# NumPy Introduction

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.

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 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++.

The source code for NumPy is located at this github repository https://github.com/numpy/numpy

**github:** enables many people to work on the same codebase.

### Import NumPy

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

```
import numpy
```

## Example

```
1.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.

```
import numpy as np
```

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

## Example

```
import numpy as np

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

print(arr)
```

### Checking NumPy Version

The version string is stored under __version__ attribute.

```
import numpy as np

print(np.__version__)
```

### Create a NumPy ndarray Object

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

We can create a NumPy ndarray object by using the array() function.

### Example

```
import numpy as np

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

print(arr)

print(type(arr))
```

## Dimensions in Arrays

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

### Example

Create a 0-D array with value 42

```
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.

## Example

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

```
import numpy as np

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

print(arr)
```

### 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.

## Example

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

```
import numpy as np

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

print(arr)
```

## 3-D arrays

An array that has 2-D arrays (matrices) as its elements is called 3-D array.

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

## Example

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

```
import numpy as np

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

print(arr)
```

## Check Number of Dimensions?

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

## Example

Check how many dimensions the arrays have:

```
import numpy as np

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

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

## Higher Dimensional 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

Create an array with 5 dimensions and verify that it has 5 dimensions:

```
import numpy as np
```

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

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

## NumPy Array Indexing

### Access Array Elements

Array indexing is the same as accessing an array element.

You can access an array element by referring to its index number.

The indexes in NumPy arrays start with 0, meaning that the first element has index 0, and the second has index 1 etc.

## Example

Get the first element from the following array:

```
import numpy as np

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

print(arr[0])
```

## Example

Get the second element from the following array.

```
import numpy as np

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

print(arr[1])
```

## Example

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

```
import numpy as np

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

```
print(arr[2] + arr[3])
```

**Access 2-D Arrays**

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

Think of 2-D arrays like a table with rows and columns, where the dimension represents the row and the index represents the column.

## Example

Access the element on the first row, second column:

```
import numpy as np

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

print('2nd element on 1st row: ', arr[0, 1])
```

```
import numpy as np

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

print('2nd element on 1st row: ', arr[0, 1])
```

Output- 2nd element on 1st row:2

## Example

Access the element on the 2nd row, 5th column:

```
import numpy as np

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

print('5th element on 2nd row: ', arr[1, 4])
```

Output

5th element on 2nd row:10

## NumPy Array Slicing

### Slicing arrays

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

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

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

If we don't pass start its considered 0

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

If we don't pass step its considered 1

## Example

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

```
import numpy as np

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

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

```
Output:[2,3,4,5]
```

## Example

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

```
import numpy as np

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

print(arr[4:])
```

```
Output:[5,6,7]
```

## Example

Slice elements from the 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]
```

### 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 2-D 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]

### Example

From both elements, return index 2:

```
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]

### Example

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

```
import numpy as np

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

print(arr[0:2, 1:4])
```

Output-

```
[[2,3,4]

,[7,8,9]]
```

## Data Types in Python

By default Python have these data types:

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

## Data Types in NumPy

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

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

- i - integer
- b - boolean
- u - unsigned integer
- f - float
- c - complex float
- m - timedelta
- M - datetime
- O - object
- S - string

## Checking the Data Type of an Array

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

- U - unicode string
- V - fixed chunk of memory for other type ( void )

### Example

Get the data type of an array object:

```
import numpy as np

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

print(arr.dtype)
```

Output-int64

### 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

## 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)
```

## NumPy Array Copy vs View

### 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:

## 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)
```

## VIEW:

## 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()
arr[0] = 42

print(arr)
print(x)
```

# Pandas

Pandas is a Python library.

Pandas is used to analyze data.

In our ‘Try it Yourself‘ editor, you can use the Pandas module, and modify the code to see the result.

## Example

Load a CSV file into a Pandas DataFrame:

```
import pandas as pd

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

print(df.to_string())
```

## What is Pandas?

Pandas is a Python library used for working with data sets.

It has functions for analyzing, cleaning, exploring, and manipulating data.

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

## Why Use Pandas ?

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

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

Relevant data is very important in data science.

## Import Pandas

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

```
import pandas
```

### Example

```
import pandas

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

myvar = pandas.DataFrame(mydataset)

print(myvar)
```

## Pandas as pd

Pandas is usually imported under the `pd` alias.

### Example

```
import pandas as pd

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

myvar = pd.DataFrame(mydataset)

print(myvar)
```

## Checking Pandas Version

The version string is stored under __version__ attribute.

## Example

```
import pandas as pd

print(pd.__version__)
```

## Pandas Series

## What is a Series?

A Pandas Series is like a column in a table.

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

## Example

Create a simple Pandas Series from a list:

```
import pandas as pd

a = [1, 7, 2]

myvar = pd.Series(a)

print(myvar)
```

# Labels

If nothing else is specified, the values are labeled with their index number. First value has index 0, second value has index 1 etc.

This label can be used to access a specified value.

## Example

Return the first value of the Series:

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

### Create Labels

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

## Example

Create your own labels:

```
import pandas as pd

a = [1, 7, 2]

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

print(myvar)
```

When you have created labels, you can access an item by referring to the label.

## Example

Return the value of ‘y‘:

```
print(myvar[‘y‘])
```

### Key/Value Objects as Series

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

## Example

Create a simple Pandas Series from a dictionary:

```
import pandas as pd

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

myvar = pd.Series(calories)

print(myvar)
```

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

## Example

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

```
import pandas as pd

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

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

print(myvar)
```

## DataFrames

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

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

## Example

Create a DataFrame from two Series:

```
import pandas as pd

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

myvar = pd.DataFrame(data)
```

```
print(myvar)
```

## What is a DataFrame?

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

### Example

Create a simple Pandas DataFrame:

```
import pandas as pd

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

#load data into a DataFrame object:
df = pd.DataFrame(data)

print(df)
```

## Locate Row

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

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

### Example

Return row 0:

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

## Load Files Into a DataFrame

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

### Example

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

```
import pandas as pd

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

print(df)
```

## Read CSV Files

A simple way to store big data sets is to use CSV files (comma separated files).

CSV files contains plain text and is a well know format that can be read by everyone including Pandas.

In our examples we will be using a CSV file called 'data.csv'.

Download data.csv. or Open data.csv

### Example

Load the CSV into a DataFrame:

```
import pandas as pd

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

print(df.to_string())
```

If you have a large DataFrame with many rows, Pandas will only return the first 5 rows, and the last 5 rows:

### Example

Print the DataFrame without the `to_string()` method:

```
import pandas as pd
```

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

print(df)
```

### max_rows

The number of rows returned is defined in Pandas option settings.

You can check your system's maximum rows with the pd.options.display.max_rows statement.

## Example

Check the number of maximum returned rows:

```
import pandas as pd

print(pd.options.display.max_rows)
```

In my system the number is 60, which means that if the DataFrame contains more than 60 rows, the print(df) statement will return only the headers and the first and last 5 rows.

You can change the maximum rows number with the same statement.

## Example

Increase the maximum number of rows to display the entire DataFrame:

```
import pandas as pd

pd.options.display.max_rows = 9999

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

print(df)
```

### Pandas Read JSON

# Read JSON

Big data sets are often stored, or extracted as JSON.

JSON is plain text, but has the format of an object, and is well known in the world of programming, including Pandas.

In our examples we will be using a JSON file called 'data.json'.

Open data.json.

## Example

Load the JSON file into a DataFrame:

```
import pandas as pd

df = pd.read_json('data.json')

print(df.to_string())
```

## Pandas - Analyzing DataFrames

# Viewing the Data

One of the most used method for getting a quick overview of the DataFrame, is the `head()` method.

The `head()` method returns the headers and a specified number of rows, starting from the top.

## Example

Get a quick overview by printing the first 10 rows of the DataFrame:

```
import pandas as pd

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

print(df.head(10))
```

There is also a tail() method for viewing the last rows of the DataFrame.

The tail() method returns the headers and a specified number of rows, starting from the bottom.

## Example

Print the last 5 rows of the DataFrame:

```
print(df.tail())
```

## Pandas - Cleaning Data

Data cleaning means fixing bad data in your data set.

Bad data could be:

- Empty cells
- Data in wrong format
- Wrong data
- Duplicates

# Empty Cells

Empty cells can potentially give you a wrong result when you analyze data.

### Remove Rows

One way to deal with empty cells is to remove rows that contain empty cells.

This is usually OK, since data sets can be very big, and removing a few rows will not have a big impact on the result.

## Example

Return a new Data Frame with no empty cells:

```
import pandas as pd

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

new_df = df.dropna( )

print(new_df.to_string( ))

If you want to change the original DataFrame, use the inplace = True argument:

## Example

Remove all rows with NULL values:

import pandas as pd

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

df.dropna(inplace = True)

print(df.to_string( ))

## Replace Empty Values

Another way of dealing with empty cells is to insert a new value instead.

This way you do not have to delete entire rows just because of some empty cells.

The fillna( ) method allows us to replace empty cells with a value:

## Example

Replace NULL values with the number 130:

import pandas as pd

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

df.fillna(130, inplace = True)