbutton.pack( side = BOTTOM)
root.mainloop()
```



Label

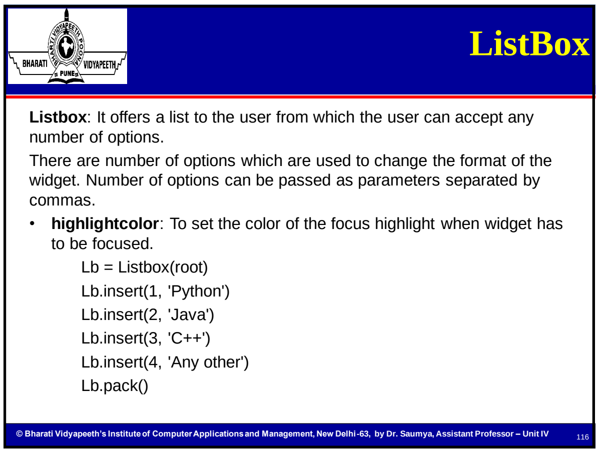
Label: It refers to the display box where you can put any text or image which can be updated any time as per the code.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas. Some of them are listed below.

- **bg**: to set he normal background color.
- **bg** to set he normal background color.
- **command**: to call a function.
- **font**: to set the font on the button label.
- **image**: to set the image on the button.
- **width**: to set the width of the button.
- **height**" to set the height of the button.

```
w = Label(root, text='Python Programming')
w.pack()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 115



ListBox

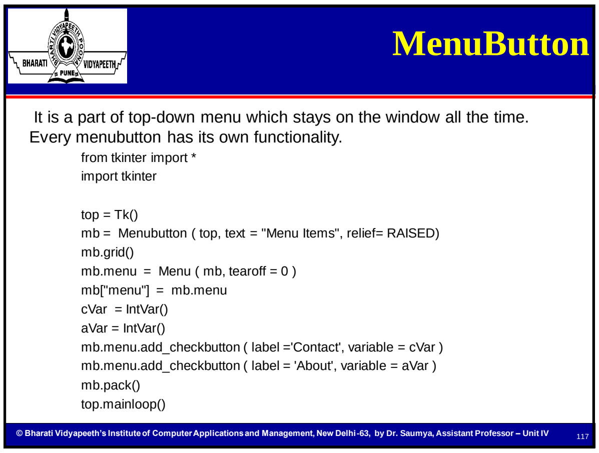
Listbox: It offers a list to the user from which the user can accept any number of options.

There are number of options which are used to change the format of the widget. Number of options can be passed as parameters separated by commas.

- **highlightcolor**: To set the color of the focus highlight when widget has to be focused.

```
Lb = Listbox(root)
Lb.insert(1, 'Python')
Lb.insert(2, 'Java')
Lb.insert(3, 'C++')
Lb.insert(4, 'Any other')
Lb.pack()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 116



MenuButton

It is a part of top-down menu which stays on the window all the time. Every menubutton has its own functionality.

```
from tkinter import *
import tkinter

top = Tk()
mb =  Menubutton ( top, text = "Menu Items", relief= RAISED)
mb.grid()
mb.menu  =  Menu ( mb, tearoff = 0 )
mb["menu"]  =  mb.menu
cVar  = IntVar()
aVar = IntVar()
mb.menu.add_checkbutton ( label ='Contact', variable = cVar )
mb.menu.add_checkbutton ( label = 'About', variable = aVar )
mb.pack()
top.mainloop()
```

© Bharati Vidyapeeth's Institute of Computer Applications and Management, New Delhi-63, by Dr. Saumya, Assistant Professor – Unit IV 117

Menu

Menu: It is used to create all kinds of menus used by the application

```
from tkinter import *

root = Tk()
menu = Menu(root)
root.config(menu=menu)
filemenu = Menu(menu)
menu.add_cascade(label='File', menu=filemenu)
filemenu.add_command(label='New')
filemenu.add_command(label='Open...')
filemenu.add_separator()
filemenu.add_command(label='Exit', command=root.quit)
helpmenu = Menu(menu)
menu.add_cascade(label='Help', menu=helpmenu)
helpmenu.add_command(label='About')
mainloop()
```

Message

Message: It refers to the multi-line and non-editable text.
The message widget is similar in its functionality to the Label widget, but it is **more flexible in displaying text**, e.g. the font can be changed while the Label widget can only display text in a single font. It provides a multiline object, that is the text may span more than one line

```
from tkinter import *
main = Tk()
ourMessage ='This is our Message'
messageVar = Message(main, text = ourMessage)
messageVar.config(bg='lightgreen')
messageVar.pack( )
main.mainloop( )
```

RadioButtons

The **Radiobutton** is a standard Tkinter widget used to implement one-of-many selections. **Radiobuttons** can contain text or images, and you can associate a Python function or method with each button. When the button is pressed, Tkinter automatically calls that function or method.

General Syntax:

button = Radiobutton(master, text="Name on Button", variable = "shared variable", value = "values of each button", options = values, …)
***shared variable** = A Tkinter variable shared among all Radio buttons*
***value** = each radiobutton should have different value otherwise more than 1 radiobutton will get selected.*

RadioButtons (Example)

```
from tkinter import *

def sel():
   selection = "You selected the option " + str(var.get())
   label.config(text = selection)

root = Tk()
var = IntVar()
R1 = Radiobutton(root, text="Option 1", variable=var, value=1, command=sel)
R1.pack( anchor = W )

R2 = Radiobutton(root, text="Option 2", variable=var, value=2, command=sel)
R2.pack( anchor = W )

R3 = Radiobutton(root, text="Option 3", variable=var, value=3, command=sel)
R3.pack( anchor = W)
label = Label(root)
label.pack()
root.mainloop()
```

Database Access

- Python can be used in database applications.
- One of the most popular databases is MySQL.
- You can download a free MySQL database at https://www.mysql.com/downloads/.
- Python needs a MySQL driver to access the MySQL database.
- use PIP to install "MySQL Connector".
- Use command python -m pip install mysql-connector-python

Create Connection

- We can start by creating a connection to the database.
- Use the username and password from your MySQL database:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword"
)

print(mydb)
```

Creating a Database

To create a database in MySQL, use the "CREATE DATABASE" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE DATABASE mydatabase")
```

Creating a Table

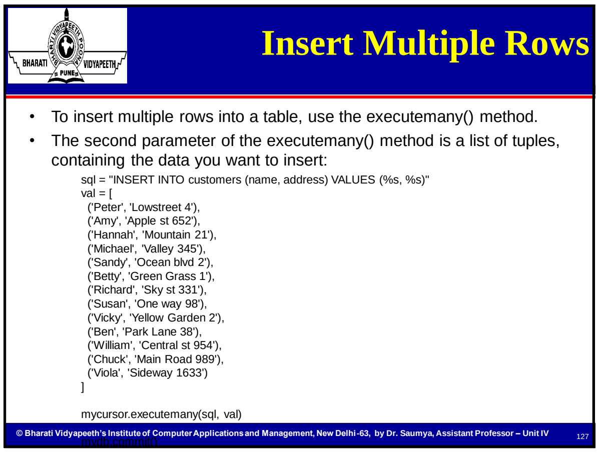
- To create a table in MySQL, use the "CREATE TABLE" statement.
- Make sure you define the name of the database when you create the connection

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("CREATE TABLE customers (id INT AUTO_INCREMENT PRIMARY KEY, name
VARCHAR(255), address VARCHAR(255))")
```

Insert Into Table

To fill a table in MySQL, use the "INSERT INTO" statement.

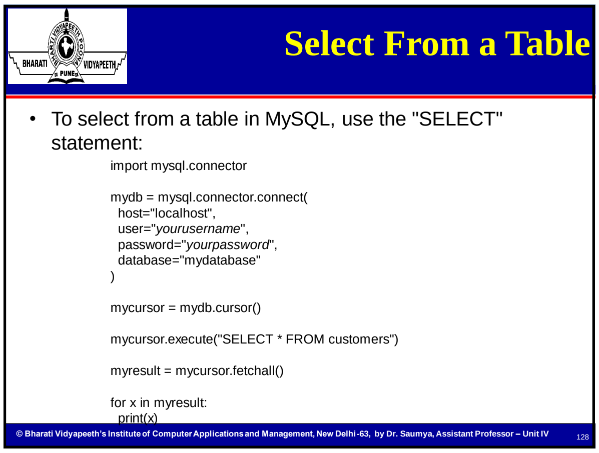
```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = ("John", "Highway 21")
mycursor.execute(sql, val)

mydb.commit()

print(mycursor.rowcount, "record inserted.")
```



Insert Multiple Rows

- To insert multiple rows into a table, use the executemany() method.
- The second parameter of the executemany() method is a list of tuples, containing the data you want to insert:

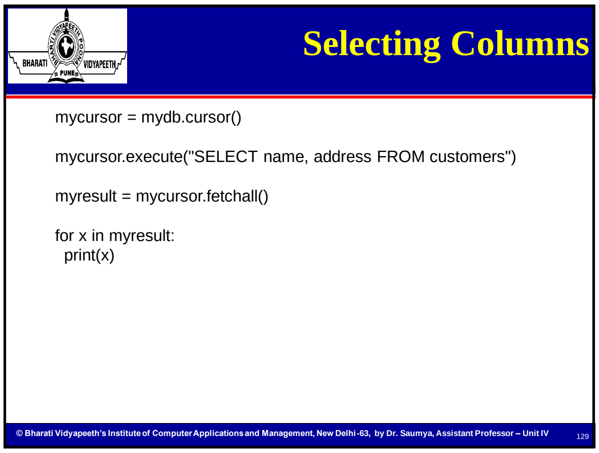
```
sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"
val = [
  ('Peter', 'Lowstreet 4'),
  ('Amy', 'Apple st 652'),
  ('Hannah', 'Mountain 21'),
  ('Michael', 'Valley 345'),
  ('Sandy', 'Ocean blvd 2'),
  ('Betty', 'Green Grass 1'),
  ('Richard', 'Sky st 331'),
  ('Susan', 'One way 98'),
  ('Vicky', 'Yellow Garden 2'),
  ('Ben', 'Park Lane 38'),
  ('William', 'Central st 954'),
  ('Chuck', 'Main Road 989'),
  ('Viola', 'Sideway 1633')
]

mycursor.executemany(sql, val)
mydb.commit()
```



Select From a Table

- To select from a table in MySQL, use the "SELECT" statement:

```
import mysql.connector

mydb = mysql.connector.connect(
  host="localhost",
  user="yourusername",
  password="yourpassword",
  database="mydatabase"
)

mycursor = mydb.cursor()

mycursor.execute("SELECT * FROM customers")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```



Selecting Columns

```
mycursor = mydb.cursor()

mycursor.execute("SELECT name, address FROM customers")

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Select With a Filter

When selecting records from a table, you can filter the selection by using the "WHERE" statement:

```
mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address ='Park Lane 38'"

mycursor.execute(sql)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Prevent SQL Injection

- When query values are provided by the user, you should escape the values.
- This is to prevent SQL injections, which is a common web hacking technique to destroy or misuse your database.
- The mysql.connector module has methods to escape query values:

```
mycursor = mydb.cursor()

sql = "SELECT * FROM customers WHERE address = %s"
adr = ("Yellow Garden 2", )

mycursor.execute(sql, adr)

myresult = mycursor.fetchall()

for x in myresult:
  print(x)
```

Delete Record

You can delete records from an existing table by using the "DELETE FROM" statement:

```
mycursor = mydb.cursor()

sql = "DELETE FROM customers WHERE address = 'Mountain 21'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) deleted")
```

Update Records in a Table

You can update existing records in a table by using the "UPDATE" statement:

```
mycursor = mydb.cursor()

sql = "UPDATE customers SET address = 'Canyon 123' WHERE address = 'Valley
345'"

mycursor.execute(sql)

mydb.commit()

print(mycursor.rowcount, "record(s) affected")
```

Limit the Result

You can limit the number of records returned from the query, by using the "LIMIT" statement:

```
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers LIMIT 5")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Start From Another Position

- If you want to return five records, starting from the third record, you can use the "OFFSET" keyword:

```
mycursor = mydb.cursor()
mycursor.execute("SELECT * FROM customers LIMIT 5 OFFSET 2")
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```

Drop a Table

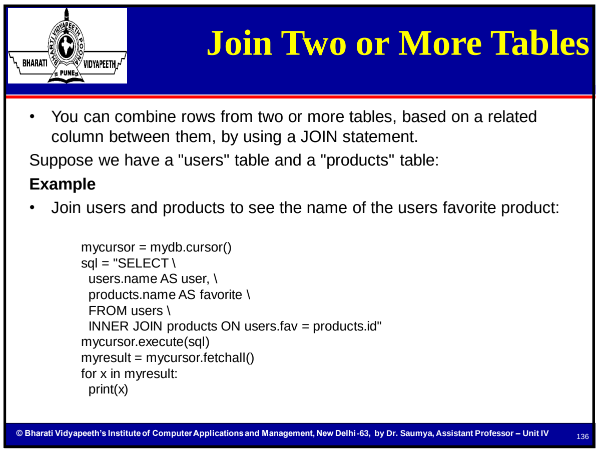
You can delete an existing table by using the "DROP TABLE" statement:

```
mycursor = mydb.cursor()
sql = "DROP TABLE customers"
mycursor.execute(sql)
```

Drop Only if Exist

- If the the table you want to delete is already deleted, or for any other reason does not exist, you can use the IF EXISTS keyword to avoid getting an error.

```
mycursor = mydb.cursor()
sql = "DROP TABLE IF EXISTS customers"
mycursor.execute(sql)
```



Join Two or More Tables

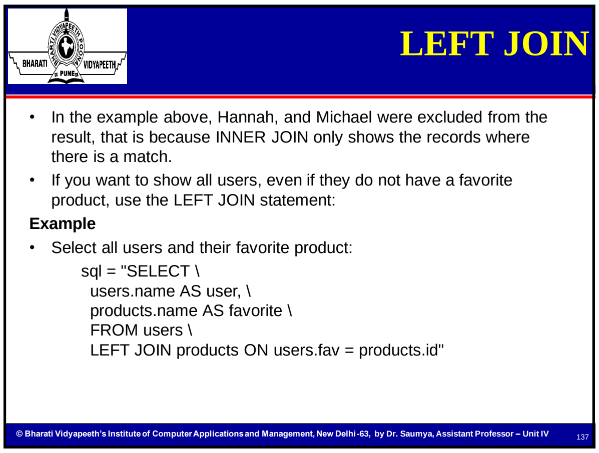
- You can combine rows from two or more tables, based on a related column between them, by using a JOIN statement.

Suppose we have a "users" table and a "products" table:

Example

- Join users and products to see the name of the users favorite product:

```
mycursor = mydb.cursor()
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  INNER JOIN products ON users.fav = products.id"
mycursor.execute(sql)
myresult = mycursor.fetchall()
for x in myresult:
  print(x)
```



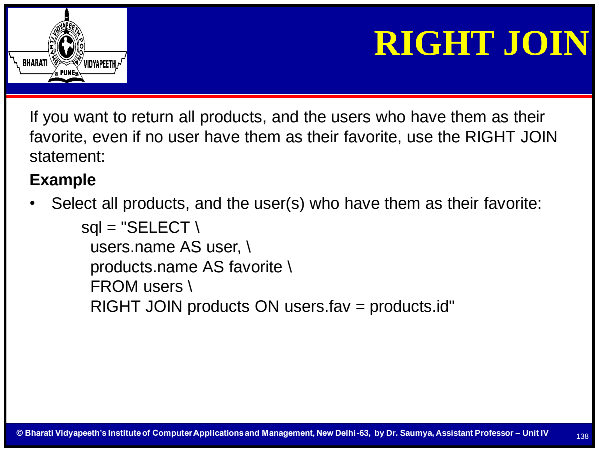
LEFT JOIN

- In the example above, Hannah, and Michael were excluded from the result, that is because INNER JOIN only shows the records where there is a match.
- If you want to show all users, even if they do not have a favorite product, use the LEFT JOIN statement:

Example

- Select all users and their favorite product:

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  LEFT JOIN products ON users.fav = products.id"
```



RIGHT JOIN

If you want to return all products, and the users who have them as their favorite, even if no user have them as their favorite, use the RIGHT JOIN statement:

Example

- Select all products, and the user(s) who have them as their favorite:

```
sql = "SELECT \
  users.name AS user, \
  products.name AS favorite \
  FROM users \
  RIGHT JOIN products ON users.fav = products.id"
```