Contents

[NumPy 5](#_Toc95384106)

[What is NumPy? 5](#_Toc95384107)

[Why Use NumPy? 5](#_Toc95384108)

[Why is NumPy Faster Than Lists? 5](#_Toc95384109)

[Which Language is NumPy written in? 5](#_Toc95384110)

[Where is the NumPy Codebase? 5](#_Toc95384111)

[Installation of NumPy 6](#_Toc95384112)

[Import NumPy 6](#_Toc95384113)

[NumPy as np 6](#_Toc95384114)

[Checking NumPy Version 6](#_Toc95384115)

[NumPy Creating Arrays 7](#_Toc95384116)

[Create a NumPy ndarray Object 7](#_Toc95384117)

[Dimensions in Arrays 7](#_Toc95384118)

[0-D Arrays 7](#_Toc95384119)

[1-D Arrays 7](#_Toc95384120)

[2-D Arrays 8](#_Toc95384121)

[3-D arrays 8](#_Toc95384122)

[Check Number of Dimensions? 8](#_Toc95384123)

[Higher Dimensional Arrays 8](#_Toc95384124)

[NumPy Array Indexing 10](#_Toc95384125)

[Access Array Elements 10](#_Toc95384126)

[Access 2-D Arrays 10](#_Toc95384127)

[Access 3-D Arrays 10](#_Toc95384128)

[Negative Indexing 11](#_Toc95384129)

[NumPy Array Slicing 12](#_Toc95384130)

[Slicing arrays 12](#_Toc95384131)

[Negative Slicing 12](#_Toc95384132)

[STEP 12](#_Toc95384133)

[Slicing 2-D Arrays 13](#_Toc95384134)

[NumPy Copy vs View 14](#_Toc95384135)

[COPY 14](#_Toc95384136)

[VIEW 14](#_Toc95384137)

[Make Changes in the VIEW: 14](#_Toc95384138)

[Check if Array Owns it's Data 15](#_Toc95384139)

[NumPy Array Iterating 16](#_Toc95384140)

[Iterating 2-D Arrays 16](#_Toc95384141)

[Iterating 3-D Arrays 16](#_Toc95384142)

[Iterating Arrays Using nditer() 17](#_Toc95384143)

[Iterating on Each Scalar Element 17](#_Toc95384144)

[Iterating Array With Different Data Types 17](#_Toc95384145)

[Iterating With Different Step Size 17](#_Toc95384146)

[Enumerated Iteration Using ndenumerate() 18](#_Toc95384147)

[NumPy Joining Arrays 19](#_Toc95384148)

[Joining Arrays Using Stack Functions 19](#_Toc95384149)

[Stacking Along Rows 19](#_Toc95384150)

[Stacking Along Columns 20](#_Toc95384151)

[Stacking Along Height (depth) 20](#_Toc95384152)

[NumPy Splitting Array 21](#_Toc95384153)

[Split Into Arrays 21](#_Toc95384154)

[Splitting 2-D Arrays 21](#_Toc95384155)

[NumPy Array Search 23](#_Toc95384156)

[Search Sorted 23](#_Toc95384157)

[Search From the Right Side 24](#_Toc95384158)

[Multiple Values 24](#_Toc95384159)

[NumPy Array Sort 25](#_Toc95384160)

[Sorting a 2-D Array 25](#_Toc95384161)

[NumPy Array Filter 26](#_Toc95384162)

[Creating the Filter Array 26](#_Toc95384163)

[Creating Filter Directly from Array 27](#_Toc95384164)

[NumPy Matplotlib 28](#_Toc95384165)

[pyplot() 28](#_Toc95384166)

[Formatting Characters 29](#_Toc95384167)

[Color Abbreviations 30](#_Toc95384168)

[bar() 31](#_Toc95384169)

[Pandas 32](#_Toc95384170)

[Why Use Pandas? 32](#_Toc95384171)

[What Can Pandas Do? 32](#_Toc95384172)

[Where is the Pandas Codebase? 32](#_Toc95384173)

[Installation of Pandas 32](#_Toc95384174)

[Import Pandas 32](#_Toc95384175)

[Pandas as pd 33](#_Toc95384176)

[Checking Pandas Version 33](#_Toc95384177)

[Pandas Series 34](#_Toc95384178)

[Labels 34](#_Toc95384179)

[Create Labels 34](#_Toc95384180)

[Key/Value Objects as Series 34](#_Toc95384181)

[DataFrames 35](#_Toc95384182)

[Pandas DataFrames 36](#_Toc95384183)

[Locate Row 36](#_Toc95384184)

[Named Indexes 36](#_Toc95384185)

[Locate Named Indexes 37](#_Toc95384186)

[Load Files Into a DataFrame 37](#_Toc95384187)

[Pandas Read CSV 38](#_Toc95384188)

[max\_rows 38](#_Toc95384189)

[Pandas Read JSON 39](#_Toc95384190)

[Dictionary as JSON 39](#_Toc95384191)

[Pandas Analyzing DataFrames 41](#_Toc95384192)

[Viewing the Data 41](#_Toc95384193)

[Info About the Data 41](#_Toc95384194)

[Null Values 42](#_Toc95384195)

[Pandas Cleaning Data 43](#_Toc95384196)

[Our Data Set 43](#_Toc95384197)

[Cleaning Empty Cells 44](#_Toc95384198)

[Empty Cells 44](#_Toc95384199)

[Remove Rows 44](#_Toc95384200)

[Replace Empty Values 44](#_Toc95384201)

[Replace Only For Specified Columns 44](#_Toc95384202)

[Replace Using Mean, Median, or Mode 45](#_Toc95384203)

[Pandas Removing Duplicates 45](#_Toc95384204)

[Discovering Duplicates 45](#_Toc95384205)

[Removing Duplicates 46](#_Toc95384206)

[Pandas Correlations 47](#_Toc95384207)

[Finding Relationships 47](#_Toc95384208)

[Perfect Correlation: 47](#_Toc95384209)

[Good Correlation: 48](#_Toc95384210)

[Bad Correlation: 48](#_Toc95384211)

[Pandas Plotting 49](#_Toc95384212)

[Scatter Plot 49](#_Toc95384213)

[Histogram 51](#_Toc95384214)

# NumPy

* NumPy is a Python library.
* NumPy is used for working with arrays.
* NumPy is short for "Numerical Python".

**Create a NumPy array:**

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

## 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?

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.

Data Science: is a branch of computer science where we study how to store, use and analyze data for deriving information from it.

## 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.

## Which Language is NumPy written in?

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++.

## Where is the NumPy Codebase?

The source code for NumPy is located at this GitHub repository [**https://github.com/numpy/numpy**](https://github.com/numpy/numpy)

# 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.

## 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.

**Example**

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

print(arr)

## NumPy as np

NumPy is usually imported under the np alias.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

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.

**Example**

import numpy as np  
print(np.\_\_version\_\_)

# NumPy Creating Arrays

## 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))

type(): This built-in Python function tells us the type of the object passed to it. Like in above code it shows that arr is numpy.ndarray type.

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

**Example:** Use a tuple to create a NumPy array:

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

## Dimensions in Arrays

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

nested array: are arrays that have arrays as their elements.

## 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.

NumPy has a whole sub module dedicated towards matrix operations called numpy.mat

**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)

In this array the innermost dimension (5th dim) has 4 elements, the 4th dim has 1 element that is the vector, the 3rd dim has 1 element that is the matrix with the vector, the 2nd dim has 1 element that is 3D array and 1st dim has 1 element that is a 4D array.

# 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 row represents the dimension 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])

**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])

## Access 3-D Arrays

To access elements from 3-D arrays we can use comma separated integers representing the dimensions and the index of the element.

**Example:** Access the third element of the second array of the first array:

import numpy as np  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  
print(arr[0, 1, 2])

**Example Explained**

arr[0, 1, 2] prints the value 6.

And this is why:

The first number represents the first dimension, which contains two arrays:  
[[1, 2, 3], [4, 5, 6]]  
and:  
[[7, 8, 9], [10, 11, 12]]  
Since we selected 0, we are left with the first array:  
[[1, 2, 3], [4, 5, 6]]

The second number represents the second dimension, which also contains two arrays:  
[1, 2, 3]  
and:  
[4, 5, 6]  
Since we selected 1, we are left with the second array:  
[4, 5, 6]

The third number represents the third dimension, which contains three values:  
4  
5  
6  
Since we selected 2, we end up with the third value:  
6

## Negative Indexing

Use negative indexing to access an array from the end.

**Example:** Print the last element from the 2nd dim:

import numpy as np  
arr = np.array([[1,2,3,4,5], [6,7,8,9,10]])  
print('Last element from 2nd dim: ', arr[1, -1])

# 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])

Note: The result includes the start index, but excludes the end index.

**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:])

**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])

## 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])

## 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])

**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])

## 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])

Note: Remember that second element has index 1.

**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])

**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])

# NumPy Copy vs 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)

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

## 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)

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

## Make Changes in the VIEW:

**Example:** Make a view, change the view, 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)

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

## Check if Array Owns it's Data

As mentioned above, copies owns the data, and views does not own the data, but how can we check this?

Every NumPy array has the attribute base that returns None if the array owns the data.

Otherwise, the base  attribute refers to the original object.

**Example:** Print the value of the base attribute to check if an array owns it's data or not:

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

x = arr.copy()

y = arr.view()

print(x.base)  
print(y.base)

The copy returns None.

The view returns the original array.

# NumPy Array Iterating

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 1-D array it will go through each element one by one.

**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)

## Iterating 2-D 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)

If we iterate on a n-D array it will go through n-1th dimension one by one.

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)

## Iterating 3-D Arrays

In a 3-D array it will go through all the 2-D arrays.

**Example:** Iterate on the elements of the following 3-D array:

import numpy as np  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  
for x in arr:  
  print(x)

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

**Example:** Iterate down to the scalars:

import numpy as np  
arr = np.array([[[1, 2, 3], [4, 5, 6]], [[7, 8, 9], [10, 11, 12]]])  
for x in arr:  
  for y in x:  
    for z in y:  
      print(z)

## 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.

### 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 3-D array:

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

## 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'].

**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)

## Iterating With Different Step Size

We can use filtering and followed by iteration.

**Example:** Iterate through every scalar element of the 2D array skipping 1 element:

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

## 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 usecases.

**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)

**Example:** Enumerate on following 2D array's 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)

# NumPy Joining 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.

**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)

**Example:** Join two 2-D 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)

## 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.

**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)

## 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)

## 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)

## Stacking Along Height (depth)

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

**Example**

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

# NumPy Splitting Array

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.

**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)

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

If the array has less elements than required, it will adjust from the end accordingly.

**Example:** Split the array in 4 parts:

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

Note: We also have the method split() available but it will not adjust the elements when elements are less in source array for splitting like in example above, array\_split() worked properly but split() would fail.

## Split Into Arrays

The return value of the array\_split() method is an array containing each of the split as an array. If you split an array into 3 arrays, you can access them from the result just like any array element:

**Example:** Access the splitted arrays:

import numpy as np  
arr = np.array([1, 2, 3, 4, 5, 6])  
newarr = np.array\_split(arr, 3)  
print(newarr[0])  
print(newarr[1])  
print(newarr[2])

## Splitting 2-D Arrays

Use the same syntax when splitting 2-D arrays. Use the array\_split() method, pass in the array you want to split and the number of splits you want to do.

**Example:** Split the 2-D array into three 2-D 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)

The example above returns three 2-D arrays. Let's look at another example, this time each element in the 2-D arrays contains 3 elements.

**Example:** Split the 2-D array into three 2-D arrays.

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]])

newarr = np.array\_split(arr, 3)

print(newarr)

The example above returns three 2-D arrays. In addition, you can specify which axis you want to do the split around. The example below also returns three 2-D arrays, but they are split along the row (axis=1).

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

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]])

newarr = np.array\_split(arr, 3, axis=1)

print(newarr)

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

**Example:** Use the hsplit() method to split the 2-D array into three 2-D arrays along rows.

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]])

newarr = np.hsplit(arr, 3)

print(newarr)

# NumPy Array Search

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)

The example above will return a tuple: (array([3, 5, 6],)

Which means that the value 4 is present at index 3, 5, and 6.

**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)

**Example:** Find the indexes where the values are odd:

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

## 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.

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

**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)

Example explained: The number 7 should be inserted on index 1 to remain the sort order.

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

### 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)

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)

# NumPy Array Sort

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:** Sort the array:

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

Note: This method returns a copy of the array, leaving the original array unchanged.

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))

**Example:** Sort a boolean array:

import numpy as np  
arr = np.array([True, False, True])  
print(np.sort(arr))

## Sorting a 2-D Array

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

**Example:** Sort a 2-D array:

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

# NumPy Array Filter

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.

**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)

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

In the example above we hard-coded the True and False values, but the common use is to create a filter array based on conditions.

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

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 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)

**Example:** Create a filter array that will return only even elements from the original array:

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

# Create an empty list

filter\_arr = []

# go through each element in arr

for element in arr:  
  #if element is completely divisble by 2, set the value to True, else False  
  if element % 2 == 0:

    filter\_arr.append(True)  
  else:  
    filter\_arr.append(False)

newarr = arr[filter\_arr]

print(filter\_arr)  
print(newarr)

## Creating Filter Directly from Array

The above 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.

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

import numpy as np  
arr = np.array([41, 42, 43, 44])

filter\_arr = arr > 42

newarr = arr[filter\_arr]

print(filter\_arr)  
print(newarr)

**Example:** Create a filter array that will return only even elements from the original array:

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)

# NumPy Matplotlib

Matplotlib is a plotting library for Python. It is used along with NumPy to provide an environment that is an effective open source alternative for MatLab. It can also be used with graphics toolkits like PyQt and wxPython.

Matplotlib module was first written by John D. Hunter. Since 2012, Michael Droettboom is the principal developer. Currently, Matplotlib ver. 1.5.1 is the stable version available. The package is available in binary distribution as well as in the source code form on [**www.matplotlib.org**](http://www.matplotlib.org/).

Conventionally, the package is imported into the Python script by adding the following statement −

from matplotlib import pyplot as plt

Here **pyplot()** is the most important function in matplotlib library, which is used to plot 2D data.

## pyplot()

The following script plots the equation **y = 2x + 5**

**Example**

import numpy as np

from matplotlib import pyplot as plt

x = np.arange(1,11)

y = 2 \* x + 5

plt.title("Matplotlib demo")

plt.xlabel("x axis caption")

plt.ylabel("y axis caption")

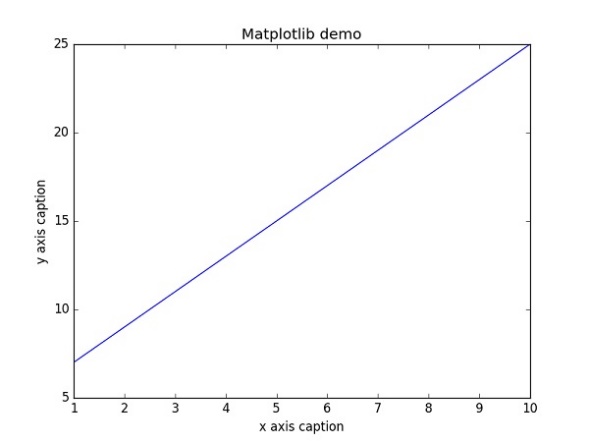
plt.plot(x,y)

plt.show()

An ndarray object x is created from **np.arange() function** as the values on the **x axis**. The corresponding values on the **y axis** are stored in another **ndarray object y**. These values are plotted using **plot()** function of pyplot submodule of matplotlib package.

The graphical representation is displayed by **show()** function.

The above code should produce the following output −



## Formatting Characters

Instead of the linear graph, the values can be displayed discretely by adding a format string to the **plot()** function. Following formatting characters can be used.

|  |  |
| --- | --- |
| **Sr.No.** | **Character & Description** |
| 1 | '-' Solid line style |
| 2 | '--' Dashed line style |
| 3 | '-.' Dash-dot line style |
| 4 | ':' Dotted line style |
| 5 | '.' Point marker |
| 6 | ',' Pixel marker |
| 7 | 'o' Circle marker |
| 8 | 'v' Triangle\_down marker |
| 9 | '^' Triangle\_up marker |
| 10 | '<' Triangle\_left marker |
| 11 | '>' Triangle\_right marker |
| 12 | '1' Tri\_down marker |
| 13 | '2' Tri\_up marker |
| 14 | '3' Tri\_left marker |
| 15 | '4' Tri\_right marker |
| 16 | 's' Square marker |
| 17 | 'p' Pentagon marker |
| 18 | '\*' Star marker |
| 19 | 'h' Hexagon1 marker |
| 20 | 'H' Hexagon2 marker |
| 21 | '+' Plus marker |
| 22 | 'x' X marker |
| 23 | 'D' Diamond marker |
| 24 | 'd' Thin\_diamond marker |
| 25 | '|' Vline marker |
| 26 | '\_' Hline marker |

## Color Abbreviations

The following color abbreviations are also defined.

|  |  |
| --- | --- |
| **Character** | **Color** |
| 'b' | Blue |
| 'g' | Green |
| 'r' | Red |
| 'c' | Cyan |
| 'm' | Magenta |
| 'y' | Yellow |
| 'k' | Black |
| 'w' | White |

To display the circles representing points, instead of the line in the above example, use **“ob”** as the format string in plot() function.

**Example**

import numpy as np

from matplotlib import pyplot as plt

x = np.arange(1,11)

y = 2 \* x + 5

plt.title("Matplotlib demo")

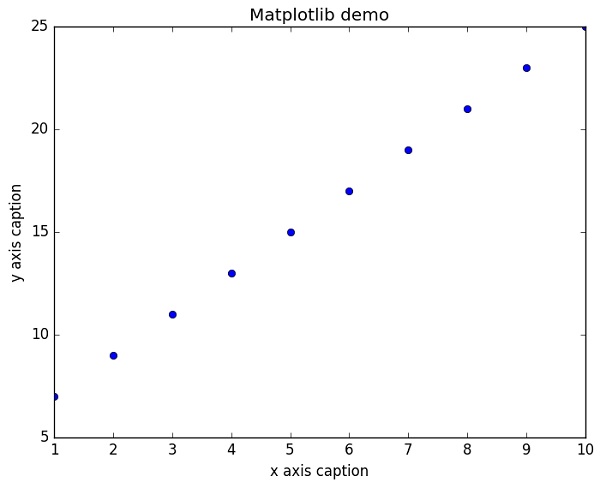
plt.xlabel("x axis caption")

plt.ylabel("y axis caption")

plt.plot(x,y,"ob")

plt.show()

The above code should produce the following output −



## bar()

The **pyplot submodule** provides **bar()** function to generate bar graphs. The following example produces the bar graph of two sets of **x** and **y** arrays.

**Example**

from matplotlib import pyplot as plt

x = [5,8,10]

y = [12,16,6]

x2 = [6,9,11]

y2 = [6,15,7]

plt.bar(x, y, align = 'center')

plt.bar(x2, y2, color = 'g', align = 'center')

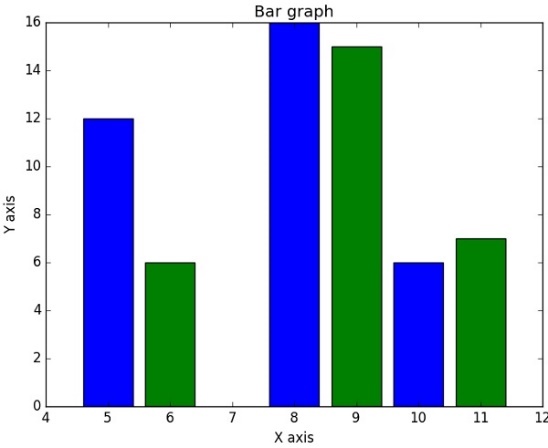
plt.title('Bar graph')

plt.ylabel('Y axis')

plt.xlabel('X axis')

plt.show()

This code should produce the following output −



# Pandas

Pandas is a Python library.

Pandas is used to analyze data.

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.

## What Can Pandas Do?

Pandas gives you answers about the data. Like:

* Is there a correlation between two or more columns?
* What is average value?
* Max value?
* Min value?

Pandas are also able to delete rows that are not relevant, or contains wrong values, like empty or NULL values. This is called cleaning the data.

## Where is the Pandas Codebase?

The source code for Pandas is located at this github repository [**https://github.com/pandas-dev/pandas**](https://github.com/pandas-dev/pandas)

## Installation of Pandas

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

Install it using this command:

C:\Users\Your Name>pip install pandas

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

## Import Pandas

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

import pandas

Now Pandas is imported and ready to use.

**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.

alias: In Python alias are an alternate name for referring to the same thing.

Create an alias with the as keyword while importing:

import pandas as pd

Now the Pandas package can be referred to as pd instead of pandas.

**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

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 you 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)

Note: 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.

**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)

# Pandas DataFrames

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)

**Result**

calories duration

0 420 50

1 380 40

2 390 45

## 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])

**Result**

calories 420

duration 50

Name: 0, dtype: int64

Note: This example returns a Pandas Series.

**Example:** Return row 0 and 1:

#use a list of indexes:

print(df.loc[[0, 1]])

**Result**

calories duration

0 420 50

1 380 40

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

## Named Indexes

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

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

import pandas as pd

data = {

  "calories": [420, 380, 390],

  "duration": [50, 40, 45]  
}  
  
df = pd.DataFrame(data, index = ["day1", "day2", "day3"])

print(df)

**Result**

calories duration

day1 420 50

day2 380 40

day3 390 45

## Locate Named Indexes

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

**Example:** Return "day2":

#refer to the named index:

print(df.loc["day2"])

Result

calories 380

duration 40

Name: 0, dtype: int64

## Load Files Into a DataFrame

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

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

import pandas as pd

df = pd.read\_csv('data.csv')

print(df)

# Pandas Read CSV

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' (*File available in the BigDataTutorials\Data\Pandas Data Files folder*).

**Example:** Load the CSV into a DataFrame:

import pandas as pd

df = pd.read\_csv('data.csv')

print(df.to\_string())

Tip: use to\_string() to print the entire DataFrame.

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

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' (*File available in the BigDataTutorials\Data\Pandas Data Files folder*).

**Example:** Load the JSON file into a DataFrame:

import pandas as pd

df = pd.read\_json('data.json')

print(df.to\_string())

Tip: use to\_string() to print the entire DataFrame.

## Dictionary as JSON

JSON = Python Dictionary

JSON objects have the same format as Python dictionaries.

If your JSON code is not in a file, but in a Python Dictionary, you can load it into a DataFrame directly:

**Example:** Load a Python Dictionary into a DataFrame:

import pandas as pd

data = {

  "Duration":{  
    "0":60,  
    "1":60,  
    "2":60,  
    "3":45,  
    "4":45,  
    "5":60  
  },  
  "Pulse":{  
    "0":110,  
    "1":117,  
    "2":103,  
    "3":109,  
    "4":117,  
    "5":102  
  },

  "Maxpulse":{

    "0":130,  
    "1":145,  
    "2":135,  
    "3":175,  
    "4":148,  
    "5":127  
  },  
  "Calories":{  
    "0":409,  
    "1":479,  
    "2":340,  
    "3":282,  
    "4":406,  
    "5":300  
  }

}  
  
df = pd.DataFrame(data)

print(df)

# 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:

In our examples we will be using a CSV file called 'dataframe.csv' (*File available in the BigDataTutorials\Data\Pandas Data Files folder*).

import pandas as pd

df = pd.read\_csv('dataframe.csv')

print(df.head(10))

Note: if the number of rows is not specified, the head() method will return the top 5 rows.

**Example:** Print the first 5 rows of the DataFrame:

import pandas as pd

df = pd.read\_csv('dataframe.csv')

print(df.head())

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())

## Info About the Data

The DataFrames object has a method called info(), that gives you more information about the data set.

**Example:** Print information about the data:

print(df.info())

**Result**

<class 'pandas.core.frame.DataFrame'>

RangeIndex: 169 entries, 0 to 168

Data columns (total 4 columns):

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Duration 169 non-null int64

1 Pulse 169 non-null int64

2 Maxpulse 169 non-null int64

3 Calories 164 non-null float64

dtypes: float64(1), int64(3)

memory usage: 5.4 KB

None

**Result Explained**

The result tells us there are 169 rows and 4 columns:

RangeIndex: 169 entries, 0 to 168

Data columns (total 4 columns):

And the name of each column, with the data type:

# Column Non-Null Count Dtype

--- ------ -------------- -----

0 Duration 169 non-null int64

1 Pulse 169 non-null int64

2 Maxpulse 169 non-null int64

3 Calories 164 non-null float64

## Null Values

The info() method also tells us how many Non-Null values there are present in each column, and in our data set it seems like there are 164 of 169 Non-Null values in the "Calories" column.

Which means that there are 5 rows with no value at all, in the "Calories" column, for whatever reason.

Empty values, or Null values, can be bad when analyzing data, and you should consider removing rows with empty values. This is a step towards what is called cleaning data.

# 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

## Our Data Set

In the next few sections, we will use this data set:

Duration Date Pulse Maxpulse Calories

0 60 '2020/12/01' 110 130 409.1

1 60 '2020/12/02' 117 145 479.0

2 60 '2020/12/03' 103 135 340.0

3 45 '2020/12/04' 109 175 282.4

4 45 '2020/12/05' 117 148 406.0

5 60 '2020/12/06' 102 127 300.0

6 60 '2020/12/07' 110 136 374.0

7 450 '2020/12/08' 104 134 253.3

8 30 '2020/12/09' 109 133 195.1

9 60 '2020/12/10' 98 124 269.0

10 60 '2020/12/11' 103 147 329.3

11 60 '2020/12/12' 100 120 250.7

12 60 '2020/12/12' 100 120 250.7

13 60 '2020/12/13' 106 128 345.3

14 60 '2020/12/14' 104 132 379.3

15 60 '2020/12/15' 98 123 275.0

16 60 '2020/12/16' 98 120 215.2

17 60 '2020/12/17' 100 120 300.0

18 45 '2020/12/18' 90 112 NaN

19 60 '2020/12/19' 103 123 323.0

20 45 '2020/12/20' 97 125 243.0

21 60 '2020/12/21' 108 131 364.2

22 45 NaN 100 119 282.0

23 60 '2020/12/23' 130 101 300.0

24 45 '2020/12/24' 105 132 246.0

25 60 '2020/12/25' 102 126 334.5

26 60 2020/12/26 100 120 250.0

27 60 '2020/12/27' 92 118 241.0

28 60 '2020/12/28' 103 132 NaN

29 60 '2020/12/29' 100 132 280.0

30 60 '2020/12/30' 102 129 380.3

31 60 '2020/12/31' 92 115 243.0

* The data set contains some empty cells ("Date" in row 22, and "Calories" in row 18 and 28).
* The data set contains wrong format ("Date" in row 26).
* The data set contains wrong data ("Duration" in row 7).
* The data set contains duplicates (row 11 and 12).

## Cleaning Empty Cells

### 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.

In our cleaning examples we will be using a CSV file called 'dirtydata.csv' (*File available in the BigDataTutorials\Data\Pandas Data Files folder*).

**Example:** Return a new Data Frame with no empty cells:

import pandas as pd

df = pd.read\_csv('dirtydata.csv')

new\_df = df.dropna()

print(new\_df.to\_string())

Note: By default, the dropna() method returns a new DataFrame, and will not change the original.

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('dirtydata.csv')

df.dropna(inplace = True)

print(df.to\_string())

Note: Now, the dropna(inplace = True) will NOT return a new DataFrame, but it will remove all rows containg NULL values from the original DataFrame.

### 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('dirtydata.csv')

df.fillna(130, inplace = True)

### Replace Only For Specified Columns

The example above replaces all empty cells in the whole Data Frame.

To only replace empty values for one column, specify the column name for the DataFrame:

**Example:** Replace NULL values in the "Calories" columns with the number 130:

import pandas as pd

df = pd.read\_csv('dirtydata.csv')

df["Calories"].fillna(130, inplace = True)

### Replace Using Mean, Median, or Mode

A common way to replace empty cells, is to calculate the mean, median or mode value of the column.

Pandas uses the mean() median() and mode() methods to calculate the respective values for a specified column:

**Example:** Calculate the MEAN, and replace any empty values with it:

import pandas as pd

df = pd.read\_csv('dirtydata.csv')

x = df["Calories"].mean()

df["Calories"].fillna(x, inplace = True)

Mean = the average value (the sum of all values divided by number of values).

**Example:** Calculate the MEDIAN, and replace any empty values with it:

import pandas as pd

df = pd.read\_csv('dirtydata.csv')

x = df["Calories"].median()

df["Calories"].fillna(x, inplace = True)

Median = the value in the middle, after you have sorted all values ascending.

**Example:** Calculate the MODE, and replace any empty values with it:

import pandas as pd

df = pd.read\_csv('dirtydata.csv')

x = df["Calories"].mode()[0]

df["Calories"].fillna(x, inplace = True)

Mode = the value that appears most frequently.

## Pandas Removing Duplicates

### Discovering Duplicates

Duplicate rows are rows that have been registered more than one time.

We will use the same “dirtydata.csv” from the Cleaning Empty Cells section.

By taking a look at our test data set, we can assume that row 11 and 12 are duplicates.

To discover duplicates, we can use the duplicated() method.

The duplicated() method returns a Boolean values for each row:

**Example:** Returns True for every row that is a duplicate, othwerwise False:

print(df.duplicated())

### Removing Duplicates

To remove duplicates, use the drop\_duplicates() method.

**Example:** Remove all duplicates:

df.drop\_duplicates(inplace = True)

Remember: The (inplace = True) will make sure that the method does NOT return a new DataFrame, but it will remove all duplicates from the original DataFrame.

# Pandas Correlations

### Finding Relationships

A great aspect of the Pandas module is the corr() method.

The corr() method calculates the relationship between each column in your data set.

The examples in this page uses a CSV file called: 'data\_corr.csv' (*File available in the BigDataTutorials\Data\Pandas Data Files folder*).

**Example:** Show the relationship between the columns:

df.corr()

**Result**

Duration Pulse Maxpulse Calories

Duration 1.000000 -0.155408 0.009403 0.922721

Pulse -0.155408 1.000000 0.786535 0.025120

Maxpulse 0.009403 0.786535 1.000000 0.203814

Calories 0.922721 0.025120 0.203814 1.000000

Note: The corr() method ignores "not numeric" columns.

**Result Explained**

The Result of the corr() method is a table with a lot of numbers that represents how well the relationship is between two columns.

The number varies from -1 to 1.

1 means that there is a 1 to 1 relationship (a perfect correlation), and for this data set, each time a value went up in the first column, the other one went up as well.

0.9 is also a good relationship, and if you increase one value, the other will probably increase as well.

-0.9 would be just as good relationship as 0.9, but if you increase one value, the other will probably go down.

0.2 means NOT a good relationship, meaning that if one value goes up does not mean that the other will.

What is a good correlation? It depends on the use, but I think it is safe to say you have to have at least 0.6 (or -0.6) to call it a good correlation.

### Perfect Correlation:

We can see that "Duration" and "Duration" got the number 1.000000, which makes sense, each column always has a perfect relationship with itself.

### Good Correlation:

"Duration" and "Calories" got a 0.922721 correlation, which is a very good correlation, and we can predict that the longer you work out, the more calories you burn, and the other way around: if you burned a lot of calories, you probably had a long work out.

### Bad Correlation:

"Duration" and "Maxpulse" got a 0.009403 correlation, which is a very bad correlation, meaning that we can not predict the max pulse by just looking at the duration of the work out, and vice versa.

# Pandas Plotting

Chart, histogram

Description automatically generated

Pandas uses the plot() method to create diagrams.

We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.

**Example:** Import pyplot from Matplotlib and visualize our DataFrame:

The examples in this page uses a CSV file called: ‘data\_pandas\_plot.csv' (*File available in the BigDataTutorials\Data\Pandas Data Files folder*).

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv('data\_pandas\_plot.csv')

df.plot()  
plt.show()

### Scatter Plot

Specify that you want a scatter plot with the kind argument:

kind = 'scatter'

A scatter plot needs an x- and a y-axis.

In the example below we will use "Duration" for the x-axis and "Calories" for the y-axis.

Include the x and y arguments like this:

x = 'Duration', y = 'Calories'

**Example**

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv('data\_pandas\_plot.csv')

df.plot(kind = 'scatter', x = 'Duration', y = 'Calories')

plt.show()

**Result**

Chart, scatter chart

Description automatically generated

Remember: In the previous example, we learned that the correlation between "Duration" and "Calories" was 0.922721, and we conluded with the fact that higher duration means more calories burned.

By looking at the scatterplot, I will agree.

Let's create another scatterplot, where there is a bad relationship between the columns, like "Duration" and "Maxpulse", with the correlation 0.009403:

**Example:** A scatterplot where there are no relationship between the columns:

import pandas as pd

import matplotlib.pyplot as plt

df = pd.read\_csv('data\_pandas\_plot.csv')

df.plot(kind = 'scatter', x = 'Duration', y = 'Maxpulse')

plt.show()

**Result**

Chart, scatter chart

Description automatically generated

### Histogram

Use the kind argument to specify that you want a histogram:

kind = 'hist'

A histogram needs only one column.

A histogram shows us the frequency of each interval, e.g. how many workouts lasted between 50 and 60 minutes?

In the example below we will use the "Duration" column to create the histogram:

**Example**

df["Duration"].plot(kind = 'hist')

**Result**

Chart, histogram

Description automatically generated

Note: The histogram tells us that there were over 100 workouts that lasted between 50 and 60 minutes.