# Chapter 5: Using Important Python Packages

Python Packages

What is meant by package in Python?

A package is **a namespace that organizes a set of related classes and interfaces**. Conceptually you can think of packages as being similar to different folders on your computer.

A Python package usually consists of several modules. Physically, a package is **a folder containing modules and maybe other folders that themselves may contain more folders and modules**.

## Download a Package

Downloading a package is very easy.

Open the command line interface and tell PIP to download the package you want.

Navigate your command line to the location of Python's script directory, and type the following:

Example

Download a package named "camelcase":

C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip install camelcase

Now you have downloaded and installed your first package!

## Using a Package

Once the package is installed, it is ready to use.

Import the "camelcase" package into your project.

Example

Import and use "camelcase":

import camelcase  
  
c = camelcase.CamelCase()  
  
txt = "hello world"  
  
print(c.hump(txt))

## Find Packages

## List Packages

Use the list command to list all the packages installed on your system:

Example

List installed packages:

C:\Users\Your Name\AppData\Local\Programs\Python\Python36-32\Scripts>pip list

# NumPy

NumPy is a Python library.

NumPy is used for working with arrays.

NumPy is short for "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?

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.

## 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++.

## Installation of NumPy

If you have [Python](https://www.w3schools.com/python/default.asp) and [PIP](https://www.w3schools.com/python/python_pip.asp) 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\_\_)

## 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))

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).

## 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

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.

## 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])

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])

Top of Form

Bottom of Form

## 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])

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])

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])

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

## 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

Get the data type of an array object:

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

Example

Get the data type of an array containing strings:

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

## 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)

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)

## What if a Value Can Not Be Converted?

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

Example

A non integer string like 'a' can not be converted to integer (will raise an error):

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

## 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.

Example

Change 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)

Example

Change 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)

Example

Change 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)

Top of Form

Test Yourself th Exercises

## 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)

## 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)

## Check if Array Owns its 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)

## Shape of an Array

The shape of an 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 2-D array:

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

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.

Example

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:

import numpy as np  
  
arr = np.array([1, 2, 3, 4], ndmin=5)  
  
print(arr)  
print('shape of array :', arr.shape)

## What does the shape tuple represent?

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

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

## Reshaping arrays

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 1-D to 2-D

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)

## Reshape From 1-D to 3-D

Example

Convert the following 1-D array with 12 elements into a 3-D 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)

## 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.

Example

Try converting 1D array with 8 elements to a 2D array with 3 elements in each dimension (will raise an error):

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

## 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)

The Example above returns the original array, so it is a view.

## 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.

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)

## Flattening the arrays

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

We can use reshape(-1) to do this.

Example

Convert the array into a 1D array:

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

## 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 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)

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, lets 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)

## 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.

**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)

Top of Form

## Test Yourself With Exercises

## Exercise:

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))

Submit Answer »

[Start the Exercise](https://www.w3schools.com/python/numpy/exercise.asp?filename=exercise_array_join1)

Bottom of Form

## 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.

**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 a list 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)

## 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)

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)

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.

Top of Form

## Test Yourself With Exercises

## Exercise:

Use the correct NumPy method to find all items with the value 4.

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

x = np.(arr == 4)

Bottom of Form

## 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**

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))

Top of Form

## Test Yourself With Exercises

## Exercise:

Use the correct NumPy method to return a **sorted**array.

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

x = np.(arr)

Bottom of Form

## 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.

**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 array 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 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)

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 the element is completely divisble by 2, set the value to True, otherwise 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)

## What are ufuncs?

ufuncs stands for "Universal Functions" and they are NumPy functions that operate on the ndarray object.

## Why use ufuncs?

ufuncs are used to implement vectorization in NumPy which is way faster than iterating over elements.

They also provide broadcasting and additional methods like reduce, accumulate etc. that are very helpful for computation.

ufuncs also take additional arguments, like:

where boolean array or condition defining where the operations should take place.

dtype defining the return type of elements.

out output array where the return value should be copied.

## What is Vectorization?

Converting iterative statements into a vector based operation is called vectorization.

It is faster as modern CPUs are optimized for such operations.

## Add the Elements of Two Lists

list 1: [1, 2, 3, 4]

list 2: [4, 5, 6, 7]

One way of doing it is to iterate over both of the lists and then sum each elements.

**Example**

Without ufunc, we can use Python's built-in zip() method:

x = [1, 2, 3, 4]  
y = [4, 5, 6, 7]  
z = []  
  
for i, j in zip(x, y):  
  z.append(i + j)  
print(z)

NumPy has a ufunc for this, called add(x, y) that will produce the same result.

Example

With ufunc, we can use the add() function:

import numpy as np  
  
x = [1, 2, 3, 4]  
y = [4, 5, 6, 7]  
z = np.add(x, y)  
  
print(z)

## How To Create Your Own ufunc

To create your own ufunc, you have to define a function, like you do with normal functions in Python, then you add it to your NumPy ufunc library with the frompyfunc() method.

The frompyfunc() method takes the following arguments:

1. function - the name of the function.
2. inputs - the number of input arguments (arrays).
3. outputs - the number of output arrays.

**Example**

Create your own ufunc for addition:

import numpy as np  
  
def myadd(x, y):  
  return x+y  
  
myadd = np.frompyfunc(myadd, 2, 1)  
  
print(myadd([1, 2, 3, 4], [5, 6, 7, 8]))

## Check if a Function is a ufunc

Check the type of a function to check if it is a ufunc or not.

A ufunc should return <class 'numpy.ufunc'>.

Example

Check if a function is a ufunc:

import numpy as np  
  
print(type(np.add))

If it is not a ufunc, it will return another type, like this built-in NumPy function for joining two or more arrays:

Example

Check the type of another function: concatenate():

import numpy as np  
  
print(type(np.concatenate))

If the function is not recognized at all, it will return an error:

Example

Check the type of something that does not exist. This will produce an error:

import numpy as np  
  
print(type(np.blahblah))

To test if the function is a ufunc in an if statement, use the numpy.ufunc value (or np.ufunc if you use np as an alias for numpy):

Example

Use an if statement to check if the function is a ufunc or not:

import numpy as np  
  
if type(np.add) == np.ufunc:  
  print('add is ufunc')  
else:  
  print('add is not ufunc')

## 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 conditionally.

**Arithmetic Conditionally:** means that we can define conditions where the arithmetic operation should happen.

All of the discussed arithmetic functions take a where parameter in which we can specify that condition.

## 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)

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)

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)

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)

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.

## Power

The power() function rises the values from the first array to the power of the values of the second array, and return the results in a new array.

Example

Raise the valules in arr1 to the power of values in arr2:

import numpy as np  
  
arr1 = np.array([10, 20, 30, 40, 50, 60])  
arr2 = np.array([3, 5, 6, 8, 2, 33])  
  
newarr = np.power(arr1, arr2)  
  
print(newarr)

The Example above will return [1000 3200000 729000000 6553600000000 2500 0] which is the result of 10\*10\*10, 20\*20\*20\*20\*20, 30\*30\*30\*30\*30\*30 etc.

## Remainder

Both the mod() and the remainder() functions return the remainder of the values in the first array corresponding to the values in the second array, and return the results in a new array.

Example

Return the remainders:

import numpy as np  
  
arr1 = np.array([10, 20, 30, 40, 50, 60])  
arr2 = np.array([3, 7, 9, 8, 2, 33])  
  
newarr = np.mod(arr1, arr2)  
  
print(newarr)

The Example above will return [1 6 3 0 0 27] which is the remainders when you divide 10 with 3 (10%3), 20 with 7 (20%7) 30 with 9 (30%9) etc.

You get the same result when using the remainder() function:

Example

Return the remainders:

import numpy as np  
  
arr1 = np.array([10, 20, 30, 40, 50, 60])  
arr2 = np.array([3, 7, 9, 8, 2, 33])  
  
newarr = np.remainder(arr1, arr2)  
  
print(newarr)

## Quotient and Mod

The divmod() function return both the quotient and the the mod. The return value is two arrays, the first array contains the quotient and second array contains the mod.

Example

Return the quotient and mod:

import numpy as np  
  
arr1 = np.array([10, 20, 30, 40, 50, 60])  
arr2 = np.array([3, 7, 9, 8, 2, 33])  
  
newarr = np.divmod(arr1, arr2)  
  
print(newarr)

The Example above will return:  
(array([3, 2, 3, 5, 25, 1]), array([1, 6, 3, 0, 0, 27]))  
The first array represents the quotients, (the integer value when you divide 10 with 3, 20 with 7, 30 with 9 etc.  
The second array represents the remainders of the same divisions.

## Absolute Values

Both the absolute() and the abs() functions do the same absolute operation element-wise but we should use absolute() to avoid confusion with python's inbuilt math.abs()

Example

Return the quotient and mod:

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

## Rounding Decimals

There are primarily five ways of rounding off decimals in NumPy:

* truncation
* fix
* rounding
* floor
* ceil

## Truncation

Remove the decimals, and return the float number closest to zero. Use the trunc() and fix() functions.

**Example**

Truncate elements of following array:

import numpy as np  
  
arr = np.trunc([-3.1666, 3.6667])  
  
print(arr)

Example

Same Example, using fix():

import numpy as np  
  
arr = np.fix([-3.1666, 3.6667])  
  
print(arr)

## Rounding

The around() function increments preceding digit or decimal by 1 if >=5 else do nothing.

E.g. round off to 1 decimal point, 3.16666 is 3.2

Example

Round off 3.1666 to 2 decimal places:

import numpy as np  
  
arr = np.around(3.1666, 2)  
  
print(arr)

## Floor

The floor() function rounds off decimal to nearest lower integer.

E.g. floor of 3.166 is 3.

Example

Floor the elements of following array:

import numpy as np  
  
arr = np.floor([-3.1666, 3.6667])  
  
print(arr)

## Ceil

The ceil() function rounds off decimal to nearest upper integer.

E.g. ceil of 3.166 is 4.

Example

Ceil the elements of following array:

import numpy as np  
  
arr = np.ceil([-3.1666, 3.6667])  
  
print(arr)

## Logs

NumPy provides functions to perform log at the base 2, e and 10.

We will also explore how we can take log for any base by creating a custom ufunc.

All of the log functions will place -inf or inf in the elements if the log can not be computed.

## Log at Base 2

Use the log2() function to perform log at the base 2.

**Example**

Find log at base 2 of all elements of following array:

import numpy as np  
  
arr = np.arange(1, 10)  
  
print(np.log2(arr))

**Note:** The arange(1, 10) function returns an array with integers starting from 1 (included) to 10 (not included).

## Log at Base 10

Use the log10() function to perform log at the base 10.

Example

Find log at base 10 of all elements of following array:

import numpy as np  
  
arr = np.arange(1, 10)  
  
print(np.log10(arr))

## Natural Log, or Log at Base e

Use the log() function to perform log at the base e.

Example

Find log at base e of all elements of following array:

import numpy as np  
  
arr = np.arange(1, 10)  
  
print(np.log(arr))

## Log at Any Base

NumPy does not provide any function to take log at any base, so we can use the frompyfunc() function along with inbuilt function math.log() with two input parameters and one output parameter:

Example

from math import log  
import numpy as np  
  
nplog = np.frompyfunc(log, 2, 1)  
  
print(nplog(100, 15))

## Summations

What is the difference between summation and addition?

Addition is done between two arguments whereas summation happens over n elements.

**Example**

Add the values in arr1 to the values in arr2:

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
arr2 = np.array([1, 2, 3])  
  
newarr = np.add(arr1, arr2)  
  
print(newarr)

**Returns:** [2 4 6]

Example

Sum the values in arr1 and the values in arr2:

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
arr2 = np.array([1, 2, 3])  
  
newarr = np.sum([arr1, arr2])  
  
print(newarr)

**Returns:** 12

## Summation Over an Axis

If you specify axis=1, NumPy will sum the numbers in each array.

Example

Perform summation in the following array over 1st axis:

import numpy as np  
  
arr1 = np.array([1, 2, 3])  
arr2 = np.array([1, 2, 3])  
  
newarr = np.sum([arr1, arr2], axis=1)  
  
print(newarr)

**Returns:** [6 6]

## Cummulative Sum

Cummulative sum means partially adding the elements in array.

E.g. The partial sum of [1, 2, 3, 4] would be [1, 1+2, 1+2+3, 1+2+3+4] = [1, 3, 6, 10].

Perfom partial sum with the cumsum() function.

Example

Perform cummulative summation in the following array:

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

## Products

To find the product of the elements in an array, use the prod() function.

**Example**

Find the product of the elements of this array:

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

**Returns:** 24 because 1\*2\*3\*4 = 24

Example

Find the product of the elements of two arrays:

import numpy as np  
  
arr1 = np.array([1, 2, 3, 4])  
arr2 = np.array([5, 6, 7, 8])  
  
x = np.prod([arr1, arr2])  
  
print(x)

**Returns:** 40320 because 1\*2\*3\*4\*5\*6\*7\*8 = 40320

## Product Over an Axis

If you specify axis=1, NumPy will return the product of each array.

Example

Perform summation in the following array over 1st axis:

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

**Returns:** [24 1680]

## Cummulative Product

Cummulative product means taking the product partially.

E.g. The partial product of [1, 2, 3, 4] is [1, 1\*2, 1\*2\*3, 1\*2\*3\*4] = [1, 2, 6, 24]

Perfom partial sum with the cumprod() function.

Example

Take cummulative product of all elements for following array:

import numpy as np  
  
arr = np.array([5, 6, 7, 8])  
  
newarr = np.cumprod(arr)  
  
print(newarr)

## Differences

A discrete difference means subtracting two successive elements.

E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]

To find the discrete difference, use the diff() function.

**Example**

Compute discrete difference of the following array:

import numpy as np  
  
arr = np.array([10, 15, 25, 5])  
  
newarr = np.diff(arr)  
  
print(newarr)

**Returns:** [5 10 -20] because 15-10=5, 25-15=10, and 5-25=-20

We can perform this operation repeatedly by giving parameter n.

E.g. for [1, 2, 3, 4], the discrete difference with n = 2 would be [2-1, 3-2, 4-3] = [1, 1, 1] , then, since n=2, we will do it once more, with the new result: [1-1, 1-1] = [0, 0]

Example

Compute discrete difference of the following array twice:

import numpy as np  
  
arr = np.array([10, 15, 25, 5])  
  
newarr = np.diff(arr, n=2)  
  
print(newarr)

## Differences

A discrete difference means subtracting two successive elements.

E.g. for [1, 2, 3, 4], the discrete difference would be [2-1, 3-2, 4-3] = [1, 1, 1]

To find the discrete difference, use the diff() function.

**Example**

Compute discrete difference of the following array:

import numpy as np  
  
arr = np.array([10, 15, 25, 5])  
  
newarr = np.diff(arr)  
  
print(newarr)

**Returns:** [5 10 -20] because 15-10=5, 25-15=10, and 5-25=-20

We can perform this operation repeatedly by giving parameter n.

E.g. for [1, 2, 3, 4], the discrete difference with n = 2 would be [2-1, 3-2, 4-3] = [1, 1, 1] , then, since n=2, we will do it once more, with the new result: [1-1, 1-1] = [0, 0]

Example

Compute discrete difference of the following array twice:

import numpy as np  
  
arr = np.array([10, 15, 25, 5])  
  
newarr = np.diff(arr, n=2)  
  
print(newarr)

## Finding GCD (Greatest Common Denominator)

The GCD (Greatest Common Denominator), also known as HCF (Highest Common Factor) is the biggest number that is a common factor of both of the numbers.

**Example**

Find the HCF of the following two numbers:

import numpy as np  
  
num1 = 6  
num2 = 9  
  
x = np.gcd(num1, num2)  
  
print(x)

**Returns:** 3 because that is the highest number both numbers can be divided by (6/3=2 and 9/3=3).

## Finding GCD in Arrays

To find the Highest Common Factor of all values in an array, you can use the reduce() method.

The reduce() method will use the ufunc, in this case the gcd() function, on each element, and reduce the array by one dimension.

Example

Find the GCD for all of the numbers in the following array:

import numpy as np  
  
arr = np.array([20, 8, 32, 36, 16])  
  
x = np.gcd.reduce(arr)  
  
print(x)

## Trigonometric Functions

NumPy provides the ufuncs sin(), cos() and tan() that take values in radians and produce the corresponding sin, cos and tan values.

**Example**

Find sine value of PI/2:

import numpy as np  
  
x = np.sin(np.pi/2)  
  
print(x)

Example

Find sine values for all of the values in arr:

import numpy as np  
  
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])  
  
x = np.sin(arr)  
  
print(x)

## Convert Degrees Into Radians

By default all of the trigonometric functions take radians as parameters but we can convert radians to degrees and vice versa as well in NumPy.

**Note:** radians values are pi/180 \* degree\_values.

Example

Convert all of the values in following array arr to radians:

import numpy as np  
  
arr = np.array([90, 180, 270, 360])  
  
x = np.deg2rad(arr)  
  
print(x)

## Radians to Degrees

Example

Convert all of the values in following array arr to degrees:

import numpy as np  
  
arr = np.array([np.pi/2, np.pi, 1.5\*np.pi, 2\*np.pi])  
  
x = np.rad2deg(arr)  
  
print(x)

## Finding Angles

Finding angles from values of sine, cos, tan. E.g. sin, cos and tan inverse (arcsin, arccos, arctan).

NumPy provides ufuncs arcsin(), arccos() and arctan() that produce radian values for corresponding sin, cos and tan values given.

Example

Find the angle of 1.0:

import numpy as np  
  
x = np.arcsin(1.0)  
  
print(x)

## Angles of Each Value in Arrays

Example

Find the angle for all of the sine values in the array

import numpy as np  
  
arr = np.array([1, -1, 0.1])  
  
x = np.arcsin(arr)  
  
print(x)

## Hypotenues

Finding hypotenues using pythagoras theorem in NumPy.

NumPy provides the hypot() function that takes the base and perpendicular values and produces hypotenues based on pythagoras theorem.

Example

Find the hypotenues for 4 base and 3 perpendicular:

import numpy as np  
  
base = 3  
perp = 4  
  
x = np.hypot(base, perp)  
  
print(x)

## Hyperbolic Functions

NumPy provides the ufuncs sinh(), cosh() and tanh() that take values in radians and produce the corresponding sinh, cosh and tanh values..

**Example**

Find sinh value of PI/2:

import numpy as np  
  
x = np.sinh(np.pi/2)  
  
print(x)

Example

Find cosh values for all of the values in arr:

import numpy as np  
  
arr = np.array([np.pi/2, np.pi/3, np.pi/4, np.pi/5])  
  
x = np.cosh(arr)  
  
print(x)

## Finding Angles

Finding angles from values of hyperbolic sine, cos, tan. E.g. sinh, cosh and tanh inverse (arcsinh, arccosh, arctanh).

Numpy provides ufuncs arcsinh(), arccosh() and arctanh() that produce radian values for corresponding sinh, cosh and tanh values given.

Example

Find the angle of 1.0:

import numpy as np  
  
x = np.arcsinh(1.0)  
  
print(x)

## Angles of Each Value in Arrays

Example

Find the angle for all of the tanh values in array:

import numpy as np  
  
arr = np.array([0.1, 0.2, 0.5])  
  
x = np.arctanh(arr)  
  
print(x)

## What is a Set

A set in mathematics is a collection of unique elements.

Sets are used for operations involving frequent intersection, union and difference operations.

## Create Sets in NumPy

We can use NumPy's unique() method to find unique elements from any array. E.g. create a set array, but remember that the set arrays should only be 1-D arrays.

**Example**

Convert following array with repeated elements to a set:

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

## Finding Union

To find the unique values of two arrays, use the union1d() method.

Example

Find union of the following two set arrays:

import numpy as np  
  
arr1 = np.array([1, 2, 3, 4])  
arr2 = np.array([3, 4, 5, 6])  
  
newarr = np.union1d(arr1, arr2)  
  
print(newarr)

## Finding Intersection

To find only the values that are present in both arrays, use the intersect1d() method.

Example

Find intersection of the following two set arrays:

import numpy as np  
  
arr1 = np.array([1, 2, 3, 4])  
arr2 = np.array([3, 4, 5, 6])  
  
newarr = np.intersect1d(arr1, arr2, assume\_unique=True)  
  
print(newarr)

## Finding Difference

To find only the values in the first set that is NOT present in the seconds set, use the setdiff1d() method.

Example

Find the difference of the set1 from set2:

import numpy as np  
  
set1 = np.array([1, 2, 3, 4])  
set2 = np.array([3, 4, 5, 6])  
  
newarr = np.setdiff1d(set1, set2, assume\_unique=True)  
  
print(newarr)

Finding Symmetric Difference

To find only the values that are NOT present in BOTH sets, use the setxor1d() method.

Example

Find the symmetric difference of the set1 and set2:

import numpy as np  
  
set1 = np.array([1, 2, 3, 4])  
set2 = np.array([3, 4, 5, 6])  
  
newarr = np.setxor1d(set1, set2, assume\_unique=True)  
  
print(newarr)

# Pandas

## Pandas Introduction

### 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.

### 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.

## Installation and import

### Installation of Pandas

If you have [Python](https://www.w3schools.com/python/default.asp) and [PIP](https://www.w3schools.com/python/python_pip.asp) 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

### 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"])

### 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)

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

## 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: day2, 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)

Top of Form

## 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](https://www.w3schools.com/python/pandas/data.csv). or [Open data.csv](https://www.w3schools.com/python/pandas/data.csv.txt)

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)

## 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](https://www.w3schools.com/python/pandas/data.js).

**Example**

Load the JSON file into a DataFrame:

import pandas as pd  
  
df = pd.read\_json('data.json')  
  
print(df.to\_string())

## Dictionary as JSON

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)

## 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))

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

Download [data.csv](https://www.w3schools.com/python/pandas/data.csv), or open [data.csv](https://www.w3schools.com/python/pandas/data.csv.txt) in your browser.

**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('data.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, and you will learn more about that in the next chapters.

## Data Cleaning

Data cleaning means fixing bad data in your data set.

Bad data could be:

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

In this tutorial you will learn how to deal with all of them.

### Our Data Set

In the next chapters 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).

### 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())

**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('data.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 containing 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('data.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('data.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('data.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('data.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('data.csv')  
  
x = df["Calories"].mode()[0]  
  
df["Calories"].fillna(x, inplace = True)

**Mode** = the value that appears most frequently.

### Data of Wrong Format

Cells with data of wrong format can make it difficult, or even impossible, to analyze data.

To fix it, you have two options: remove the rows, or convert all cells in the columns into the same format.

### Convert Into a Correct Format

In our Data Frame, we have two cells with the wrong format. Check out row 22 and 26, the 'Date' column should be a string that represents a date:

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 20201226 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

Let's try to convert all cells in the 'Date' column into dates.

Pandas has a to\_datetime() method for this:

Example

Convert to date:

import pandas as pd  
  
df = pd.read\_csv('data.csv')  
  
df['Date'] = pd.to\_datetime(df['Date'])  
  
print(df.to\_string())

Result:

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 NaT 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

As you can see from the result, the date in row 26 was fixed, but the empty date in row 22 got a NaT (Not a Time) value, in other words an empty value. One way to deal with empty values is simply removing the entire row.

### Removing Rows

The result from the converting in the Example above gave us a NaT value, which can be handled as a NULL value, and we can remove the row by using the dropna() method.

Example

Remove rows with a NULL value in the "Date" column:

df.dropna(subset=['Date'], inplace = True)

### Wrong Data

"Wrong data" does not have to be "empty cells" or "wrong format", it can just be wrong, like if someone registered "199" instead of "1.99".

Sometimes you can spot wrong data by looking at the data set, because you have an expectation of what it should be.

If you take a look at our data set, you can see that in row 7, the duration is 450, but for all the other rows the duration is between 30 and 60.

It doesn't have to be wrong, but taking in consideration that this is the data set of someone's workout sessions, we conclude with the fact that this person did not work out in 450 minutes.

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 20201226 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

How can we fix wrong values, like the one for "Duration" in row 7?

### Replacing Values

One way to fix wrong values is to replace them with something else.

In our Example, it is most likely a typo, and the value should be "45" instead of "450", and we could just insert "45" in row 7:

Example

Set "Duration" = 45 in row 7:

df.loc[7, 'Duration'] = 45

For small data sets you might be able to replace the wrong data one by one, but not for big data sets.

To replace wrong data for larger data sets you can create some rules, e.g. set some boundaries for legal values, and replace any values that are outside of the boundaries.

Example

Loop through all values in the "Duration" column.

If the value is higher than 120, set it to 120:

for x in df.index:  
  if df.loc[x, "Duration"] > 120:  
    df.loc[x, "Duration"] = 120

### Removing Rows

Another way of handling wrong data is to remove the rows that contains wrong data.

This way you do not have to find out what to replace them with, and there is a good chance you do not need them to do your analyses.

Example

Delete rows where "Duration" is higher than 120:

for x in df.index:  
  if df.loc[x, "Duration"] > 120:  
    df.drop(x, inplace = True)

### Discovering Duplicates

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

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 20201226 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

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.

### 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.csv'.

[Download data.csv](https://www.w3schools.com/python/pandas/data.csv). or [Open data.csv](https://www.w3schools.com/python/pandas/data.csv.txt)

**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.

Top of Form

## Plotting

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.

Read more about Matplotlib in our [Matplotlib Tutorial](https://www.w3schools.com/python/matplotlib_intro.asp).

Example

Import pyplot from Matplotlib and visualize our DataFrame:

import pandas as pd  
import matplotlib.pyplot as plt  
  
df = pd.read\_csv('data.csv')  
  
df.plot()  
  
plt.show()

The Examples in this page uses a CSV file called: 'data.csv'.

[Download data.csv](https://www.w3schools.com/python/pandas/data.csv) or [Open data.csv](https://www.w3schools.com/python/pandas/data.csv.txt)

### 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.

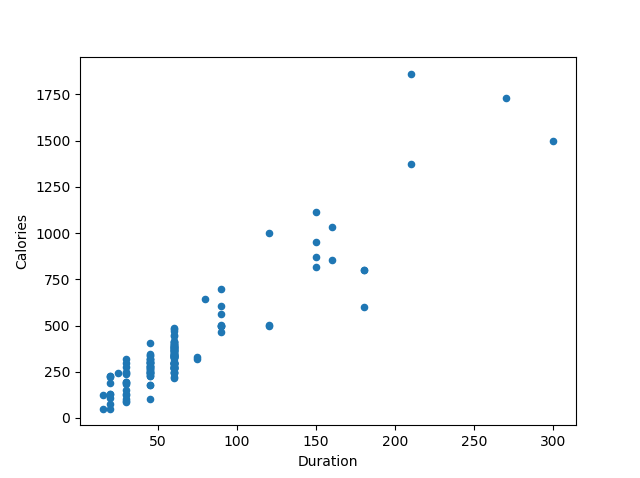
Include the x and y arguments like this:

x = 'Duration', y = 'Calories'

Example

import pandas as pd  
import matplotlib.pyplot as plt  
  
df = pd.read\_csv('data.csv')  
  
df.plot(kind = 'scatter', x = 'Duration', y = 'Calories')  
  
plt.show()

Result



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

By looking at the scatterplot, I will agree.

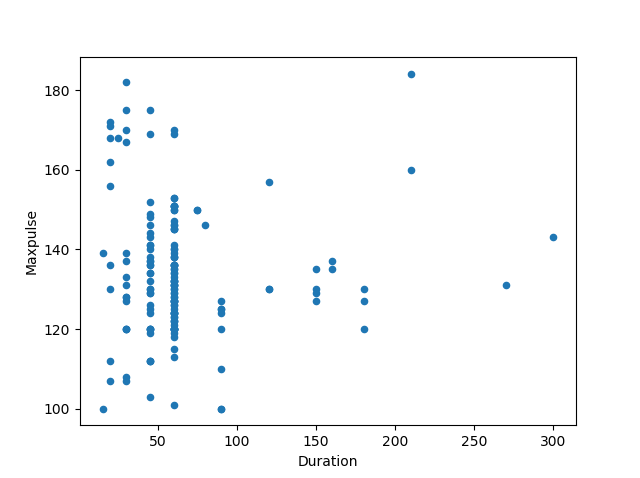
Let's create another scatterplot, where there is a bad relationship between the columns, like "Duration" and "Maxpulse", with the correlation 0.009403:

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

Result



### Histogram

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

kind = 'hist'

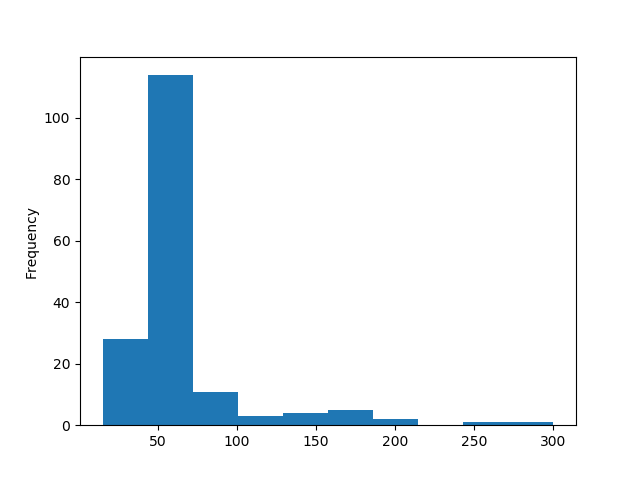
A histogram needs only one column.

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')



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