Introduction to NumPy

Numerical Python

What is NumPy

NumPy is a Python library used for working with arrays.

It also has functions for working in domain of linear algebra, fourier transform, and matrices.

NumPy was created in 2005 by Travis Oliphant. It is an open source project and you can use it freely.

NumPy stands for Numerical Python.

Why Use NumPy over a typical array and list?

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 arrays are not an Python array nor a List!

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.

Why learn NumPy?

- **Efficient data storage and manipulation**: NumPy provides efficient ways to store and manipulate large, multi-dimensional arrays of data. This is essential for working with data in many fields, including physics, engineering, finance, and machine learning.
- **Data cleaning and preprocessing**: Business analysts often need to clean and preprocess data before performing analysis. NumPy provides powerful tools for data cleaning, such as replacing missing values and handling outliers.
- **Statistical analysis**: NumPy provides many functions for statistical analysis, such as computing means, medians, and standard deviations. These tools can help business analysts extract insights from data and make informed decisions.
- **Data visualization**: NumPy arrays can be easily plotted using visualization libraries like Matplotlib. Business analysts can use these tools to create clear and informative data visualizations that help stakeholders understand the results of their analysis.
- **Integration with other tools**: NumPy is a widely-used library in the scientific Python ecosystem, and many other data analysis libraries, such as Pandas, build on top of NumPy arrays. Learning NumPy can help business analysts use these other tools more effectively and integrate them into their workflow.

Installation of NumPy

If you have Python and PIP already installed on a system, then installation of NumPy is very easy.

Install it using this command:

- **C:\Users\<your name>\pip install numpy**

If this command fails, then use a python distribution that already has NumPy installed like, Anaconda, Spyder etc.

Make sure to install it in your virtual environment.

Import Numpy

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

```
import numpy
```

Now NumPy is imported and ready to use.

code:

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

output:

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

NumPy as np

NumPy is usually imported under the `np` alias.

alias: in Python alias is an alternate name for referring to the same thing.

```
import numpy as np
```

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

NumPy Code With Alias

code:

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

output:

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

Checking NumPy Version

The version string is stored under `__version__` attribute.

REMEMBER: Checking the version is important to prevent deprecated code or inconsistencies.

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

NumPy Creating Arrays

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

We can create a NumPy `ndarray` object by passing a list, tuple or any array-like object into the `array()` method.

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

output:

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

Dimensions in Arrays

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

0D Arrays

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

Example:

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

```
output:
42
```

1D Arrays

An array that has 0D arrays as its elements is called uni-dimensional, a 1D array, or a ***Vector***.

These are the most common and basic arrays.

Example:

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

```
output:
[1 2 3 4 5]
```

2D Arrays

An array that has 1D arrays as its elements is called a 2D array or a ***Matrix***.

These are often used to represent matrix or 2nd order tensors.

Example:

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

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

3D Arrays

An array that has 2D arrays (matrices) as its elements is called 3D array or ***Tensor***.
Higher than 3D, a >=4D are also referred to as a tensor with 4-dimensions.

These are often used to represent a 3rd order tensor.

Example:

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

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

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

Check Number of Dimensions?

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

Example:

```
import numpy as np

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

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

```
output:
0
1
```

Higher Dimension Arrays

An array can have any number of dimensions.

When the array is created, you can define the number of dimensions by using the `ndmin` argument.

Example:

```
import numpy as np

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

print('dimensions :', arr.ndim)
```

```
output:
[[[[[1 2 3 4]]]]]
dimensions : 5
```

Exercise 1

E1 - Q1

Insert the correct method for creating a NumPy array.

```
arr = np.____([1, 2, 3, 4])
```

E1 - Q2

Insert the correct syntax for checking the number of dimension of a NumPy array.

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

E1 - Q3

Insert the correct argument for creating a NumPy array with 2 dimensions.

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

NumPy Array Slicing

Slicing in Python means taking elements from one given index to another given index

Slicing Arrays

- Slicing in python means taking elements from one given index to another given index.
- We pass slice instead of index like this: `[start:end]`.
- We can also define the step, like this: `[start:end:step]`.
- If we don't pass start its considered 0
- If we don't pass end its considered length of array in that dimension
- If we don't pass step it’s considered 1

Slicing Arrays

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

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5])
```

```
output:
[2 3 4 5]
```

Slicing Arrays

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

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[4:])
```

```
output:
[5 6 7]
```

Slicing Arrays

Example: Slice elements from beginning to index 4 (not included):

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[:4])
```

```
output:
[1 2 3 4]
```

Negative Slicing

Use the minus operator to refer to an index from the end.

Example: Slice from the index 3 from the end to index 1 from the end.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[-3:-1])
```

```
output:
[5 6]
```

Step

Use the `step` value to determine the step of the slicing:

Example: Return every other element from index 1 to index 5.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[1:5:2])
```

```
output:
[2 4]
```

Step

Example: Return every other element from the entire array

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])

print(arr[::2])
```

```
output:
[1 3 5 7]
```

Slicing 2D Arrays

Example: From the second element, slice elements from index 1 to index 4 (not included)

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5],
                [6, 7, 8, 9, 10]])

print(arr[1, 1:4])
```

```
output:
[7 8 9]
```

Slicing 2D Arrays

Example: From both elements, return index 2

```
import numpy as np    COLUMNS

arr = np.array([[1, 2, 3, 4, 5],
                [6, 7, 8, 9, 10]])  ROWS

print(arr[0:2, 2])
```

ROWS

COLUMNS

```
output:
[3 8]
```

Slicing 2D Arrays

Example: From both elements, slice index 1 to index 4 (not included), this will return a 2D array

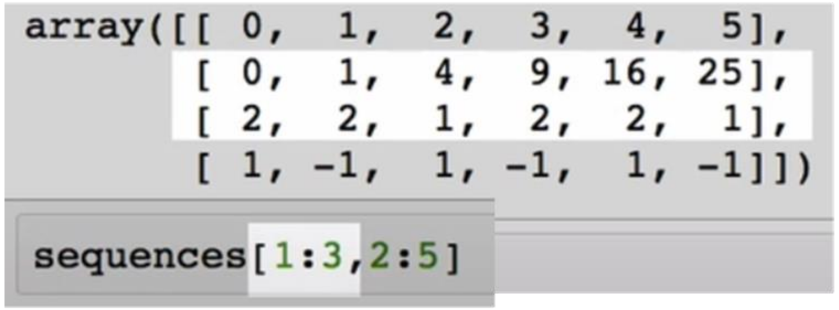
arr[start:end ***(ROW)****,* ***column****]*

```
import numpy as np

arr = np.array([[1, 2, 3, 4, 5],
                [6, 7, 8, 9, 10]])

print(arr[0:2, 2])
```

output:

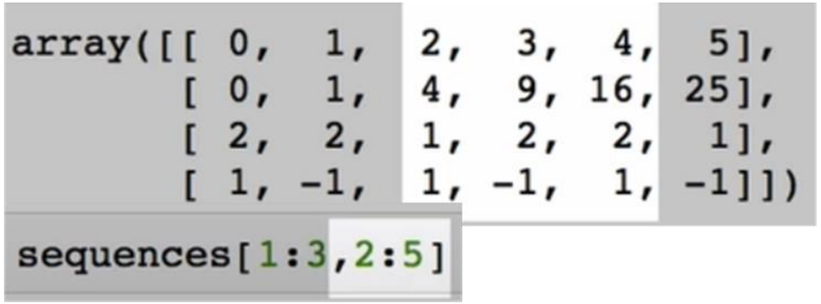
```
[3 8]
```

Slicing Arrays



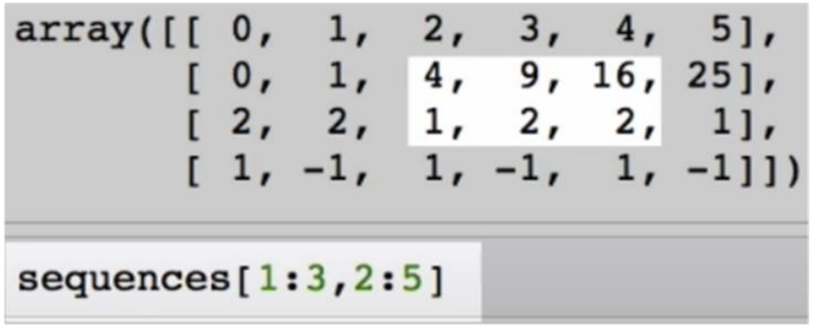
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 0,  1,  4,  9, 16, 25],
       [ 2,  2,  1,  2,  2,  1],
       [ 1, -1,  1, -1,  1, -1]])
```

```
sequences[1:3,2:5]
```



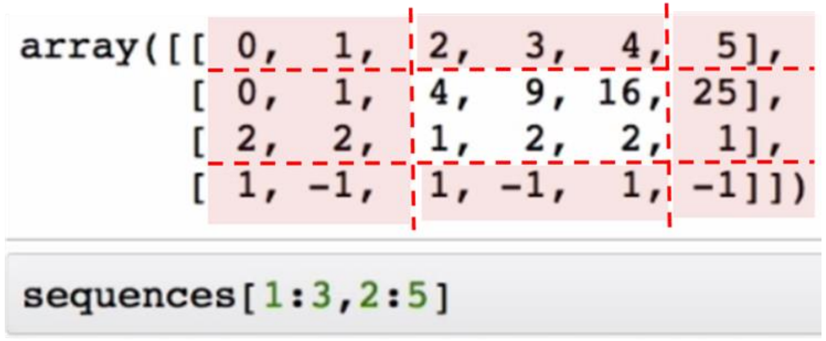
```
array([[ 0,  1,  2,  3,  4,  5],
       [ 0,  1,  4,  9, 16, 25],
       [ 2,  2,  1,  2,  2,  1],
       [ 1, -1,  1, -1,  1, -1]])
```

```
sequences[1:3,2:5]
```



```
array([[ 0,  1,  2,  3,  4,  5],
       [ 0,  1,  4,  9, 16, 25],
       [ 2,  2,  1,  2,  2,  1],
       [ 1, -1,  1, -1,  1, -1]])
```

```
sequences[1:3,2:5]
```



```
array([[ 0,  1,  2,  3,  4,  5],
       [ 0,  1,  4,  9, 16, 25],
       [ 2,  2,  1,  2,  2,  1],
       [ 1, -1,  1, -1,  1, -1]])
```

```
sequences[1:3,2:5]
```

Tip

- When slicing with a single **range** at the **first position**, it grabs the **column** of the **range** and **takes the values of the numbers in x position** [**range**, **x**].
- When having [**x**, **range**], the **x takes the index of the array and takes the range values of it**.
- Having **two range values** [**row range**, **column range**] **will slice the row and columns, respectively.**

Exercise 2

E2 - Q1

Insert the correct slicing syntax to print the following selection of the array:

Everything from (including) the second item to (not including) the fifth item.

```
arr = np.array([10, 15, 20, 25, 30,35, 40])
print(arr_____)
```

E2 - Q2

Insert the correct slicing syntax to print the following selection of the array:

Everything from (including) the third item to (not including) the fifth item.

```
arr = np.array([10, 15, 20, 25, 30, 35, 40])
print(arr____)
```

E2 - Q3

Insert the correct slicing syntax to print the following selection of the array:

Every other item from (including) the second item to (not including) the fifth item.

Tip: use the step syntax.

```
arr = np.array([10, 15, 20, 25, 30, 35, 40])
print(arr_____)
```

E2 - Q4

Insert the correct slicing syntax to print the following selection of the array:

Every other item from the entire array.

Tip: use the step syntax.

```
arr = np.array([10, 15, 20, 25, 30, 35, 40])
print(arr___)
```

NumPy Data Types

Data Types in Numpy

Example: From both elements, slice index 1 to index 4 (not included), this will return a 2D array

- `i` - integer
- `b` - boolean
- `u` - unsigned integer
- `f` - float
- `c` - complex float
- `V` - fixed chunk of memory for other type (void)

- `m` - timedelta
- `M` - datetime
- `O` - object
- `S` - string
- `U` - unicode string

Checking the Data Type of an Array

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

Example:

```
import numpy as np

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

print(arr.dtype)
```

```
output:
int64
```

Checking the Data Type of an Array

Example: Get the data type of an array containing strings

```
import numpy as np

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

```
output:
U6
U - indicates that
it is a string
Where 6 indicates
that its has <=6
characters
```

Creating Arrays with a Defined Data Type

We use the `array()` function to create arrays, this function can take an optional argument:

`dtype` that allows us to define the expected data type of the array elements:

Example: Create an array with data type String

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

output:

```
[b'1' b'2' b'3' b'4']
|S1
b - indicates bytes
|S1 - data type string
with 1 byte
```

Creating Arrays with a Defined Data Type

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

Example: Create an array with data type 4 bytes integer

```
import numpy as np

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

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

```
output:
[1 2 3 4]
int32
```

```
i4 means that it is a
32-bit integer in
numpy
```

What if a Value Cannot be Converted?

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

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

Example: A Non integer string like ‘a’ cannot be converted to integer.

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

Converting Data Type on Existing Arrays

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

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

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

Converting Data Type on Existing Arrays

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

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype('i')

print(newarr)
print(newarr.dtype)
```

```
output:
[1 2 3]
int32
```

Converting Data Type on Existing Arrays

Example: Change the data type from float to integer by using **int** as parameter value.

```
import numpy as np

arr = np.array([1.1, 2.1, 3.1])
newarr = arr.astype(int)

print(newarr)
print(newarr.dtype)
```

```
output:
[1 2 3]
int64
```

Converting Data Type on Existing Arrays

Example: Change the data type from integer to boolean

```
import numpy as np

arr = np.array([1, 0, 3])
newarr = arr.astype(bool)

print(newarr)
print(newarr.dtype)
```

```
output:
[ True False  True]
bool
```

Values >0 == True

Exercise 3

E3 - Q1

NumPy uses a character to represent each of the following data types, which one?

i = **integer**
_ = **boolean**
_ = **unsigned integer**
_ = **float**
_ = **complex float**
_ = **timedelta**
_ = **datetime**
_ = **object**
_ = **string**

E3 - Q2

Insert the correct NumPy syntax to print the data type of an array.

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

E3 - Q3

Insert the correct argument to specify that the array should be of type STRING.

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

E3 - Q4

Insert the correct method to change the data type to integer.

```
arr = np.array([1.1, 2.1, 3.1])
newarr = arr._____
```

NumPy Array Copy vs View

Shallow Copy vs Deep Copy

- Understanding shallow and deep copy in Python is crucial for effective data manipulation and maintaining data integrity.
- These concepts are essential when working with complex data structures like lists and dictionaries.
- Shallow copies create efficient references to existing data, which can be beneficial for memory and performance.
- However, they require careful handling to avoid unintended modifications to the original data. In contrast, deep copies provide complete data isolation, ensuring that changes in one object don't affect others.
- By comprehending the distinctions between these copying methods, programmers can make informed decisions to protect their data and optimize their code, allowing for more precise control over how data is shared and modified within Python programs.

Shallow Copy

- A shallow copy **creates a new array**, but it **does not create new copies of the elements within the array**. Instead, it points to the same elements as the original array. A deep copy, on the other hand, creates a completely independent copy of both the array and its data. It does not share any data with the original array.
- Shallow copy **creates a new object that references the same inner objects as the original**, **sharing** data references. **Changes to inner objects affect all references**, but **changes to top-level objects may not**.
- The following code demonstrates the impact of modifying the inner and top-level items of a list (origin) on shallow copies (shallow_copy) and a reference (referenced).

Shallow Copy Code

```
from copy import copy

# changing inner item
origin = [[1], [2], [3]]
shallow_copy = copy(origin)
referenced = origin
origin[1][0] = 0

# Result
print(origin)        # [[1], [0], [3]]
print(shallow_copy)  # [[1], [0], [3]]
print(referenced)    # [[1], [0], [3]]

# changing a top-level item
origin = [[1], [2], [3]]
shallow_copy = copy(origin)
referenced = origin
origin[1] = 0

# Result
print(origin)        # [[1], 0, [3]]
print(shallow_copy)  # [[1], [2], [3]]
print(referenced)    # [[1], 0, [3]]
```

- When you change the inner item (origin[1][0]), all collections reflect the change.
- When you change a top-level item (origin[1]), the reference reflects the change, but the shallow copy remains unchanged because it shares references to the inner lists.
- Shallow copy creates a new object that references the same inner objects as the original. Changing the inner object affects all references. **However, if you change the top-level object (origin[1]), only references directly pointing to that object (like referenced) reflect the change, while the shallow copy (shallow_copy) retains the original reference.**

Deep Copy

```
from copy import deepcopy

origin = [[1],[2],[3]]
deep_copy = deepcopy(origin)
referenced = origin
origin[1][0]= 0

# Result
print(origin)       # [[1], [0], [3]]
print(deep_copy)  # [[1], [2], [3]]
print(referenced)    # [[1], [0], [3]]
```

- The first line imports the deepcopy function from the copy module, allowing you to create a deep copy of objects. This deep copy duplicates the entire structure, including all nested sublists, ensuring independence.
- When we modify the second element of the first sublist within the origin list, changing it from 2 to 0.
- origin is changed to [[1], [0], [3]] because you directly modified the original list.
- deep_copy remains [[1], [2], [3]] because it's a deep copy, and changes to the origin list do not affect it.
- referenced also reflects the change, becoming [[1], [0], [3]], as it points to the same list object as origin.

Shallow and Deep Copy in NumPy

- When working with arrays and data structures in Python, it's crucial to grasp the concepts of shallow copy and deep copy. These concepts become even more relevant when dealing with NumPy arrays, a fundamental part of scientific computing in Python.
- In NumPy, you can easily create references to the same data buffer as the original array using the view() method.
- The changes made to the original data affect the view, and changes to the view affect the original data.

NumPy does not have shallow copies.

NumPy arrays, by their nature, do not have a concept of shallow copies.
NumPy arrays are contiguous blocks of memory where elements are stored, and there is no hierarchy of elements or references within a NumPy array like you would find in nested Python lists.

Using Copy with a NumPy Array

- In the context of NumPy, the term "shallow copy" can be misleading because it implies a sharing of data that doesn't actually occur in the same way it does with regular Python lists.
- In NumPy, views are the mechanism that allows you to create alternative perspectives on the same data, but it's not a true shallow copy in the sense of copying references or hierarchies of elements.
- In NumPy, the np.copy() method is your go-to tool for creating deep copies.

```
import numpy as np

original_array = np.array([1, 2, 3, 4])
shallow_copy = original_array.view()
print(original_array)
print(shallow_copy)

shallow_copy[0] = 2

print(original_array)
print(shallow_copy)

Output:
[1 2 3 4]
[1 2 3 4]
[2 2 3 4]
[2 2 3 4]
```

NumPy and Python lists are different

- The last important point to emphasize is that Python's copy and deepcopy functions do not behave in the same way as they do with Python lists.
- This is primarily due to the fact that NumPy arrays lack a hierarchical structure.
- As a result, when copying NumPy arrays, they are always deeply copied, preserving their original structure without sharing references.

Summary of Shallow and Deep Copies

- copy.deepcopy(): Deep copy creates an entirely independent copy of the list object, so changes in the original do not affect it.
- References point to the same object, so changes in the original object are reflected in both references.
- copy.copy(): A shallow copy duplicates the outer structure of a list object but not the inner elements.
- np.copy(): Creates an independent copy of a NumPy array, fully duplicating the data.
- np.view(): Creates an alternative perspective, sharing the same data, and changes to one affect the other.
- Python's built-in copy() and deepcopy() both create an independent copy of the original data.

The Difference Between Copy and View

The main difference between a copy and a view of an array is that the copy is a new array, and the view is just a view of the original array.

The copy *owns* the data and any changes made to the copy will not affect original array, and any changes made to the original array will not affect the copy.

The view *does not own* the data and any changes made to the view will affect the original array, and any changes made to the original array will affect the view.

Copy

The copy **SHOULD NOT be affected** by the changes made to the original array.

Example: Make a copy, change the original array, and display both arrays:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.copy()
arr[0] = 42

print(arr)
print(x)
```

output:

```
[42  2  3  4  5]
[1 2 3 4 5]
```

View

The view **SHOULD be affected** by the changes made to the original array.

Example: Make a view, change the original array, and display both arrays

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5])
x = arr.view()
x[0] = 31

print(arr)
print(x)
```

```
output:
[31  2  3  4  5]
[31  2  3  4  5]
```

Exercise 4

E4 - Q1

Use the correct method to make a copy of the array.

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

E4 - Q2

Use the correct method to make a view of the array.

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

x = arr._______
```

NumPy Array Shape

The shape of the array is the number of elements in each dimension

Get the Shape of an Array

NumPy arrays have an attribute called `shape` that returns a tuple with each index having the number of corresponding elements.

Example: Print the shape of a 2D array.

```
import numpy as np
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
print(arr.shape)
```

```
output:
(2, 4)
```

The example above returns `(2, 4)`, which means that the array has 2 dimensions, where the first dimension has 2 elements and the second has 4.

Get the Shape of an Array

Create an array with 5 dimensions using `ndmin` using a vector with values 1,2,3,4 and verify that last dimension has value 4:

```python
import numpy as np

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

print(arr)
print('shape :', arr.shape)
```

output:

```
[[[[[1 2 3 4]]]]]
shape : (1, 1, 1, 1, 4)
```

What Does the Shape Tuple Represent?

Integers at every index tells about the number of elements the corresponding dimension has.

In the example in the previous slide at index-4 we have value 4, so we can say that 5th (4 + 1 th) dimension has 4 elements.

Numpy Array Reshaping

Reshaping means changing the shape of an array.

The shape of an array is the number of elements in each dimension.

By reshaping we can add or remove dimensions or change number of elements in each dimension.

Reshape From 1D to 2D

Example: Convert the following 1-D array with 12 elements into a 2-D array.

The outermost dimension will have 4 arrays, each with 3 elements.

```
import numpy as np

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

newarr = arr.reshape(4, 3)

print(newarr)
```

```
output:
[[ 1  2  3]
 [ 4  5  6]
 [ 7  8  9]
 [10 11 12]]
```

Reshape From 1D to 3D

Example: Convert the following 1D array with 12 elements into a 3D array.

The outermost dimension will have 2 arrays that contains 3 arrays, each with 2 elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7,
                8, 9, 10, 11, 12])
newarr = arr.reshape(2, 3, 2)

print(newarr)
```

output:

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

 [[ 7  8]
  [ 9 10]
  [11 12]]]
```

Can We Reshape Into Any Shape?

Yes, as long as the elements required for reshaping are equal in both shapes.

We can reshape an 8 elements 1D array into 4 elements in 2 rows 2D array but we cannot reshape it into a 3 elements 3 rows 2D array as that would require 3x3 = 9 elements.

Returns Copy or View?

Example: Check if the returned array is a copy or a view.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7,8])
print(arr.reshape(2, 4).base)
```

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

The example above returns the original array, so its a view.

Base is None if its from its own memory.

Unknown Dimension

You are allowed to have one "unknown" dimension.

Meaning that you do not have to specify an exact number for one of the dimensions in the reshape method.

Pass `-1` as the value, and NumPy will calculate this number for you.

Unknown Dimension

Example: Convert 1D array with 8 elements to 3D array with 2x2 elements:

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
newarr = arr.reshape(2, 2, -1)

print(newarr)
```

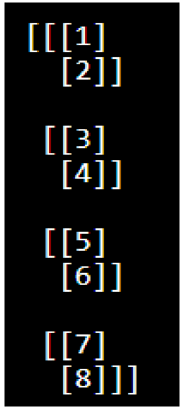
```
output:
[[[1 2]
  [3 4]]

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

Note: We cannot pass -1 to more than one dimension

Unknown Dimension

- Having an unknown dimension (-1) provides a way to easily turn a 1D array to a 3D array regardless of the size of its last dimension.
- It is important that the first two arguments meets the total size of the 1D array.

```python
import numpy as np

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

newarr = arr.reshape(4, 2, -1)

print(newarr)
```



```
[[[1]
  [2]]

 [[3]
  [4]]

 [[5]
  [6]]

 [[7]
  [8]]]
```

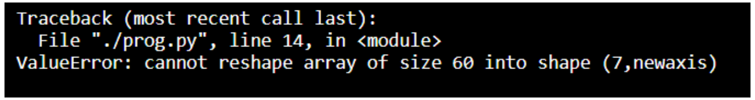
Increasing more than -1 for Unknown Dim

- When you set the unknown dimension to -2 or any negative number, NumPy treats it as a special value that means "calculate this dimension based on the total size of the array and the other dimensions, but leave out this dimension from the calculation". This can be useful in some cases where you want NumPy to automatically calculate the size of one or more dimensions based on the size of the array and the other dimensions, but you don't want to specify the exact size of those dimensions.

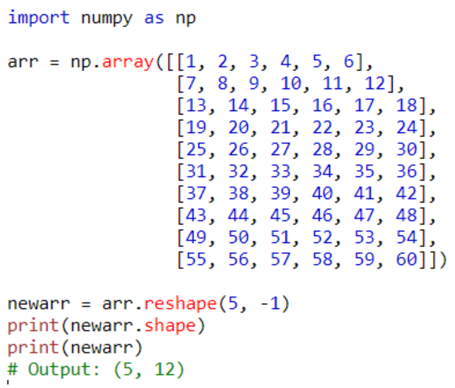
- In your example, when you set the unknown dimension to -1, NumPy calculates the value of that dimension as 2 (since 2x2=4, and 4x2=8), and reshapes the original array into a 2x2x2 array. However, when you set the unknown dimension to -2, NumPy calculates the value of that dimension as 4 (since 2x2x4=16, which is the size of the original array), but this dimension is not actually used in the reshaping operation, because you have already specified two other dimensions (2 and 2). Therefore, setting the unknown dimension to -2 has no effect in this case. Similarly, setting the unknown dimension to any other negative value that is not used in the calculation of the other dimensions will also have no effect.

Unknown Dimension

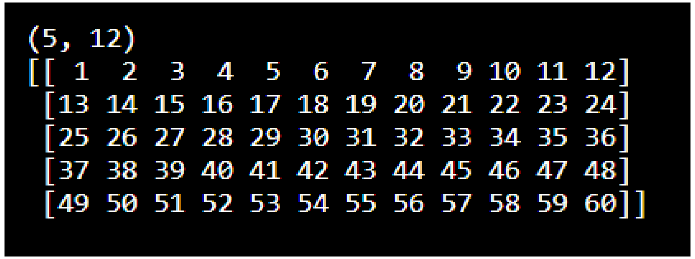
- Just remember that the unknown dimension gives the user a pre-defined number to multiply to the given missing value.
- Giving a value that is not possible to divide the given length of the array will cause an error. Ex: 60 is not divisible by 7.



```
Traceback (most recent call last):
  File "./prog.py", line 14, in <module>
ValueError: cannot reshape array of size 60 into shape (7,newaxis)
```



```python
import numpy as np

arr = np.array([[1, 2, 3, 4, 5, 6],
                [7, 8, 9, 10, 11, 12],
                [13, 14, 15, 16, 17, 18],
                [19, 20, 21, 22, 23, 24],
                [25, 26, 27, 28, 29, 30],
                [31, 32, 33, 34, 35, 36],
                [37, 38, 39, 40, 41, 42],
                [43, 44, 45, 46, 47, 48],
                [49, 50, 51, 52, 53, 54],
                [55, 56, 57, 58, 59, 60]])

newarr = arr.reshape(5, -1)
print(newarr.shape)
print(newarr)
# Output: (5, 12)
```



```
(5, 12)
[[ 1  2  3  4  5  6  7  8  9 10 11 12]
 [13 14 15 16 17 18 19 20 21 22 23 24]
 [25 26 27 28 29 30 31 32 33 34 35 36]
 [37 38 39 40 41 42 43 44 45 46 47 48]
 [49 50 51 52 53 54 55 56 57 58 59 60]]
```

Flattening the Arrays

Flattening array means converting a multidimensional array into a 1D array.

We can use `reshape(-1)` to do this. Having (-1, m) will consider it as a typical slicing.

Example: Convert the array into a 1D array.

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

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

NOTE

- Remember! Arrays need to have an equally sized number of elements.
- Having [1, 2, 3] [4, 5, 6] [7, 8, 9] is acceptable.
- This example [1, 2] [3, 4, 5, 6] [7, 8, 9] is not acceptable, as this is not an array.
- All elements in each dimension should always be equal to be considered an n-dimensional array.

Exercise 5

E5 - Q1

Use the correct NumPy syntax to check the shape of an array

.

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

print(arr._______)
```

E5 - Q2

Use the correct NumPy method to change the shape of an array from 1D to 2D.

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

newarr = arr.______(4, 3)
```

E5 - Q3

Exercise: Use a correct NumPy method to change the shape of an array from 2D to 1D.

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

newarr = arr.______
```

NumPy Array Iterating

Iterating Arrays

Iterating means going through elements one by one.

As we deal with multi-dimensional arrays in numpy, we can do this using basic `for` loop of python.

If we iterate on a 1D array it will go through each element one by one.

Iterating Arrays

Example: Iterate on the elements of the following 1-D array.

```
import numpy as np

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

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

```
output:
1
2
3
```

Iterating 2D Arrays

In a 2-D array it will go through all the rows.

Example

Iterate on the elements of the following 2-D array:

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

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

Iterating 2D Arrays

To return the actual values, the scalars, we have to iterate the arrays in each dimension.

Example: Iterate on each scalar element of the 2-D array.

```
import numpy as np

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

for x in arr:
  for y in x:
    print(y)
```

```
output:
1
2
3
4
5
6
```

Iterating Arrays Using nditer()

The function `nditer()` is a helping function that can be used from very basic to very advanced iterations. It solves some basic issues which we face in iteration, let’s go through it with examples.

```
import numpy as np

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

for x in arr:
  for y in x:
    print(y)
```

```
output:
1
2
3
4
5
6
```

Iterating on Each Scalar Element

In basic `for` loops, iterating through each scalar of an array we need to use *n* `for` loops which can be difficult to write for arrays with very high dimensionality.

Example: Iterate through the following 3D array.

```
import numpy as np
arr = np.array([[[1], [2]], [[3], [4]]])
for x in np.nditer(arr):
  print(x)
```

```
output:
1
2
3
4
```

Iterating Array With Different Data Types

We can use `op_dtypes` argument and pass it the expected datatype to change the datatype of elements while iterating.

NumPy does not change the data type of the element in-place (where the element is in array) so it needs some other space to perform this action, that extra space is called buffer, and in order to enable it in `nditer()` we pass `flags=['buffered']`.

Iterating Array With Different Data Types

Example: Iterate through the array as a string.

```
import numpy as np
arr = np.array([1, 2, 3])
for x in np.nditer(arr,
                   flags=['buffered'],
                   op_dtypes=['S']):
  print(x)
```

```
output:
b'1'
b'2'
b'3'
```

Enumerated Iteration Using ndenumerate()

Enumeration means mentioning sequence number of somethings one by one.

Sometimes we require corresponding index of the element while iterating, the `ndenumerate()` method can be used for those use cases.

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

```
output:
1
3
5
7
```

Enumerated Iteration Using **ndenumerate()**

Example: Enumerate on following 1D arrays elements

```
import numpy as np
arr = np.array([1, 2, 3])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

```
output:
(0,) 1
(1,) 2
(2,) 3
```

Enumerated Iteration Using **ndenumerate()**

Example: Enumerate on following 2D arrays elements

```
import numpy as np
arr = np.array([[1, 2, 3, 4],
                [5, 6, 7, 8]])
for idx, x in np.ndenumerate(arr):
  print(idx, x)
```

```
output:
(0, 0) 1
(0, 1) 2
(0, 2) 3
(0, 3) 4
(1, 0) 5
(1, 1) 6
(1, 2) 7
(1, 3) 8
```

NumPy Joining Array

Joining NumPy Arrays

Joining means putting contents of two or more arrays in a single array.

In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

We pass a sequence of arrays that we want to join to the `concatenate()` function, along with the axis. If axis is not explicitly passed, it is taken as 0.

Joining NumPy Array

Example: Join two arrays

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.concatenate((arr1, arr2))
print(arr)
```

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

Joining NumPy Array

Example: Join two 2D arrays along rows (axis = 1).

```
import numpy as np
arr1 = np.array([[1, 2], [3, 4]])
arr2 = np.array([[5, 6], [7, 8]])
arr = np.concatenate((arr1, arr2), axis=1)
print(arr)
```

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

Joining Arrays Using Stack Functions

Stacking is same as concatenation, the only difference is that stacking is done along a new axis.

We can concatenate two 1-D arrays along the second axis which would result in putting them one over the other, ie. stacking.

We pass a sequence of arrays that we want to join to the `stack()` method along with the axis. If axis is not explicitly passed it is taken as 0.

Joining Arrays Using Stack Functions

Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.stack((arr1, arr2), axis=1)
print(arr)
```

output:

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

Stacking Along Rows

NumPy provides a helper function: `hstack()` to stack along rows.

Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.hstack((arr1, arr2))
print(arr)
```

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

Stacking Along Columns

NumPy provides a helper function: `vstack()` to stack along columns.

Example:

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack((arr1, arr2))
print(arr)
```

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

Stacking Along Height (depth)

NumPy provides a helper function: `dstack()` to stack along height, which is the same as depth.

```
import numpy as np
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.dstack((arr1, arr2))
print(arr)
```

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

Exercise 6

E6 - Q1

Use a correct NumPy method to join two arrays into a single array.

```
arr1 = np.array([1, 2, 3])

arr2 = np.array([4, 5, 6])

arr = np._______((arr1, arr2))
```

E6 - Q2

Stack the two arrays alongside its column

```
arr1 = np.array([1, 2, 3])
arr2 = np.array([4, 5, 6])
arr = np.vstack(__________)
```

NumPy Splitting Array

Splitting NumPy Arrays

Splitting is reverse operation of Joining.

Joining merges multiple arrays into one and Splitting breaks one array into multiple.

We use `array_split()` for splitting arrays, we pass it the array we want to split and the number of splits.

Splitting Numpy Arrays

Example: Split the array in 3 parts.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6])
newarr = np.array_split(arr, 3)
print(newarr)
```

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

Note: The return value is an **array** containing three arrays

Splitting 2D Arrays

Use the same syntax when splitting 2D arrays.

Use the `array_split()` method, pass in the array you want to split and the number of splits you want to do.

Splitting 2D Arrays

Example: Split the 2D array into three 2D arrays

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

```
output:
[array([[1, 2],
        [3, 4]]),
 array([[5, 6],
        [7, 8]]),
 array([[9, 10],
        [11, 12]])]
```

The example above returns three 2D arrays. In addition, you can specify which axis you want to do the split around.

Splitting 2D Arrays

Example: Split the 2D array into three 2D arrays along rows.

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

```
output:
[array([[ 1],
       [ 4],
       [ 7],
       [10]]),
array([[ 2],
       [ 5],
       [ 8],
       [11]]),
array([[ 3],
       [ 6],
       [ 9],
       [12]])]
```

Splitting 2D Arrays

An alternate solution is using `hsplit()` opposite of `hstack()`

Example: Split the 2D array into three 2D arrays along rows.

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

output:

```
[array([[ 1],
       [ 4],
       [ 7],
       [10]]),
array([[ 2],
       [ 5],
       [ 8],
       [11]]),
array([[ 3],
       [ 6],
       [ 9],
       [12]])]
```

Splitting 2D Arrays

An alternate solution is using `hsplit()` opposite of `hstack()`

Example: Split the 2D array into three 2D arrays along rows.

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

```
output:
[array([[ 1],
        [ 4],
        [ 7],
        [10]]),
 array([[ 2],
        [ 5],
        [ 8],
        [11]]),
 array([[ 3],
        [ 6],
        [ 9],
        [12]])]
```

Splitting 2D Arrays

Similar alternatives to `vstack()` and `dstack()` are available as `vsplit()` and `dsplit()`.

NumPy Searching Arrays

Searching Arrays

You can search an array for a certain value, and return the indexes that get a match.

To search an array, use the `where()` method.

Example: Find the indexes where the value is 4.

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

Output:

```
(array([3, 5, 6]),)
```

Searching Arrays

Example: Find the indexes where the values are even.

```
import numpy as np
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
x = np.where(arr%2 == 0)
print(x)
```

Output:

```
(array([1, 3, 5, 7]),)
```

Search Sorted

There is a method called `searchsorted()` which performs a binary search in the array, and returns the index where the specified value would be inserted to maintain the search order.

Example: Find the indexes where the value 7 should be inserted.

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7)
print(x)
```

```
Output:
1
```

The `searchsorted()` method is assumed to be used on sorted arrays.

Search From the Right Side

By default the left-most index is returned, but we can give `side='right'` to return the right-most index instead.

Example: Find the indexes where the value 7 should be inserted, starting from the right.

```
import numpy as np
arr = np.array([6, 7, 8, 9])
x = np.searchsorted(arr, 7, side='right')
print(x)
```

```
Output:
2
```

Example explained: The number 7 should be inserted on index 2 to remain the sort order.

The method starts the search from the right and returns the first index where the number 7 is no longer less than the next value.

Multiple Values

To search for more than one value, use an array with the specified values.

Example: Find the indexes where the values 2, 4, and 6 should be inserted.

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

Output:

```
[1 2 3]
```

The return value is an array: `[1 2 3]` containing the three indexes where 2, 4, 6 would be inserted in the original array to maintain the order.

NumPy Sorting Arrays

Sorting Arrays

Sorting means putting elements in an *ordered sequence*.

Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.

The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

Example:

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

Output:

```
[0 1 2 3]
```

Sorting Arrays

You can also sort arrays of strings, or any other data type:

Example: Sort the array alphabetically.

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

```
Output:
['apple'
'banana'
'cherry']
```

Sorting a 2D Array

If you use the sort() method on a 2-D array, both arrays will be sorted:

Example: Sort the array alphabetically.

```
import numpy as np
arr = np.array([[3, 2, 4], [5, 0, 1]])
print(np.sort(arr))
```

Output:

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

NumPy Filter Array

Filtering Arrays

Getting some elements out of an existing array and creating a new array out of them is called *filtering*.

In NumPy, you filter an array using a *boolean index list*.

A *boolean index list* is a list of booleans corresponding to indexes in the array.

- If the value at an index is `True` that element is contained in the filtered array, if the value at that index is `False` that element is excluded from the filtered array.

Filtering Arrays

Example: Create an array from the elements on index 0 and 2:

```
import numpy as np
arr = np.array([41, 42, 43, 44])
x = [True, False, True, False]
newarr = arr[x]
print(newarr)
```

```
Output:
[41 43]
```

The example above will return `[41, 43]`, why?

Because the new filter contains only the values where the filter array had the value `True`, in this case, index 0 and 2.

Creating the Filter Array

It is common to create a filter array based on conditions instead of hard coding boolean arrays.

Example: Create a filter array that will return only values higher than 42

Continued on the next page.

Creating the Filter Array

```
import numpy as np
arr = np.array([41, 42, 43, 44])
# Create an empty list
filter_arr = []
# go through each element in arr
for element in arr:
  # if the element is higher than 42,
    set the value to True, otherwise False:
  if element > 42:
          filter_arr.append(True)
  else:
     filter_arr.append(False)
newarr = arr[filter_arr]
print(filter_arr)
print(newarr)
```

Output:

```
[False, False, True, True]
[43 44]
```

Creating Filter Directly From Array

The previous example is quite a common task in NumPy and NumPy provides a nice way to tackle it.

We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

```
import numpy as np

arr = np.array([41, 42, 43, 44])
filter_arr = arr > 42
newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

Output:

```
[False False  True
True]
[43 44]
```

Creating Filter Directly From Array

The previous example is quite a common task in NumPy and NumPy provides a nice way to tackle it.

We can directly substitute the array instead of the iterable variable in our condition and it will work just as we expect it to.

```
import numpy as np

arr = np.array([1, 2, 3, 4, 5, 6, 7])
filter_arr = arr % 2 == 0
newarr = arr[filter_arr]

print(filter_arr)
print(newarr)
```

Output:

```
[False  True False  True False
True False]
[2 4 6]
```

Exercise 7

E7 - Q1

Sort the following array and filter out all the values that are less than 10. Print the final array.

```
arr = np.array([5, 1, 23, 2, 67, 82, 11, 8])
```

NumPy Mathematics

This section lists some of the important universal functions and mathematical operations in NumPy.

Simple Arithmetic

You could use arithmetic operators `+ - * /` directly between NumPy arrays, but this section discusses an extension of the same where we have functions that can take any array-like objects e.g. lists, tuples etc. and perform arithmetic.

Addition

The `add()` function sums the content of two arrays, and return the results in a new array.

Example: Add the values in arr1 to the values in arr2.

```
import numpy as np

arr1 = np.array([10, 11, 12, 13, 14, 15])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.add(arr1, arr2)
print(newarr)
```

```
Output:
[30 32 34 36 38 40]
```

The example above will return [30 32 34 36 38 40] which is the sums of 10+20, 11+21, 12+22 etc.

Subtraction

The `subtract()` function subtracts the values from one array with the values from another array, and return the results in a new array.

Example: Subtract the values in arr2 from the values in arr1:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.subtract(arr1, arr2)
print(newarr)
```

```
Output:
[-10  -1   8  17  26  35]
```

The example above will return [-10 -1 8 17 26 35] which is the result of 10-20, 20-21, 30-22 etc.

Multiplication

The `multiply()` function multiplies the values from one array with the values from another array, and return the results in a new array.

Example: Multiply the values in arr1 with the values in arr2:

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([20, 21, 22, 23, 24, 25])
newarr = np.multiply(arr1, arr2)
print(newarr)
```

```
Output:
[ 200  420  660  920 1200 1500]
```

The example above will return [200 420 660 920 1200 1500] which is the result of 10*20, 20*21, 30*22 etc.

Division

The `divide()` function divides the values from one array with the values from another array, and return the results in a new array.

Example: Divide the values in arr1 with the values in arr2.

```
import numpy as np

arr1 = np.array([10, 20, 30, 40, 50, 60])
arr2 = np.array([3, 5, 10, 8, 2, 33])
newarr = np.divide(arr1, arr2)

print(newarr)
```

```
Output:
[ 3.33333333 4. 3. 5. 25. 1.81818182]
```

The example above will return [3.33333333 4. 3. 5. 25. 1.81818182] which is the result of 10/3, 20/5, 30/10 etc.

Numpy Linear Algebra

NumPy package contains numpy.linalg module that provides all the functionality required for linear algebra. Some of the important functions in this module are described as ff.

- **dot** - Dot product of two arrays.
- **vdot** - Dot product of two vectors.
- **inner** - Inner product of two arrays.
- **outer** - Outer product of two arrays.
- **matmul** - Matrix product of two arrays.
- **linalg.det** - Computes the determinant of the array.
- **linalg.solve** - Solves the linear matrix equation.
- **linalg.inv** - Find the multiplicative inverse of the matrix

Numpy Statistical Functions

In Numpy, we can perform various statistical calculations using the various functions that are provided in the library like Order statistics, Averages and variances, correlating, Histograms.

The functions listed here are in numpy module usually used with **np** alias prefix, e.g. **np.average,** stated otherwise.

Order Statistics Functions

- **ptp** - Range of values (maximum - minimum) along an axis.
- **percentile** - Compute the q-th percentile of the data along the specified axis.
- **nanpercentile** - Compute the qth percentile of the data along the specified axis, while ignoring nan values.
- **quantile** - Compute the q-th quantile of the data along the specified axis.
- **nanquantile** - Compute the qth quantile of the data along the specified axis, while ignoring nan values.

Averages and Variances

- **median** Compute the median along the specified axis.
- **average** - Compute the weighted average along the specified axis.
- **mean** - Compute the arithmetic mean along the specified axis.
- **std** - Compute the standard deviation along the specified axis.
- **var** - Compute the variance along the specified axis.
- **nanmedian** - Compute the median along the specified axis, while ignoring NaNs.
- **nanmean** - Compute the arithmetic mean along the specified axis, ignoring NaNs.
- **nanstd** - Compute the standard deviation along the specified axis, while ignoring NaNs.
- **nanvar** - Compute the variance along the specified axis, while ignoring NaNs.

Note: These functions are in numpy module usually used with **np** alias prefix, e.g. **np.median**

Numpy Data Distribution

Data Distribution is a list of all possible values, and how often each value occurs.

Such lists are important when working with statistics and data science.

The random module offer methods that returns randomly generated data distributions.

Normal Distribution

The Normal Distribution is one of the most important distributions.

It is also called the Gaussian Distribution after the German mathematician Carl Friedrich Gauss.

Use the `random.normal()` method to get a Normal Data Distribution.

It has three parameters:

`loc` - (Mean) where the peak of the bell exists.

`scale` - (Standard Deviation) how flat the graph distribution should be.

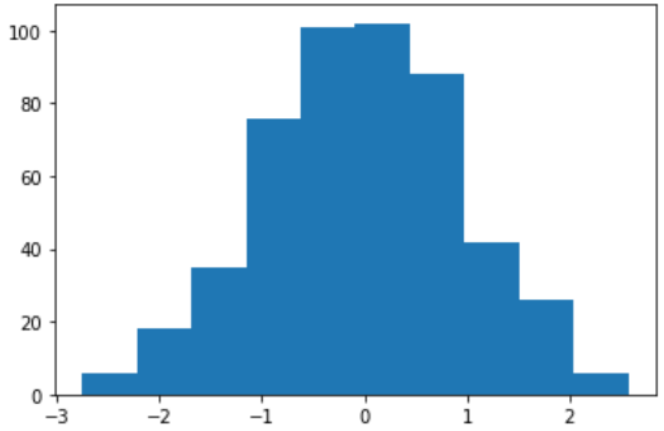
`size` - The shape of the returned array.

Normal Distribution

Example: Generate a random normal distribution of size 2x3:

```
from numpy import random
import matplotlib.pyplot as plt
import seaborn as sns

x = random.normal(size = (500))
plt.hist(x)
plt.show()
```

Output:



Other Notable Distribution Functions

Continuous distribution functions from NumPy **random** module

- **random.pareto** - A distribution following Pareto's law i.e. 80-20 distribution (20% factors cause 80% outcome).
- **random.uniform** - Used to describe probability where every event has equal chances of occuring.
- **random.logistic** - Logistic Distribution is used to describe growth.
- **random.exponential** - Exponential distribution is used for describing time till next event e.g. failure/success etc.
- **random.chisquare** - Chi Square distribution is used as a basis to verify the hypothesis.
- **random.rayleigh** - Rayleigh distribution is used in signal processing.

Other Notable Distribution Functions

Discrete Distribution functions from NumPy **random** module

- **random.binomial** - It describes the outcome of binary scenarios, e.g. toss of a coin, it will either be head or tails.
- **random.multinomial** - It describes outcomes of multinomial scenarios unlike binomial where scenarios must be only one of two. e.g. Blood type of a population, dice roll outcome.
- **random.poisson** - It estimates how many times an event can happen in a specified time. e.g. If someone eats twice a day what is probability he will eat thrice?
- **random.zipf** - Zipf distributions are used to sample data based on zipf's law.