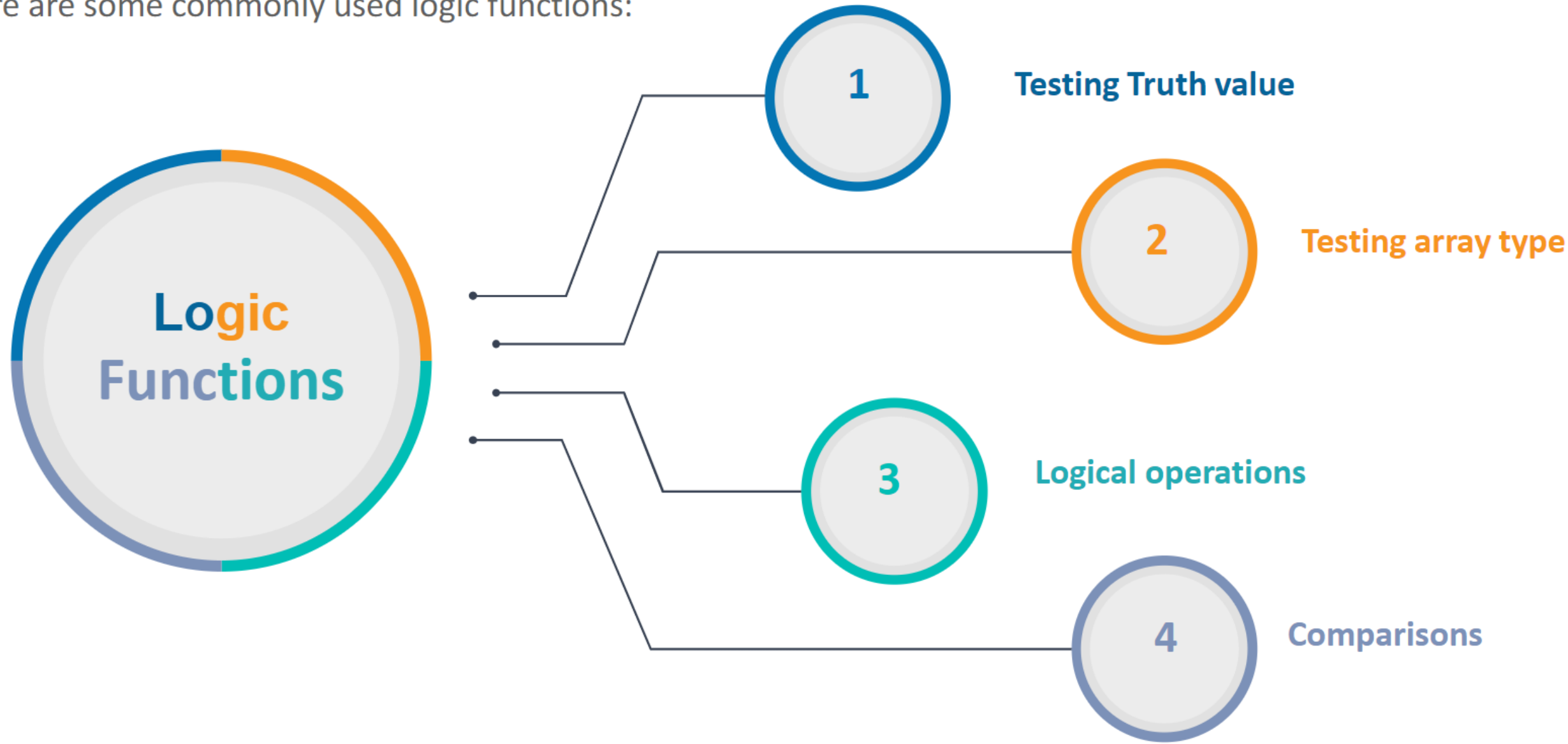
Logic Functions

Here are some commonly used logic functions:



Logic functions – Testing Truth Value – *all()*

numpy.all() checks whether all the elements of the given array along a particular axis evaluates to True

For Boolean Values

```
np.all([[True,False],[True,True]])
```

```
False
```

For Boolean Values, axis = 0

```
np.all([[True,False],[True,True]], axis=0)
```

```
array([ True, False])
```

For Numbers

```
np.all([-1, 4, 5])
```

```
True
```

Note: **Not a Number (NaN),** positive infinity and negative infinity are not equal to zero and hence evaluate to ***True***

Logic functions – Testing Truth Value – *any()*

numpy.any() tests whether any array element along a given axis evaluates to True

For Boolean Values

```
print(np.any([[True,False],[True,True]]))
```

```
True
```

For Boolean Values, axis = 0

```
print(np.any([[True,False],[True,True]],axis=0))
```

```
[ True  True]
```

For Numbers

```
print(np.any([-1,4,5]))
```

```
True
```

Logic Functions – Testing Array Type

Syntax	Returns a bool array	Example	Output
np.iscomplex	TRUE if input element is complex	np.iscomplex([2+1j, 3+0j, 3, 4.5, 7+5j])	array([True, False, False, False, True])
np.isreal	TRUE if input element is real	np.isreal([2+1j, 3+0j, 3 ,4.5, 10])	array([False, True, True, True, True])

Logic Functions – Element-wise Logical Operations

Consider x = [0,1,2,3,4]

Syntax	Example	Output
np.logical_and()	np.logical_and(x>1, x<4)	array([False, False, True, True, False])
np.logical_or()	np.logical_or(x>1, x<4)	array([True, True, True, True, True])
np.logical_not()	np.logical_not(x>1, x<4)	array([False, False, False, False, True])
np.logical_xor()	np.logical_xor(x>1, x<4)	array([True, True, False, False, True])

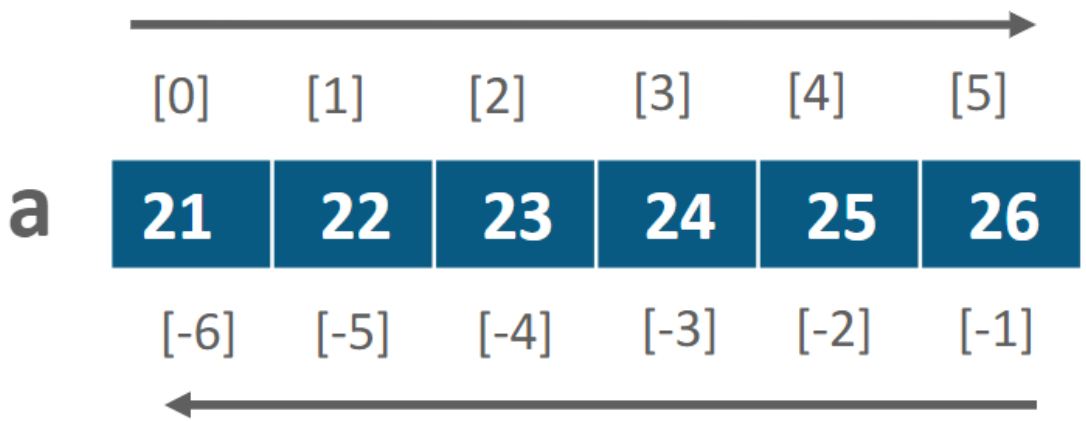
Logic Functions – Element-wise Comparisons

Syntax	True When	Example	Output
np.greater()	x1 > x2	np.greater([3,5,2,7,9],[1,7,4,5,10])	array([True, False, False, True, False])
np.greater_equal()	x1 >=x2	np.greater_equal([3,5,2,7,9],[1,7,4,5,10])	array([True, False, True, True, False])
np.less()	x1 < x2	np.less([3,5,2,7,9],[1,7,4,5,10])	array([False, True, False, False, True])
np.less_equal()	x1<= x2	np.less_equal([3,5,2,7,9],[1,7,4,5,10])	array([False, True, True, False, True])
np.equal()	x1= = x2	np.equal([3,5,2,7,9],[1,7,4,5,10])	array([False, False, True, False, False])
np.not_equal()	x1! = x2	np.not_equal([3,5,2,7,9],[1,7,4,5,10])	array([True, True, False, True, True])

Demo 2: Array Creation and Logic Functions

Indexing

- Array indexing uses square bracket “[ ]” to index the elements of the array so that the elements can then be referred individually for various uses such as extracting a value, selecting items, or even assigning a new value
- When you create a new array, an appropriate scale index is also automatically created



Here is the indexing of a mono-dimensional ndarray

Indexing – Example

For positive values

```
import numpy as np
arr=np.arange(21,26)
element=arr[3]
print(element)
```

For negative values

```
import numpy as np
arr=np.arange(21,26)
element=arr[-5]
print(element)
```

Indexing – Two-dimensional Array

Indexing in a 2-D array is represented by a pair of values –

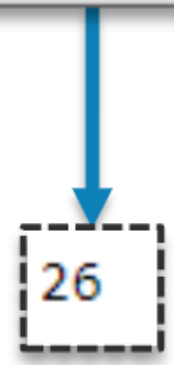
i) Index of the row and

ii) Index of the column

A	[,0]	[,1]	[,2]
[0,]	21	22	23
[1,]	24	25	26
[2,]	27	28	29

Here is the indexing of a Bi-dimensional *ndarray*

```
import numpy as np
arr=np.arange(21,30).reshape((3,3))
element=arr[1,2]
print(element)
```



Fancy Indexing

- Through Fancy indexing we can access multiple array elements at once by passing an array of indices
- This allows us to very quickly access and modify complicated subsets of an array's values

```
import numpy as np
a = np.array([3,78,23,12,90,10,45,56])
ind = np.array([[3,5],[2,4]])
print(a[ind])
```

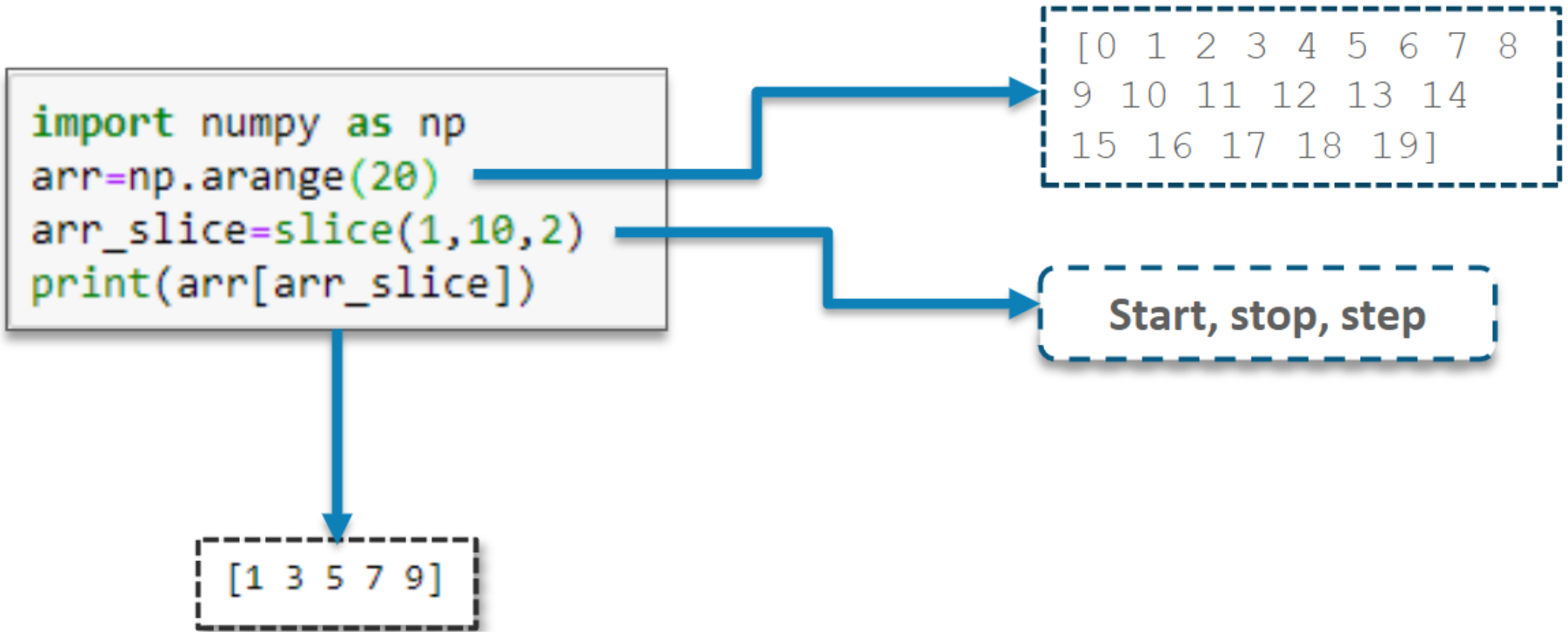
```
[[12 10]
 [23 90]]
```

Output

By using fancy indexing, the shape of the result reflects the shape of the index array rather than the shape of the array being indexed

Slicing

- Slicing allows you to extract portion of an array to generate a new array
- The slice object is constructed by using start, stop and step parameters in ***slice()*** function



Slicing (Cont.)

Slicing items beginning with a specified index

```
import numpy as np
arr=np.arange(20)
print(arr[2:])
```

```
[ 2  3  4  5  6  7  8  9 10 11 12 13 14 15 16 17 18 19]
```

all elements starting from the index 2

Slicing items until a specified index

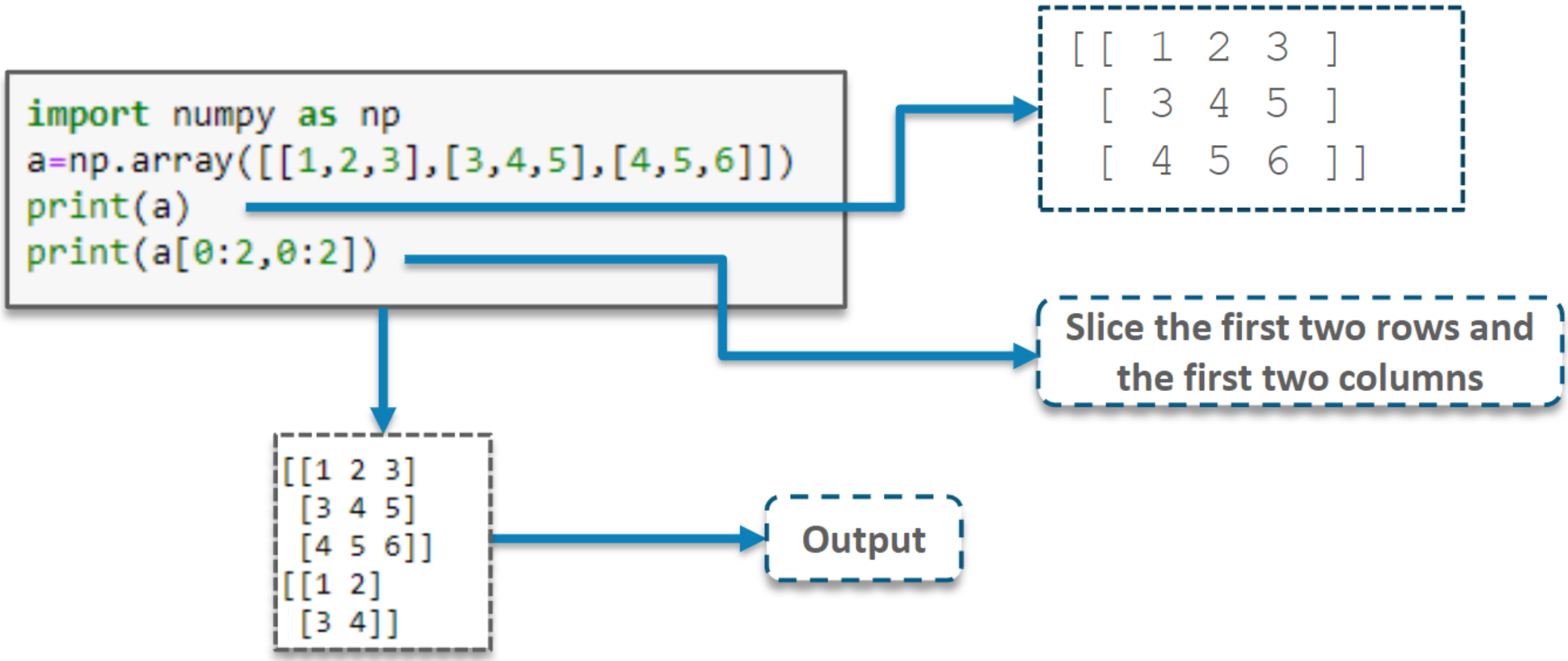
```
import numpy as np
arr=np.arange(20)
print(arr[:15])
```

```
[ 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14]
```

all elements with index lesser than 15

Slicing Multi-dimensional Arrays

Extracting specific rows and columns using slicing



Iterating in a numpy Array

- ***numpy.nditer*** is an iterator object of NumPy package
- It is an efficient multidimensional iterator object through which iteration over an array is possible

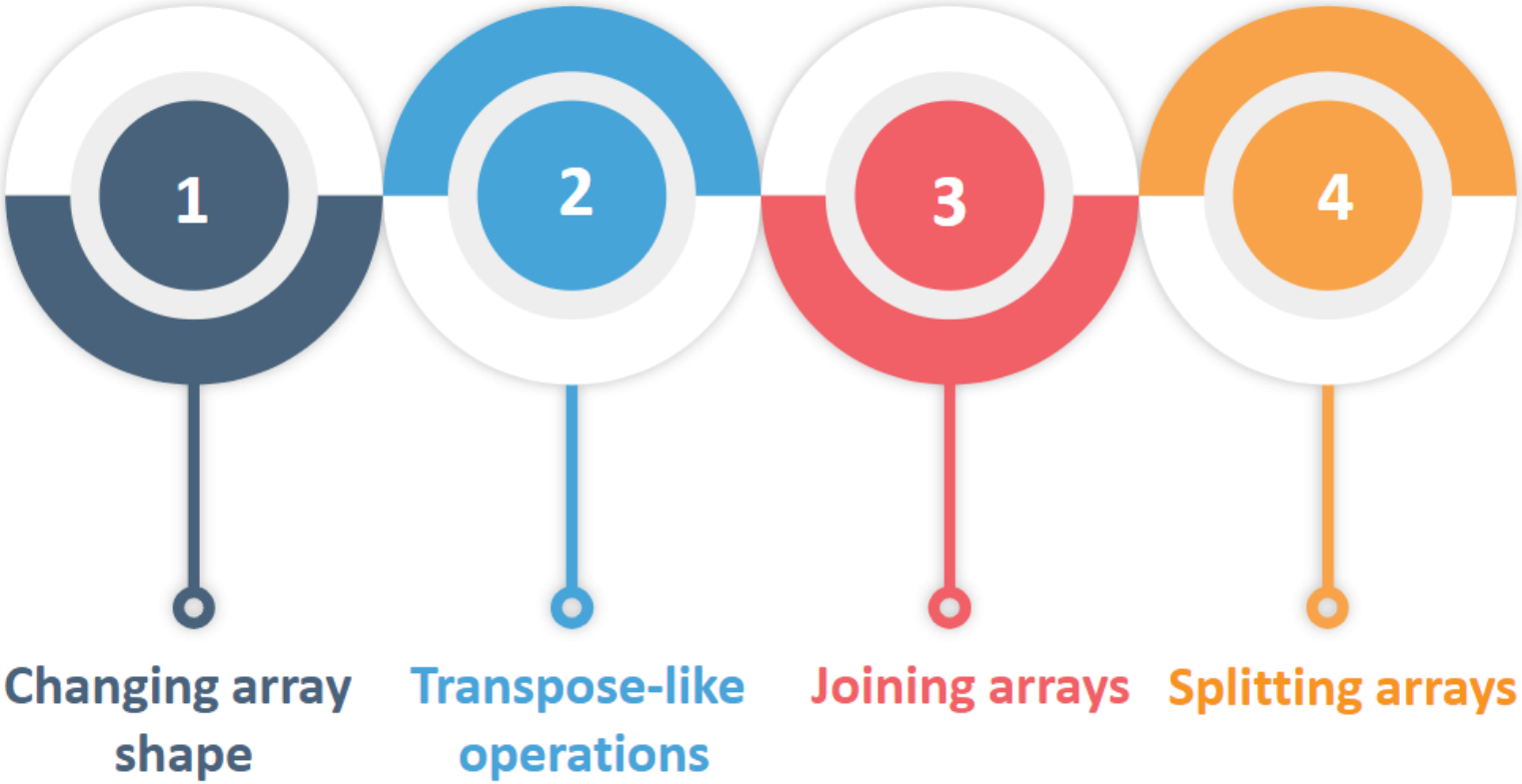
```
import numpy as np
a = np.arange(9)
a = a.reshape(3,3)
print('Original array is:')
print(a)
print('Modified array is:')
for x in np.nditer(a):
    print(x)
```

```
Original array is:
[[0 1 2]
 [3 4 5]
 [6 7 8]]
Modified array is:
0
1
2
3
4
5
6
7
8
```

Note: Here we can access individual elements of the arrays

Array Manipulation

- Often we need to shape an array or create an array using already created arrays; we need to perform array manipulation
- Here are some array manipulation routines:



Changing Array Shape – *reshape()* and *ravel()*

reshape() provides a new shape to an array without altering its data

```
x = [[0,1,2], [3,4,5]]
arr = np.array(x)
print(arr)
arr1 = np.reshape(arr, (3, 2))
print(arr1)
```

```
[[0 1 2]
 [3 4 5]]
```

```
[[0 1]
 [2 3]
 [4 5]]
```

ravel() is used to convert a two-dimensional array into a one-dimensional array

```
import numpy as np
a= np.arange(0,9).reshape(3,3)
print('Original Array is:')
print(a)
print('After ravel is:')
b= a.ravel()
print(b)
```

```
Original Array is:
[[0 1 2]
 [3 4 5]
 [6 7 8]]
After ravel is:
[0 1 2 3 4 5 6 7 8]
```

Output

Transpose-like Operations – *transpose()*

The ***transpose()*** function is used to invert columns with the rows

```
import numpy as np
a = np.arange(12).reshape(3,4)
print('Original Array:')
print(a)
b = np.transpose(a)
print('After Transpose:')
print(b)
```

```
Original Array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
After Transpose:
[[ 0  4  8]
 [ 1  5  9]
 [ 2  6 10]
 [ 3  7 11]]
```

Note: .T gives the same output as ***transpose(). transpose()*** provides more flexibility

Joining Arrays – *concatenate()*

- Multiple arrays are merged to form a new one that contains all the array
- Here are some functions of joining arrays :
  - ***concatenate()*** joins a sequence of arrays along an existing axis

For axis = None

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=None)
```

```
array([1, 2, 3, 4, 5, 6])
```

For axis = 1

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b.T), axis=1)
```

```
array([[1, 2, 5],
       [3, 4, 6]])
```

For axis = 0

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([[5, 6]])
np.concatenate((a, b), axis=0)
```

```
array([[1, 2],
       [3, 4],
       [5, 6]])
```

Note: The axis parameter specifies the index of the new axis in the dimensions of the results

Joining Arrays – *stack()* and *column_stack()*

stack() joins a sequence of arrays along a new axis

```
import numpy as np
a = np.array([1, 2, 3])
b = np.array([2, 3, 4])
np.stack((a, b))
```

```
array([[1, 2, 3],
       [2, 3, 4]])
```

column_stack() stacks 1-D arrays as columns into a 2-D array

```
import numpy as np
a = np.array([0,1,2])
b = np.array([3,4,5])
c = np.array([6,7,8])
print(np.column_stack((a,b,c)))
```

```
[[0 3 6]
 [1 4 7]
 [2 5 8]]
```

Note: Size of the arrays must be same in ***stack()***

Joining Arrays – *hstack()* and *vstack()*

hstack() adds the second array to the columns of the first array

```
a = np.array((1,2,3))
b = np.array((2,3,4))
np.hstack((a,b))
```

```
array([1, 2, 3, 2, 3, 4])
```

vstack() combines the second array as new rows in the first array

```
a = np.array((1,2,3))
b = np.array((2,3,4))
np.vstack((a,b))
```

```
array([[1, 2, 3],
       [2, 3, 4]])
```

Splitting Arrays – *split()*, *hsplit()* and *vsplit()*

split() splits an array into multiple sub-arrays

```
import numpy as np
x = np.arange(9)
np.split(x, 3)
```

```
[array([0, 1, 2]), array([3, 4, 5]), array([6, 7, 8])]
```

hsplit() splits the array horizontally

```
import numpy as np
a = np.arange(16).reshape((4,4))
[b,c]= np.hsplit(a,2)
print('b=',b)
print('c=',c)
```

```
b= [[ 0  1]
 [ 4  5]
 [ 8  9]
 [12 13]]
c= [[ 2  3]
 [ 6  7]
 [10 11]
 [14 15]]
```

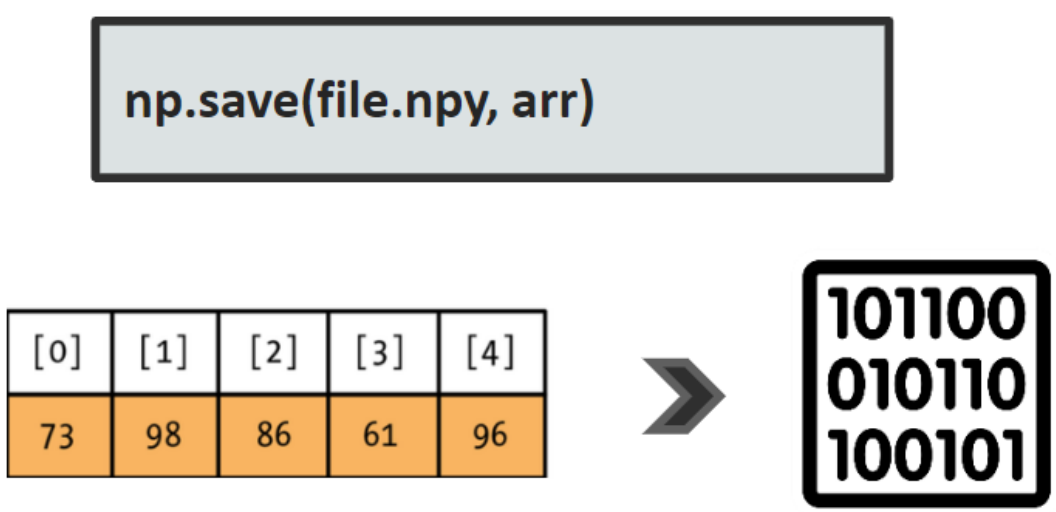
vsplit() splits the array vertically

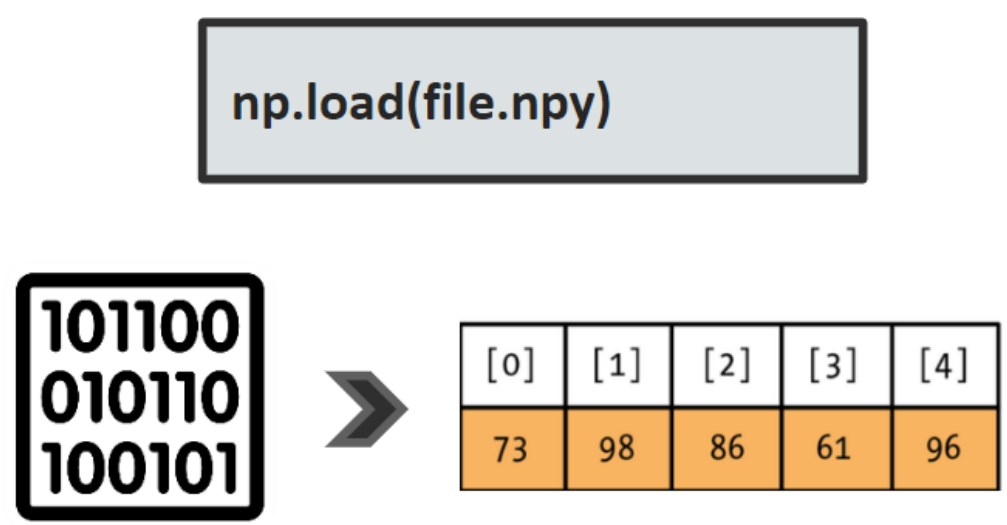
```
import numpy as np
a = np.arange(16).reshape((4,4))
[b,c]= np.vsplit(a,2)
print('b=',b)
print('c=',c)
```

```
b= [[0 1 2 3]
 [4 5 6 7]]
c= [[ 8  9 10 11]
 [12 13 14 15]]
```

Loading and Saving Data in Binary File (.npy, .npz)

- NumPy provides a pair of functions called ***save()*** and ***load()*** that enables to save and retrieve data stored in binary format
- To recover data from binary files ***load()*** function is used and to save an array to binary file ***save()*** function is used





Loading and Saving Data in Text Files (.txt, .csv)

- NumPy provides a set of functions called ***savetxt()*** and ***genfromtxt()*** that enable us to save and retrieve data stored in text format
- To save an array to text file ***savetxt()*** is used and to load data from text file ***genfromtxt()*** is used

np.savetxt(file.txt,arr,delimiter =',')

[0]	[1]	[2]	[3]	[4]
73	98	86	61	96

np.genfromtxt(file.txt,arr,delimiter =',')

[0]	[1]	[2]	[3]	[4]
73	98	86	61	96

Note: Same functions are used for .csv files

Demo 3: File Handling Using NumPy