

SACHIN SIROHI

Python - Data Science

Data is the new Oil. This statement shows how every modern IT system is driven by capturing, storing and analysing data for various needs. Be it about making decision for business, forecasting weather, studying protein structures in biology or designing a marketing campaign. All of these scenarios involve a multidisciplinary approach of using mathematical models, statistics, graphs, databases and of course the business or scientific logic behind the data analysis. So we need a programming language which can cater to all these diverse needs of data science. Python shines bright as one such language as it has numerous libraries and built in features which makes it easy to tackle the needs of Data science.

In this tutorial we will cover these the various techniques used in data science using the Python programming language.

Audience

This tutorial is designed for Computer Science graduates as well as Software Professionals who are willing to learn data science in simple and easy steps using Python as a programming language.

Prerequisites

Before proceeding with this tutorial, you should have a basic knowledge of writing code in Python programming language, using any python IDE and execution of Python programs.

Python - Data Science Introduction

Data science is the process of deriving knowledge and insights from a huge and diverse set of data through organizing, processing and analysing the data. It involves many different disciplines like mathematical and statistical modelling, extracting data from its source and applying data visualization techniques. Often it also involves handling big data technologies to gather both structured and unstructured data. Below we will see some example scenarios where Data science is used.

Python in Data Science

The programming requirements of data science demands a very versatile yet flexible language which is simple to write the code but can handle highly complex mathematical processing. Python is most suited for such requirements as it has already established itself both as a language for general computing as well as scientific computing. Moreover it is being continuously upgraded in form of new addition to its plethora of libraries aimed at different programming requirements. Below we will discuss such features of python which makes it the preferred language for data science.

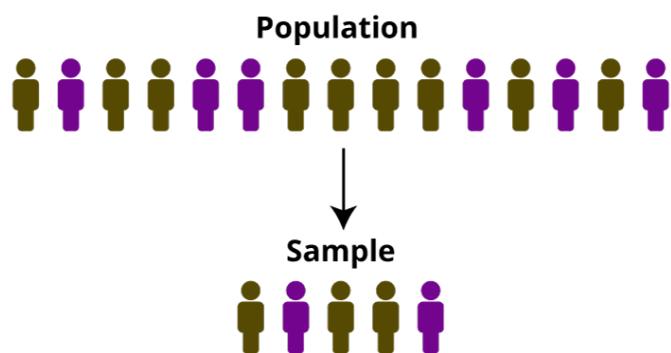
- A simple and easy to learn language which achieves result in fewer lines of code than other similar languages like R. Its simplicity also makes it robust to handle complex scenarios with minimal code and much less confusion on the general flow of the program.
- It is cross platform, so the same code works in multiple environments without needing any change. That makes it perfect to be used in a multi-environment setup easily.
- Its excellent memory management capability, especially garbage collection makes it versatile in gracefully managing very large volume of data transformation, slicing, dicing and visualization.
- Most importantly Python has got a very large collection of libraries which serve as special purpose analysis tools. For example – the NumPy package deals with scientific computing and its array needs much less memory than the conventional python list for managing numeric data. And the number of such packages is continuously growing.



Statistical Analysis in Data Science

Statistics is a science concerned with collection, analysis, interpretation, and presentation of data. In Statistics, we generally want to study a population. You may consider a population as a collection of things, persons, or objects under experiment or study. It is usually not possible to gain access to all of the information from the entire population due to logistical reasons. So, when we want to study a population, we generally select a sample.

In sampling, we select a portion (or subset) of the larger population and then study the portion (or the sample) to learn about the population. [Data](#) is the result of sampling from a population.



What is Statistical Analysis? Types, Methods and Examples

What Is Statistical Analysis?

Statistical analysis is the process of collecting and analyzing data in order to discern patterns and trends. It is a method for removing bias from evaluating data by employing numerical analysis. This technique is useful for collecting the interpretations of research, developing statistical models, and planning surveys and studies.

Statistical analysis is a scientific tool that helps collect and analyze large amounts of [data](#) to identify common patterns and trends to convert them into meaningful information. In simple words, statistical analysis is a [data analysis tool](#) that helps draw meaningful conclusions from raw and unstructured data.

The conclusions are drawn using statistical analysis facilitating decision-making and helping businesses make future predictions on the basis of past trends. It can be defined as a science of collecting and analyzing data to identify trends and patterns and presenting them. Statistical analysis involves working with numbers and is used by businesses and other institutions to make use of data to derive meaningful information.

Types of Statistical Analysis

Given below are the 6 types of statistical analysis:

- Descriptive Analysis

[Descriptive statistical analysis](#) involves collecting, interpreting, analyzing, and summarizing data to present them in the form of charts, graphs, and tables. Rather than drawing conclusions, it simply makes the complex data easy to read and understand.

- **Inferential Analysis**

The [inferential statistical analysis](#) focuses on drawing meaningful conclusions on the basis of the data analyzed. It studies the relationship between different variables or makes predictions for the whole population.

- **Predictive Analysis**

Predictive statistical analysis is a type of statistical analysis that analyzes data to derive past trends and predict future events on the basis of them. It uses [machine learning](#) algorithms, [data mining](#), [data modelling](#), and [artificial intelligence](#) to conduct the statistical analysis of data.

- **Prescriptive Analysis**

The prescriptive analysis conducts the analysis of data and prescribes the best course of action based on the results. It is a type of statistical analysis that helps you make an informed decision.

- **Exploratory Data Analysis**

[Exploratory analysis](#) is similar to inferential analysis, but the difference is that it involves exploring the unknown data associations. It analyzes the potential relationships within the data.

- **Causal Analysis**

The causal statistical analysis focuses on determining the cause and effect relationship between different variables within the raw data. In simple words, it

determines why something happens and its effect on other variables. This methodology can be used by businesses to determine the reason for failure.

Benefits of Statistical Analysis

Statistical analysis can be called a boon to mankind and has many benefits for both individuals and organizations. Given below are some of the reasons why you should consider investing in statistical analysis:

- It can help you determine the monthly, quarterly, yearly figures of sales profits, and costs making it easier to make your decisions.
- It can help you make informed and correct decisions.
- It can help you identify the problem or cause of the failure and make corrections. For example, it can identify the reason for an increase in total costs and help you cut the wasteful expenses.
- It can help you conduct market analysis and make an effective marketing and sales strategy.
- It helps improve the efficiency of different processes.

Statistical Analysis Process

Given below are the 5 steps to conduct a statistical analysis that you should follow:

- Step 1: Identify and describe the nature of the data that you are supposed to analyze.
- Step 2: The next step is to establish a relation between the data analyzed and the sample population to which the data belongs.
- Step 3: The third step is to create a model that clearly presents and summarizes the relationship between the population and the data.

- Step 4: Prove if the model is valid or not.
- Step 5: Use [predictive analysis](#) to predict future trends and events likely to happen.

Statistical Analysis Methods

Although there are various methods used to perform data analysis,

Statistics

A statistic is merely a portion of a target sample. It refers to the measure of the values calculated from the population.

A parameter is a fixed and unknown numerical value used for describing the entire population. The most commonly used parameters are:

- **Mean**
- **Median**
- **Mode**

Mean :

The mean is the average or the most common value in a data sample or a population. It is also referred to as the expected value.

Formula: Sum of the total number of observations/the number of observations.

```
Experimental data set: 2, 4, 6, 8, 10, 12, 14, 16, 18, 20
```

Calculating mean:

$$\begin{aligned} & (2 + 4 + 6 + 8 + 10 + 12 + 14 + 16 + 18 + 20) / 10 \\ & = 110 / 10 \end{aligned}$$

```
= 11
```

Copy Code

Median:

In statistics, the median is the value separating the higher half from the lower half of a data sample, a population, or a probability distribution. It's the mid-value obtained by arranging the data in increasing order or descending order.

Formula:

Let n be the data set (increasing order)

When data set is odd: Median = $n+1/2$ th term

```
Case-I: (n is odd)
```

```
Experimental data set = 1, 2, 3, 4, 5
```

```
Median (n = 5) = [(5 +1)/2]th term
```

```
= 6/2 term
```

```
= 3rd term
```

```
Therefore, the median is 3
```

Copy Code

When data set is even: Median = $[n/2\text{th} + (n/2 + 1)\text{th}] / 2$

```
Case-II: (n is even)
```

```
Experimental data set = 1, 2, 3, 4, 5, 6
```

```
Median (n = 6) = [n/2\text{th} + (n/2 + 1)\text{th}] / 2
```

```
= ( 6/2\text{th} + (6/2 +1)\text{th}) / 2
```

$$= (3\text{rd} + 4\text{th})/2$$

$$= (3 + 4)/2$$

$$= 7/2$$

$$= 3.5$$

Therefore, the median is 3.5

Copy Code

Mode:

The mode is the value that appears most often in a set of data or a population.

```
Experimental data set= 1, 2, 2, 2, 3, 3, 3, 3, 3, 4, 4, 4, 5, 6
```

```
Mode = 3
```

Copy Code

(Since 3 is the most repeated element in the sequence.)

- Standard Deviation

Standard deviation is another very widely used statistical tool or method. It analyzes the deviation of different data points from the mean of the entire data set. It determines how data of the data set is spread around the mean. You can use it to decide whether the research outcomes can be generalized or not.

- Regression

Regression is a statistical tool that helps determine the cause and effect relationship between the variables. It determines the relationship between a dependent and an independent variable. It is generally used to predict future trends and events.

- Hypothesis Testing

[Hypothesis testing](#) can be used to test the validity or trueness of a conclusion or argument against a data set. The hypothesis is an assumption made at the beginning of the research and can hold or be false based on the analysis results.

- Sample Size Determination

Sample size determination or [data sampling](#) is a technique used to derive a sample from the entire population, which is representative of the population. This method is used when the size of the population is very large. You can choose from among the various data sampling techniques such as snowball sampling, convenience sampling, and random sampling.

Statistical Analysis Software

Everyone can't perform very complex statistical calculations with accuracy making statistical analysis a time-consuming and costly process. Statistical software has become a very important tool for companies to perform their data analysis. The software uses [Artificial Intelligence and Machine Learning](#) to perform complex calculations, identify trends and patterns, and create charts, graphs, and tables accurately within minutes.

Statistical Analysis Examples

Look at the standard deviation sample calculation given below to understand more about statistical analysis. The weights of 5 pizza bases in cms are as follows:

| Particulars (Weight in cms) | Mean Deviation | Square of Mean Deviation |
|-----------------------------|----------------|--------------------------|
| 9 | $9-6.4 = 2.6$ | $(2.6)^2 = 6.76$ |
| 2 | $2-6.4 = -4.4$ | $(-4.4)^2 = 19.36$ |
| 5 | $5-6.4 = -1.4$ | $(-1.4)^2 = 1.96$ |
| 4 | $4-6.4 = -2.4$ | $(-2.4)^2 = 5.76$ |
| 12 | $12-6.4 = 5.6$ | $(5.6)^2 = 31.36$ |

Calculation of Mean = $(9+2+5+4+12)/5 = 32/5 = 6.4$

Calculation of mean of squared mean deviation = $(6.76+19.36+1.96+5.76+31.36)/5 = 13.04$

Sample Variance = 13.04

Standard deviation = $\sqrt{13.04} = 3.611$

Statistics with Python

Statistics, in general, is the method of collection of data, tabulation, and interpretation of numerical data. It is an area of applied mathematics concern with data collection analysis, interpretation, and presentation. With statistics, we can see how data can be used to solve complex problems.

In this tutorial, we will learn about solving statistical problems with Python and will also learn the concept behind it. Let's start by understanding some concepts that will be useful throughout the article.

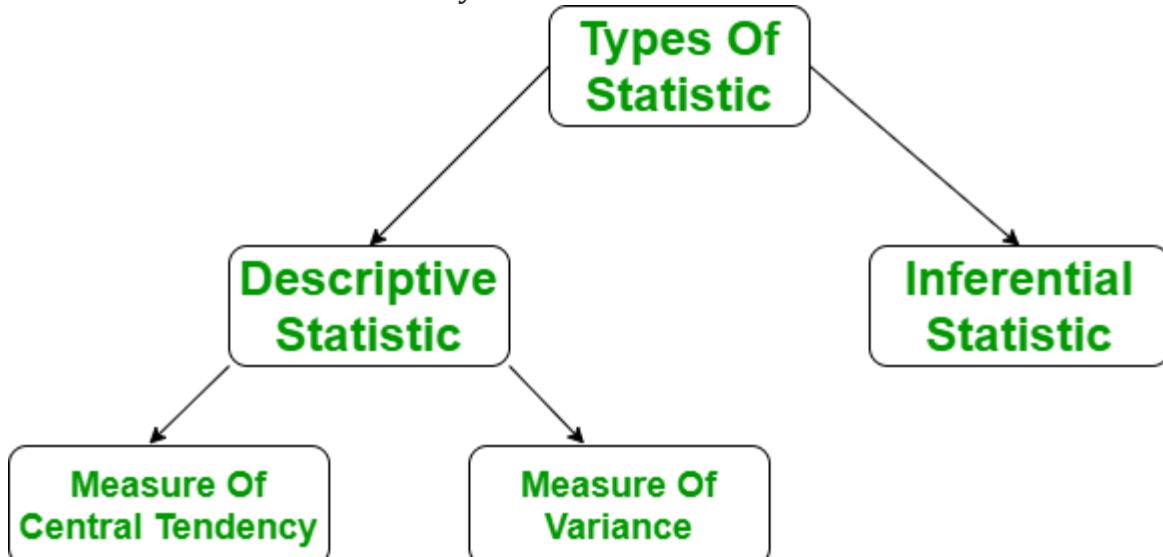
Note: We will be covering descriptive statistics with the help of the **statistics** module provided by Python.

Understanding the Descriptive Statistics

In a layman's term, descriptive statistics generally means describing the data with the help of some representative methods like charts, tables, Excel files, etc. The data is described in such a way that it can express some meaningful information that can also be used to find some future trends. Describing and summarizing a single variable is called **univariate analysis**. Describing a statistical relationship between two variables is called **bivariate analysis**. Describing the statistical relationship between multiple variables is called **multivariate analysis**.

There are two types of descriptive Statistics -

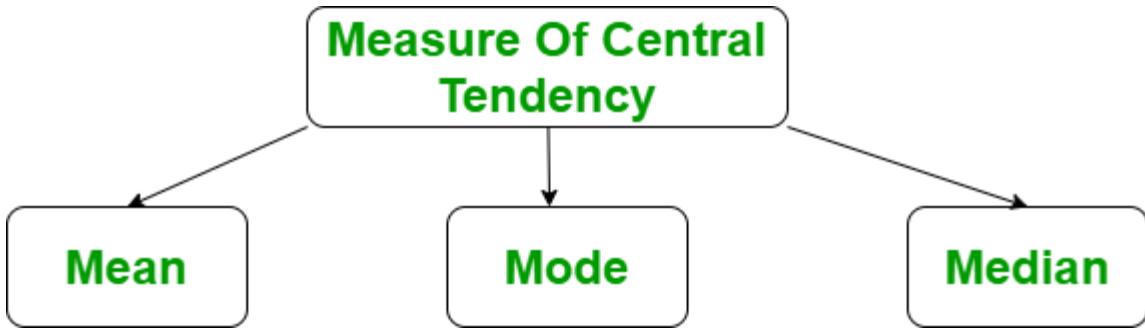
- Measure of central tendency
- Measure of variability



Measure of central tendency

The **measure of central tendency** is a single value that attempts to describe the whole set of data. There are three main features of central tendency -

- Mean
- Median
 - Median Low
 - Median High
- Mode



Mean

It is the sum of observations divided by the total number of observations. It is also defined as average which is the sum divided by count.

The [mean\(\)](#) function returns the mean or average of the data passed in its arguments. If passed argument is empty, **StatisticsError** is raised.

Example:

- Python3

```

# Python code to demonstrate the working of
# mean()

# importing statistics to handle statistical
# operations
import statistics

# initializing list
li = [1, 2, 3, 3, 2, 2, 2, 1]

# using mean() to calculate average of list
# elements
print ("The average of list values is : ",end="")
print (statistics.mean(li))
  
```

Output:

The average of list values is : 2

Median

It is the middle value of the data set. It splits the data into two halves. If the number of elements in the data set is odd then the center element is median and if it is even then the median would be the average of two central elements.

For Odd Numbers:

For Even Numbers:

[**median\(\)**](#) function is used to calculate the median, i.e middle element of data. If the passed argument is empty, **StatisticsError** is raised.

Example:

- Python3

```
# Python code to demonstrate the
# working of median() on various
# range of data-sets

# importing the statistics module
from statistics import median

# Importing fractions module as fr
from fractions import Fraction as fr

# tuple of positive integer numbers
data1 = (2, 3, 4, 5, 7, 9, 11)

# tuple of floating point values
data2 = (2.4, 5.1, 6.7, 8.9)

# tuple of fractional numbers
```

```

data3 = (fr(1, 2), fr(44, 12),
         fr(10, 3), fr(2, 3))

# tuple of a set of negative integers
data4 = (-5, -1, -12, -19, -3)

# tuple of set of positive
# and negative integers
data5 = (-1, -2, -3, -4, 4, 3, 2, 1)

# Printing the median of above datasets
print("Median of data-set 1 is % s" % (median(data1)))
print("Median of data-set 2 is % s" % (median(data2)))
print("Median of data-set 3 is % s" % (median(data3)))
print("Median of data-set 4 is % s" % (median(data4)))
print("Median of data-set 5 is % s" % (median(data5)))

```

Output:

```

Median of data-set 1 is 5
Median of data-set 2 is 5.9
Median of data-set 3 is 2
Median of data-set 4 is -5
Median of data-set 5 is 0.0

```

Median Low

median_low() function returns the median of data in case of odd number of elements, but in case of even number of elements, returns the lower of two middle elements. If the passed argument is empty, **StatisticsError** is raised

Example:

- Python3

```

# Python code to demonstrate the
# working of median_low()

# importing the statistics module
import statistics

# simple list of a set of integers
set1 = [1, 3, 3, 4, 5, 7]

# Print median of the data-set

# Median value may or may not
# lie within the data-set
print("Median of the set is % s"
      % (statistics.median(set1)))

# Print low median of the data-set
print("Low Median of the set is % s "
      % (statistics.median_low(set1)))

```

Output:

Median of the set is 3.5

Low Median of the set is 3

Median High

median_high() function returns the median of data in case of odd number of elements, but in case of even number of elements, returns the higher of two middle elements. If passed argument is empty, **StatisticsError** is raised.

Example:

- Python3

```

# Working of median_high() and median() to
# demonstrate the difference between them.

# importing the statistics module
import statistics

# simple list of a set of integers
set1 = [1, 3, 3, 4, 5, 7]

# Print median of the data-set

# Median value may or may not
# lie within the data-set
print("Median of the set is %s"
      % (statistics.median(set1)))

# Print high median of the data-set
print("High Median of the set is %s "
      % (statistics.median_high(set1)))

```

Output:

Median of the set is 3.5

High Median of the set is 4

Mode

It is the value that has the highest frequency in the given data set. The data set may have no mode if the frequency of all data points is the same. Also, we can have more than one mode if we encounter two or more data points having the same frequency.

mode() function returns the number with the maximum number of occurrences. If the passed argument is empty, **StatisticsError** is raised.

Example:

- Python3

```
# Python code to demonstrate the
# working of mode() function

# on a various range of data types

# Importing the statistics module
from statistics import mode

# Importing fractions module as fr
# Enables to calculate harmonic_mean of a
# set in Fraction

from fractions import Fraction as fr

# tuple of positive integer numbers
data1 = (2, 3, 3, 4, 5, 5, 5, 5, 6, 6, 6, 7)

# tuple of a set of floating point values
data2 = (2.4, 1.3, 1.3, 1.3, 2.4, 4.6)

# tuple of a set of fractional numbers
data3 = (fr(1, 2), fr(1, 2), fr(10, 3), fr(2, 3))

# tuple of a set of negative integers
data4 = (-1, -2, -2, -2, -7, -7, -9)

# tuple of strings
data5 = ("red", "blue", "black", "blue", "black", "black", "brown")
```

```

# Printing out the mode of the above data-sets

print("Mode of data set 1 is % s" % (mode(data1)))
print("Mode of data set 2 is % s" % (mode(data2)))
print("Mode of data set 3 is % s" % (mode(data3)))
print("Mode of data set 4 is % s" % (mode(data4)))
print("Mode of data set 5 is % s" % (mode(data5)))

```

Output:

Mode of data set 1 is 5
 Mode of data set 2 is 1.3
 Mode of data set 3 is 1/2
 Mode of data set 4 is -2
 Mode of data set 5 is black

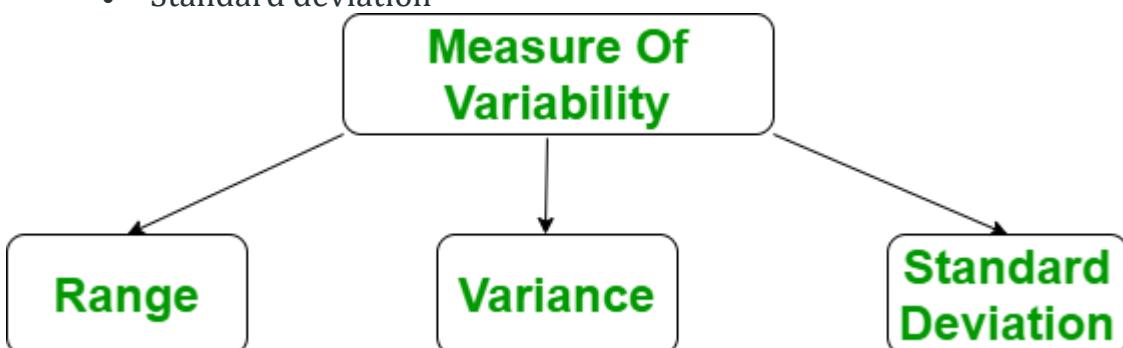
Refer to the below article to get detailed information about averages and Measures of central tendency.

- [Statistical Functions in Python | Set 1 \(Averages and Measure of Central Location\)](#)

Measure of variability

Till now, we have studied the measure of central tendency but this alone is not sufficient to describe the data. To overcome this we need the **measure of variability**. Measure of variability is known as the spread of data or how well is our data is distributed. The most common variability measures are:

- Range
- Variance
- Standard deviation



Range

The difference between the largest and smallest data point in our data set is known as the range. The range is directly proportional to the spread of data that means the bigger the range, more the spread of data and vice versa.

Range = Largest data value – smallest data value

We can calculate the maximum and minimum value using the [max\(\)](#) and [min\(\)](#) methods respectively.

Example:

- Python3

```
# Sample Data

arr = [1, 2, 3, 4, 5]

#Finding Max
Maximum = max(arr)

# Finding Min
Minimum = min(arr)

# Difference Of Max and Min
Range = Maximum-Minimum

print("Maximum = {}, Minimum = {} and Range = {}".format(
    Maximum, Minimum, Range))
```

Output:

Maximum = 5, Minimum = 1 and Range = 4

Variance

It is defined as an average squared deviation from the mean. It is being calculated by finding the difference between every data point and the average which is also known as the mean, squaring them, adding all of them, and then dividing by the number of data points present in our data set.

where N = number of terms

u = Mean

The statistics module provides the [variance\(\)](#) method that does all the maths behind the scene. If passed argument is empty, **StatisticsError** is raised.

Example:

- Python3

```
# Python code to demonstrate variance()

# function on varying range of data-types


# importing statistics module
from statistics import variance


# importing fractions as parameter values
from fractions import Fraction as fr


# tuple of a set of positive integers
# numbers are spread apart but not very much
sample1 = (1, 2, 5, 4, 8, 9, 12)

# tuple of a set of negative integers
sample2 = (-2, -4, -3, -1, -5, -6)

# tuple of a set of positive and negative numbers
# data-points are spread apart considerably
sample3 = (-9, -1, -0, 2, 1, 3, 4, 19)

# tuple of a set of fractional numbers
sample4 = (fr(1, 2), fr(2, 3), fr(3, 4),
           fr(5, 6), fr(7, 8))
```

```

# tuple of a set of floating point values

sample5 = (1.23, 1.45, 2.1, 2.2, 1.9)

# Print the variance of each samples

print("Variance of Sample1 is % s " % (variance(sample1)))
print("Variance of Sample2 is % s " % (variance(sample2)))
print("Variance of Sample3 is % s " % (variance(sample3)))
print("Variance of Sample4 is % s " % (variance(sample4)))
print("Variance of Sample5 is % s " % (variance(sample5)))

```

Output:

Variance of Sample1 is 15.80952380952381
 Variance of Sample2 is 3.5
 Variance of Sample3 is 61.125
 Variance of Sample4 is 1/45
 Variance of Sample5 is 0.17613000000000006

Standard Deviation

It is defined as the square root of the variance. It is being calculated by finding the Mean, then subtract each number from the Mean which is also known as average and square the result. Adding all the values and then divide by the no of terms followed the square root.

where N = number of terms

u = Mean

[**stdev\(\)**](#) method of the statistics module returns the standard deviation of the data. If passed argument is empty, **StatisticsError** is raised.

Example:

- Python3

```

# Python code to demonstrate stdev()

# function on various range of datasets

```

```
# importing the statistics module
from statistics import stdev

# importing fractions as parameter values
from fractions import Fraction as fr

# creating a varying range of sample sets
# numbers are spread apart but not very much
sample1 = (1, 2, 5, 4, 8, 9, 12)

# tuple of a set of negative integers
sample2 = (-2, -4, -3, -1, -5, -6)

# tuple of a set of positive and negative numbers
# data-points are spread apart considerably
sample3 = (-9, -1, -0, 2, 1, 3, 4, 19)

# tuple of a set of floating point values
sample4 = (1.23, 1.45, 2.1, 2.2, 1.9)

# Print the standard deviation of
print("The Standard Deviation of Sample1 is % s"
      % (stdev(sample1)))

print("The Standard Deviation of Sample2 is % s"
      % (stdev(sample2)))

print("The Standard Deviation of Sample3 is % s"
      % (stdev(sample3)))
```

```
print("The Standard Deviation of Sample4 is % s"  
      % (stdev(sample4)))
```

Output:

The Standard Deviation of Sample1 is 3.9761191895520196
The Standard Deviation of Sample2 is 1.8708286933869707
The Standard Deviation of Sample3 is 7.8182478855559445
The Standard Deviation of Sample4 is 0.41967844833872525

Python – Numpy

NumPy is a Python library.

NumPy is used for working with arrays.

NumPy is short for "Numerical Python".

NumPy Introduction

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++.

Operations using NumPy

Using NumPy, a developer can perform the following operations –

- Mathematical and logical operations on arrays.
- Fourier transforms and routines for shape manipulation.
- Operations related to linear algebra. NumPy has in-built functions for linear algebra and random number generation.

NumPy – A Replacement for MatLab

NumPy is often used along with packages like **SciPy** (Scientific Python) and **Matplotlib** (plotting library). This combination is widely used as a replacement for MatLab, a popular platform for technical computing. However, Python alternative to MatLab is now seen as a more modern and complete programming language.

It is open source, which is an added advantage of 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
```

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

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`.

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

Checking NumPy Version

The version string is stored under `__version__` attribute.

```
import numpy as np  
print(np.__version__)  
1.16.3
```

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.

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

```
[1 2 3 4 5]  
<class 'numpy.ndarray'>
```

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)
```

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

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)
```

```
42
```

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)
```

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

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)

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

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)

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

Check Number of Dimensions?

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

Example

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)
```

```
0
1
2
3
```

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)

[[[[[1 2 3 4]]]]]
number of dimensions : 5
```

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])  
  
1
```

Get the second element from the following array.

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

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])
```

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])  
  
[2 3 4 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:])  
  
[5 6 7]
```

Example

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

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

Negative Slicing

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

Example

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

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

STEP

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

Example

Return every other element from index 1 to index 5:

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

Example

Return every other element from the entire array:

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

Slicing 2-D Arrays

Example

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

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

Example

From both elements, return index 2:

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

Example

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

```
import numpy as np
```

```
arr = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10]])  
print(arr[0:2, 1:4])  
[[2 3 4]  
 [7 8 9]]
```

NumPy Data Types

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)  
  
int64
```

Example

Get the data type of an array containing strings:

```
import numpy as np  
  
arr = np.array(['apple', 'banana', 'cherry'])  
  
print(arr.dtype)  
  
<U6
```

Creating Arrays With a Defined Data Type

We use the `array()` function to create arrays, this function can take an optional argument: `dtype` that allows us to define the expected data type of the array elements:

Example

Create an array with data type string:

```
import numpy as np

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

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

```
[b'1' b'2' b'3' b'4']
|S1
```

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)
```

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

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**.

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

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')
```

```
Traceback (most recent call last):
  File "./prog.py", line 3, in <module>
```

```
ValueError: invalid literal for int() with base 10: 'a'
```

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)

[1 2 3]
int32
```

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)
```

```
[1 2 3]  
int64
```

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)  
  
[ True False True]  
bool
```

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)
```

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

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)
```

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

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)
```

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

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)
```

```
None
[1 2 3 4 5]
```

NumPy Array Shape

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)  
  
(2, 4)
```

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

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

NumPy Array Reshaping

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)
```

| | | | | | |
|---|---|----|----|----|---|
| [| [| 1 | 2 | 3 |] |
| | [| 4 | 5 | 6 |] |
| | [| 7 | 8 | 9 |] |
| | [| 10 | 11 | 12 |] |

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)

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

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

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)

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

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)
```

[[[1 2]
 [3 4]]

 [[5 6]
 [7 8]]]

Note: We can not pass `-1` to more than one dimension.

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)
```

[1 2 3 4 5 6]

NumPy Array Iterating

Iterating Arrays

Iterating means going through elements one by one.

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

If we iterate on a 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)
```

```
1
2
3
```

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)
```

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

If we iterate on a n -D array it will go through $n-1$ th 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)  
  
1  
2  
3  
4  
5  
6
```

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)  
  
x represents the 2-D array:  
[[1 2 3]  
 [4 5 6]]  
x represents the 2-D array:  
[[ 7  8  9]]
```

```
[10 11 12]]
```

o 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)
```

```
1
2
3
4
5
6
7
8
9
10
11
12
```

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)
```

```
1
2
3
4
5
6
7
8
```

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)

b'1'
b'2'
b'3'
```

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)
```

```
1
3
5
7
```

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)
```

```
(0,) 1
(1,) 2
```

```
(2,) 3
```

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)
```

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

NumPy Joining Array

Joining NumPy Arrays

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

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

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

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)
```

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

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)
```

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

Joining Arrays Using Stack Functions

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

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

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

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)
```

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

NumPy Splitting Array

Splitting NumPy Arrays

Splitting is reverse operation of Joining.

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

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

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)
```

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

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)

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

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])
```

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

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)
```

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

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)
```

```
[array([[1, 2, 3],  
       [4, 5, 6]]), array([[ 7,  8,  9],  
       [10, 11, 12]]), array([[13, 14, 15],  
       [16, 17, 18]])]
```

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)
```

```
[array([[ 1],  
       [ 4],  
       [ 7],  
       [10],  
       [13],  
       [16]]), array([[ 2],  
       [ 5],  
       [ 8],  
       [11],  
       [14],  
       [17]]), array([[ 3],  
       [ 6],  
       [ 9],  
       [12],  
       [15],  
       [18]])]
```

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)
```

```
[array([[ 1],  
       [ 4],  
       [ 7],  
       [10],  
       [13],  
       [16]]), array([[ 2],  
       [ 5],  
       [ 8],  
       [11],  
       [14],  
       [17]]), array([[ 3],  
       [ 6],  
       [ 9],  
       [12],  
       [15],  
       [18]])]
```

NumPy Searching Arrays

Searching Arrays

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

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

Example

Find the indexes where the value is 4:

```
import numpy as np

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

x = np.where(arr == 4)

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

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

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)

(array([0, 2, 4, 6]),)
```

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)

1
```

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)

2
```

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

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

Multiple Values

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

Example

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

```
import numpy as np

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

x = np.searchsorted(arr, [2, 4, 6])

print(x)

[1 2 3]
```

NumPy Sorting Arrays

Sorting Arrays

Sorting means putting elements in an *ordered sequence*.

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

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

Example

Sort the array:

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

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))  
['apple' 'banana' 'cherry']
```

Example

Sort a boolean array:

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

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

NumPy Filter Array

Filtering Arrays

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

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

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

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

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)
```

[41 43]

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)

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

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 divisible 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)

[False, True, False, True, False, True, False]
[2 4 6]

```

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)

```

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

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)

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

SciPy

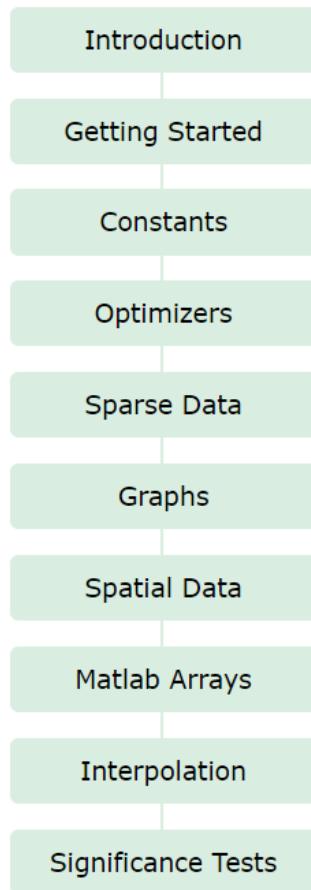
SciPy is a scientific computation library that uses [NumPy](#) underneath.

SciPy stands for Scientific Python.

Learning by Reading

We have created 10 tutorial pages for you to learn the fundamentals of SciPy:

Basic SciPy



SciPy Introduction

What is SciPy?

SciPy is a scientific computation library that uses [NumPy](#) underneath.

SciPy stands for Scientific Python.

It provides more utility functions for optimization, stats and signal processing.

Like NumPy, SciPy is open source so we can use it freely.

SciPy was created by NumPy's creator Travis Olliphant.

Why Use SciPy?

If SciPy uses NumPy underneath, why can we not just use NumPy?

SciPy has optimized and added functions that are frequently used in NumPy and Data Science.

Which Language is SciPy Written in?

SciPy is predominantly written in Python, but a few segments are written in C.

Where is the SciPy Codebase?

The source code for SciPy is located at this github repository <https://github.com/scipy/scipy>

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

SciPy Getting Started

Installation of SciPy

If you have [Python](#) and [PIP](#) already installed on a system, then installation of SciPy is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install scipy
```

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

Import SciPy

Once SciPy is installed, import the SciPy module(s) you want to use in your applications by adding the `from scipy import module` statement:

```
from scipy import constants
```

Now we have imported the *constants* module from SciPy, and the application is ready to use it:

Example

How many cubic meters are in one liter:

```
from scipy import constants  
  
print(constants.liter)
```

```
0.001
```

constants: SciPy offers a set of mathematical constants, one of them is `liter` which returns 1 liter as cubic meters.

You will learn more about constants in the next chapter.

Checking SciPy Version

The version string is stored under the `__version__` attribute.

Example

```
import scipy  
  
print(scipy.__version__)  
  
0.18.1
```

SciPy Constants

Constants in SciPy

As SciPy is more focused on scientific implementations, it provides many built-in scientific constants.

These constants can be helpful when you are working with Data Science.

PI is an example of a scientific constant.

Example

Print the constant value of PI:

```
from scipy import constants  
  
print(constants.pi)  
  
3.141592653589793
```

Constant Units

A list of all units under the constants module can be seen using the `dir()` function.

Example

List all constants:

```
from scipy import constants  
  
print(dir(constants))
```

Unit Categories

The units are placed under these categories:

- Metric
- Binary
- Mass
- Angle
- Time
- Length
- Pressure
- Volume
- Speed
- Temperature
- Energy
- Power
- Force

Metric (SI) Prefixes:

Return the specified unit in **meter** (e.g. `centi` returns `0.01`)

Example

```
from scipy import constants  
  
print(constants.yotta)      #1e+24  
print(constants.zetta)      #1e+21  
print(constants.exa)         #1e+18  
print(constants.peta)        #1000000000000000.0  
print(constants.tera)        #1000000000000.0  
print(constants.giga)        #1000000000.0  
print(constants.mega)        #1000000.0  
print(constants.kilo)         #1000.0  
print(constants.hecto)        #100.0  
print(constants.deka)         #10.0
```

```
print(constants.deci)      #0.1
print(constants centi)     #0.01
print(constants milli)     #0.001
print(constants micro)     #1e-06
print(constants nano)      #1e-09
print(constants pico)      #1e-12
print(constants femto)     #1e-15
print(constants atto)      #1e-18
print(constants zepto)     #1e-21

1e+24
1e+21
1e+18
1000000000000000.0
1000000000000.0
1000000000.0
1000000.0
1000.0
100.0
10.0
0.1
0.01
0.001
1e-06
1e-09
1e-12
1e-15
1e-18
1e-21
```

Binary Prefixes:

Return the specified unit in **bytes** (e.g. **kibi** returns **1024**)

Example

```
from scipy import constants

print(constants.kibi)      #1024
print(constants.mebi)       #1048576
print(constants.gibi)       #1073741824
print(constants.tebi)       #1099511627776
print(constants.pebi)       #1125899906842624
print(constants.exbi)       #1152921504606846976
print(constants.zebi)       #1180591620717411303424
print(constants.yobi)       #1208925819614629174706176
```

```
1024
1048576
1073741824
1099511627776
1125899906842624
1152921504606846976
1180591620717411303424
1208925819614629174706176
```

Mass:

Return the specified unit in **kg** (e.g. `gram` returns `0.001`)

Example

```
from scipy import constants

print(constants.gram)          #0.001
print(constants.metric_ton)    #1000.0
print(constants.grain)         #6.479891e-05
print(constants.lb)            #0.45359236999999997
print(constants.pound)         #0.45359236999999997
print(constants.oz)            #0.02834952312499998
print(constants.ounce)         #0.02834952312499998
print(constants.stone)         #6.350293179999995
print(constants.long_ton)      #1016.0469088
print(constants.short_ton)     #907.1847399999999
print(constants.troy_ounce)    #0.03110347679999998
print(constants.troy_pound)    #0.3732417215999996
print(constants.carat)         #0.0002
print(constants.atomic_mass)   #1.66053904e-27
print(constants.m_u)           #1.66053904e-27
print(constants.u)             #1.66053904e-27

0.001
1000.0
6.479891e-05
0.45359236999999997
0.45359236999999997
0.02834952312499998
0.02834952312499998
6.350293179999995
1016.0469088
907.1847399999999
0.03110347679999998
0.3732417215999996
0.0002
```

```
1.66053904e-27  
1.66053904e-27  
1.66053904e-27
```

Angle:

Return the specified unit in **radians** (e.g. `degree` returns `0.017453292519943295`)

Example

```
from scipy import constants

print(constants.degree)      #0.017453292519943295
print(constants.arcmin)      #0.0002908882086657216
print(constants.arcminute)    #0.0002908882086657216
print(constants.arcsec)       #4.84813681109536e-06
print(constants.arcsecond)    #4.84813681109536e-06

0.017453292519943295
0.0002908882086657216
0.0002908882086657216
4.84813681109536e-06
4.84813681109536e-06
```

Time:

Return the specified unit in **seconds** (e.g. `hour` returns `3600.0`)

Example

```
from scipy import constants

print(constants.minute)      #60.0
print(constants.hour)        #3600.0
print(constants.day)          #86400.0
print(constants.week)         #604800.0
print(constants.year)         #31536000.0
print(constants.Julian_year)  #31557600.0

60.0
3600.0
86400.0
604800.0
```

```
31536000.0  
31557600.0
```

Length:

Return the specified unit in **meters** (e.g. `nautical_mile` returns `1852.0`)

Example

```
from scipy import constants
```

```
print(constants.inch)          #0.0254
print(constants.foot)          #0.3047999999999996
print(constants.yard)          #0.914399999999999
print(constants.mile)          #1609.3439999999998
print(constants.mil)           #2.539999999999997e-05
print(constants.pt)            #0.00035277777777777776
print(constants.point)         #0.0003527777777777776
print(constants.survey_foot)   #0.3048006096012192
print(constants.survey_mile)   #1609.3472186944373
print(constants.nautical_mile) #1852.0
print(constants.fermi)         #1e-15
print(constants.angstrom)      #1e-10
print(constants.micron)        #1e-06
print(constants.au)            #149597870691.0
print(constants.astronomical_unit) #149597870691.0
print(constants.light_year)    #9460730472580800.0
print(constants.parsec)        #3.0856775813057292e+16
```

```
0.0254
0.3047999999999996
0.914399999999999
1609.3439999999998
2.539999999999997e-05
0.00035277777777777776
0.0003527777777777776
0.3048006096012192
1609.3472186944373
1852.0
1e-15
1e-10
1e-06
149597870691.0
149597870691.0
9460730472580800.0
3.0856775813057292e+16
```

Speed:

Return the specified unit in **meters per second** (e.g. `speed_of_sound` returns `340.5`)

Example

```
from scipy import constants

print(constants.kmh)          #0.2777777777777778
print(constants.mph)          #0.4470399999999994
print(constants.mach)         #340.5
print(constants.speed_of_sound) #340.5
print(constants.knot)         #0.5144444444444445

0.2777777777777778
0.4470399999999994
340.5
340.5
0.5144444444444445
```

Temperature:

Return the specified unit in **Kelvin** (e.g. `zero_Celsius` returns `273.15`)

Example

```
from scipy import constants

print(constants.zero_Celsius)    #273.15
print(constants.degree_Fahrenheit) #0.5555555555555556

273.15
0.5555555555555556
```

Python - Pandas

Pandas is a Python library.

Pandas is used to analyze data.

Learning by Reading

Starting with a basic introduction and ends up with cleaning and plotting data



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.



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

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.

Pandas Getting Started

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)
```

```
   cars  passings  
0   BMW         3  
1  Volvo        7  
2   Ford        2
```

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)
```

```
   cars  passings
0  BMW         3
1  Volvo        7
2   Ford        2
```

Checking Pandas Version

The version string is stored under `__version__` attribute.

Example

```
import pandas as pd

print(pd.__version__)

1.0.3
```

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)
```

```
0    1  
1    7  
2    2  
dtype: int64
```

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:

```
import pandas as pd  
  
a = [1, 7, 2]  
  
myvar = pd.Series(a)  
  
print(myvar[0])
```

```
1
```

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)
```

```
x    1
y    7
z    2
dtype: int64
```

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

Example

Return the value of "y":

```
import pandas as pd

a = [1, 7, 2]

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

print(myvar["y"])
```

7

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)
```

```
day1    420
day2    380
day3    390
dtype: int64
```

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)

day1    420
day2    380
dtype: int64
```

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)

   calories  duration
0        420         50
1        380         40
```

Pandas DataFrames

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: 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)
```

| | Duration | Pulse | Maxpulse | Calories |
|-----|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| .. | ... | ... | ... | ... |
| 164 | 60 | 105 | 140 | 290.8 |
| 165 | 60 | 110 | 145 | 300.4 |
| 166 | 60 | 115 | 145 | 310.2 |
| 167 | 75 | 120 | 150 | 320.4 |
| 168 | 75 | 125 | 150 | 330.4 |

[169 rows x 4 columns]

Pandas Read CSV

Read CSV Files

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

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

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

[Download data.csv](#). or [Open data.csv](#)

Example

Load the CSV into a DataFrame:

```
import pandas as pd  
  
df = pd.read_csv('data.csv')  
  
print(df.to_string())
```

```
Duration  Pulse  Maxpulse  Calories
```

| | | | | |
|----|----|-----|-----|-------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| 5 | 60 | 102 | 127 | 300.5 |
| 6 | 60 | 110 | 136 | 374.0 |
| 7 | 45 | 104 | 134 | 253.3 |
| 8 | 30 | 109 | 133 | 195.1 |
| 9 | 60 | 98 | 124 | 269.0 |
| 10 | 60 | 103 | 147 | 329.3 |
| 11 | 60 | 100 | 120 | 250.7 |
| 12 | 60 | 106 | 128 | 345.3 |
| 13 | 60 | 104 | 132 | 379.3 |
| 14 | 60 | 98 | 123 | 275.0 |
| 15 | 60 | 98 | 120 | 2 |

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)
```

| | Duration | Pulse | Maxpulse | Calories |
|-----|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| .. | ... | ... | ... | ... |
| 164 | 60 | 105 | 140 | 290.8 |
| 165 | 60 | 110 | 145 | 300.4 |
| 166 | 60 | 115 | 145 | 310.2 |
| 167 | 75 | 120 | 150 | 320.4 |
| 168 | 75 | 125 | 150 | 330.4 |

[169 rows x 4 columns]

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)
```

60

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)
```

| | Duration | Pulse | Maxpulse | Calories |
|---|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| 5 | 60 | 102 | 127 | 300.5 |
| 6 | 60 | 110 | 136 | 374.0 |
| 7 | 45 | 104 | 134 | 253.3 |
| 8 | 30 | 109 | 133 | 195.1 |

9

60

98

124

269.0

Pandas Read JSON

Read JSON

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

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

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

[Open data.json.](#)

Example

Load the JSON file into a DataFrame:

```
import pandas as pd  
  
df = pd.read_json('data.json')  
  
print(df.to_string())
```

| | Duration | Pulse | Maxpulse | Calories |
|---|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| 5 | 60 | 102 | 127 | 300.5 |
| 6 | 60 | 110 | 136 | 374.0 |
| 7 | 45 | 104 | 134 | 253.3 |
| 8 | 30 | 109 | 133 | 195.1 |

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)
```

| | Duration | Pulse | Maxpulse | Calories |
|---|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| 5 | 60 | 102 | 127 | 300.5 |

Pandas - Analyzing

DataFrames

Viewing the Data

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

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

Example

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

```
import pandas as pd

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

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

| | Duration | Pulse | Maxpulse | Calories |
|---|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |
| 5 | 60 | 102 | 127 | 300.5 |
| 6 | 60 | 110 | 136 | 374.0 |
| 7 | 45 | 104 | 134 | 253.3 |
| 8 | 30 | 109 | 133 | 195.1 |
| 9 | 60 | 98 | 124 | 269.0 |

Example

Print the first 5 rows of the DataFrame:

```
import pandas as pd  
  
df = pd.read_csv('data.csv')  
  
print(df.head())
```

| | Duration | Pulse | Maxpulse | Calories |
|---|----------|-------|----------|----------|
| 0 | 60 | 110 | 130 | 409.1 |
| 1 | 60 | 117 | 145 | 479.0 |
| 2 | 60 | 103 | 135 | 340.0 |
| 3 | 45 | 109 | 175 | 282.4 |
| 4 | 45 | 117 | 148 | 406.0 |

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:

| | Duration | Pulse | Maxpulse | Calories |
|-----|----------|-------|----------|----------|
| 164 | 60 | 105 | 140 | 290.8 |
| 165 | 60 | 110 | 145 | 300.4 |
| 166 | 60 | 115 | 145 | 310.2 |
| 167 | 75 | 120 | 150 | 320.4 |
| 168 | 75 | 125 | 150 | 330.4 |

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
```

Pandas - Cleaning Data

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).

Pandas - 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.

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())

#Notice in the result that some rows have been removed (row 18, 22 and
#28).
#These rows had cells with empty values.
```

| Duration | Date | Pulse | Maxpulse | Calories |
|----------|------|--------------|----------|----------|
| 0 | 60 | '2020/12/01' | 110 | 130 |
| 1 | 60 | '2020/12/02' | 117 | 145 |
| 2 | 60 | '2020/12/03' | 103 | 135 |
| 3 | 45 | '2020/12/04' | 109 | 175 |
| 4 | 45 | '2020/12/05' | 117 | 148 |
| 5 | 60 | '2020/12/06' | 102 | 127 |
| 6 | 60 | '2020/12/07' | 110 | 136 |
| 7 | 450 | '2020/12/08' | 104 | 134 |
| 8 | 30 | '2020/12/09' | 109 | 133 |
| 9 | 60 | '2020/12/10' | 98 | 124 |
| 10 | 60 | '2020/12/11' | 103 | 147 |
| 11 | 60 | '2020/12/12' | 100 | 120 |
| 12 | 60 | '2020/12/12' | 100 | 120 |
| 13 | 60 | '2020/12/13' | 106 | 128 |
| 14 | 60 | '2020/12/14' | 104 | 132 |
| 15 | 60 | '2020/12/15' | 98 | 123 |
| 16 | 60 | '2020/12/16' | 98 | 120 |
| 17 | 60 | '2020/12/17' | 100 | 120 |
| 19 | 60 | '2020/12/19' | 103 | 123 |
| 20 | 45 | '2020/12/20' | 97 | 125 |
| 21 | 60 | '2020/12/21' | 108 | 131 |
| 23 | 60 | '2020/12/23' | 130 | 101 |
| 24 | 45 | '2020/12/24' | 105 | 132 |

| | | | | | |
|----|----|--------------|-----|-----|-------|
| 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 |
| 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 |

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())
```

#Notice in the result that some rows have been removed (row 18, 22 and 28).

#These rows had cells with empty values.

| 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 |
| 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 |
| 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 |
| 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 |

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)
```

#Notice in the result: empty cells got the value 130 (in row 18, 22 and 28).

| 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 | 130.0 |
| 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 | 130 | 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 | 130.0 |
| 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 |

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)

#This operation inserts 130 in empty cells in the "Calories" column (row 18 and 28).
```

| | 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 | 130.0 |
| 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 | 130.0 |
| 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 |

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)
```

| Duration | Date | Pulse | Maxpulse | Calories |
|----------|------|--------------|----------|----------|
| 0 | 60 | '2020/12/01' | 110 | 130 |
| 1 | 60 | '2020/12/02' | 117 | 145 |
| 2 | 60 | '2020/12/03' | 103 | 135 |
| 3 | 45 | '2020/12/04' | 109 | 175 |

| | | | | | |
|----|-----|--------------|-----|-----|--------|
| 4 | 45 | '2020/12/05' | 117 | 148 | 406.00 |
| 5 | 60 | '2020/12/06' | 102 | 127 | 300.00 |
| 6 | 60 | '2020/12/07' | 110 | 136 | 374.00 |
| 7 | 450 | '2020/12/08' | 104 | 134 | 253.30 |
| 8 | 30 | '2020/12/09' | 109 | 133 | 195.10 |
| 9 | 60 | '2020/12/10' | 98 | 124 | 269.00 |
| 10 | 60 | '2020/12/11' | 103 | 147 | 329.30 |
| 11 | 60 | '2020/12/12' | 100 | 120 | 250.70 |
| 12 | 60 | '2020/12/12' | 100 | 120 | 250.70 |
| 13 | 60 | '2020/12/13' | 106 | 128 | 345.30 |
| 14 | 60 | '2020/12/14' | 104 | 132 | 379.30 |
| 15 | 60 | '2020/12/15' | 98 | 123 | 275.00 |
| 16 | 60 | '2020/12/16' | 98 | 120 | 215.20 |
| 17 | 60 | '2020/12/17' | 100 | 120 | 300.00 |
| 18 | 45 | '2020/12/18' | 90 | 112 | 304.68 |
| 19 | 60 | '2020/12/19' | 103 | 123 | 323.00 |
| 20 | 45 | '2020/12/20' | 97 | 125 | 243.00 |
| 21 | 60 | '2020/12/21' | 108 | 131 | 364.20 |
| 22 | 45 | NaN | 100 | 119 | 282.00 |
| 23 | 60 | '2020/12/23' | 130 | 101 | 300.00 |
| 24 | 45 | '2020/12/24' | 105 | 132 | 246.00 |
| 25 | 60 | '2020/12/25' | 102 | 126 | 334.50 |
| 26 | 60 | '2020/12/26' | 100 | 120 | 250.00 |
| 27 | 60 | '2020/12/27' | 92 | 118 | 241.00 |
| 28 | 60 | '2020/12/28' | 103 | 132 | 304.68 |
| 29 | 60 | '2020/12/29' | 100 | 132 | 280.00 |
| 30 | 60 | '2020/12/30' | 102 | 129 | 380.30 |
| 31 | 60 | '2020/12/31' | 92 | 115 | 243.00 |

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)
```

| | 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 | 291.2 |
| 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 | 291.2 |
| 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 |

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)
```

| Duration | Date | Pulse | Maxpulse | Calories |
|----------|------|--------------|----------|----------|
| 0 | 60 | '2020/12/01' | 110 | 130 |
| 1 | 60 | '2020/12/02' | 117 | 145 |
| 2 | 60 | '2020/12/03' | 103 | 135 |
| 3 | 45 | '2020/12/04' | 109 | 175 |
| 4 | 45 | '2020/12/05' | 117 | 148 |
| 5 | 60 | '2020/12/06' | 102 | 127 |

| | | | | | |
|----|-----|--------------|-----|-----|-------|
| 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 | 300.0 |
| 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 | 300.0 |
| 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 |

Pandas - Cleaning Data of Wrong Format

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())

```

| | 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 |

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:

```
import pandas as pd  
  
df = pd.read_csv('data.csv')  
  
df['Date'] = pd.to_datetime(df['Date'])  
  
df.dropna(subset=['Date'], inplace = True)  
  
print(df.to_string())
```

| | 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 |
| 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 |

Pandas - Fixing Wrong Data

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:

```
import pandas as pd

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

df.loc[7,'Duration'] = 45

print(df.to_string())
```

| Duration | Date | Pulse | Maxpulse | Calories |
|----------|------|--------------|----------|----------|
| 0 | 60 | '2020/12/01' | 110 | 130 |
| 1 | 60 | '2020/12/02' | 117 | 145 |
| 2 | 60 | '2020/12/03' | 103 | 135 |
| 3 | 45 | '2020/12/04' | 109 | 175 |
| 4 | 45 | '2020/12/05' | 117 | 148 |
| 5 | 60 | '2020/12/06' | 102 | 127 |
| 6 | 60 | '2020/12/07' | 110 | 136 |
| 7 | 45 | '2020/12/08' | 104 | 134 |
| 8 | 30 | '2020/12/09' | 109 | 133 |
| 9 | 60 | '2020/12/10' | 98 | 124 |
| 10 | 60 | '2020/12/11' | 103 | 147 |
| 11 | 60 | '2020/12/12' | 100 | 120 |
| 12 | 60 | '2020/12/12' | 100 | 120 |
| 13 | 60 | '2020/12/13' | 106 | 128 |
| 14 | 60 | '2020/12/14' | 104 | 132 |
| 15 | 60 | '2020/12/15' | 98 | 123 |
| 16 | 60 | '2020/12/16' | 98 | 120 |
| 17 | 60 | '2020/12/17' | 100 | 120 |
| 18 | 45 | '2020/12/18' | 90 | 112 |
| 19 | 60 | '2020/12/19' | 103 | 123 |
| 20 | 45 | '2020/12/20' | 97 | 125 |
| 21 | 60 | '2020/12/21' | 108 | 131 |

| | | | | | |
|----|----|--------------|-----|-----|-------|
| 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 |

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:

```
import pandas as pd

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

for x in df.index:

    if df.loc[x, "Duration"] > 120:

        df.loc[x, "Duration"] = 120

print(df.to_string())
```

| | 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 | 120 | '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 |

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:

```
import pandas as pd

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

for x in df.index:

    if df.loc[x, "Duration"] > 120:

        df.drop(x, inplace = True)

#remember to include the 'inplace = True' argument to make the changes in the original
#DataFrame object instead of returning a copy

print(df.to_string())
```

| 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 |
| 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 |

Pandas - Removing

Duplicates

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, otherwise `False`

```
import pandas as pd

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

print(df.duplicated())
```

| | |
|---|-------|
| 0 | False |
| 1 | False |
| 2 | False |

```
3    False
4    False
5    False
6    False
7    False
8    False
9    False
10   False
11   False
12   True
13   False
14   False
15   False
16   False
17   False
18   False
19   False
```

Removing Duplicates

To remove duplicates, use the `drop_duplicates()` method.

Example

Remove all duplicates:

```
import pandas as pd

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

df.drop_duplicates(inplace = True)

print(df.to_string())
```

#Notice that row 12 has been removed from the 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 |
| 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 |

Read Excel File

```
#pip install xlrd
```

#Python xlrd retrieves data from a spreadsheet using the xlrd module. It is used to read, write, or modify data.

```
import pandas as pd
df = pd.read_excel('SampleExcelFile.xlsx')
print(df)
```

Openpyxl is a Python library that is used **to read from an Excel file or write to an Excel file**. Data scientists use Openpyxl for data analysis, data copying, data mining, drawing charts, styling sheets, adding formulas, and more. Workbook: A spreadsheet is represented as a workbook in openpyxl.

```
import openpyxl  
# Define variable to load the dataframe  
dataframe = openpyxl.load_workbook("SampleExcelFile.xlsx")  
# Define variable to read sheet  
dataframe1 = dataframe.active  
# Iterate the loop to read the cell values  
for row in range(0, dataframe1.max_row):  
    for col in dataframe1.iter_cols(1, dataframe1.max_column):  
        print(col[row].value)
```

Xlwings is a Python library that **makes it easy to call Python from Excel and vice versa**. It creates reading and writing to and from Excel using Python easily. It can also be modified to act as a Python Server for Excel to synchronously exchange data between Python and Excel

```
import xlwings as xw  
# Specifying a sheet  
ws = xw.Book("SampleExcelFile.xlsx").sheets['Sheet1']  
# Selecting data from  
# a single cell  
v1 = ws.range("A1:A7").value  
v2 = ws.range("F5").value  
print("Result:", v1, v2)
```

Pandas - Data 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.csv'.

[Download data.csv](#), or [Open data.csv](#)

Example

Show the relationship between the columns:

```
import pandas as pd  
  
df = pd.read_csv('data.csv')  
  
print(df.corr())
```

| | Duration | Pulse | Maxpulse | Calories |
|----------|-----------|-----------|-----------|----------|
| Duration | 1.000000 | -0.059452 | -0.250033 | 0.344341 |
| Pulse | -0.059452 | 1.000000 | 0.269672 | 0.481791 |
| Maxpulse | -0.250033 | 0.269672 | 1.000000 | 0.335392 |
| Calories | 0.344341 | 0.481791 | 0.335392 | 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.

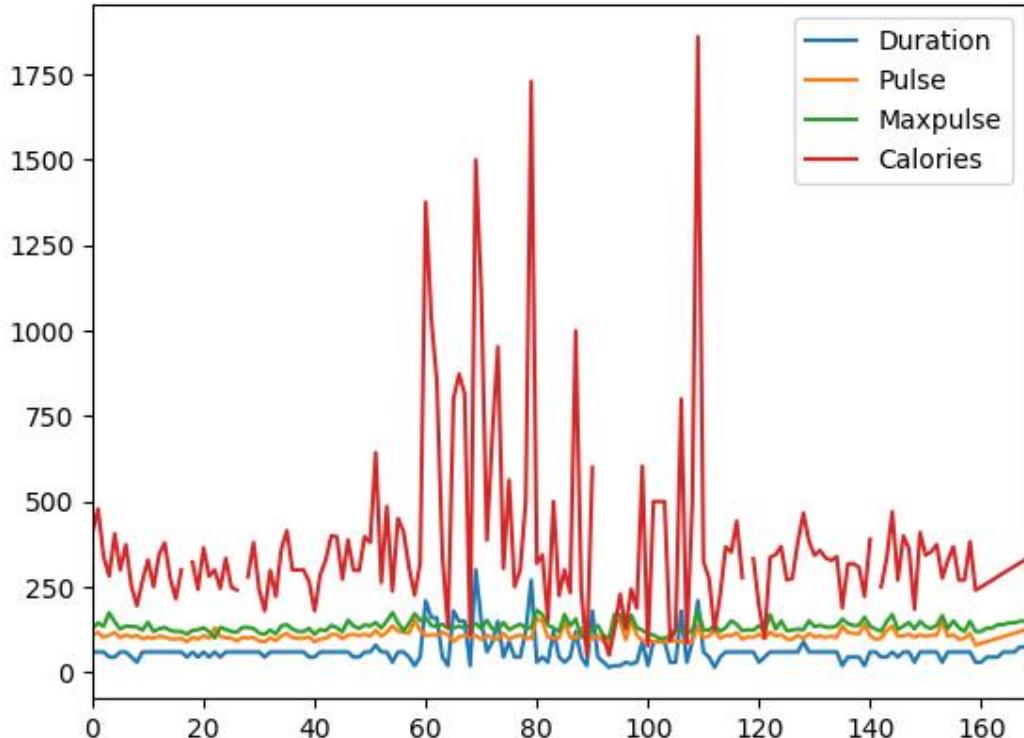
Good Correlation:

"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



Plotting

Pandas uses the `plot()` method to create diagrams.

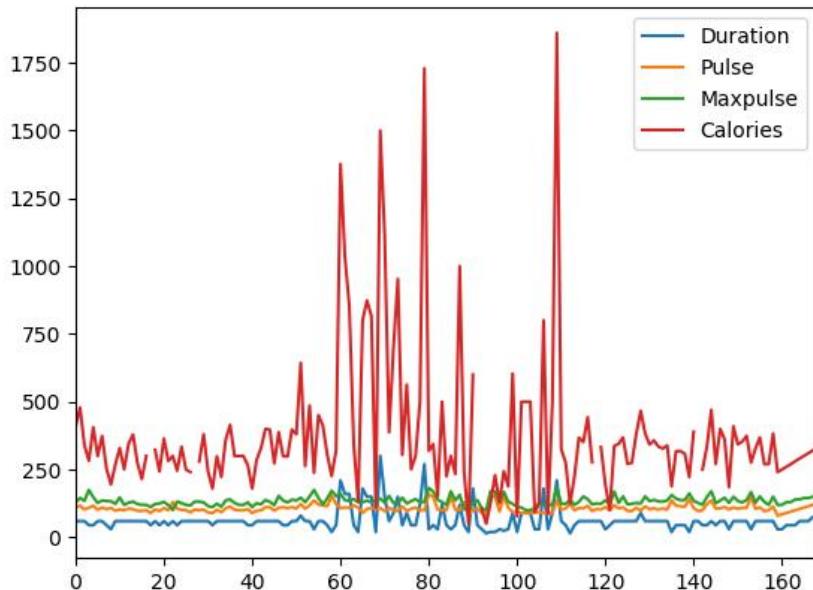
We can use Pyplot, a submodule of the Matplotlib library to visualize the diagram on the screen.

Read more about Matplotlib in our [Matplotlib Tutorial](#).

Example

Import pyplot from Matplotlib and visualize our DataFrame:

```
import matplotlib  
  
import pandas as pd  
  
import matplotlib.pyplot as plt  
  
df = pd.read_csv('data.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.

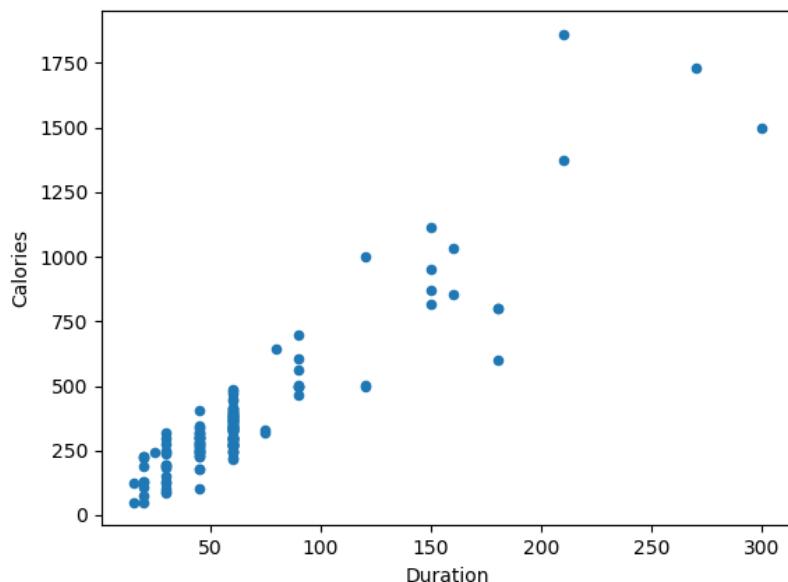
In the example below we will use "Duration" for the x-axis and "Calories" for the y-axis.

Include the x and y arguments like this:

```
x = 'Duration', y = 'Calories'
```

Example

```
#Three lines to make our compiler able to draw:  
  
import matplotlib  
  
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()
```



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.

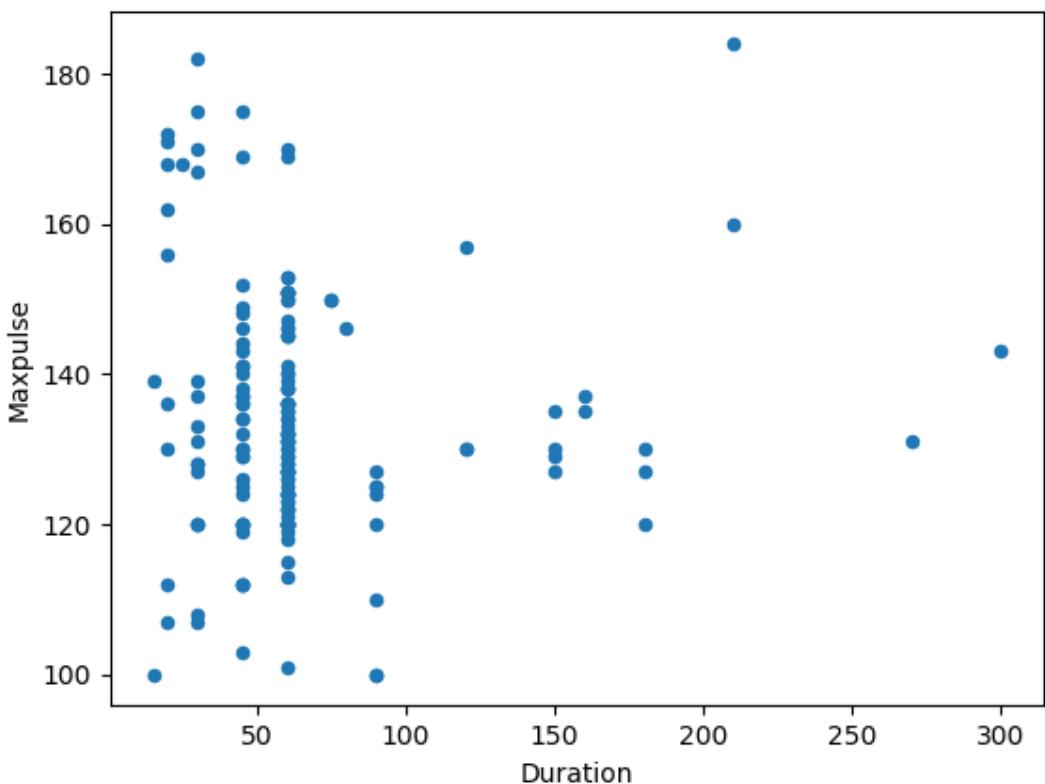
Let's create another scatterplot, where there is a bad relationship between the columns, like "Duration" and "Maxpulse", with the correlation **0.009403**:

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()
```



Histogram

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

```
kind = 'hist'
```

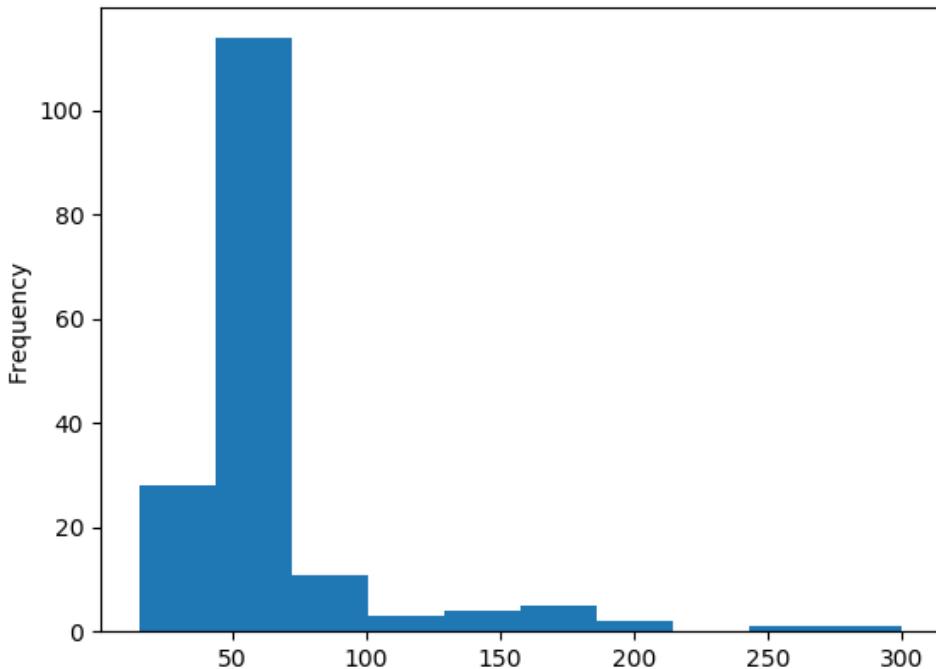
A histogram needs only one column.

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

```
import matplotlib
import pandas as pd
import matplotlib.pyplot as plt
df = pd.read_csv('data.csv')
df["Duration"].plot(kind = 'hist')
plt.show()
```



Data Visualization in python using Matplotlib

What is Matplotlib?

Matplotlib is a low level graph plotting library in python that serves as a visualization utility.

Matplotlib was created by John D. Hunter.

Matplotlib is open source and we can use it freely.

Matplotlib is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Where is the Matplotlib Codebase?

The source code for Matplotlib is located at this github repository <https://github.com/matplotlib/matplotlib>

Matplotlib Getting Started

Installation of Matplotlib

If you have [Python](#) and [PIP](#) already installed on a system, then installation of Matplotlib is very easy.

Install it using this command:

```
C:\Users\Your Name>pip install matplotlib
```

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

Import Matplotlib

Once Matplotlib is installed, import it in your applications by adding the `import module` statement:

```
import matplotlib
```

Now Matplotlib is imported and ready to use:

Checking Matplotlib Version

The version string is stored under `__version__` attribute.

Example

```
import matplotlib  
  
print(matplotlib.__version__)
```

Matplotlib Pyplot

Pyplot

Most of the Matplotlib utilities lies under the `pypot` submodule, and are usually imported under the `plt` alias:

```
import matplotlib.pyplot as plt
```

Now the Pyplot package can be referred to as `plt`.

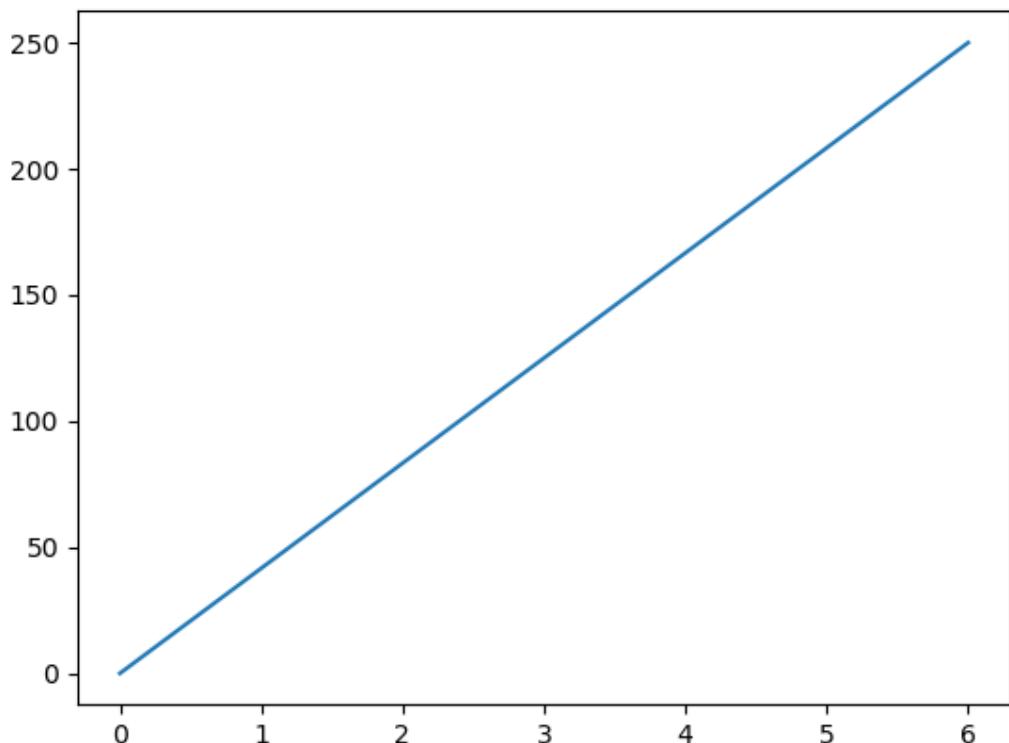
Example

Draw a line in a diagram from position (0,0) to position (6,250):

```
import matplotlib.pyplot as plt  
import numpy as np  
  
xpoints = np.array([0, 6])  
ypoints = np.array([0, 250])
```

```
plt.plot(xpoints, ypoints)  
plt.show()
```

Result:



Matplotlib Plotting

Plotting x and y points

The `plot()` function is used to draw points (markers) in a diagram.

By default, the `plot()` function draws a line from point to point.

The function takes parameters for specifying points in the diagram.

Parameter 1 is an array containing the points on the **x-axis**.

Parameter 2 is an array containing the points on the **y-axis**.

If we need to plot a line from (1, 3) to (8, 10), we have to pass two arrays [1, 8] and [3, 10] to the plot function.

Example

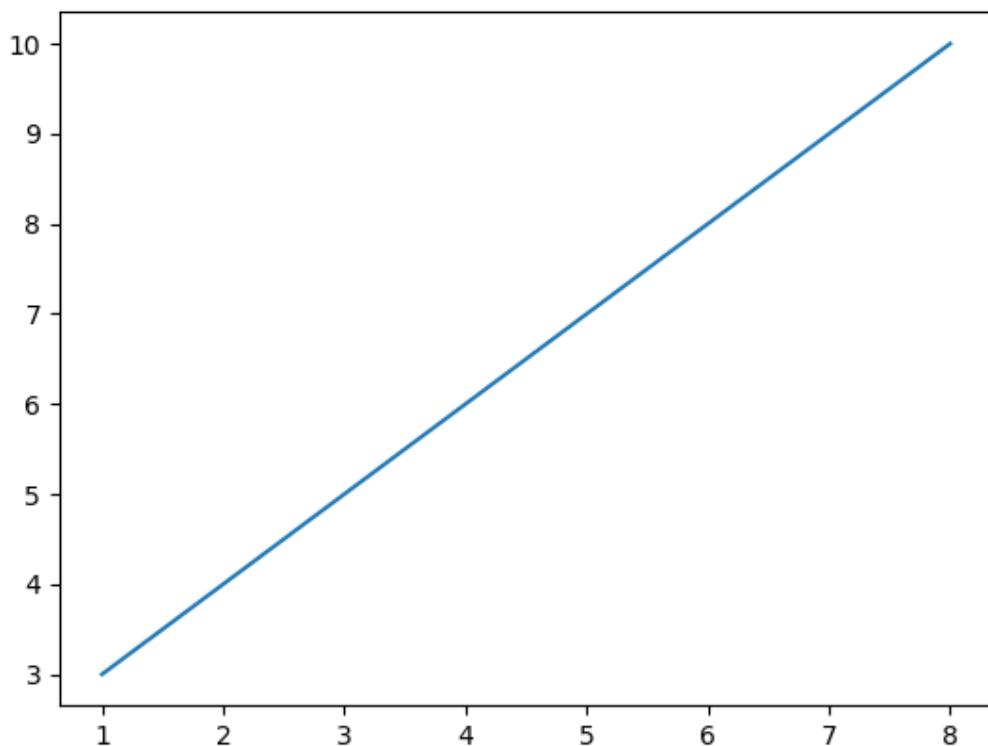
Draw a line in a diagram from position (1, 3) to position (8, 10):

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 8])
y whole points = np.array([3, 10])

plt.plot(xpoints, y whole points)
plt.show()
```

Result:



Plotting Without Line

To plot only the markers, you can use *shortcut string notation* parameter 'o', which means 'rings'.

Example

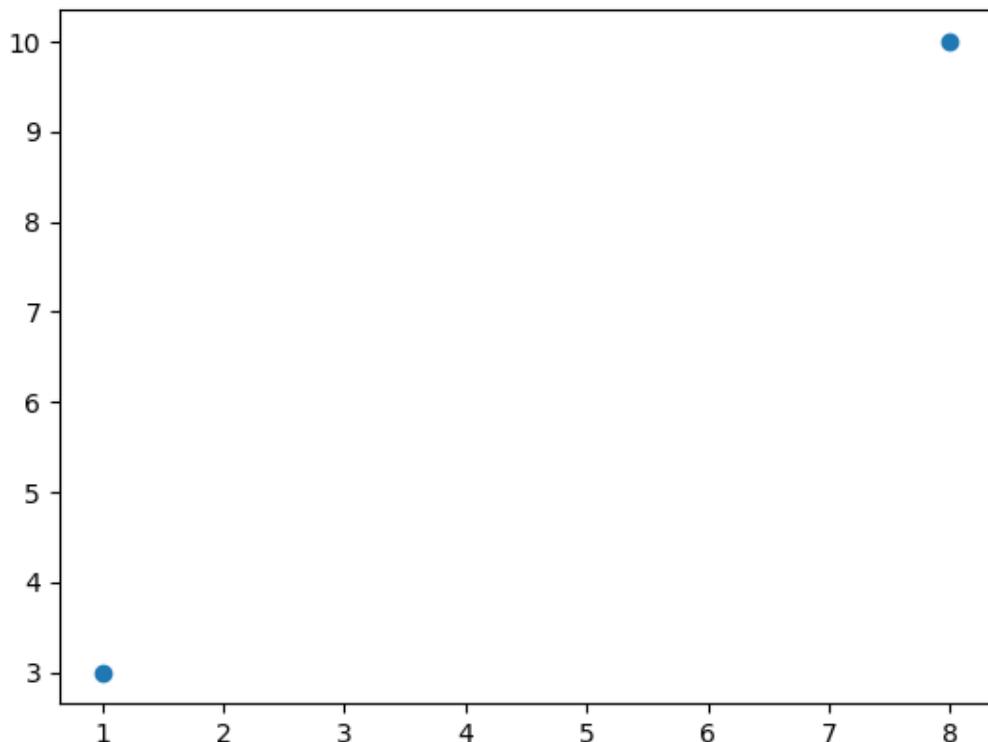
Draw two points in the diagram, one at position (1, 3) and one in position (8, 10):

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 8])
ypoints = np.array([3, 10])

plt.plot(xpoints, ypoints, 'o')
plt.show()
```

Result:



Multiple Points

You can plot as many points as you like, just make sure you have the same number of points in both axis.

Example

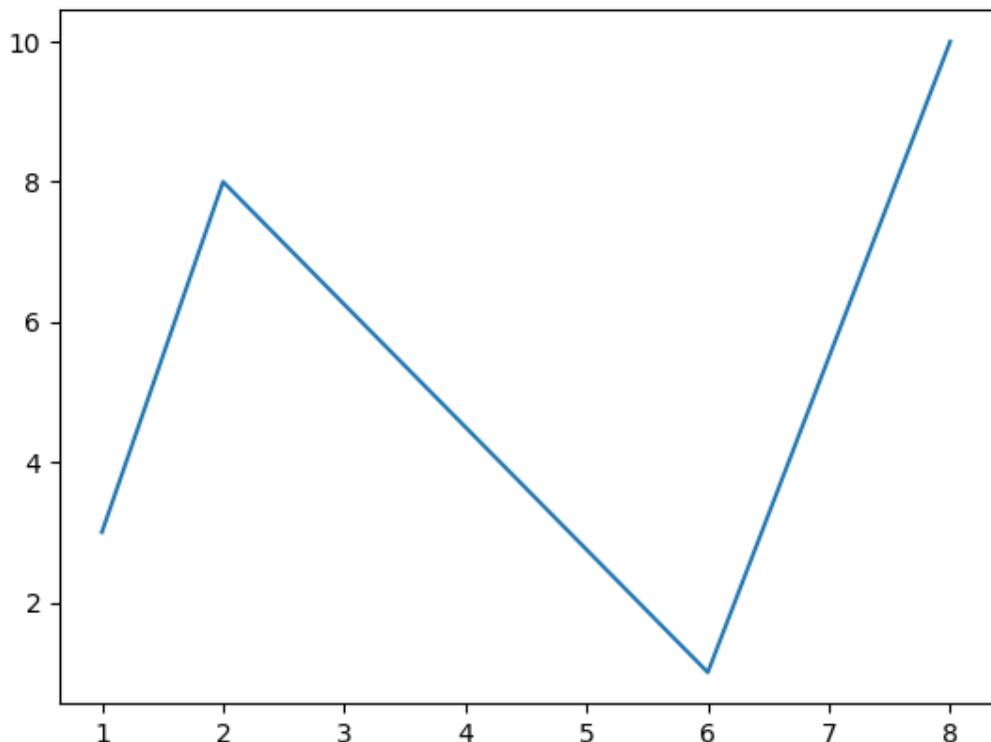
Draw a line in a diagram from position (1, 3) to (2, 8) then to (6, 1) and finally to position (8, 10):

```
import matplotlib.pyplot as plt
import numpy as np

xpoints = np.array([1, 2, 6, 8])
ypoints = np.array([3, 8, 1, 10])

plt.plot(xpoints, ypoints)
plt.show()
```

Result:



Default X-Points

If we do not specify the points in the x-axis, they will get the default values 0, 1, 2, 3, (etc. depending on the length of the y-points).

So, if we take the same example as above, and leave out the x-points, the diagram will look like this:

Example

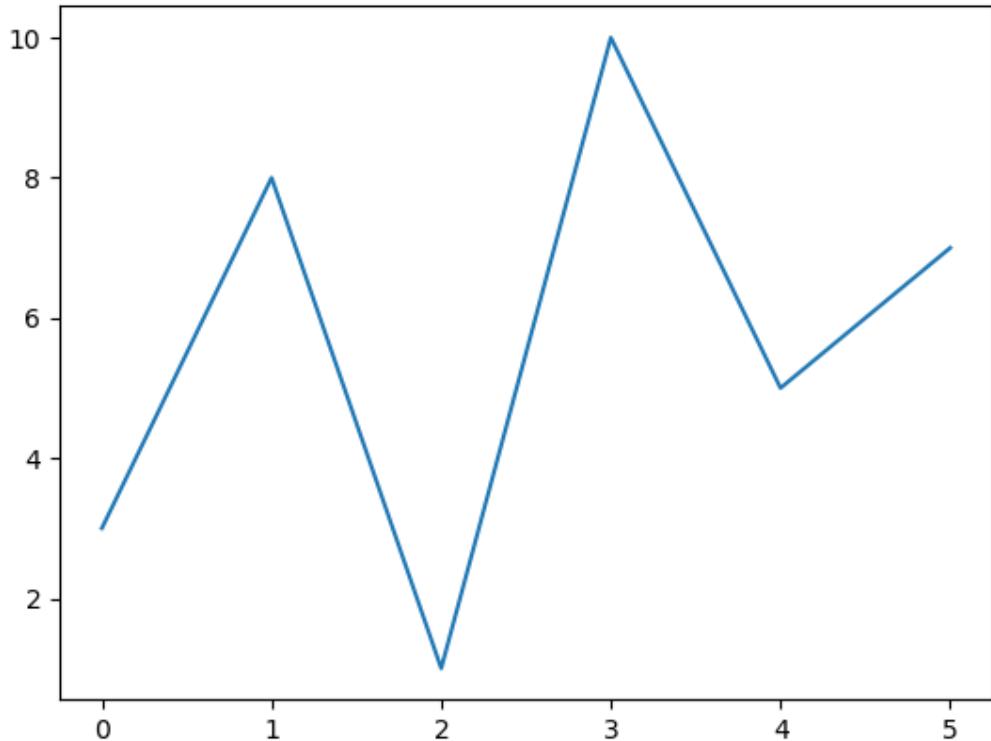
Plotting without x-points:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10, 5, 7])

plt.plot(ypoints)
plt.show()
```

Result:



Matplotlib Markers

Markers

You can use the keyword argument `marker` to emphasize each point with a specified marker:

Example

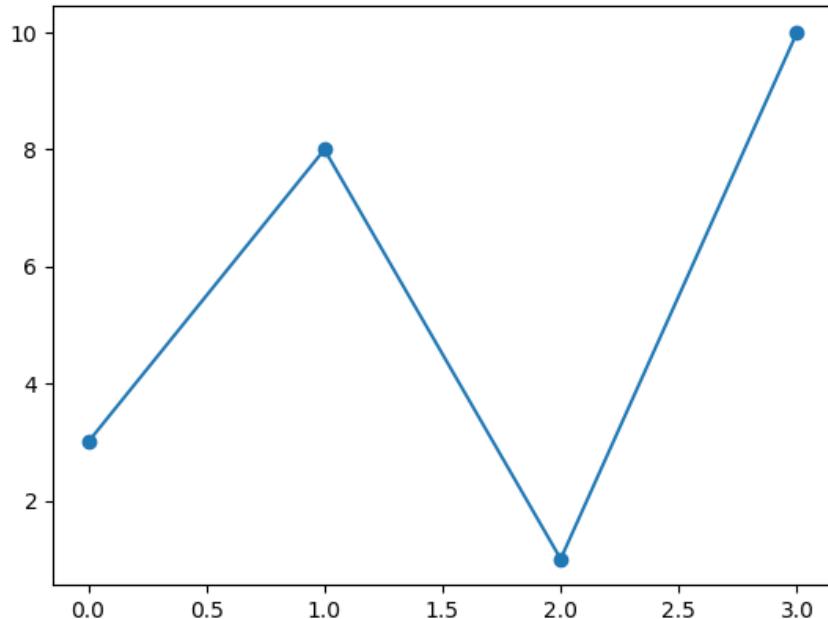
Mark each point with a circle:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o')
plt.show()
```

Result:

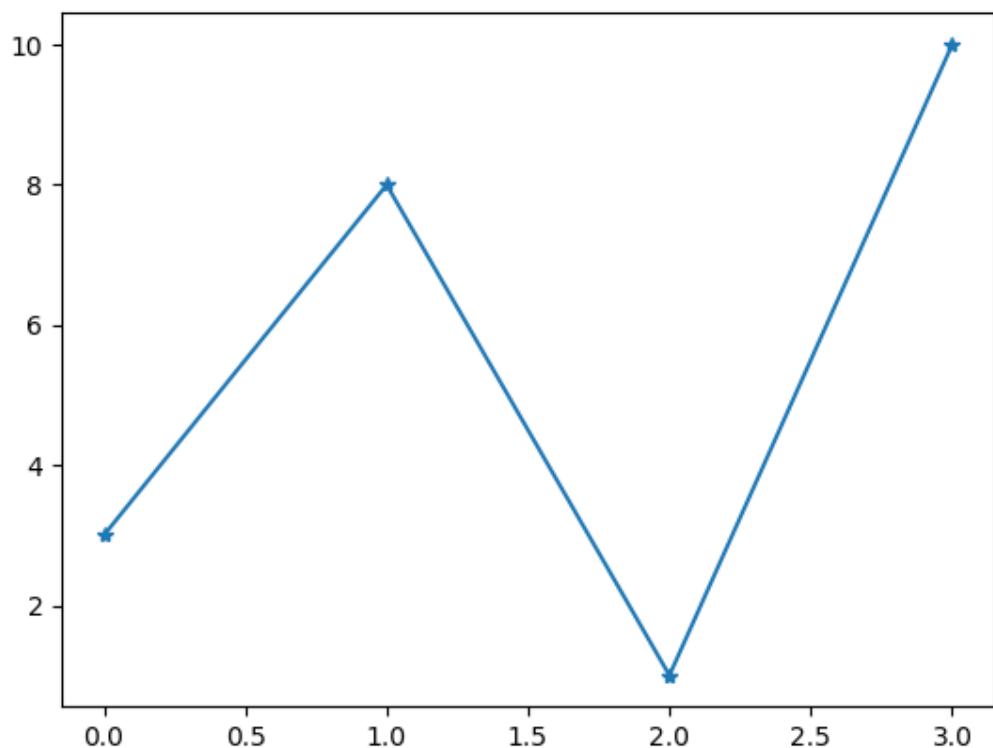


Example

Mark each point with a star:

```
...
plt.plot(ypoints, marker = '*')
...
```

Result:



Format Strings `fmt`

You can also use the *shortcut string notation* parameter to specify the marker.

This parameter is also called `fmt`, and is written with this syntax:

`marker|Line|color`

Example

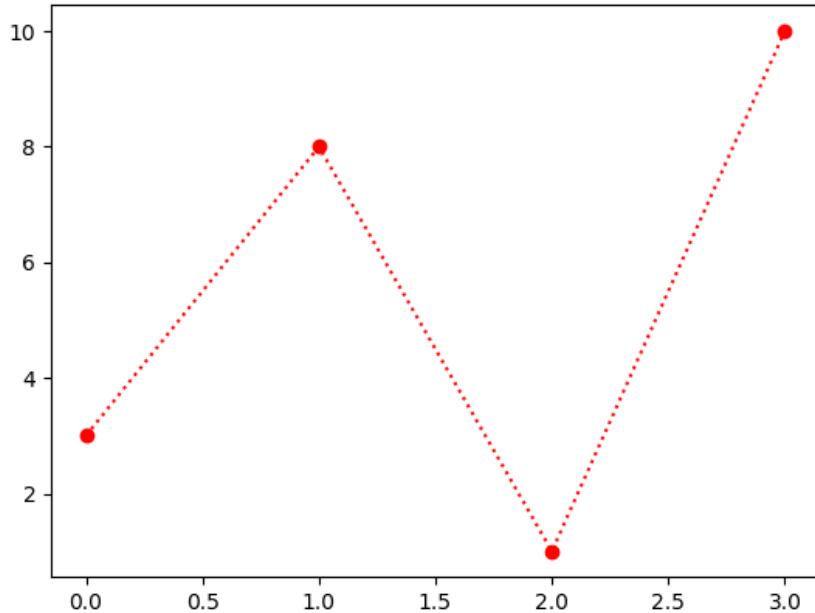
Mark each point with a circle:

```
import matplotlib.pyplot as plt
import numpy as np

y whole points = np.array([3, 8, 1, 10])

plt.plot(y whole points, 'o:r')
plt.show()
```

Result:



The marker value can be anything from the Marker Reference above.

The line value can be one of the following:

Line Reference

Line Syntax

'-' Solid line

:: Dotted line

'--' Dashed line

'-.-' Dashed/dotted line

Note: If you leave out the *line* value in the `fmt` parameter, no line will be plotted.

The short color value can be one of the following:

Color Reference

Color Syntax

'r' Red

'g' Green

| | |
|-----|---------|
| 'b' | Blue |
| 'c' | Cyan |
| 'm' | Magenta |
| 'y' | Yellow |
| 'k' | Black |
| 'w' | White |

Marker Size

You can use the keyword argument `markersize` or the shorter version, `ms` to set the size of the markers:

Example

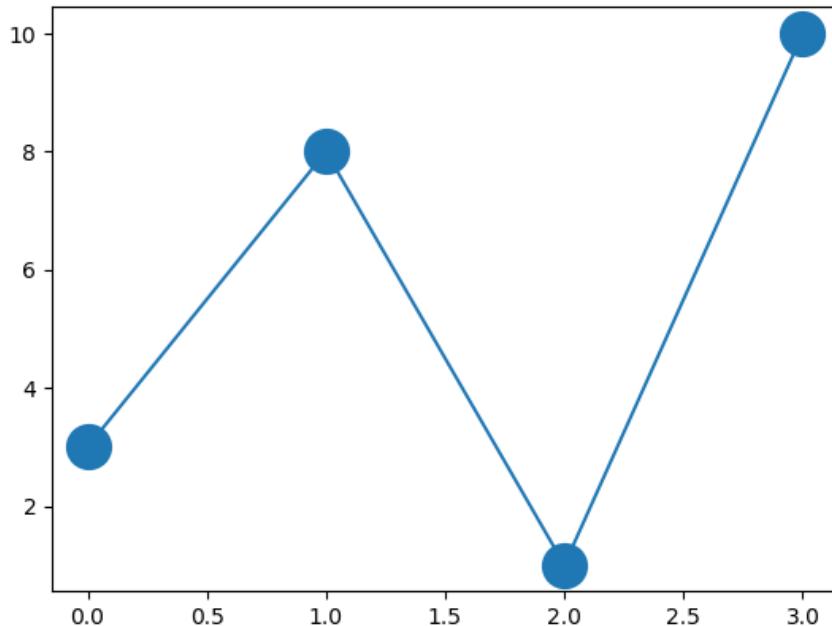
Set the size of the markers to 20:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20)
plt.show()
```

Result:



Marker Color

You can use the keyword argument `markeredgecolor` or the shorter `mec` to set the color of the *edge* of the markers:

Example

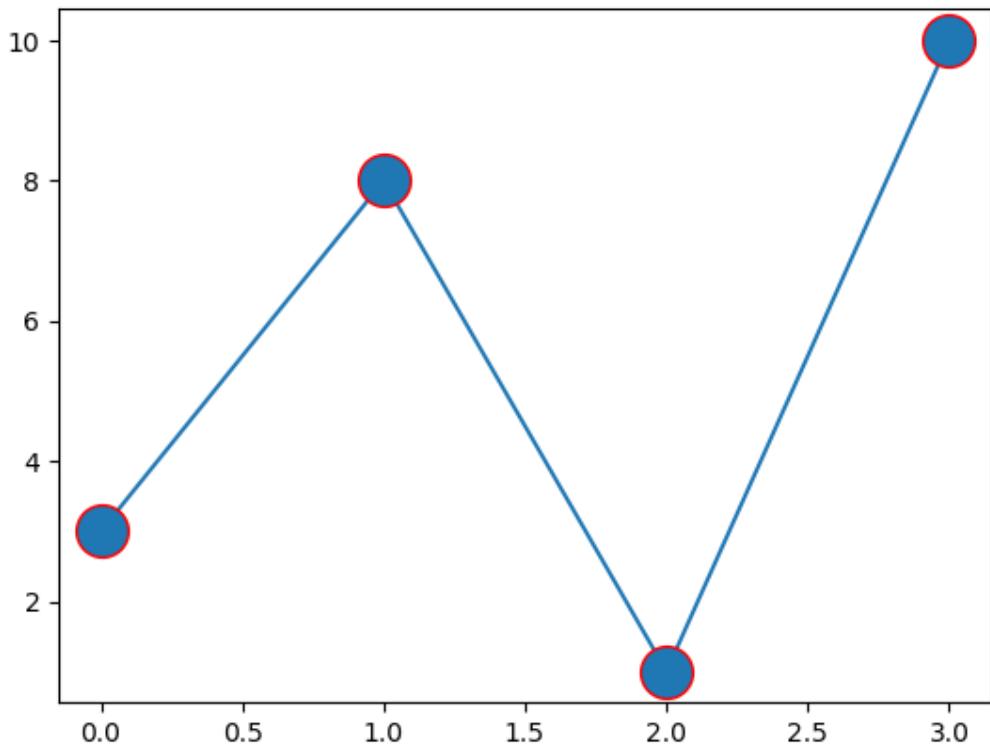
Set the EDGE color to red:

```
import matplotlib.pyplot as plt
import numpy as np

y whole = np.array([3, 8, 1, 10])

plt.plot(y whole, marker = 'o', ms = 20, mec = 'r')
plt.show()
```

Result:



You can use the keyword argument `markerfacecolor` or the shorter `mfc` to set the color inside the edge of the markers:

Example

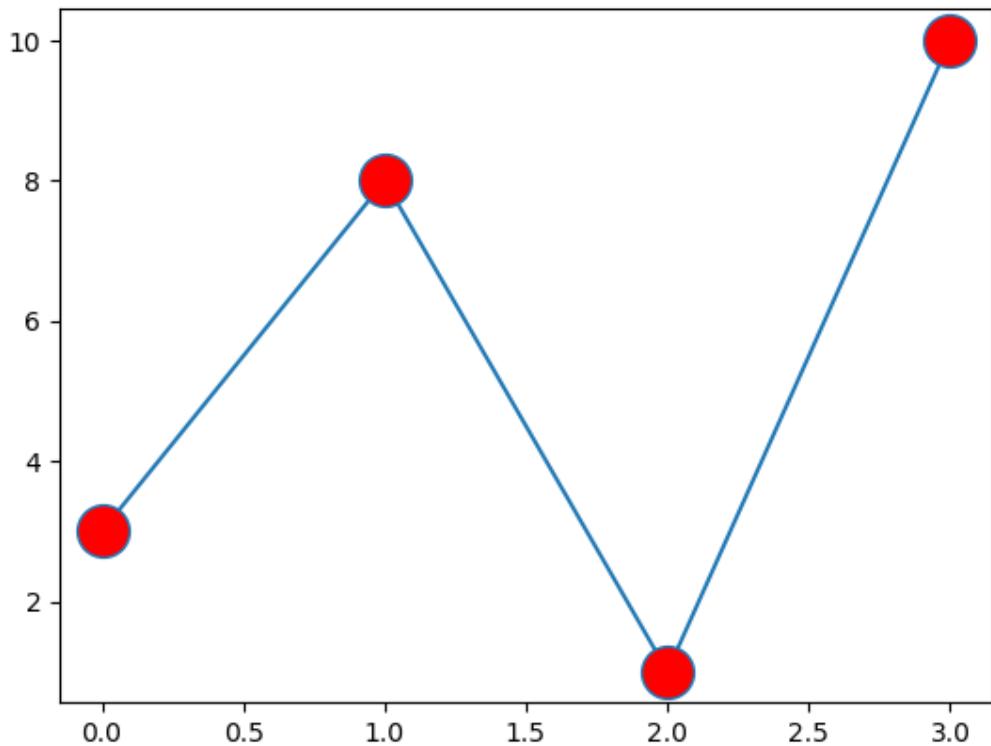
Set the FACE color to red:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mfc = 'r')
plt.show()
```

Result:



Use *both* the `mec` and `mfc` arguments to color of the entire marker:

Example

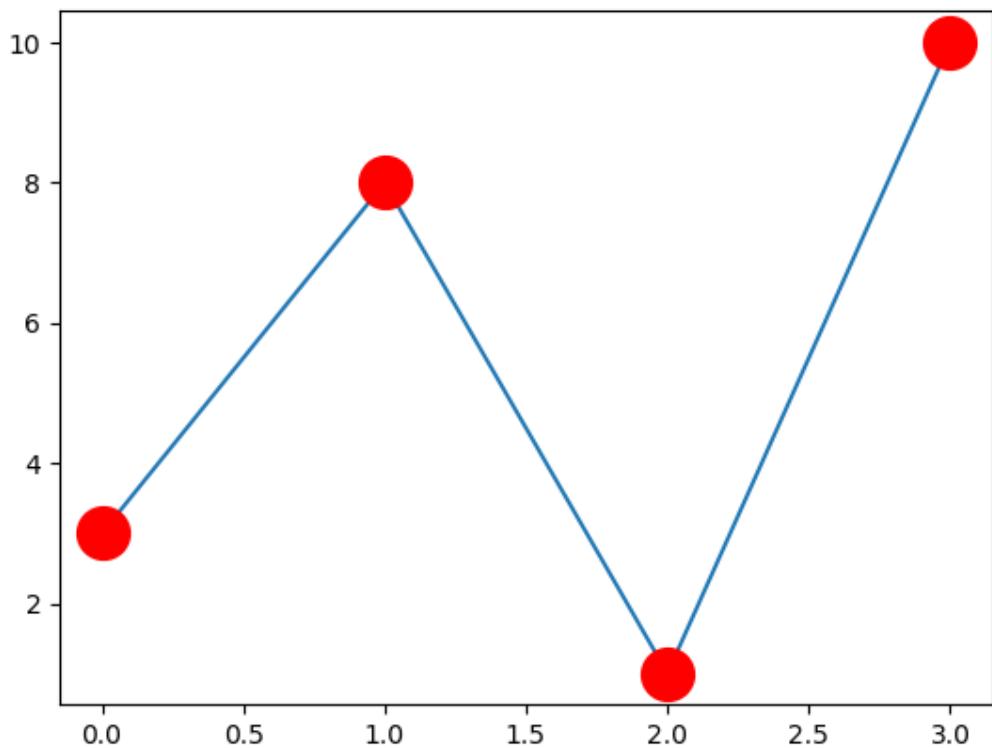
Set the color of both the *edge* and the *face* to red:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, marker = 'o', ms = 20, mec = 'r', mfc = 'r')
plt.show()
```

Result:



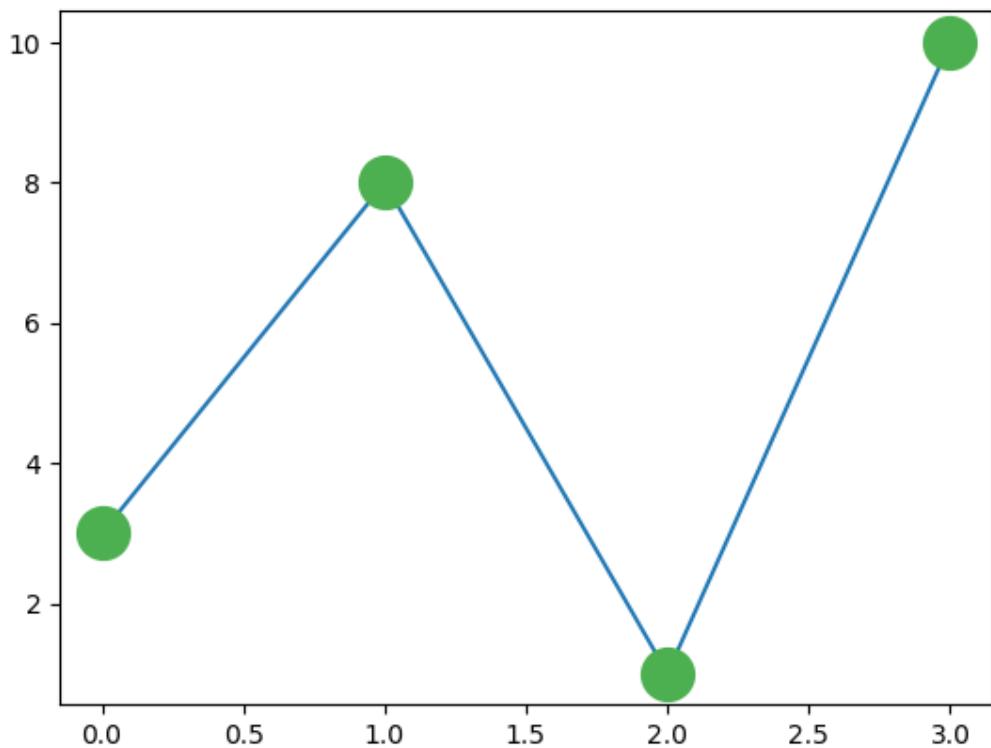
You can also use [Hexadecimal color values](#):

Example

Mark each point with a beautiful green color:

```
...
plt.plot(ypoints, marker = 'o', ms = 20, mec = '#4CAF50', mfc
= '#4CAF50')
...
```

Result:



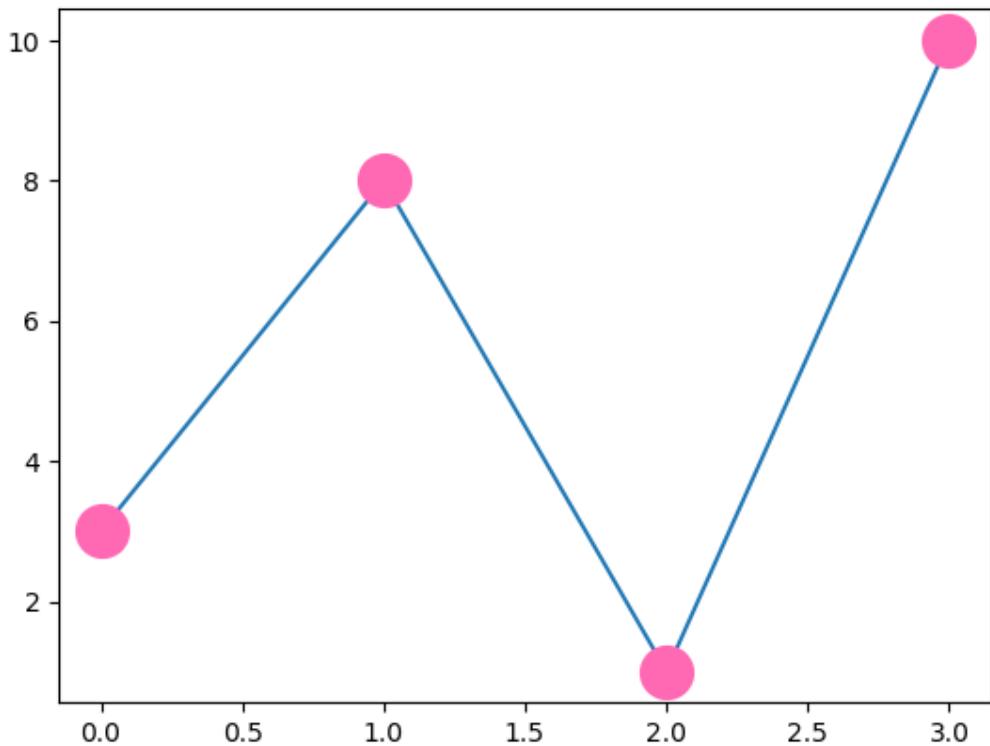
Or any of the [140 supported color names](#).

Example

Mark each point with the color named "hotpink":

```
...
plt.plot(ypoints, marker = 'o', ms = 20, mec = 'hotpink', mfc
= 'hotpink')
...
```

Result:



Matplotlib Line

Linestyle

You can use the keyword argument `linestyle`, or shorter `ls`, to change the style of the plotted line:

Example

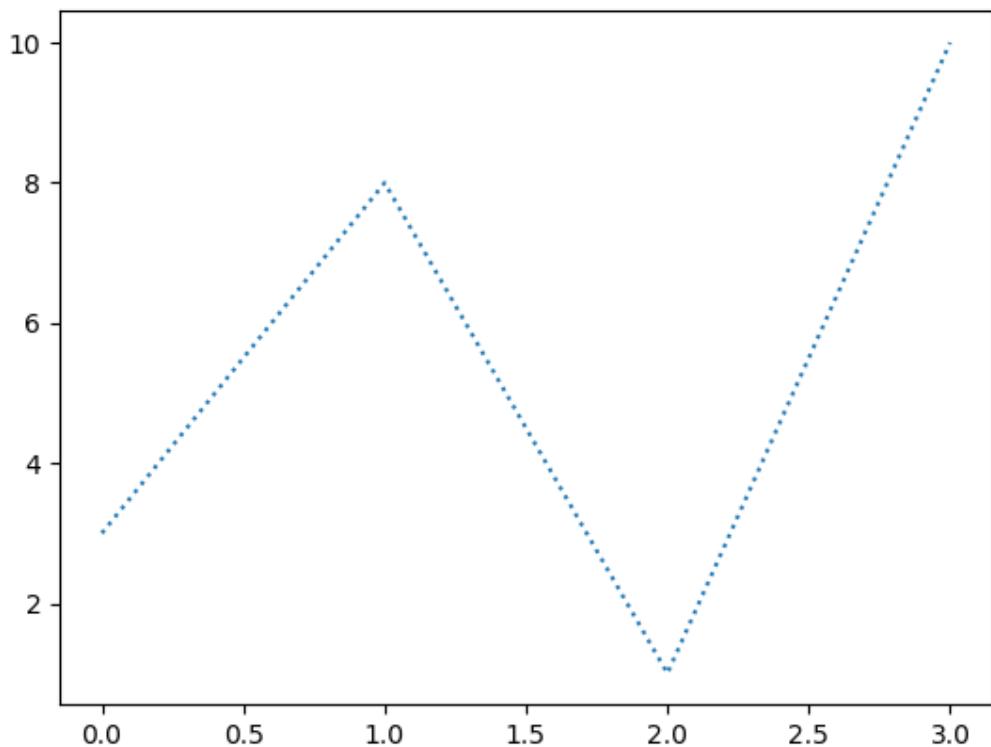
Use a dotted line:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linestyle = 'dotted')
plt.show()
```

Result:

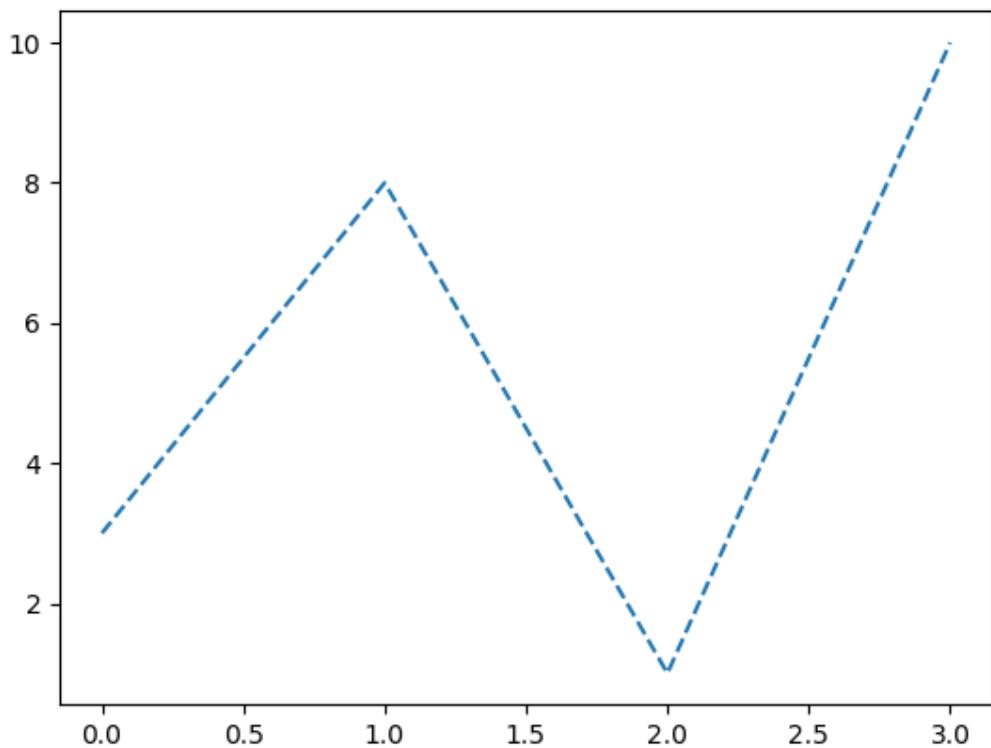


Example

Use a dashed line:

```
plt.plot(ypoints, linestyle = 'dashed')
```

Result:



Shorter Syntax

The line style can be written in a shorter syntax:

`linestyle` can be written as `ls`.

`dotted` can be written as `:.`

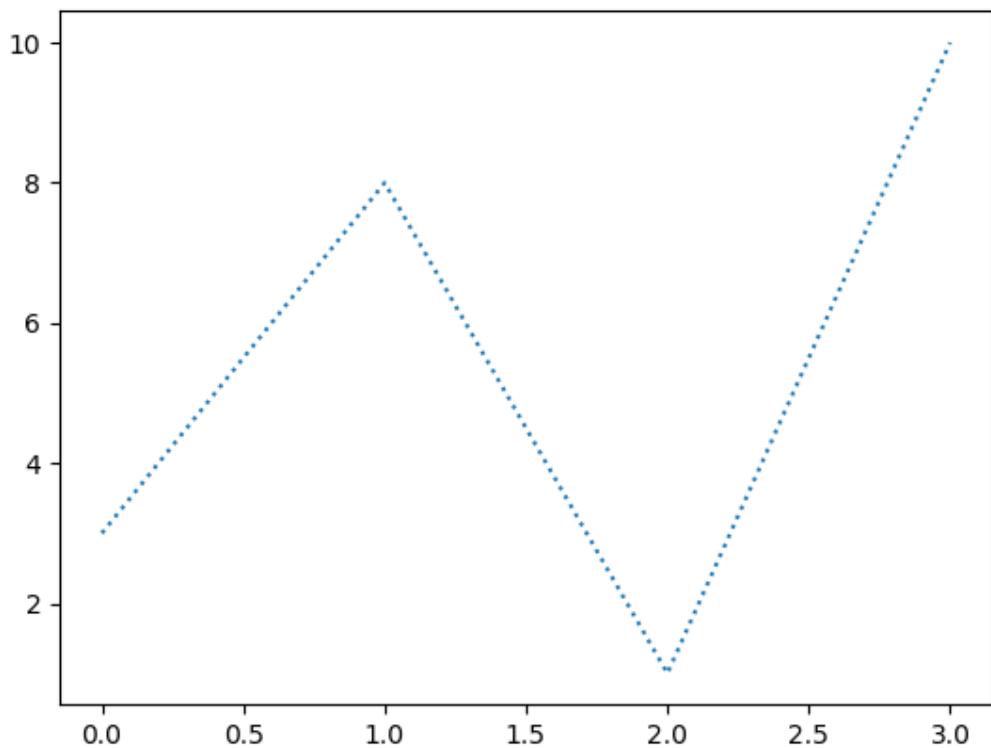
`dashed` can be written as `--`.

Example

Shorter syntax:

```
plt.plot(ypoints, ls = ':')
```

Result:



Line Styles

You can choose any of these styles:

Style

'solid' (default)

'-'

'dotted'

:

'dashed'

'--'

| | |
|-----------|---------|
| 'dashdot' | '-.' |
| 'None' | " or '' |

Line Color

You can use the keyword argument `color` or the shorter `c` to set the color of the line:

Example

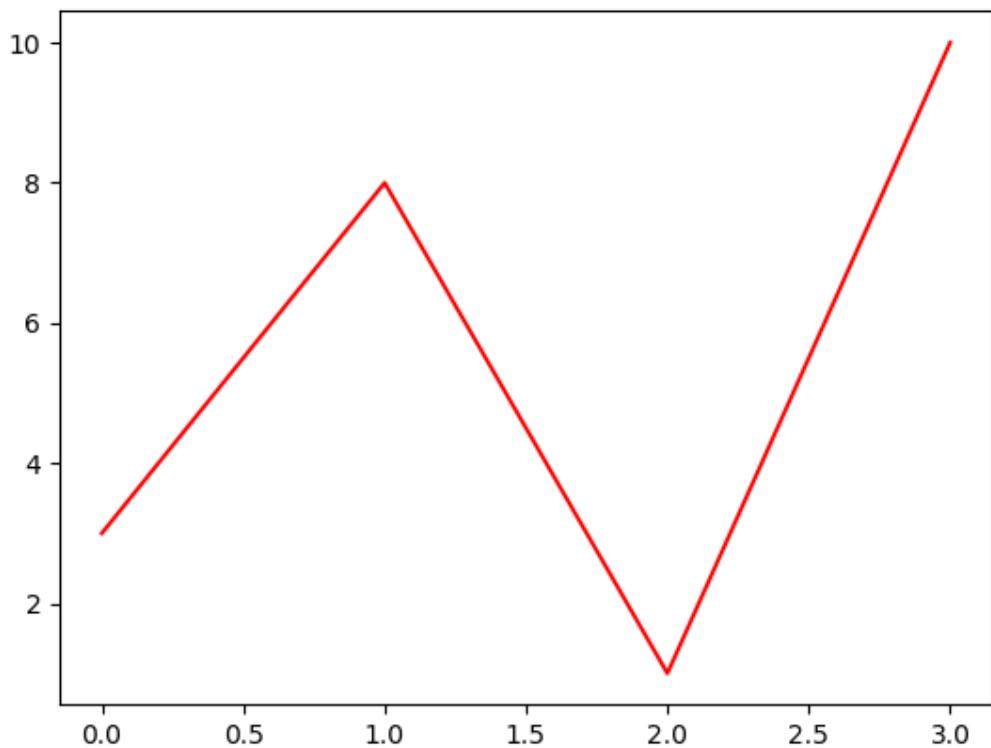
Set the line color to red:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, color = 'r')
plt.show()
```

Result:



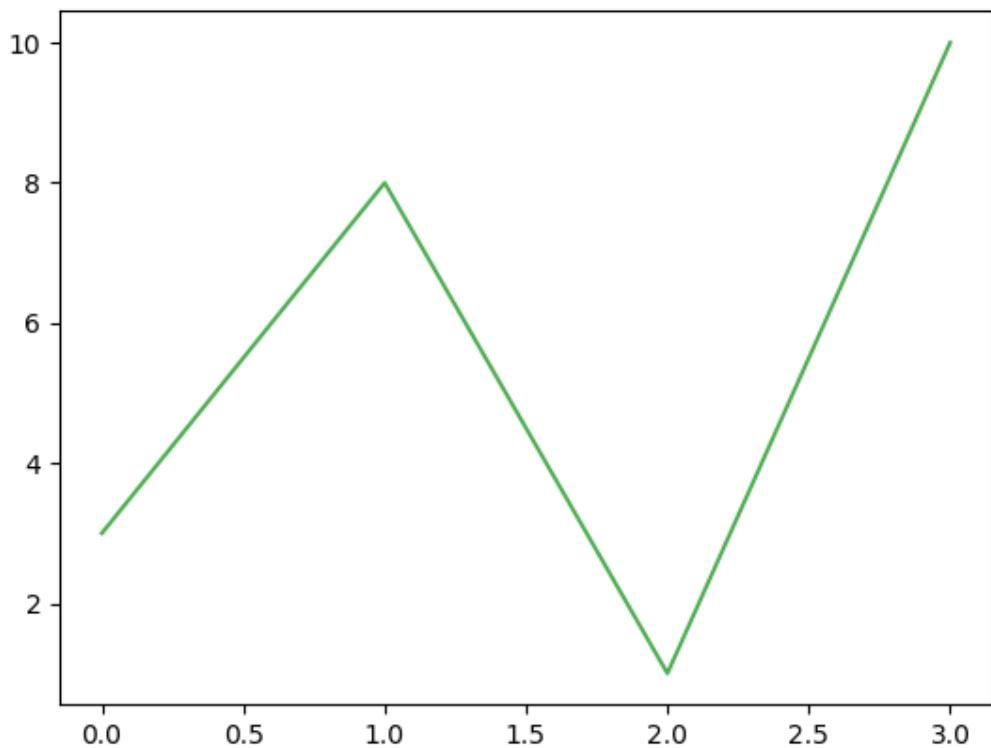
You can also use [Hexadecimal color values](#):

Example

Plot with a beautiful green line:

```
...  
plt.plot(ypoints, c = '#4CAF50')  
...
```

Result:



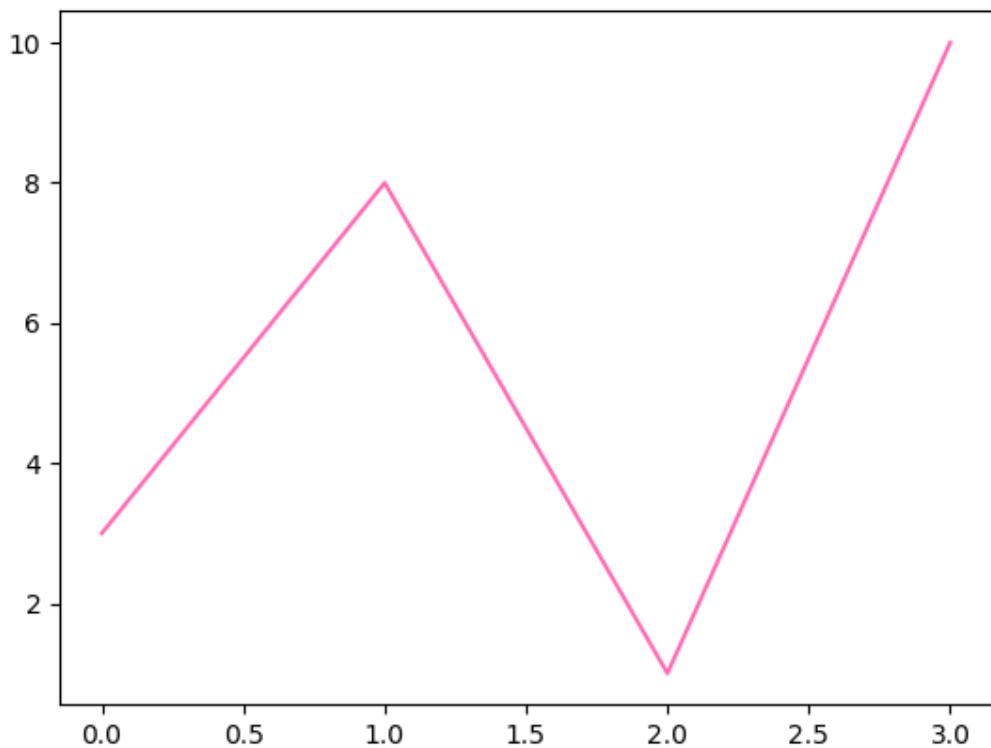
Or any of the [140 supported color names](#).

Example

Plot with the color named "hotpink":

```
...
plt.plot(ypoints, c = 'hotpink')
...
```

Result:



Line Width

You can use the keyword argument `linewidth` or the shorter `lw` to change the width of the line.

The value is a floating number, in points:

Example

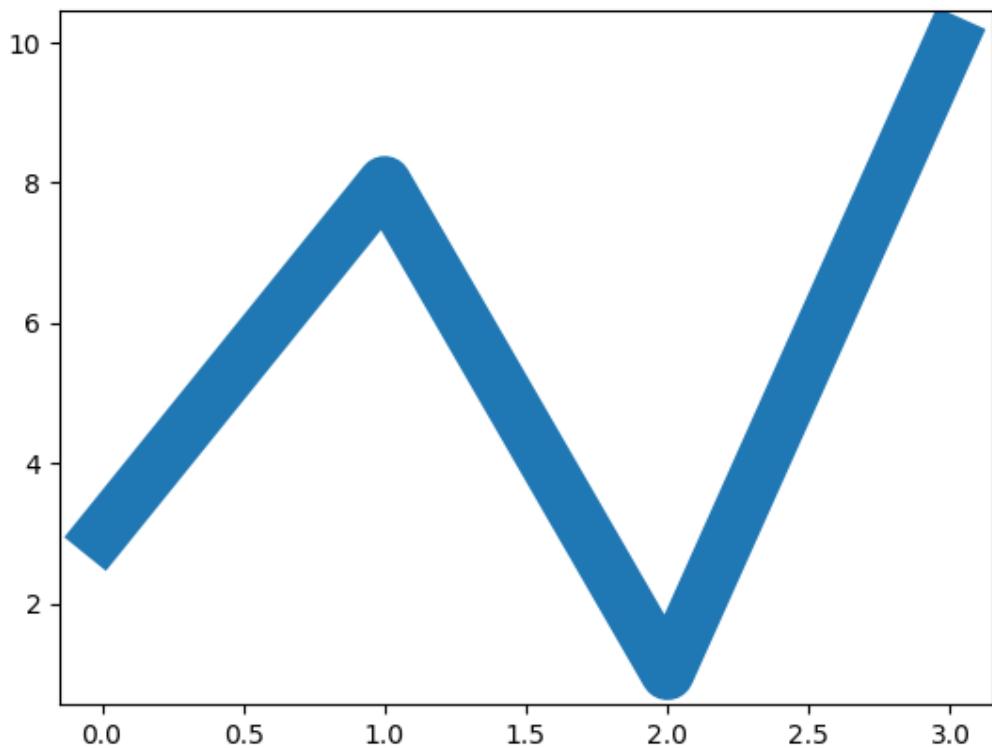
Plot with a 20.5pt wide line:

```
import matplotlib.pyplot as plt
import numpy as np

ypoints = np.array([3, 8, 1, 10])

plt.plot(ypoints, linewidth = '20.5')
plt.show()
```

Result:



Multiple Lines

You can plot as many lines as you like by simply adding more `plt.plot()` functions:

Example

Draw two lines by specifying a `plt.plot()` function for each line:

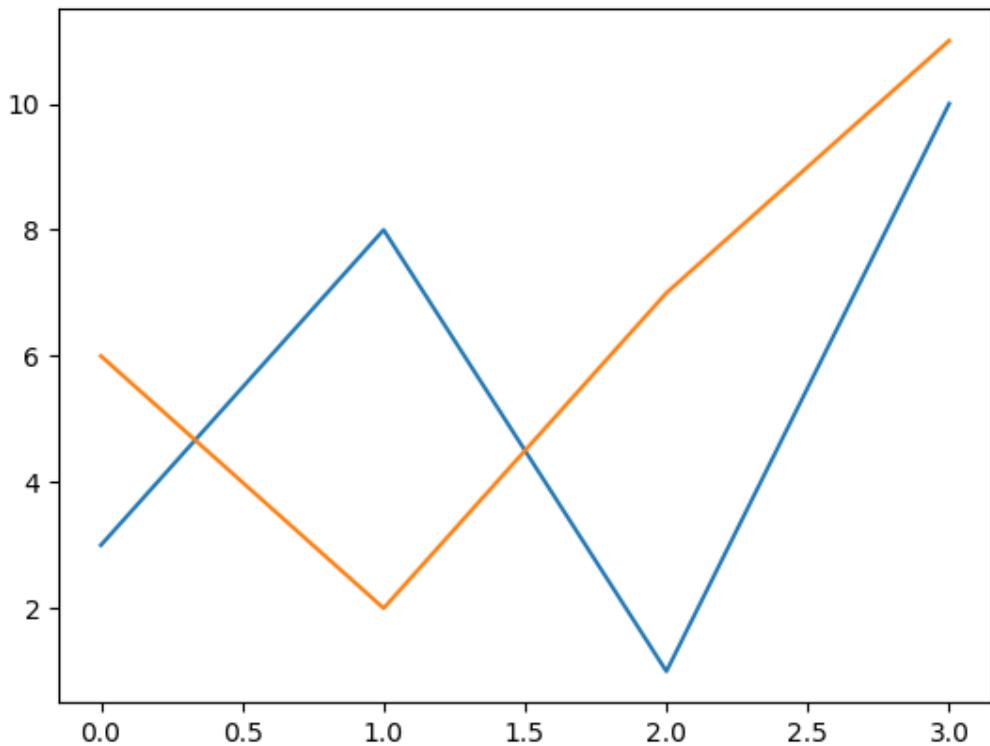
```
import matplotlib.pyplot as plt
import numpy as np

y1 = np.array([3, 8, 1, 10])
y2 = np.array([6, 2, 7, 11])

plt.plot(y1)
plt.plot(y2)

plt.show()
```

Result:



You can also plot many lines by adding the points for the x- and y-axis for each line in the same `plt.plot()` function.

(In the examples above we only specified the points on the y-axis, meaning that the points on the x-axis got the the default values (0, 1, 2, 3).)

The x- and y- values come in pairs:

Example

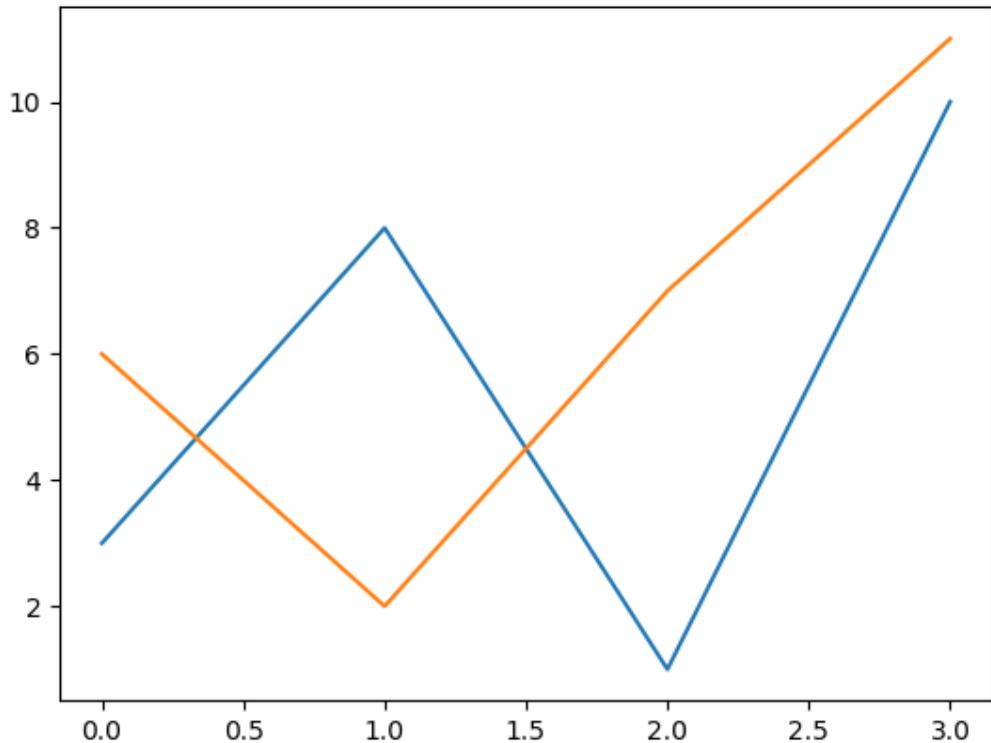
Draw two lines by specifying the x- and y-point values for both lines:

```
import matplotlib.pyplot as plt
import numpy as np

x1 = np.array([0, 1, 2, 3])
y1 = np.array([3, 8, 1, 10])
x2 = np.array([0, 1, 2, 3])
y2 = np.array([6, 2, 7, 11])

plt.plot(x1, y1, x2, y2)
plt.show()
```

Result:



Matplotlib Labels and Title

Create Labels for a Plot

With Pyplot, you can use the `xlabel()` and `ylabel()` functions to set a label for the x- and y-axis.

Example

Add labels to the x- and y-axis:

```
import numpy as np
import matplotlib.pyplot as plt

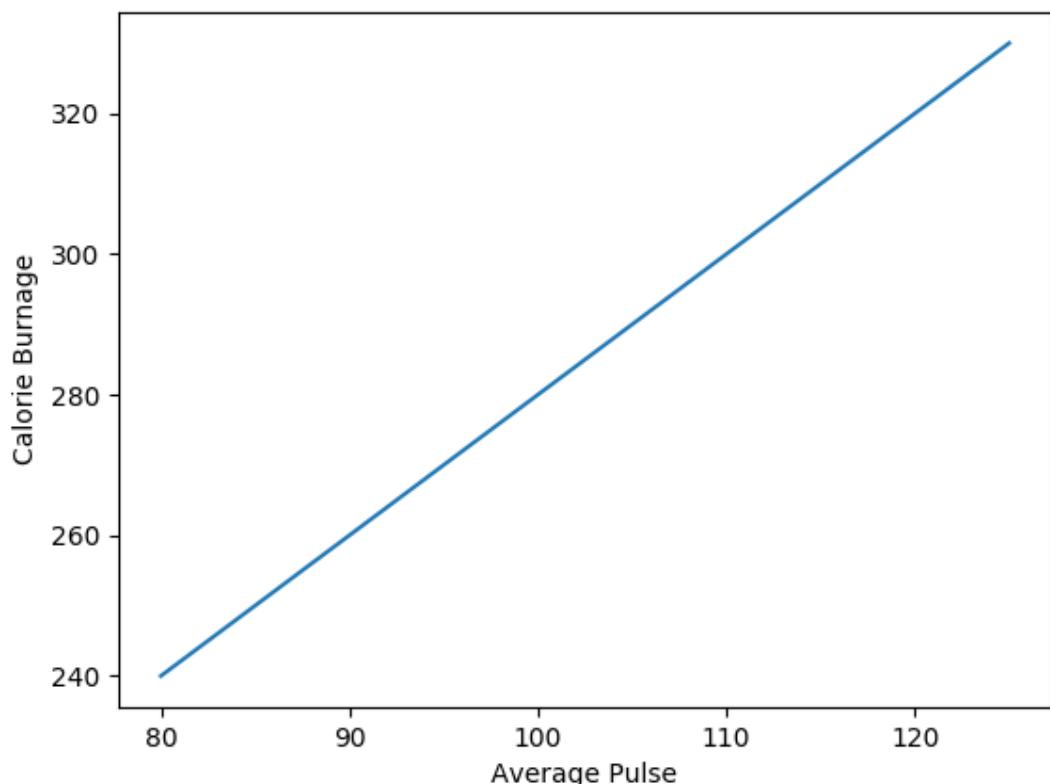
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])
```

```
plt.plot(x, y)

plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

Result:



Create a Title for a Plot

With Pyplot, you can use the `title()` function to set a title for the plot.

Example

Add a plot title and labels for the x- and y-axis:

```
import numpy as np
import matplotlib.pyplot as plt
```

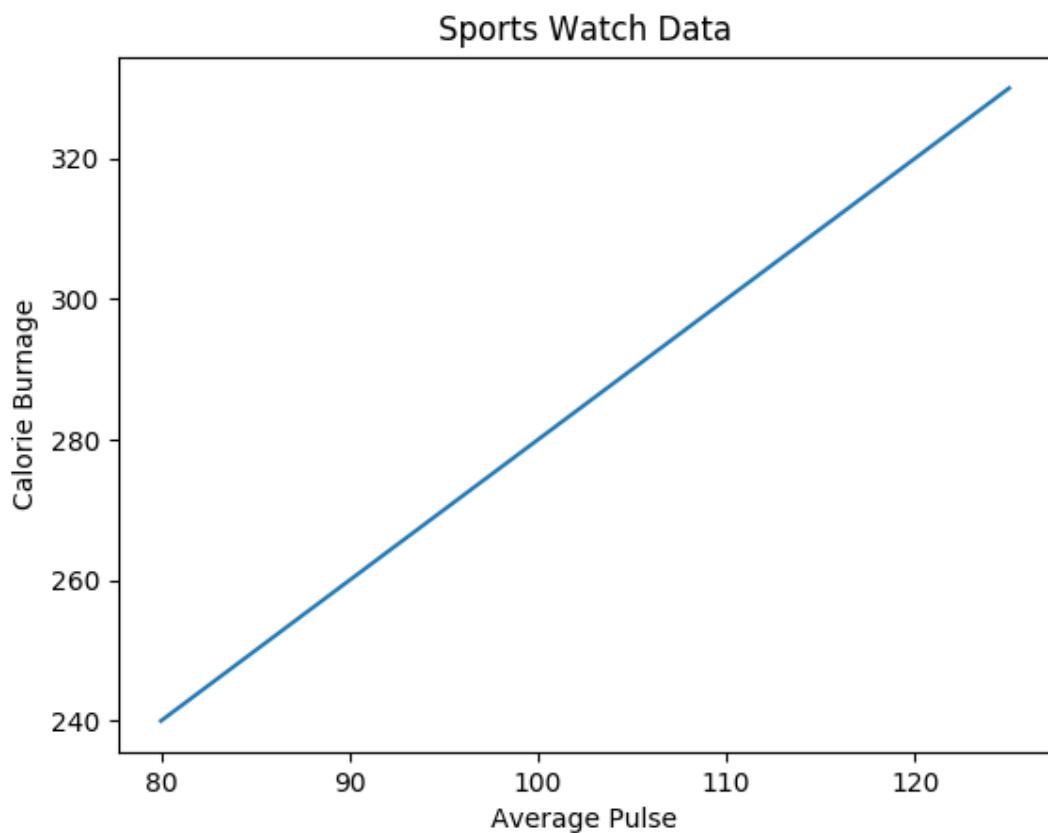
```
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.plot(x, y)

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.show()
```

Result:



Set Font Properties for Title and Labels

You can use the `fontdict` parameter in `xlabel()`, `ylabel()`, and `title()` to set font properties for the title and labels.

Example

Set font properties for the title and labels:

```
import numpy as np
import matplotlib.pyplot as plt

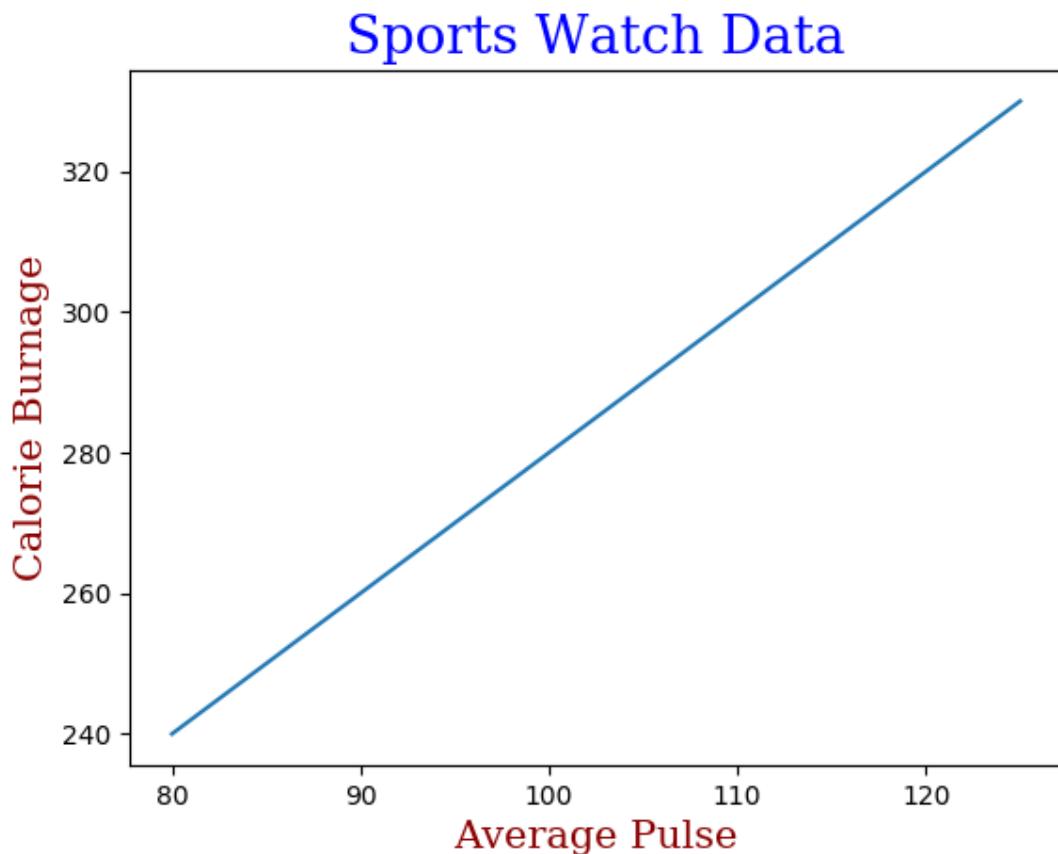
x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

font1 = {'family':'serif','color':'blue','size':20}
font2 = {'family':'serif','color':'darkred','size':15}

plt.title("Sports Watch Data", fontdict = font1)
plt.xlabel("Average Pulse", fontdict = font2)
plt.ylabel("Calorie Burnage", fontdict = font2)

plt.plot(x, y)
plt.show()
```

Result:



Position the Title

You can use the `loc` parameter in `title()` to position the title.

Legal values are: 'left', 'right', and 'center'. Default value is 'center'.

Example

Position the title to the left:

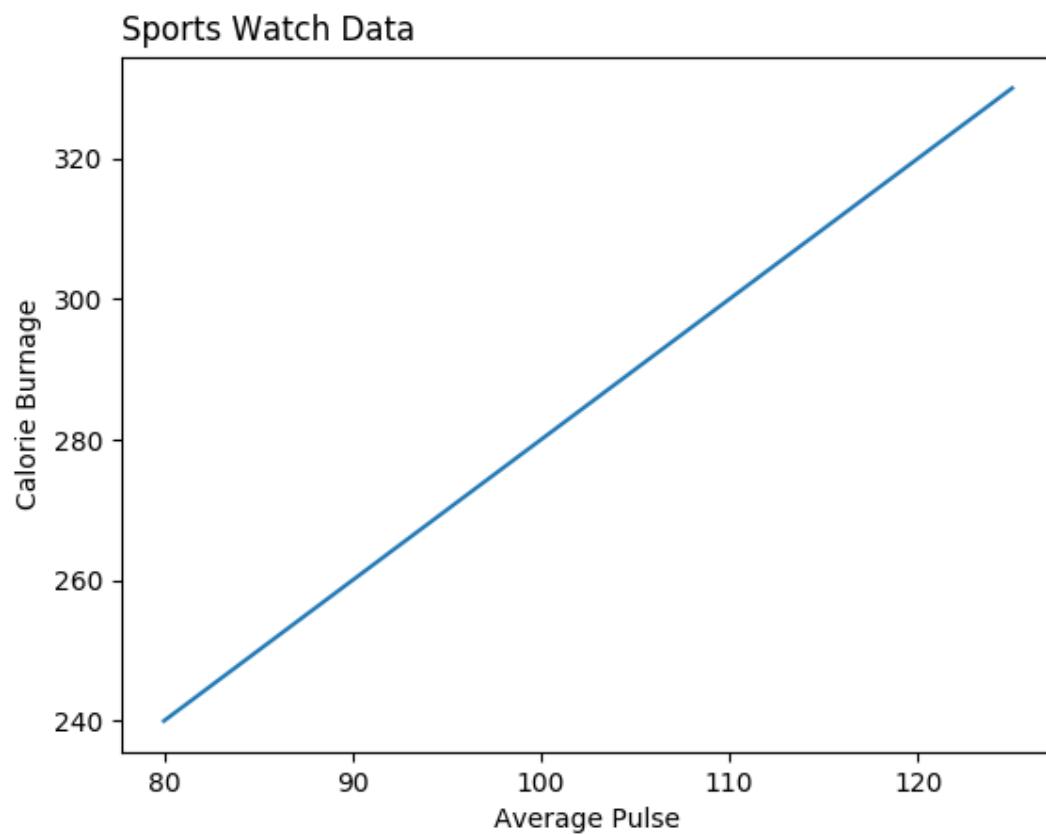
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data", loc = 'left')
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)
plt.show()
```

Result:



Matplotlib Adding Grid Lines

Add Grid Lines to a Plot

With Pyplot, you can use the `grid()` function to add grid lines to the plot.

Example

Add grid lines to the plot:

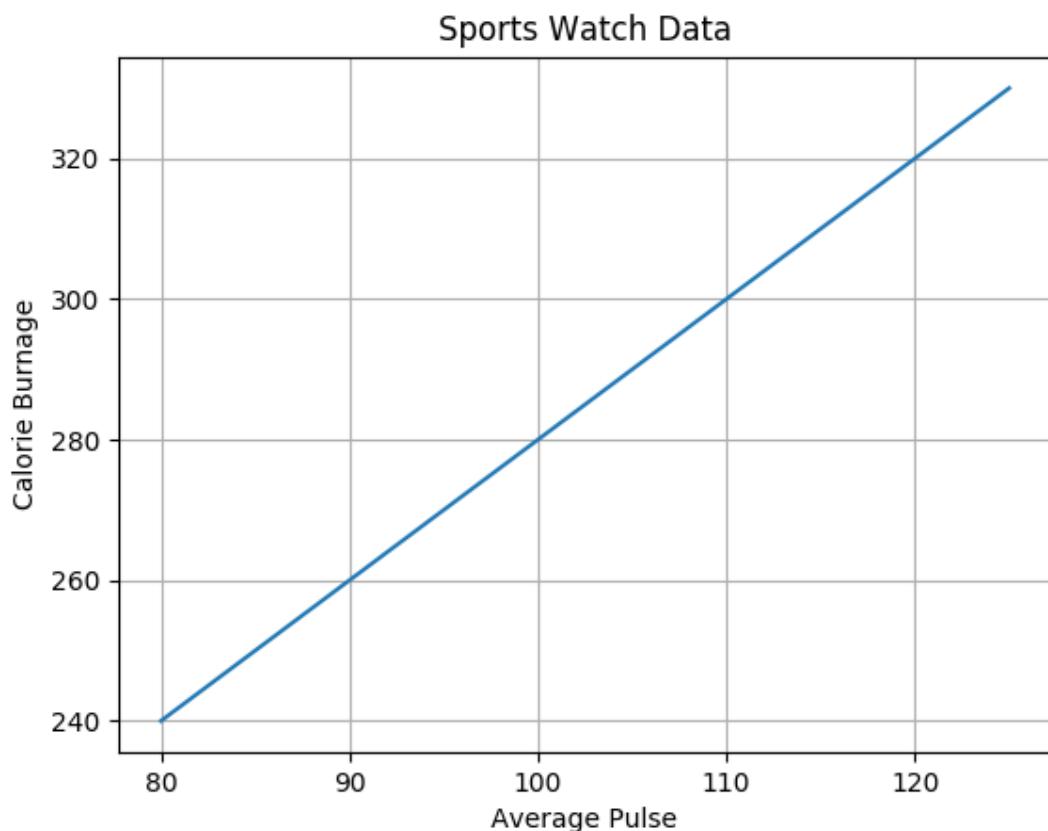
```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
```

```
plt.ylabel("Calorie Burnage")  
  
plt.plot(x, y)  
  
plt.grid()  
  
plt.show()
```

Result:



Specify Which Grid Lines to Display

You can use the `axis` parameter in the `grid()` function to specify which grid lines to display.

Legal values are: 'x', 'y', and 'both'. Default value is 'both'.

Example

Display only grid lines for the x-axis:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

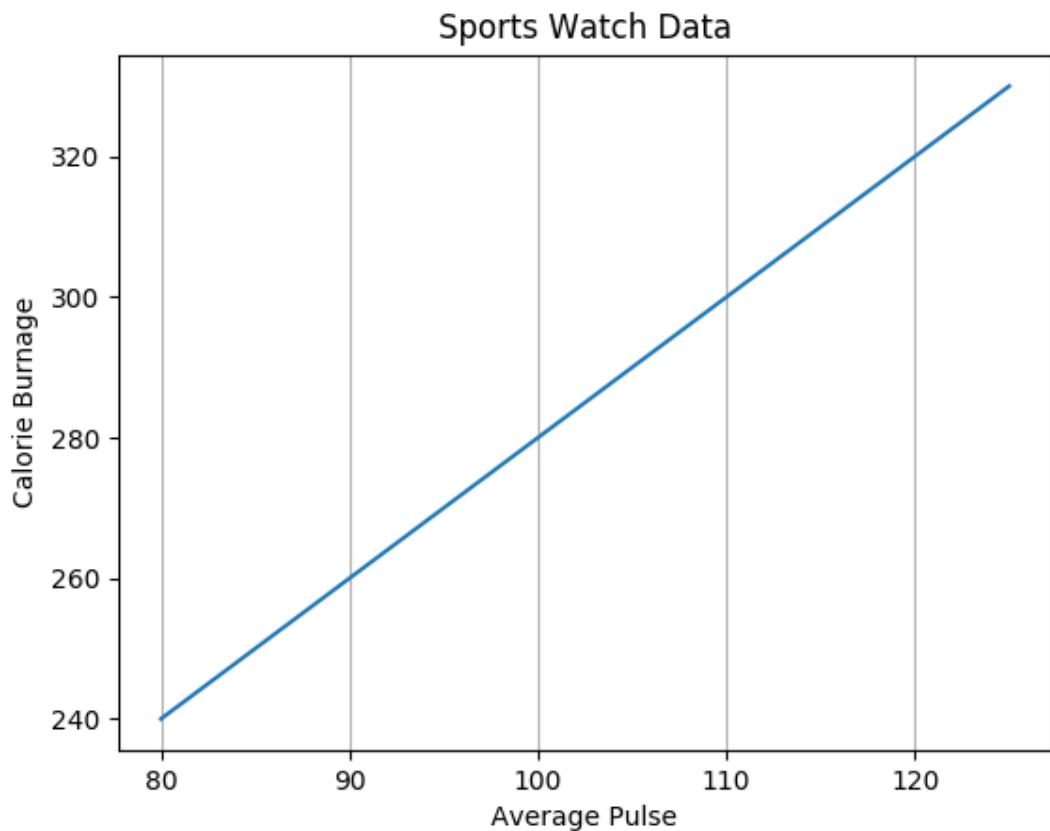
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(axis = 'x')

plt.show()
```

Result:



Example

Display only grid lines for the y-axis:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

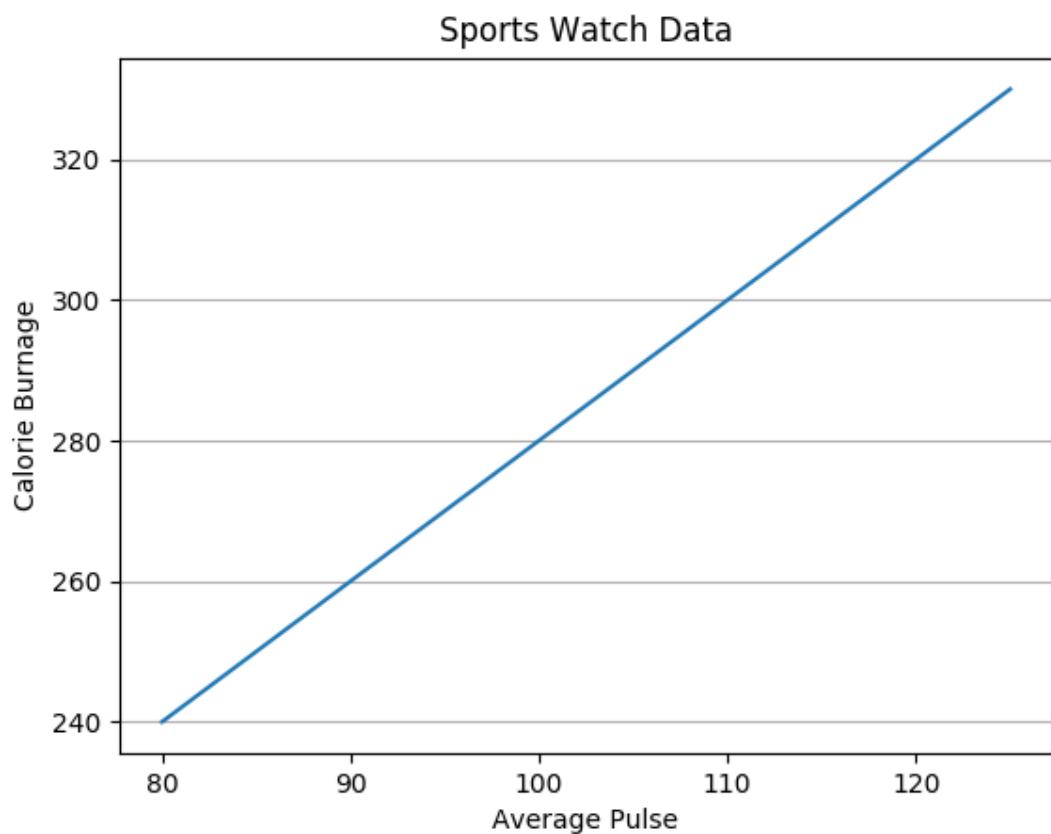
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(axis = 'y')

plt.show()
```

Result:



Set Line Properties for the Grid

You can also set the line properties of the grid, like this: `grid(color = 'color', linestyle = 'linestyle', linewidth = number)`.

Example

Set the line properties of the grid:

```
import numpy as np
import matplotlib.pyplot as plt

x = np.array([80, 85, 90, 95, 100, 105, 110, 115, 120, 125])
y = np.array([240, 250, 260, 270, 280, 290, 300, 310, 320, 330])

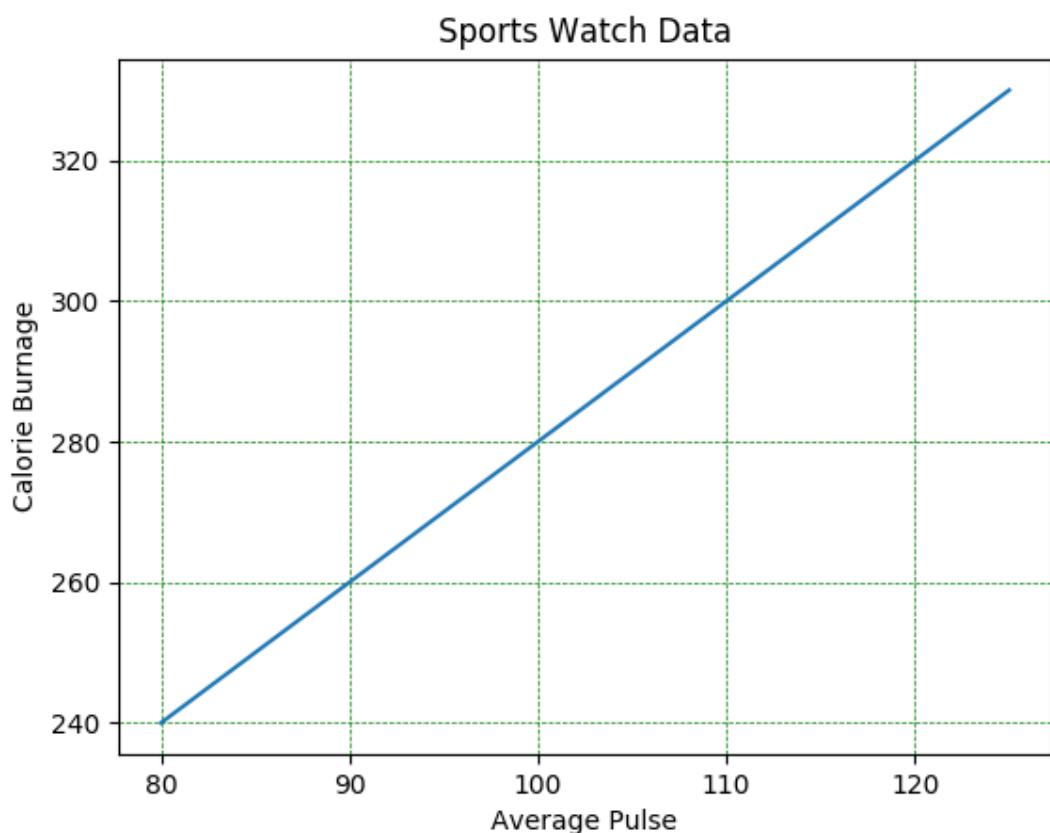
plt.title("Sports Watch Data")
plt.xlabel("Average Pulse")
plt.ylabel("Calorie Burnage")

plt.plot(x, y)

plt.grid(color = 'green', linestyle = '--', linewidth = 0.5)

plt.show()
```

Result:



Matplotlib Subplot

Display Multiple Plots

With the `subplot()` function you can draw multiple plots in one figure:

Example

Draw 2 plots:

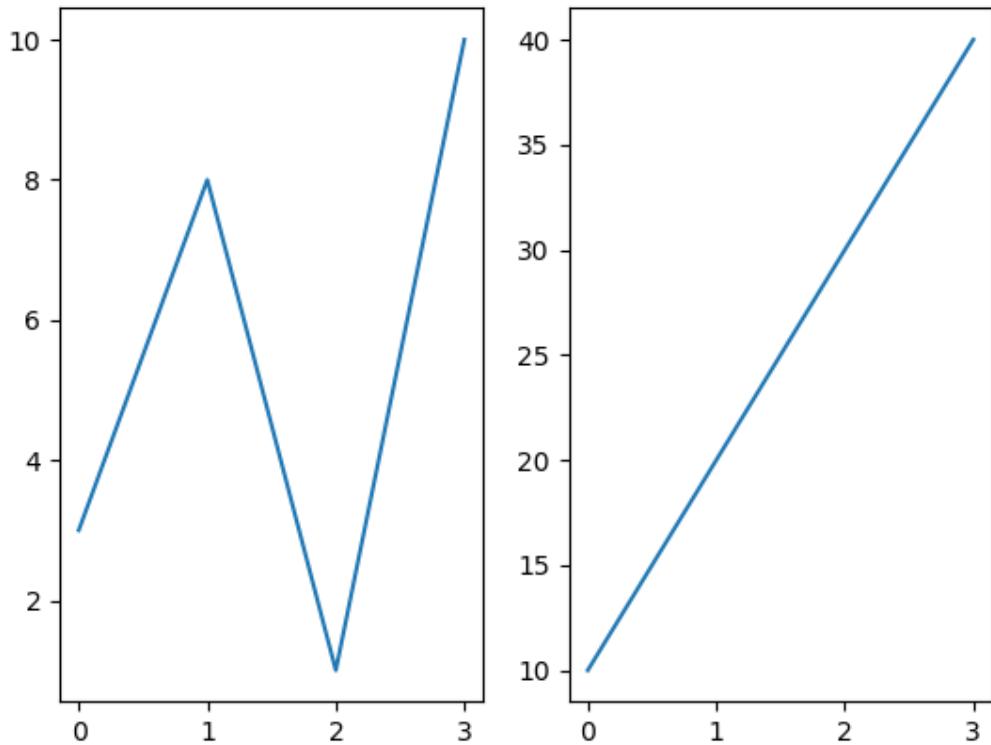
```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
```

```
#plot 2:  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(1, 2, 2)  
plt.plot(x,y)  
  
plt.show()
```

Result:



The subplot() Function

The `subplot()` function takes three arguments that describes the layout of the figure.

The layout is organized in rows and columns, which are represented by the *first* and *second* argument.

The third argument represents the index of the current plot.

```
plt.subplot(1, 2, 1)
#the figure has 1 row, 2 columns, and this plot is the first plot.
```

```
plt.subplot(1, 2, 2)
#the figure has 1 row, 2 columns, and this plot is the second plot.
```

So, if we want a figure with 2 rows an 1 column (meaning that the two plots will be displayed on top of each other instead of side-by-side), we can write the syntax like this:

Example

Draw 2 plots on top of each other:

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

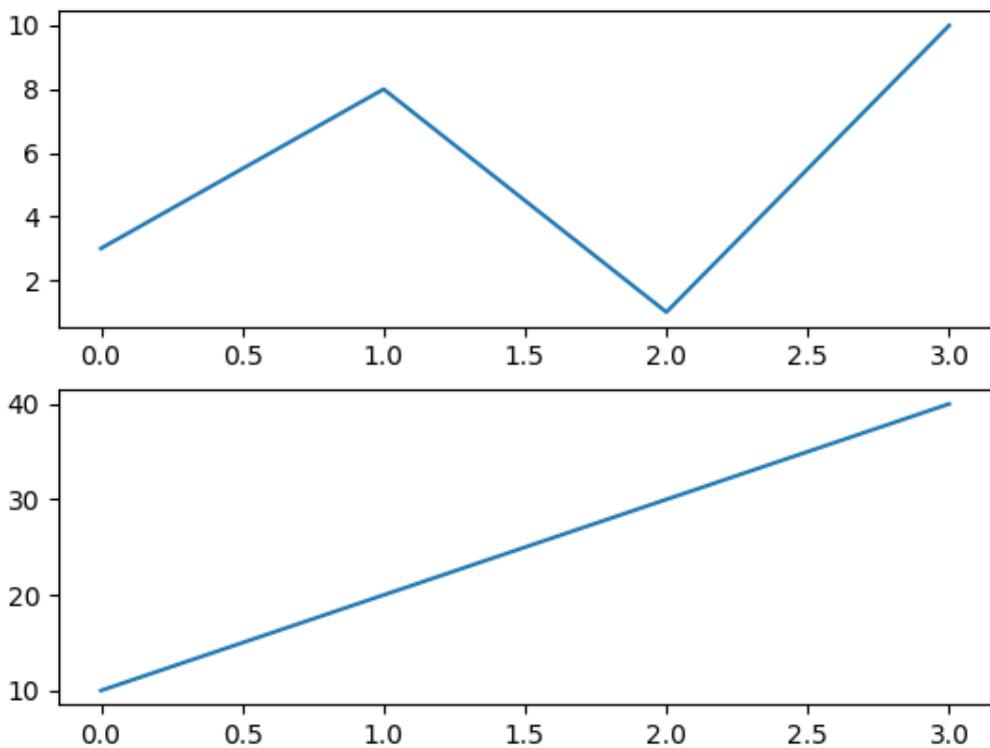
plt.subplot(2, 1, 1)
plt.plot(x,y)

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 1, 2)
plt.plot(x,y)

plt.show()
```

Result:



You can draw as many plots you like on one figure, just describe the number of rows, columns, and the index of the plot.

Example

Draw 6 plots:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 1)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 2)
plt.plot(x,y)
```

```
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 1)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(2, 3, 2)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 3)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

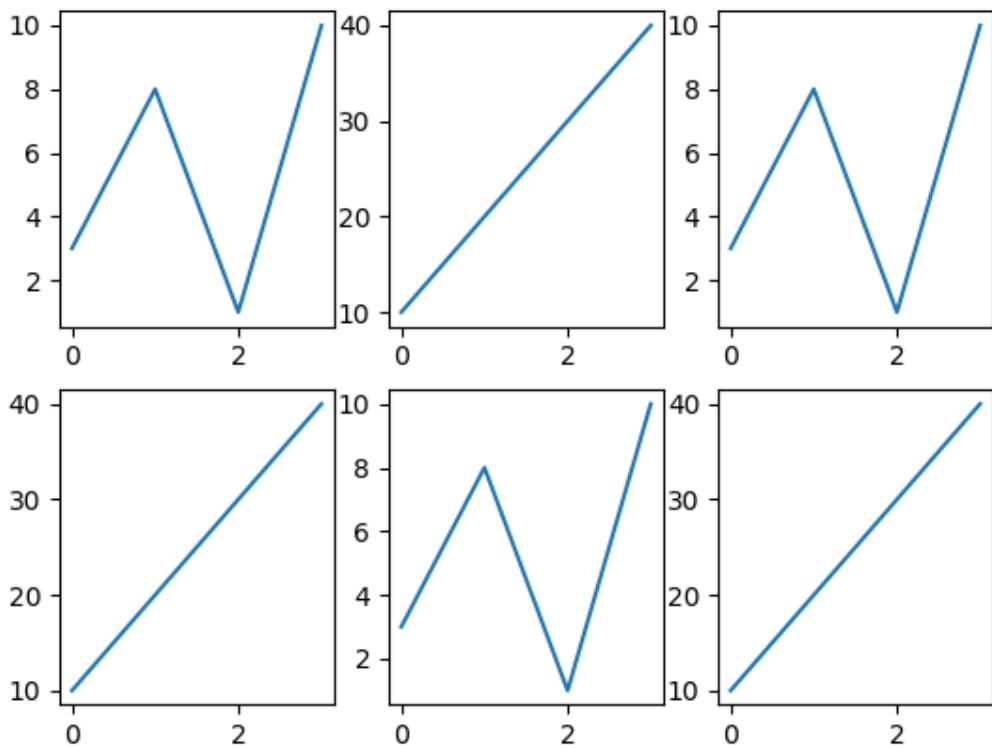
plt.subplot(2, 3, 4)
plt.plot(x,y)

x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(2, 3, 5)
plt.plot(x,y)

plt.show()
```

Result:



Title

You can add a title to each plot with the `title()` function:

Example

2 plots, with titles:

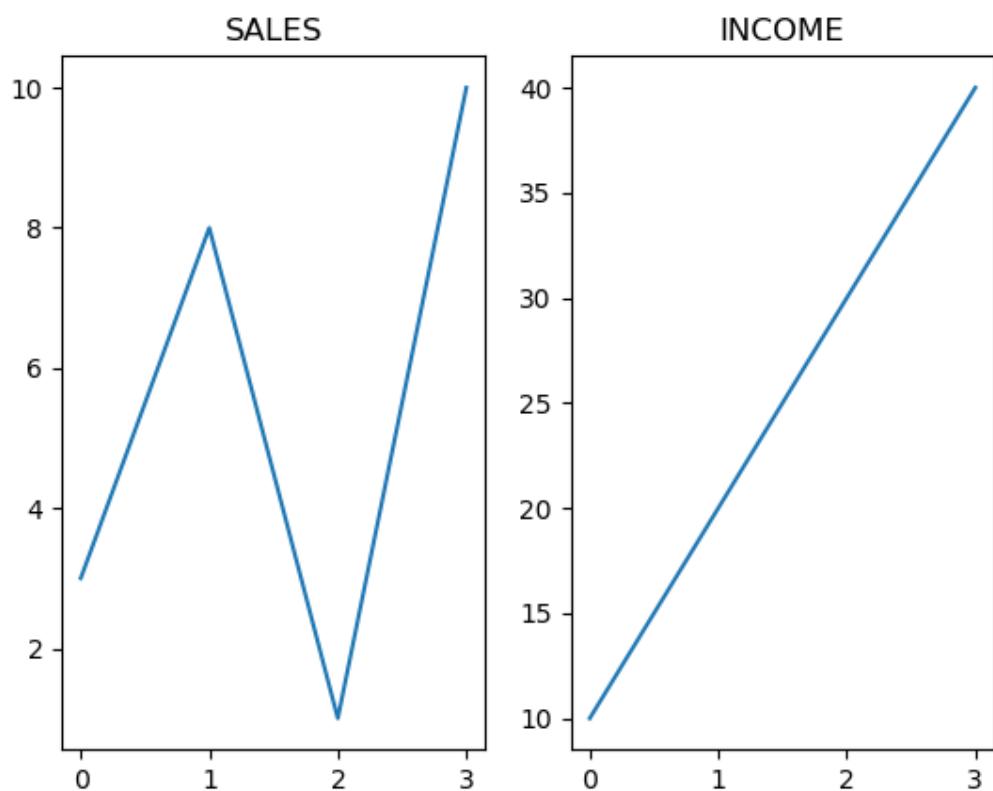
```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")
```

```
#plot 2:  
x = np.array([0, 1, 2, 3])  
y = np.array([10, 20, 30, 40])  
  
plt.subplot(1, 2, 2)  
plt.plot(x,y)  
plt.title("INCOME")  
  
plt.show()
```

Result:



Super Title

You can add a title to the entire figure with the `suptitle()` function:

Example

Add a title for the entire figure:

```
import matplotlib.pyplot as plt
import numpy as np

#plot 1:
x = np.array([0, 1, 2, 3])
y = np.array([3, 8, 1, 10])

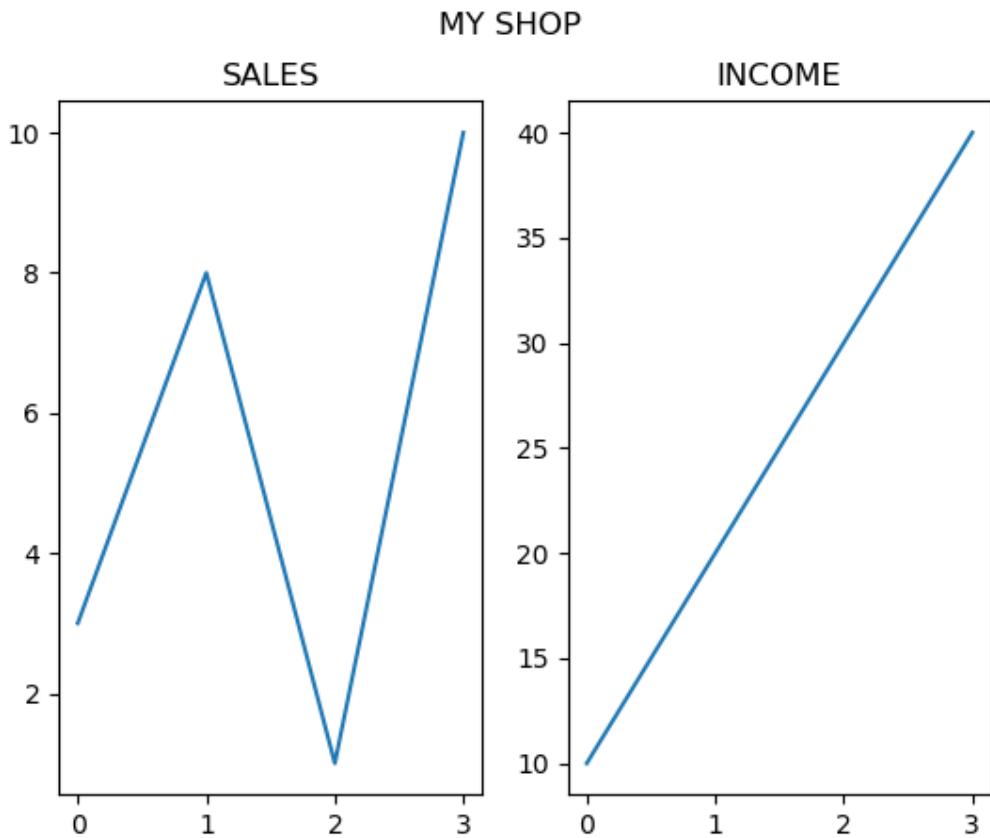
plt.subplot(1, 2, 1)
plt.plot(x,y)
plt.title("SALES")

#plot 2:
x = np.array([0, 1, 2, 3])
y = np.array([10, 20, 30, 40])

plt.subplot(1, 2, 2)
plt.plot(x,y)
plt.title("INCOME")

plt.suptitle("MY SHOP")
plt.show()
```

Result:



Matplotlib Scatter

Creating Scatter Plots

With Pyplot, you can use the `scatter()` function to draw a scatter plot.

The `scatter()` function plots one dot for each observation. It needs two arrays of the same length, one for the values of the x-axis, and one for values on the y-axis:

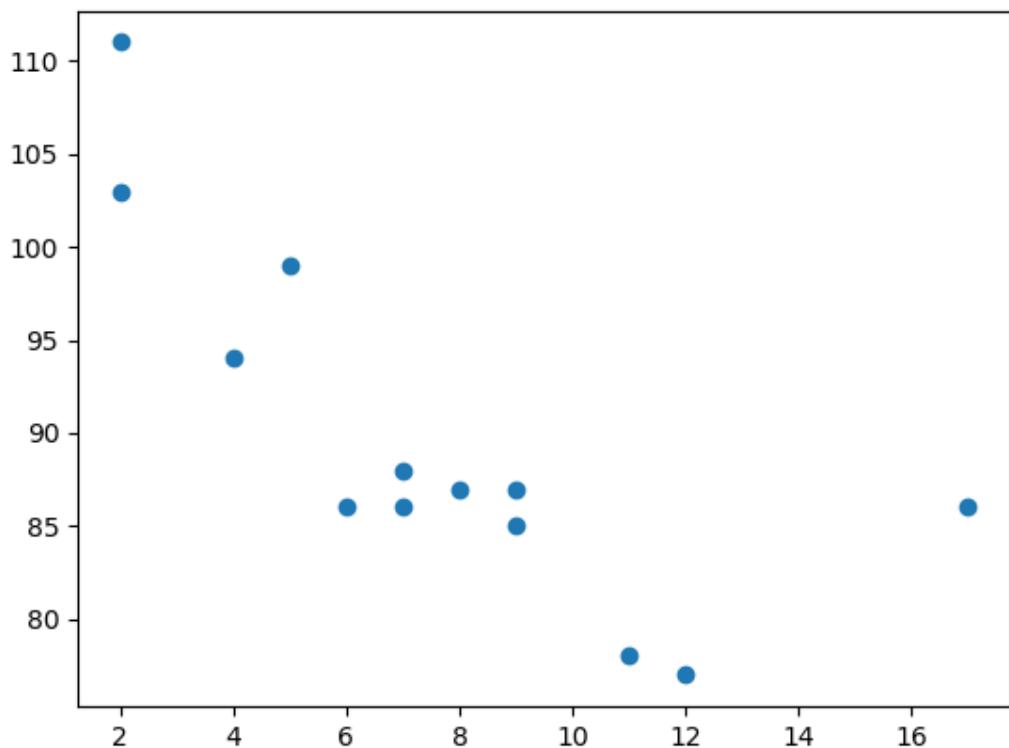
Example

A simple scatter plot:

```
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
```

```
y = np.array([99, 86, 87, 88, 111, 86, 103, 87, 94, 78, 77, 85, 86])  
  
plt.scatter(x, y)  
plt.show()
```

Result:



The observation in the example above is the result of 13 cars passing by.

The X-axis shows how old the car is.

The Y-axis shows the speed of the car when it passes.

Are there any relationships between the observations?

It seems that the newer the car, the faster it drives, but that could be a coincidence, after all we only registered 13 cars.

Compare Plots

In the example above, there seems to be a relationship between speed and age, but what if we plot the observations from another day as well? Will the scatter plot tell us something else?

Example

Draw two plots on the same figure:

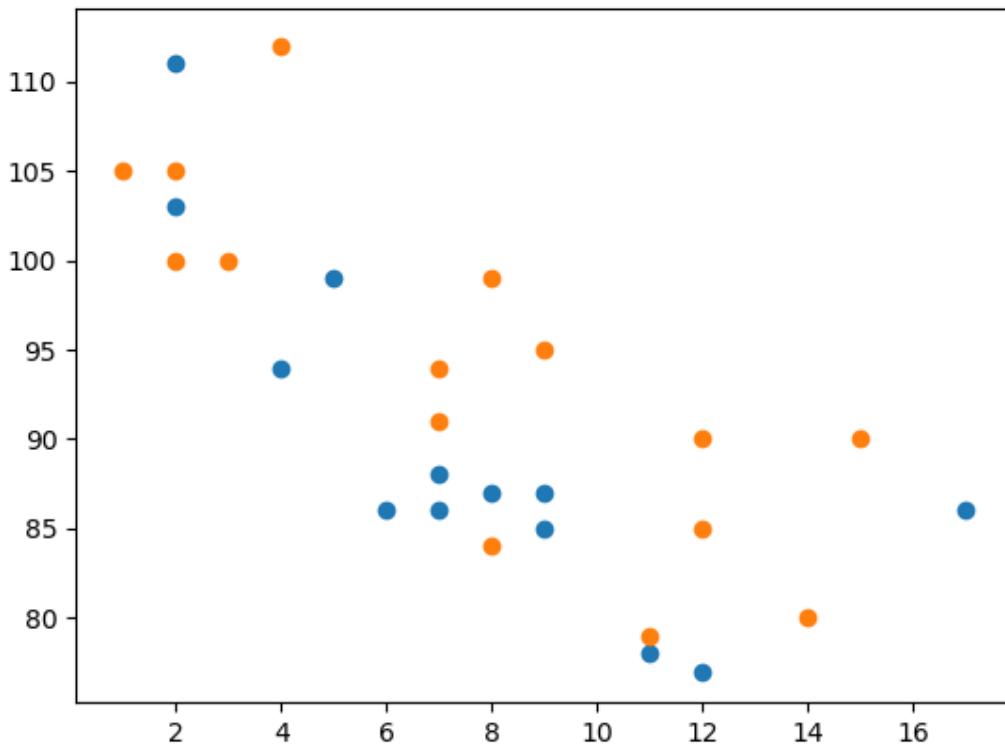
```
import matplotlib.pyplot as plt
import numpy as np

#day one, the age and speed of 13 cars:
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y)

#day two, the age and speed of 15 cars:
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y)

plt.show()
```

Result:



Note: The two plots are plotted with two different colors, by default blue and orange, you will learn how to change colors later in this chapter.

By comparing the two plots, I think it is safe to say that they both gives us the same conclusion: the newer the car, the faster it drives.

Colors

You can set your own color for each scatter plot with the `color` or the `c` argument:

Example

Set your own color of the markers:

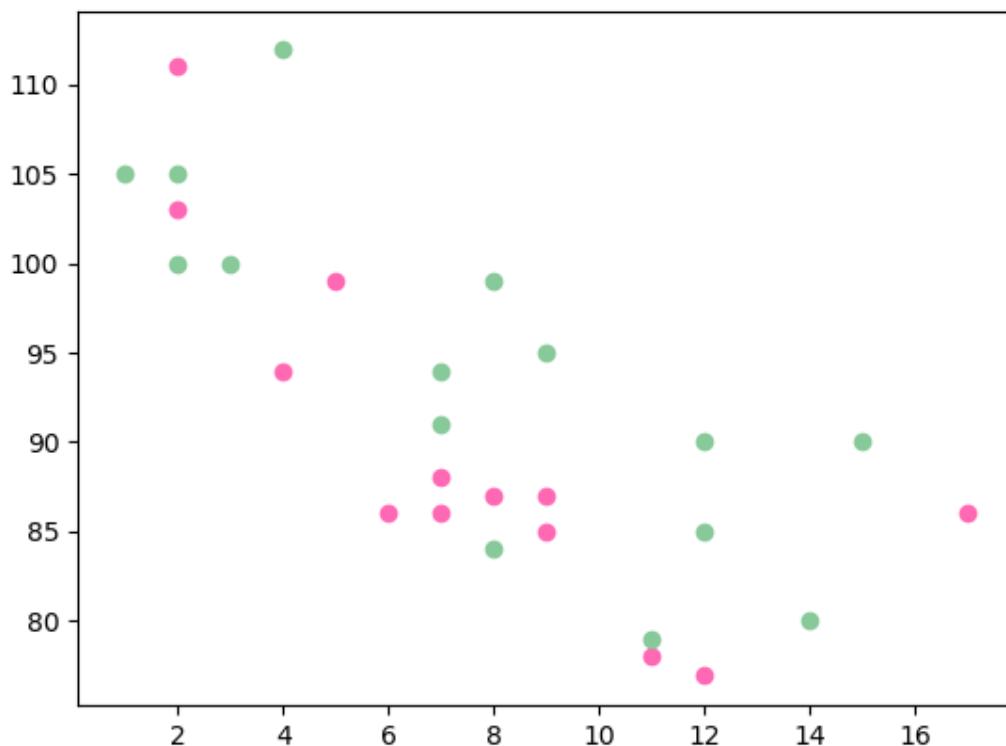
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
plt.scatter(x, y, color = 'hotpink')
```

```
x = np.array([2,2,8,1,15,8,12,9,7,3,11,4,7,14,12])
y = np.array([100,105,84,105,90,99,90,95,94,100,79,112,91,80,85])
plt.scatter(x, y, color = '#88c999')

plt.show()
```

Result:



Color Each Dot

You can even set a specific color for each dot by using an array of colors as value for the `c` argument:

Note: You *cannot* use the `color` argument for this, only the `c` argument.

Example

Set your own color of the markers:

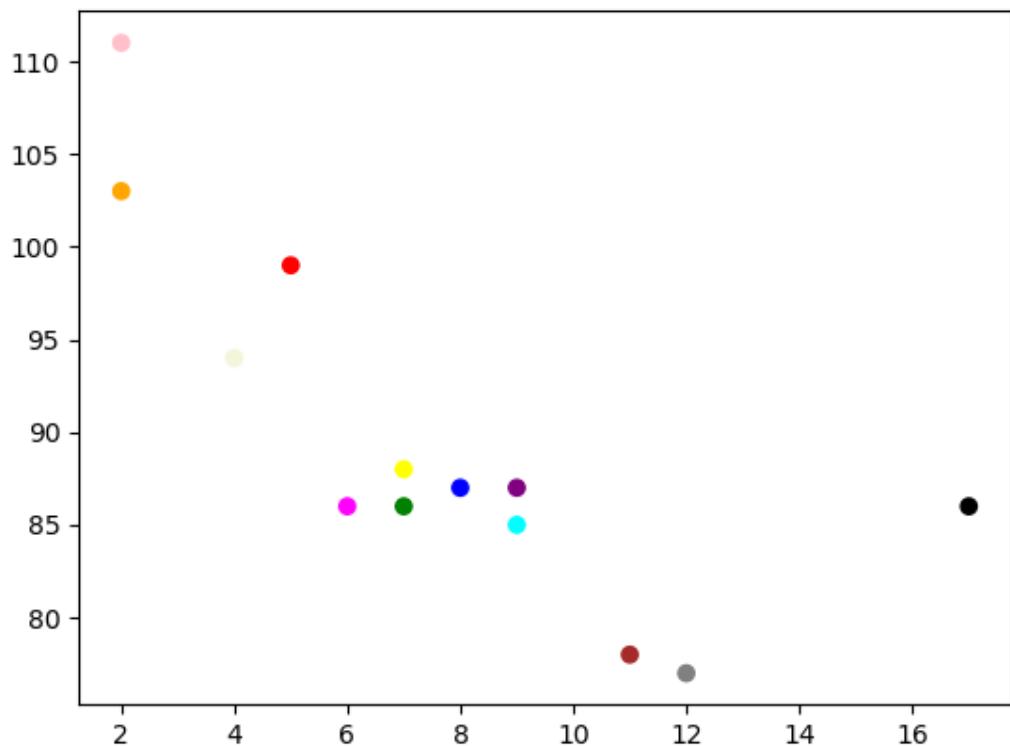
```

import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
colors =
np.array(["red","green","blue","yellow","pink","black","orange","purple",
"beige","brown","gray","cyan","magenta"])
plt.scatter(x, y, c=colors)
plt.show()

```

Result:



ColorMap

The Matplotlib module has a number of available colormaps.

A colormap is like a list of colors, where each color has a value that ranges from 0 to 100.

Here is an example of a colormap:



This colormap is called 'viridis' and as you can see it ranges from 0, which is a purple color, and up to 100, which is a yellow color.

How to Use the ColorMap

You can specify the colormap with the keyword argument `cmap` with the value of the colormap, in this case '`'viridis'`' which is one of the built-in colormaps available in Matplotlib.

In addition you have to create an array with values (from 0 to 100), one value for each of the point in the scatter plot:

Example

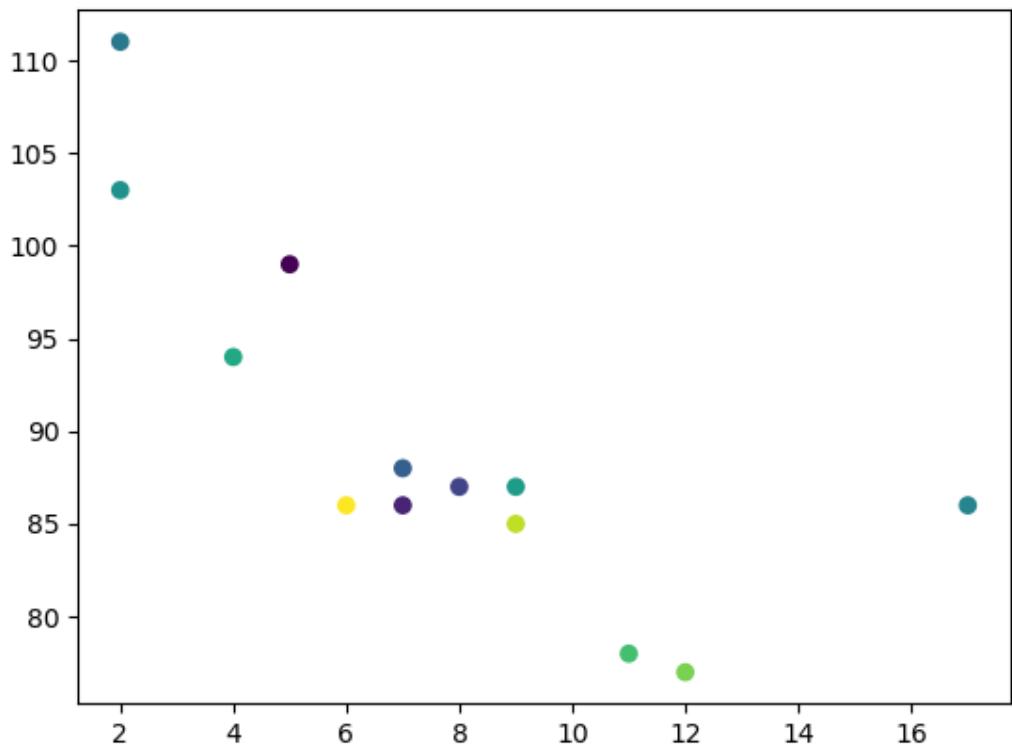
Create a color array, and specify a colormap in the scatter plot:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
```

```
colors =  
np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])  
  
plt.scatter(x, y, c=colors, cmap='viridis')  
  
plt.show()
```

Result:



You can include the colormap in the drawing by including the `plt.colorbar()` statement:

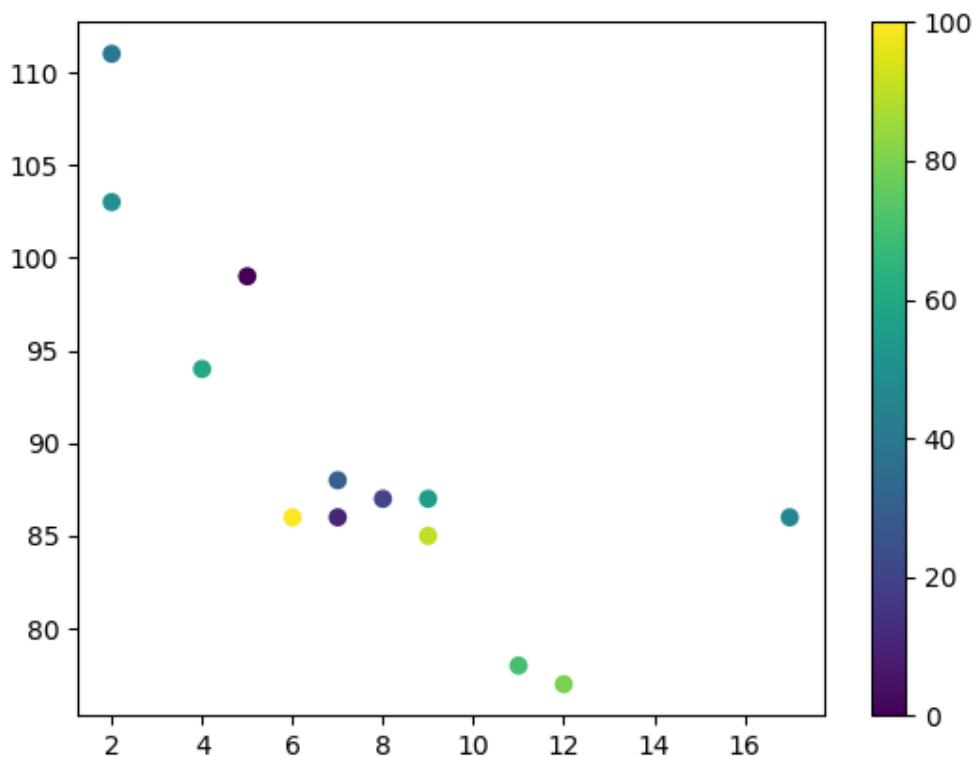
Example

Include the actual colormap:

```
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])  
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])  
colors =
```

```
np.array([0, 10, 20, 30, 40, 45, 50, 55, 60, 70, 80, 90, 100])  
  
plt.scatter(x, y, c=colors, cmap='viridis')  
  
plt.colorbar()  
  
plt.show()
```

Result:



Size

You can change the size of the dots with the `s` argument.

Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

Example

Set your own size for the markers:

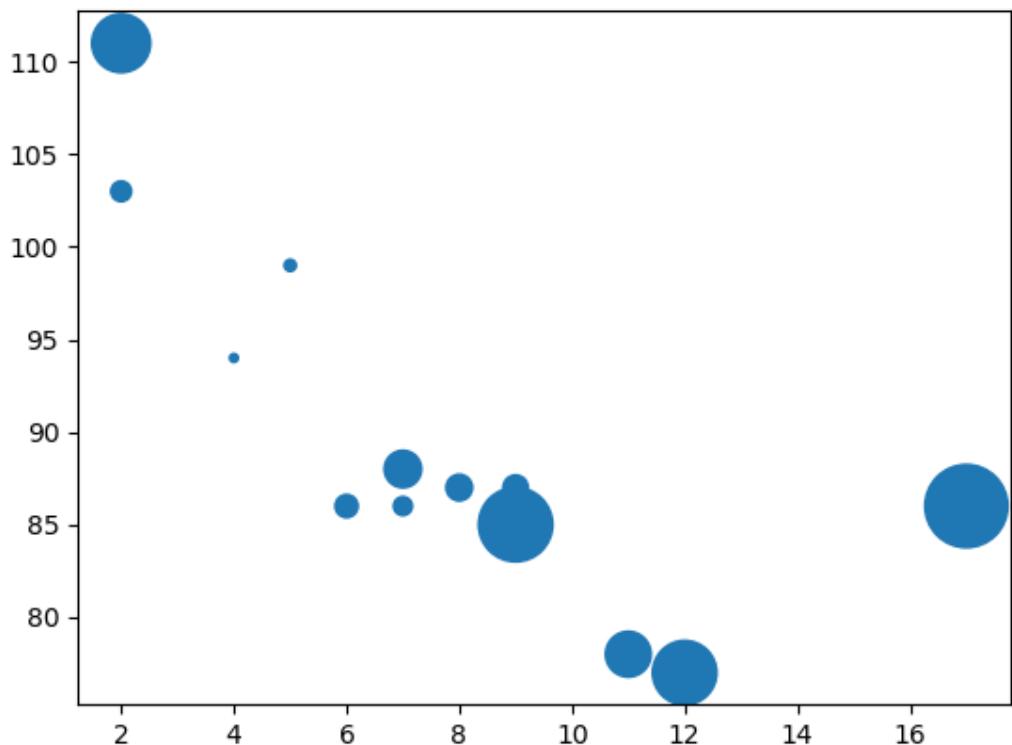
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=sizes)

plt.show()
```

Result:



Alpha

You can adjust the transparency of the dots with the `alpha` argument.

Just like colors, make sure the array for sizes has the same length as the arrays for the x- and y-axis:

Example

Set your own size for the markers:

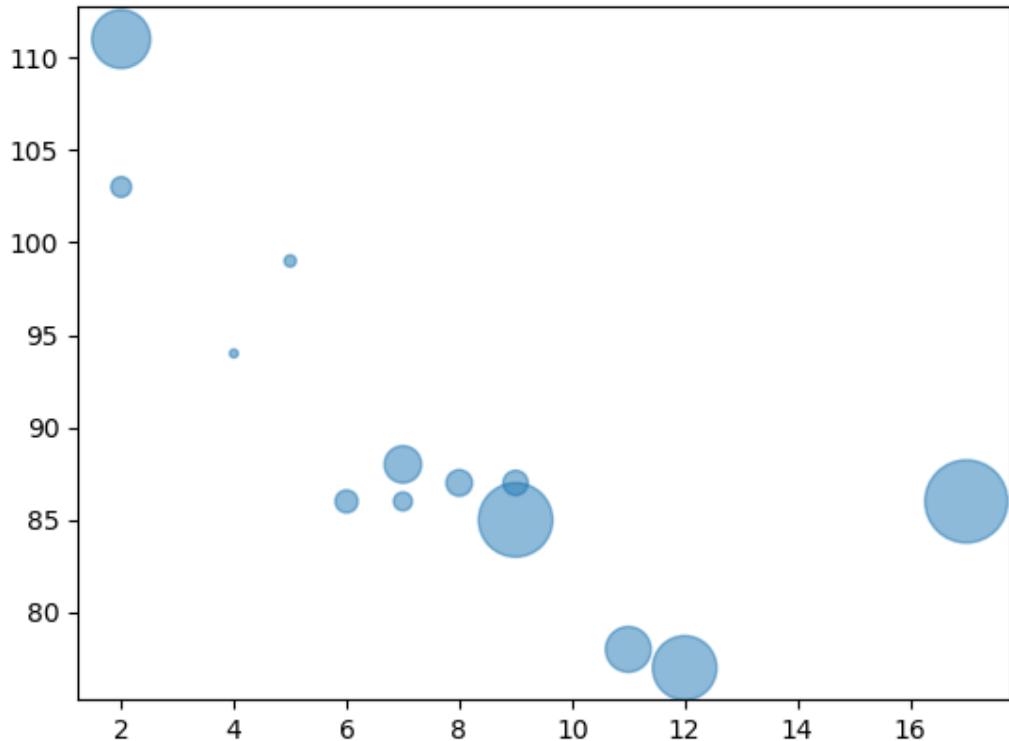
```
import matplotlib.pyplot as plt
import numpy as np

x = np.array([5,7,8,7,2,17,2,9,4,11,12,9,6])
y = np.array([99,86,87,88,111,86,103,87,94,78,77,85,86])
sizes = np.array([20,50,100,200,500,1000,60,90,10,300,600,800,75])

plt.scatter(x, y, s=sizes, alpha=0.5)

plt.show()
```

Result:



Combine Color Size and Alpha

You can combine a colormap with different sizes on the dots. This is best visualized if the dots are transparent:

Example

Create random arrays with 100 values for x-points, y-points, colors and sizes:

```
import matplotlib.pyplot as plt
import numpy as np

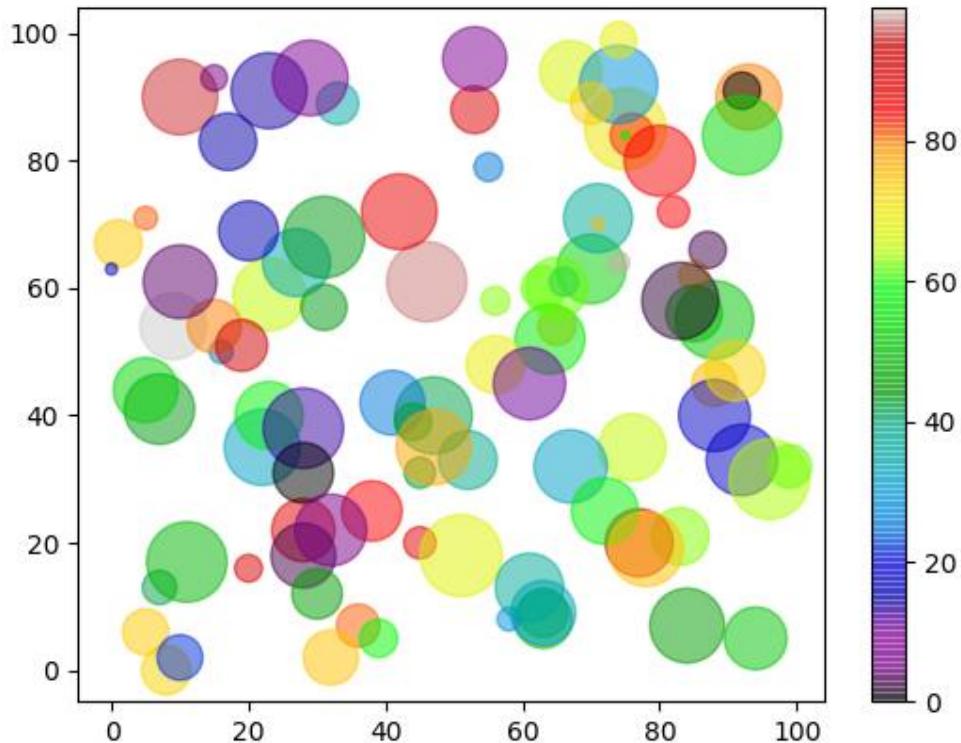
x = np.random.randint(100, size=(100))
y = np.random.randint(100, size=(100))
colors = np.random.randint(100, size=(100))
sizes = 10 * np.random.randint(100, size=(100))

plt.scatter(x, y, c=colors, s=sizes, alpha=0.5, cmap='nipy_spectral')

plt.colorbar()

plt.show()
```

Result:



Matplotlib Bars

Creating Bars

With Pyplot, you can use the `bar()` function to draw bar graphs:

Example

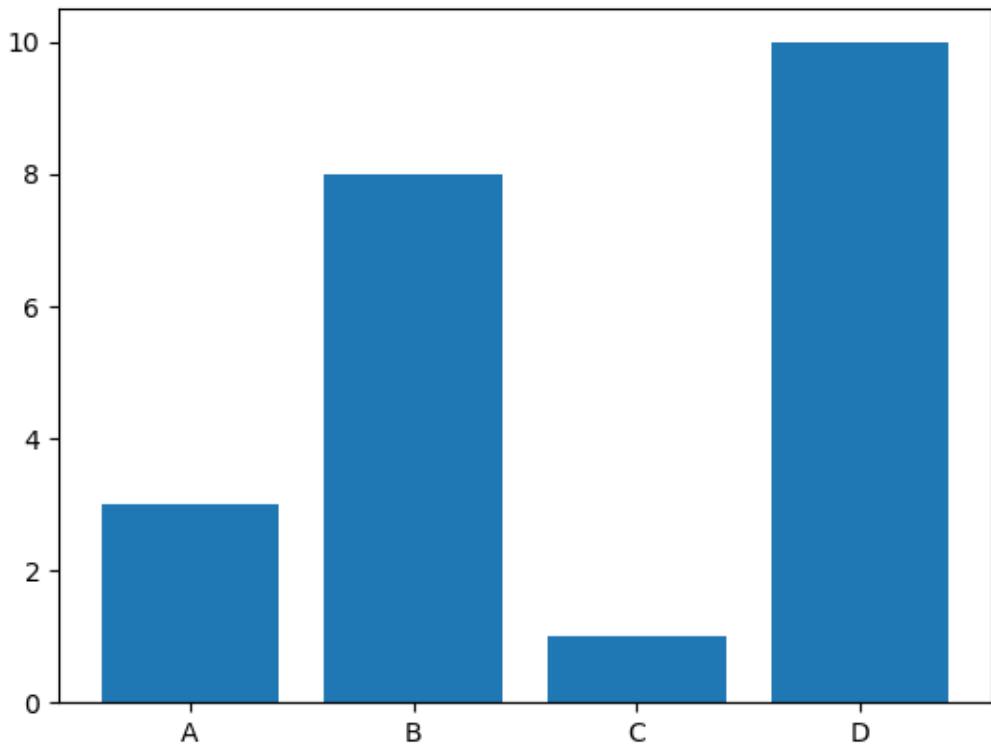
Draw 4 bars:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x,y)
plt.show()
```

Result:



The `bar()` function takes arguments that describes the layout of the bars.

The categories and their values represented by the *first* and *second* argument as arrays.

Example

```
x = ["APPLES", "BANANAS"]  
y = [400, 350]  
plt.bar(x, y)
```

Horizontal Bars

If you want the bars to be displayed horizontally instead of vertically, use the `barh()` function:

Example

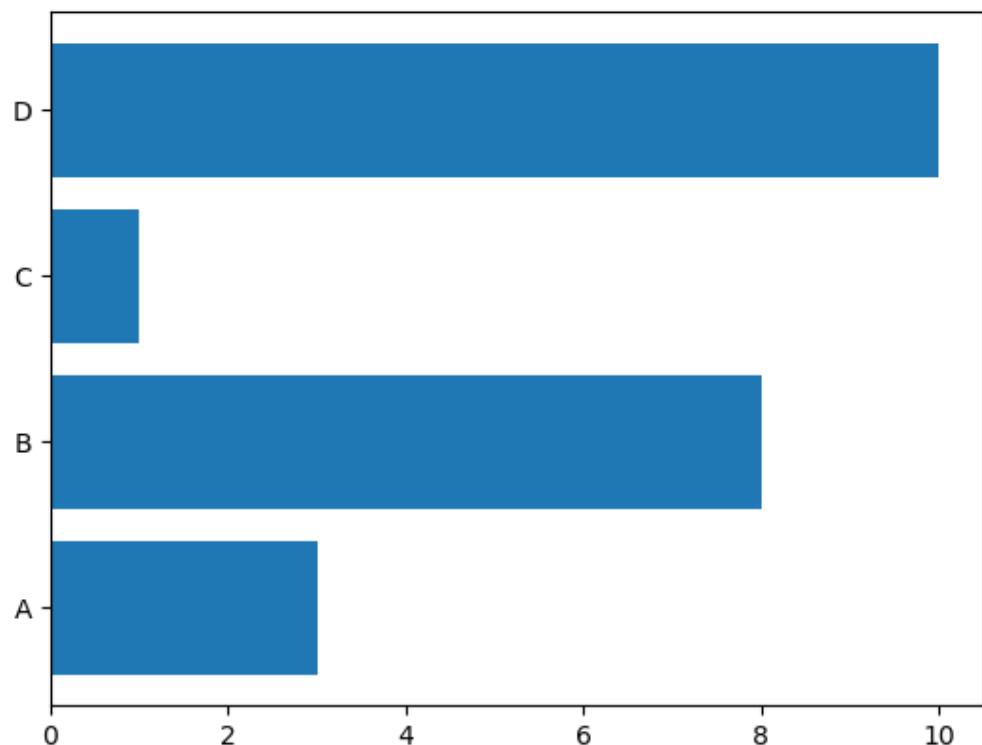
Draw 4 horizontal bars:

```
import matplotlib.pyplot as plt  
import numpy as np
```

```
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y)
plt.show()
```

Result:



Bar Color

The `bar()` and `barh()` takes the keyword argument `color` to set the color of the bars:

Example

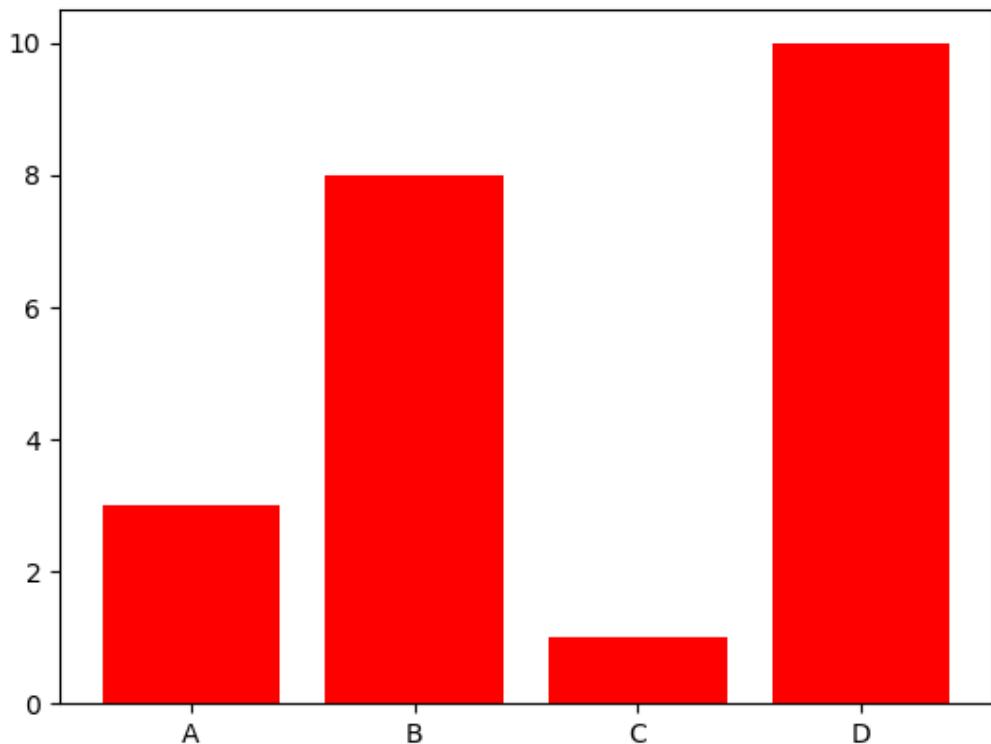
Draw 4 red bars:

```
import matplotlib.pyplot as plt
import numpy as np
```

```
x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = "red")
plt.show()
```

Result:



Color Names

You can use any of the [140 supported color names](#).

Example

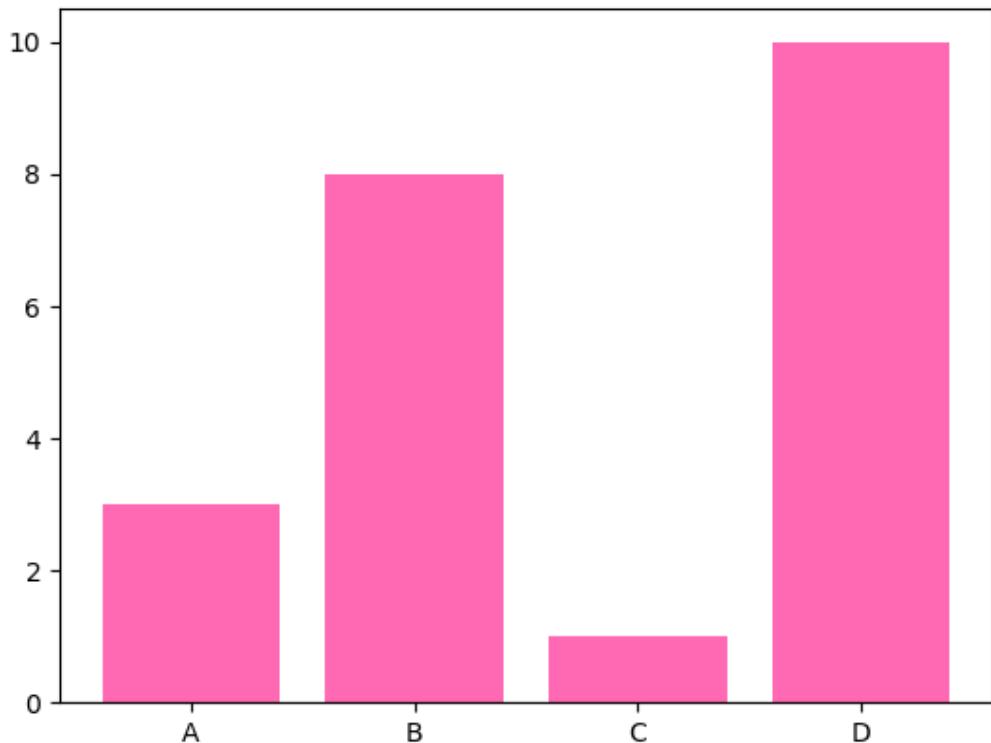
Draw 4 "hot pink" bars:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])
```

```
plt.bar(x, y, color = "hotpink")
plt.show()
```

Result:



Color Hex

Or you can use [Hexadecimal color values](#):

Example

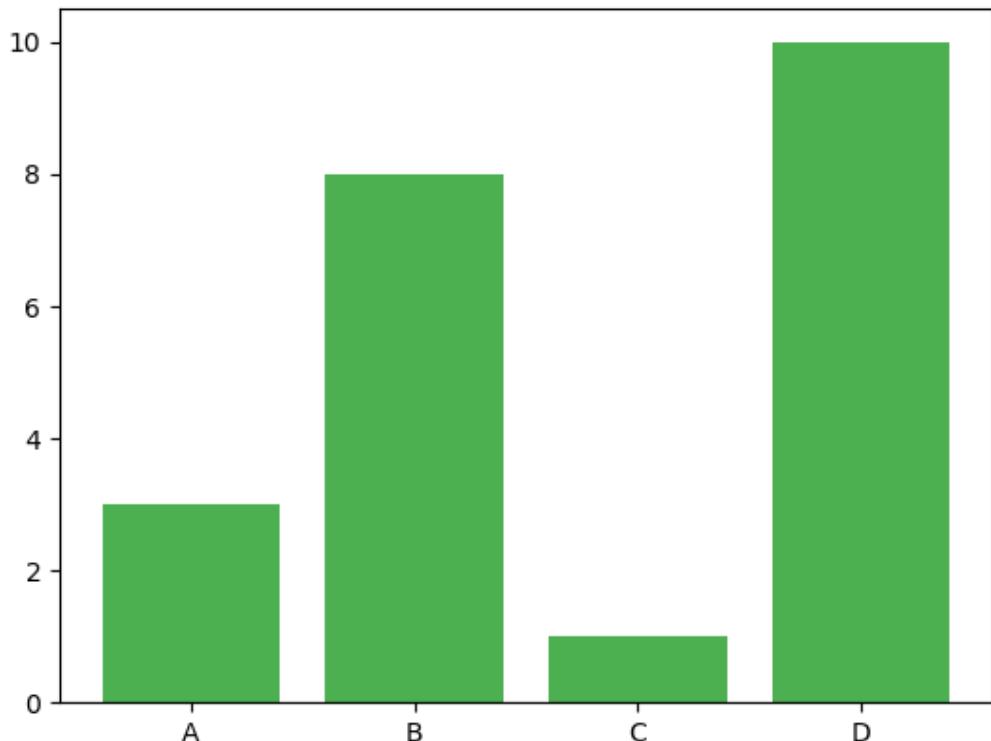
Draw 4 bars with a beautiful green color:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, color = "#4CAF50")
plt.show()
```

Result:



Bar Width

The `bar()` takes the keyword argument `width` to set the width of the bars:

Example

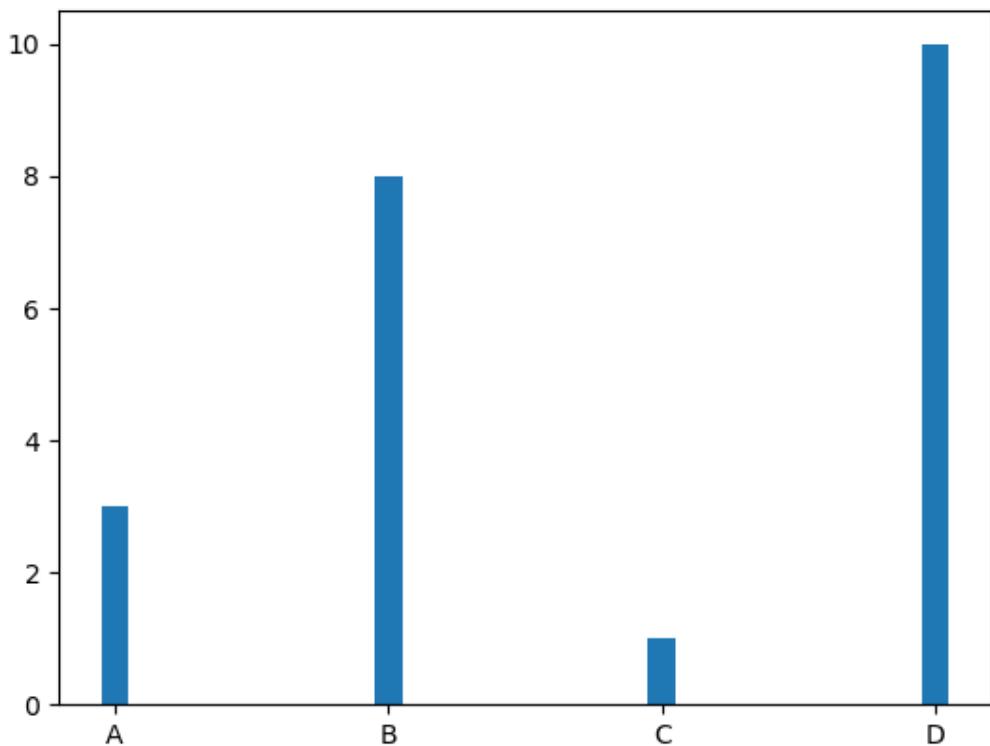
Draw 4 very thin bars:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.bar(x, y, width = 0.1)
plt.show()
```

Result:



The default width value is 0.8

Note: For horizontal bars, use `height` instead of `width`.

Bar Height

The `barh()` takes the keyword argument `height` to set the height of the bars:

Example

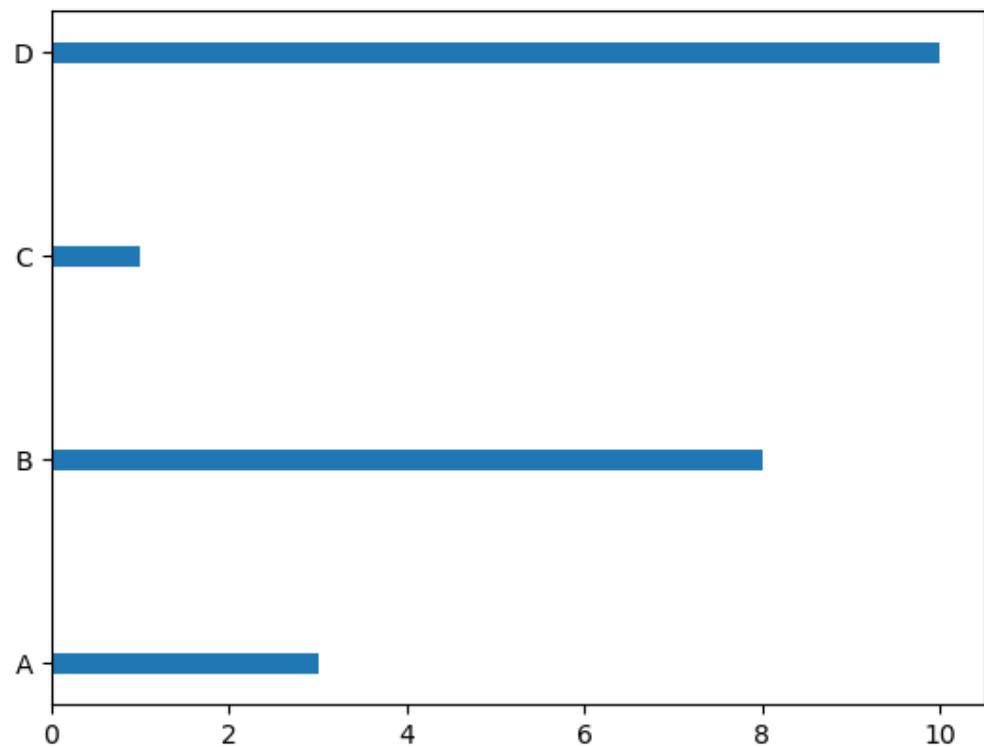
Draw 4 very thin bars:

```
import matplotlib.pyplot as plt
import numpy as np

x = np.array(["A", "B", "C", "D"])
y = np.array([3, 8, 1, 10])

plt.barh(x, y, height = 0.1)
plt.show()
```

Result:



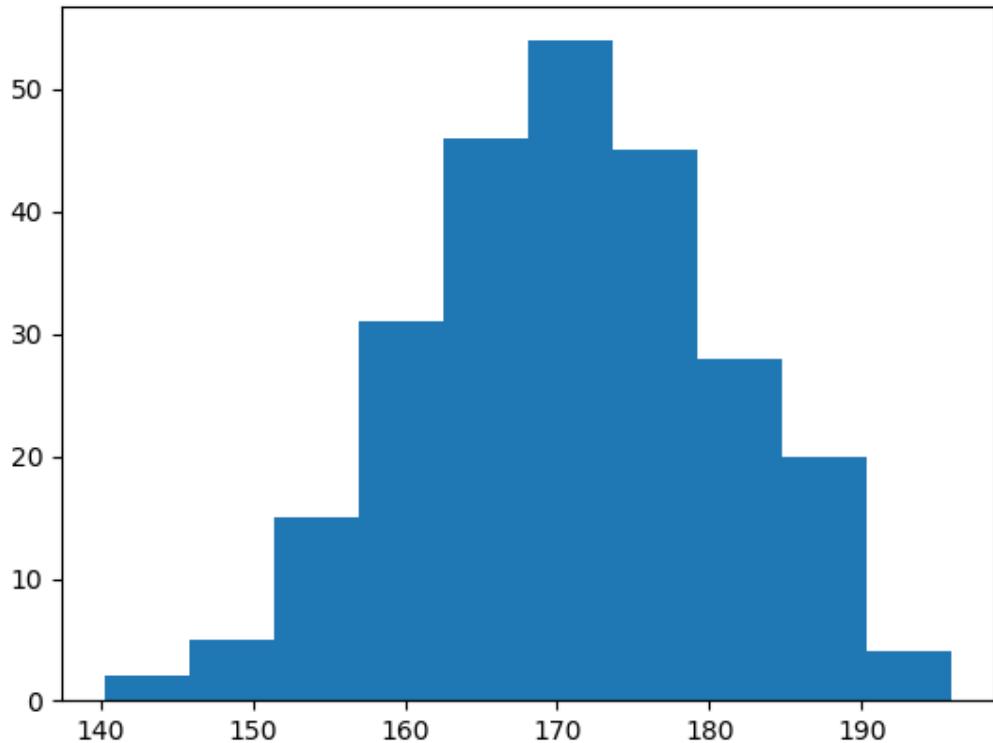
Matplotlib Histograms

Histogram

A histogram is a graph showing *frequency* distributions.

It is a graph showing the number of observations within each given interval.

Example: Say you ask for the height of 250 people, you might end up with a histogram like this:



You can read from the histogram that there are approximately:

- 2 people from 140 to 145cm
- 5 people from 145 to 150cm
- 15 people from 151 to 156cm
- 31 people from 157 to 162cm
- 46 people from 163 to 168cm
- 53 people from 168 to 173cm
- 45 people from 173 to 178cm

28 people from 179 to 184cm
21 people from 185 to 190cm
4 people from 190 to 195cm

Create Histogram

In Matplotlib, we use the `hist()` function to create histograms.

The `hist()` function will use an array of numbers to create a histogram, the array is sent into the function as an argument.

For simplicity we use NumPy to randomly generate an array with 250 values, where the values will concentrate around 170, and the standard deviation is 10. Learn more about [Normal Data Distribution](#) in our [Machine Learning Tutorial](#).

Example

A Normal Data Distribution by NumPy:

```
import numpy as np

x = np.random.normal(170, 10, 250)

print(x)
```

Result:

This will generate a *random* result, and could look like this:

```
[167.62255766 175.32495609 152.84661337 165.50264047
163.17457988
162.29867872 172.83638413 168.67303667 164.57361342
180.81120541
170.57782187 167.53075749 176.15356275 176.95378312
158.4125473
187.8842668 159.03730075 166.69284332 160.73882029
152.22378865
164.01255164 163.95288674 176.58146832 173.19849526
169.40206527
166.88861903 149.90348576 148.39039643 177.90349066
166.72462233
177.44776004 170.93335636 173.26312881 174.76534435
162.28791953]
```

```
166.77301551 160.53785202 170.67972019 159.11594186  
165.36992993 178.38979253 171.52158489 173.32636678 159.63894401  
151.95735707 175.71274153 165.00458544 164.80607211 177.50988211  
149.28106703 179.43586267 181.98365273 170.98196794 179.1093176  
176.91855744  
167.22138242
```

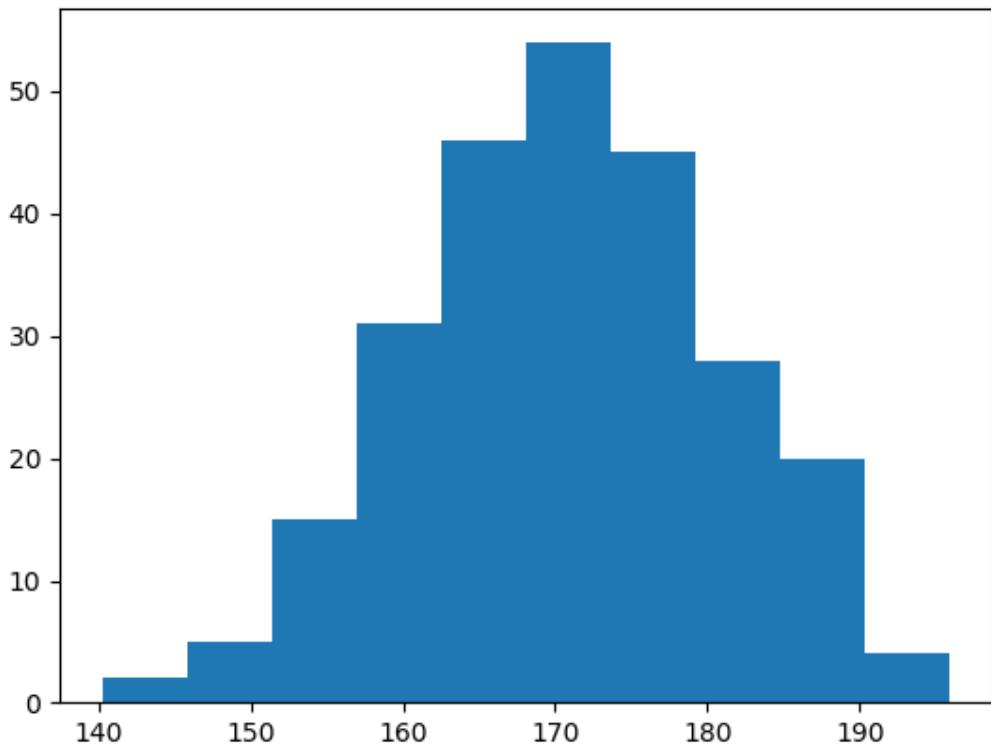
The `hist()` function will read the array and produce a histogram:

Example

A simple histogram:

```
import matplotlib.pyplot as plt  
import numpy as np  
  
x = np.random.normal(170, 10, 250)  
  
plt.hist(x)  
plt.show()
```

Result:



Matplotlib Pie Charts

Creating Pie Charts

With Pyplot, you can use the `pie()` function to draw pie charts:

Example

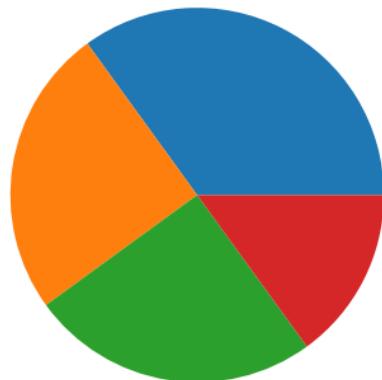
A simple pie chart:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])

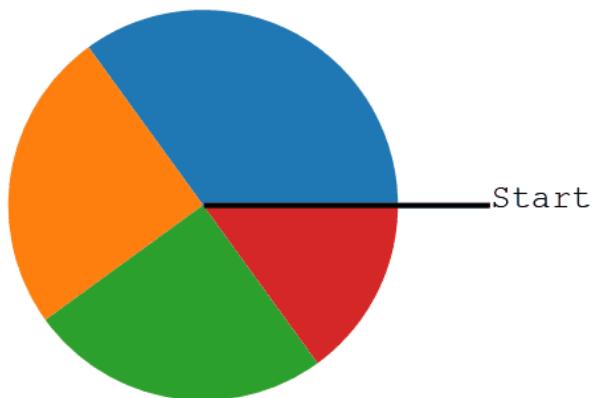
plt.pie(y)
plt.show()
```

Result:



As you can see the pie chart draws one piece (called a wedge) for each value in the array (in this case [35, 25, 25, 15]).

By default the plotting of the first wedge starts from the x-axis and move *counterclockwise*:



Note: The size of each wedge is determined by comparing the value with all the other values, by using this formula:

The value divided by the sum of all values: $x/\text{sum}(x)$

Labels

Add labels to the pie chart with the `label` parameter.

The `label` parameter must be an array with one label for each wedge:

Example

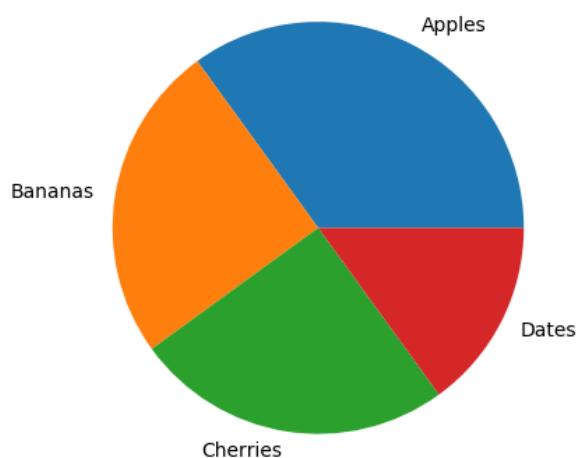
A simple pie chart:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.show()
```

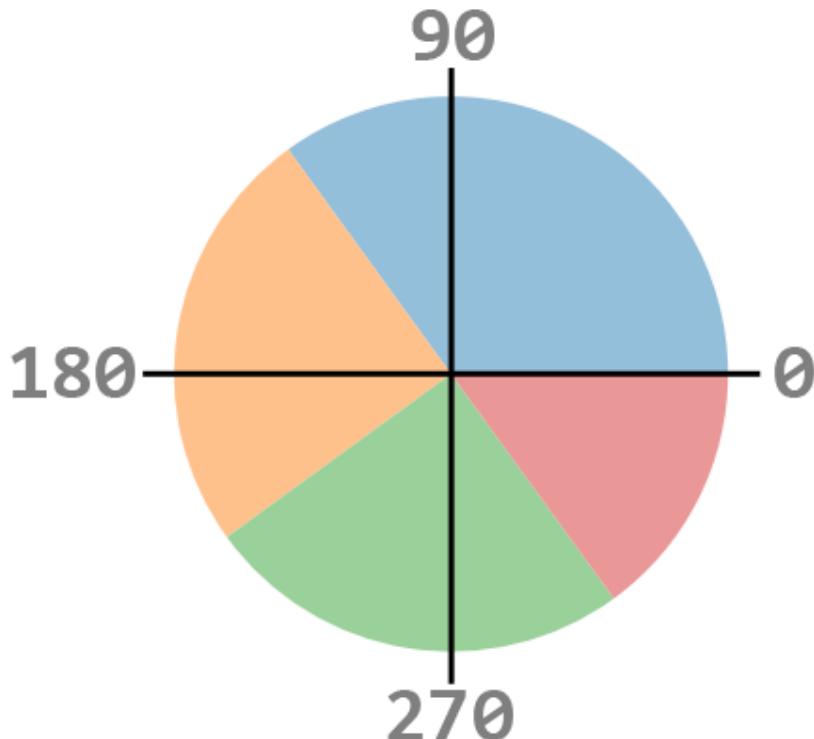
Result:



Start Angle

As mentioned the default start angle is at the x-axis, but you can change the start angle by specifying a `startangle` parameter.

The `startangle` parameter is defined with an angle in degrees, default angle is 0:



Example

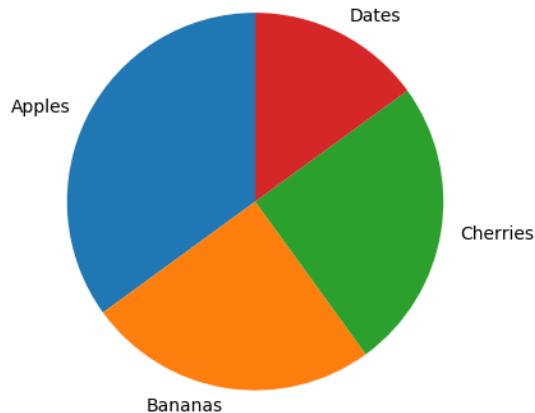
Start the first wedge at 90 degrees:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels, startangle = 90)
plt.show()
```

Result:



Explode

Maybe you want one of the wedges to stand out? The `explode` parameter allows you to do that.

The `explode` parameter, if specified, and not `None`, must be an array with one value for each wedge.

Each value represents how far from the center each wedge is displayed:

Example

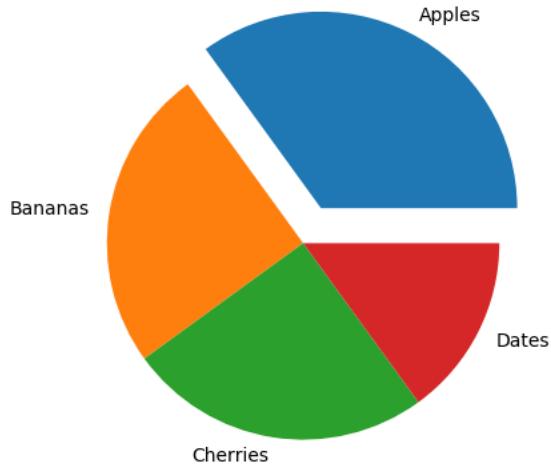
Pull the "Apples" wedge 0.2 from the center of the pie:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]

plt.pie(y, labels = mylabels, explode = myexplode)
plt.show()
```

Result:



Shadow

Add a shadow to the pie chart by setting the `shadows` parameter to `True`:

Example

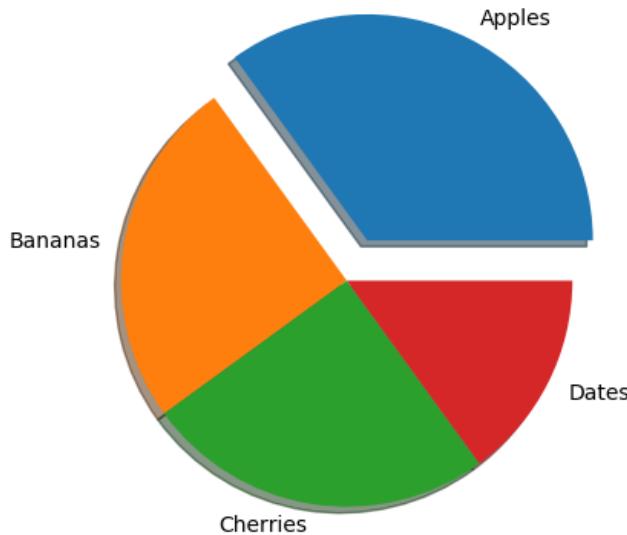
Add a shadow:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
myexplode = [0.2, 0, 0, 0]

plt.pie(y, labels = mylabels, explode = myexplode, shadow = True)
plt.show()
```

Result:



Colors

You can set the color of each wedge with the `colors` parameter.

The `colors` parameter, if specified, must be an array with one value for each wedge:

Example

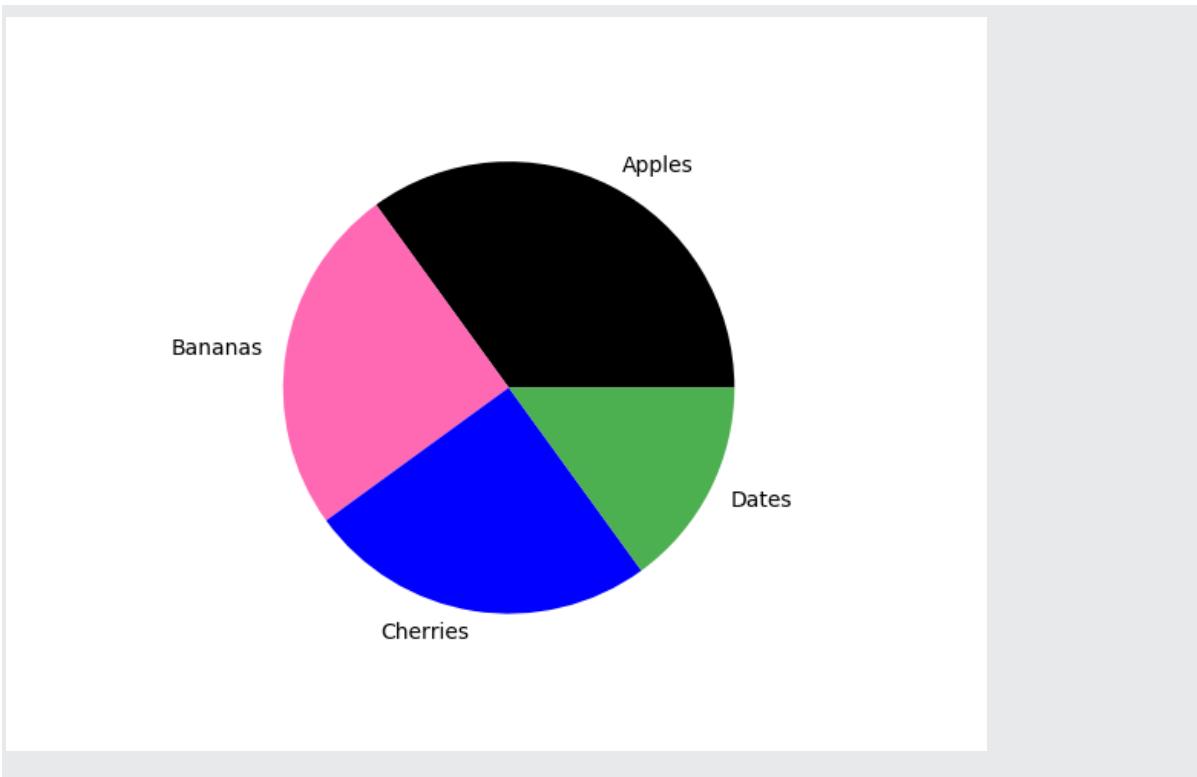
Specify a new color for each wedge:

```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]
mycolors = ["black", "hotpink", "b", "#4CAF50"]

plt.pie(y, labels = mylabels, colors = mycolors)
plt.show()
```

Result:



You can use [Hexadecimal color values](#), any of the [140 supported color names](#), or one of these shortcuts:

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

Legend

To add a list of explanation for each wedge, use the `legend()` function:

Example

Add a legend:

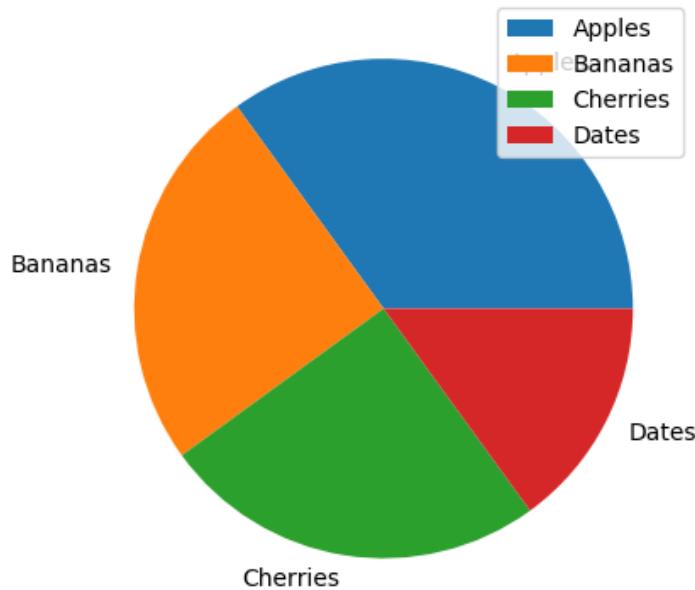
```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
```

```
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
plt.legend()
plt.show()
```

Result:



Legend With Header

To add a header to the legend, add the `title` parameter to the `legend` function.

Example

Add a legend with a header:

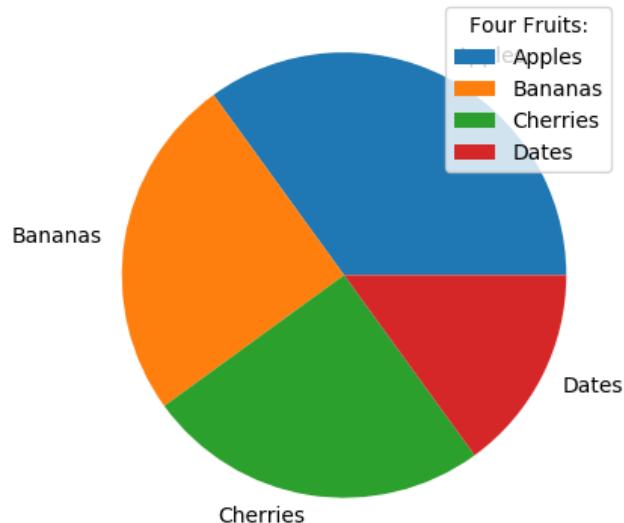
```
import matplotlib.pyplot as plt
import numpy as np

y = np.array([35, 25, 25, 15])
mylabels = ["Apples", "Bananas", "Cherries", "Dates"]

plt.pie(y, labels = mylabels)
```

```
plt.legend(title = "Four Fruits:")
plt.show()
```

Result:



Python Seaborn

Seaborn is a library mostly used for statistical plotting in Python. It is built on top of Matplotlib and provides beautiful default styles and color palettes to make statistical plots more attractive.

In this tutorial, we will learn about Python Seaborn from basics to advance using a huge dataset of seaborn basics, concepts, and different graphs that can be plotted.

Table Of Content

- [Getting Started](#)
- [Using Seaborn with Matplotlib](#)
- [Customizing Seaborn Plots](#)
 - [Changing Figure Aesthetic](#)
 - [Removal of Spines](#)
 - [Changing the figure Size](#)
 - [Scaling the plots](#)
 - [Setting the Style Temporarily](#)
- [Color Palette](#)
 - [Diverging Color Palette](#)
 - [Sequential Color Palette](#)
 - [Setting the default Color Palette](#)
- [Multiple plots with Seaborn](#)
 - [Using Matplotlib](#)
 - [Using Seaborn](#)
- [Creating Different Types of Plots](#)
 - [Relational Plots](#)
 - [Categorical Plots](#)
 - [Distribution Plots](#)
 - [Regression Plots](#)
- [More Gaphs in Seaborn](#)
- [More Topics on Seaborn](#)

Getting Started

First of all, let us install Seaborn. Seaborn can be installed using the pip. Type the below command in the terminal.

```
pip install seaborn
```

In the terminal, it will look like this –

After the installation is completed you will get a successfully installed message at the end of the terminal as shown below.

Note: Seaborn has the following dependencies –

- Python 2.7 or 3.4+
- numpy
- scipy
- pandas
- matplotlib

After the installation let us see an example of a simple plot using Seaborn. We will be plotting a simple line plot using the iris dataset. Iris dataset contains five columns such as Petal Length, Petal Width, Sepal Length, Sepal Width and Species Type. Iris is a flowering plant, the researchers have measured various features of the different iris flowers and recorded them digitally.

Example:

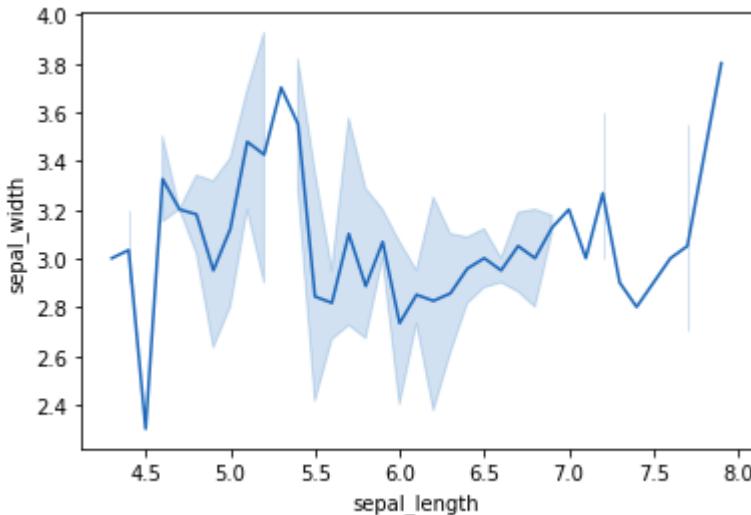
- Python3

```
# importing packages
import seaborn as sns

# loading dataset
data = sns.load_dataset("iris")

# draw lineplot
sns.lineplot(x="sepal_length", y="sepal_width", data=data)
```

Output:



In the above example, a simple line plot is created using the `lineplot()` method. Do not worry about these functions as we will be discussing them in detail in the below sections. Now after going through a simple example let us see a brief introduction about the Seaborn. Refer to the below articles to get detailed information about the same.

- [Introduction to Seaborn – Python](#)
- [Plotting graph using Seaborn](#)

In the introduction, you must have read that Seaborn is built on the top of Matplotlib. It means that Seaborn can be used with Matplotlib.

Using Seaborn with Matplotlib

Using both Matplotlib and Seaborn together is a very simple process. We just have to invoke the Seaborn Plotting function as normal, and then we can use Matplotlib's customization function.

Example 1: We will be using the above example and will add the title to the plot using the Matplotlib.

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")
```

```

# draw lineplot

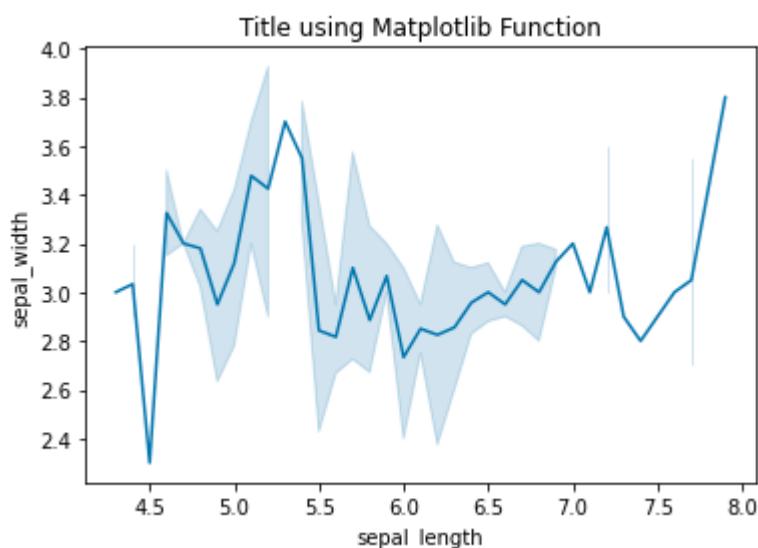
sns.lineplot(x="sepal_length", y="sepal_width", data=data)

# setting the title using Matplotlib
plt.title('Title using Matplotlib Function')

plt.show()

```

Output:



Example 2: Setting the xlim and ylim

- Python3

```

# importing packages

import seaborn as sns

import matplotlib.pyplot as plt

```

```

# loading dataset

data = sns.load_dataset("iris")

# draw lineplot

sns.lineplot(x="sepal_length", y="sepal_width", data=data)

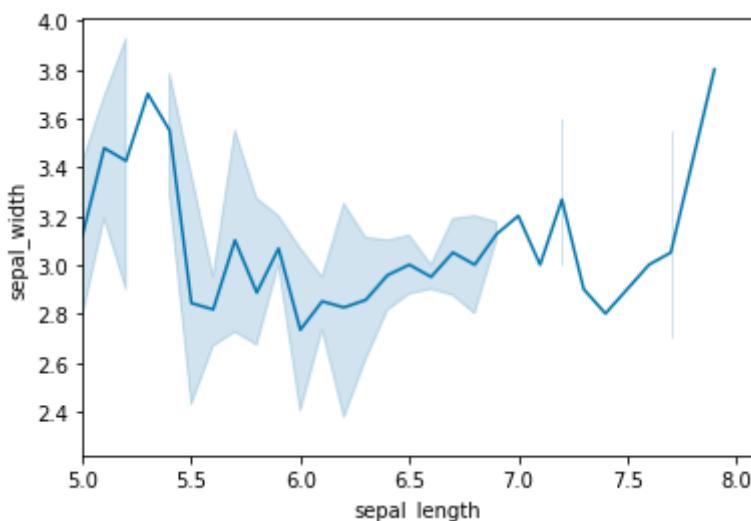
# setting the x limit of the plot

plt.xlim(5)

plt.show()

```

Output:



Customizing Seaborn Plots

Seaborn comes with some customized themes and a high-level interface for customizing the looks of the graphs. Consider the above example where the default of the Seaborn is used. It still looks nice and pretty but we can customize the graph according to our own needs. So let's see the styling of plots in detail.

Changing Figure Aesthetic

set_style() method is used to set the aesthetic of the plot. It means it affects things like the color of the axes, whether the grid is active or not, or other aesthetic elements. There are five themes available in Seaborn.

- darkgrid
- whitegrid
- dark
- white
- ticks

Syntax:

```
set_style(style=None, rc=None)
```

Example: Using the dark theme

- Python3

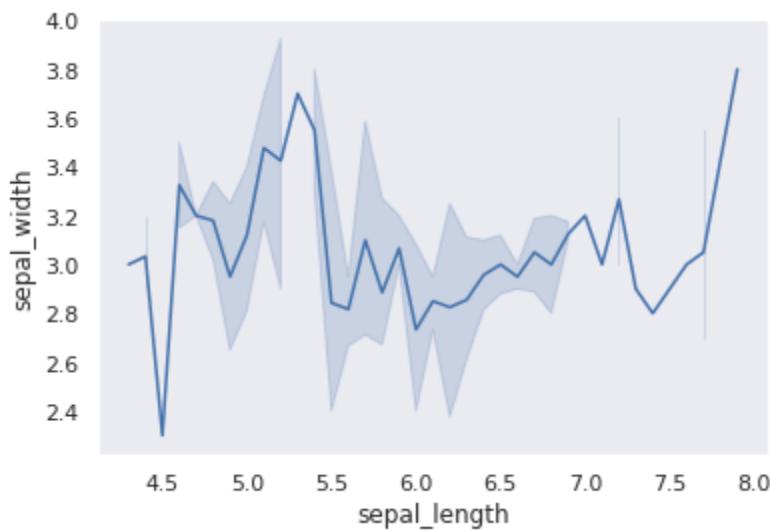
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

# draw lineplot
sns.lineplot(x="sepal_length", y="sepal_width", data=data)

# changing the theme to dark
sns.set_style("dark")
plt.show()
```

Output:



Removal of Spines

Spines are the lines noting the data boundaries and connecting the axis tick marks. It can be removed using the `despine()` method.

Syntax:

```
sns.despine(left = True)
```

Example:

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

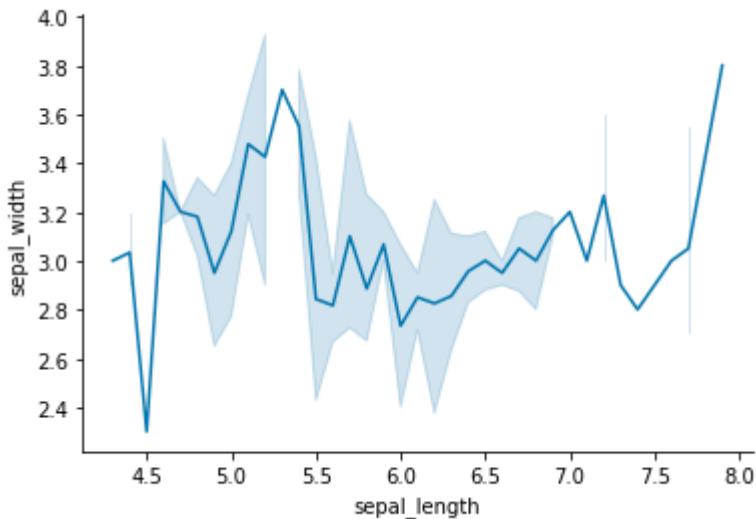
# loading dataset
data = sns.load_dataset("iris")
```

```
# draw lineplot

sns.lineplot(x="sepal_length", y="sepal_width", data=data)

# Removing the spines
sns.despine()
plt.show()
```

Output:



Changing the figure Size

The figure size can be changed using the [figure\(\)](#) method of Matplotlib. `figure()` method creates a new figure of the specified size passed in the `figsize` parameter.

Example:

- Python3

```
# importing packages
```

```

import seaborn as sns

import matplotlib.pyplot as plt


# loading dataset

data = sns.load_dataset("iris")



# changing the figure size

plt.figure(figsize = (2, 4))



# draw lineplot

sns.lineplot(x="sepal_length", y="sepal_width", data=data)



# Removing the spines

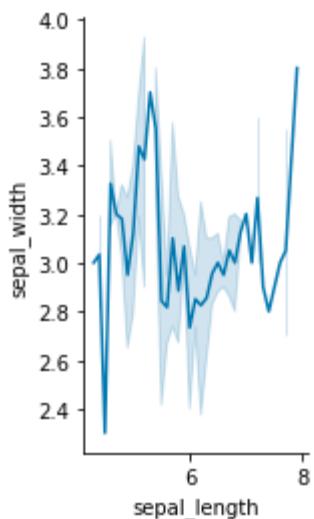
sns.despine()



plt.show()

```

Output:



Scaling the plots

It can be done using the `set_context()` method. It allows us to override default parameters. This affects things like the size of the labels, lines, and other elements of the plot, but not the overall style. The base context is “notebook”, and the other contexts are “paper”, “talk”, and “poster”. `font_scale` sets the font size.

Syntax:

```
set_context(context=None, font_scale=1, rc=None)
```

Example:

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

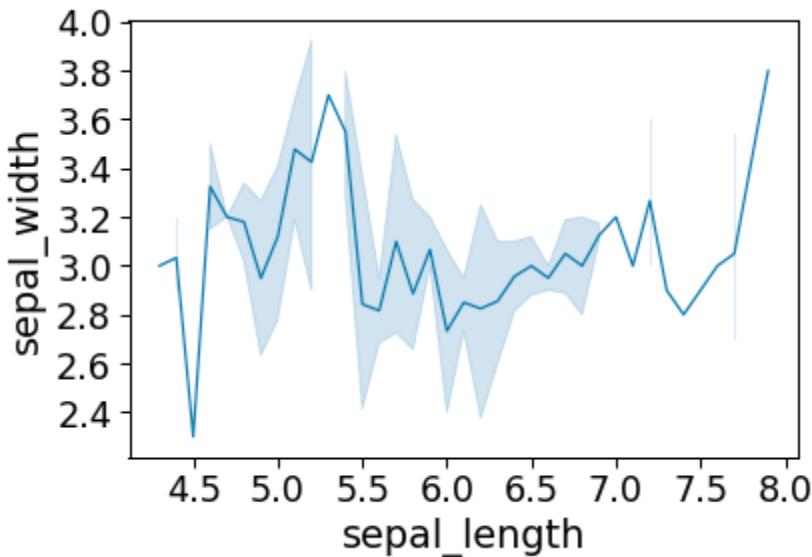
# loading dataset
data = sns.load_dataset("iris")

# draw lineplot
sns.lineplot(x="sepal_length", y="sepal_width", data=data)

# Setting the scale of the plot
sns.set_context("paper")

plt.show()
```

Output:



Setting the Style Temporarily

`axes_style()` method is used to set the style temporarily. It is used along with the **with** statement.

Syntax:

```
axes_style(style=None, rc=None)
```

Example:

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")
```

```

def plot():

    sns.lineplot(x="sepal_length", y="sepal_width", data=data)

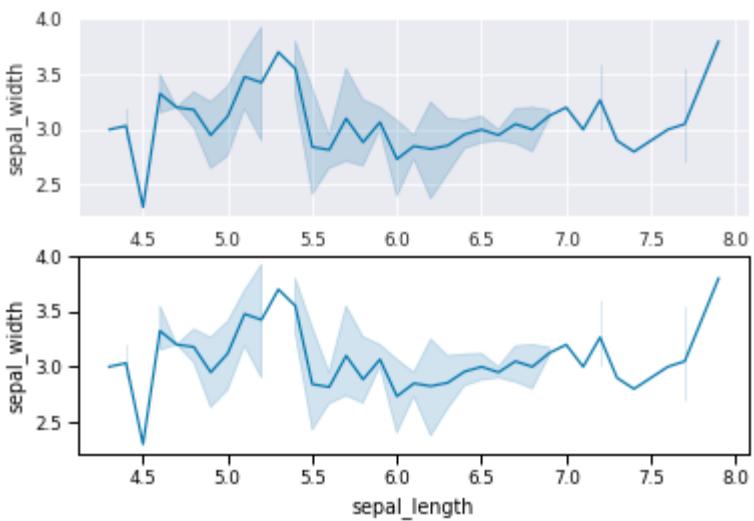
    with sns.axes_style('darkgrid'):

        # Adding the subplot
        plt.subplot(211)
        plot()

        plt.subplot(212)
        plot()

```

Output:



Refer to the below article for detailed information about styling Seaborn Plot.

- [Seaborn | Style And Color](#)

Color Palette

Colormaps are used to visualize plots effectively and easily. One might use different sorts of colormaps for different kinds of plots. **color_palette()** method is used to give colors to the plot. Another function **palplot()** is used to deal with the color palettes and plots the color palette as a horizontal array.

Example:

- Python3

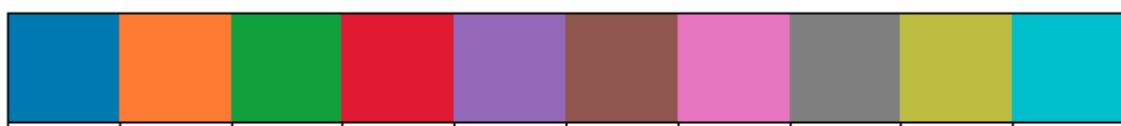
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# current color palette
palette = sns.color_palette()

# plots the color palette as a
# horizontal array
sns.palplot(palette)

plt.show()
```

Output:



Diverging Color Palette

This type of color palette uses two different colors where each color depicts different points ranging from a common point in either direction. Consider a range of -10 to 10 so the value from -10 to 0 takes one color and values from 0 to 10 take another.

Example:

- Python3

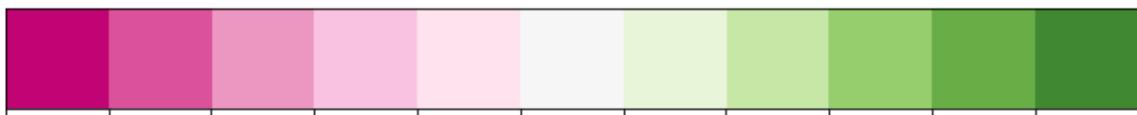
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# current colot palette
palette = sns.color_palette('PiYG', 11)

# diverging color palette
sns.palplot(palette)

plt.show()
```

Output:



In the above example, we have used an in-built diverging color palette which shows 11 different points of color. The color on the left shows pink color and color on the right shows green color.

Sequential Color Palette

A sequential palette is used where the distribution ranges from a lower value to a higher value. To do this add the character 's' to the color passed in the color palette.

Example:

- Python3

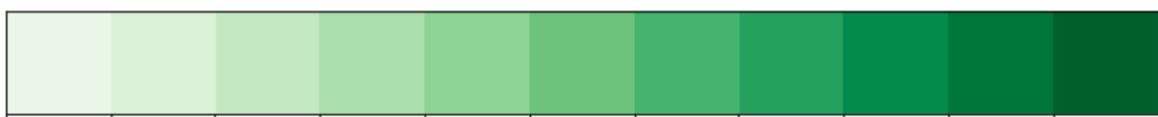
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# current colot palette
palette = sns.color_palette('Greens', 11)

# sequential color palette
sns.palplot(palette)

plt.show()
```

Output:



Setting the default Color Palette

`set_palette()` method is used to set the default color palette for all the plots. The arguments for both `color_palette()` and `set_palette()` is same. `set_palette()` changes the default matplotlib parameters.

Example:

- Python3

```
# importing packages

import seaborn as sns

import matplotlib.pyplot as plt


# loading dataset

data = sns.load_dataset("iris")


def plot():

    sns.lineplot(x="sepal_length", y="sepal_width", data=data)


    # setting the default color palette

    sns.set_palette('vlag')

    plt.subplot(211)


    # plotting with the color palette

    # as vlag

    plot()


    # setting another default color palette

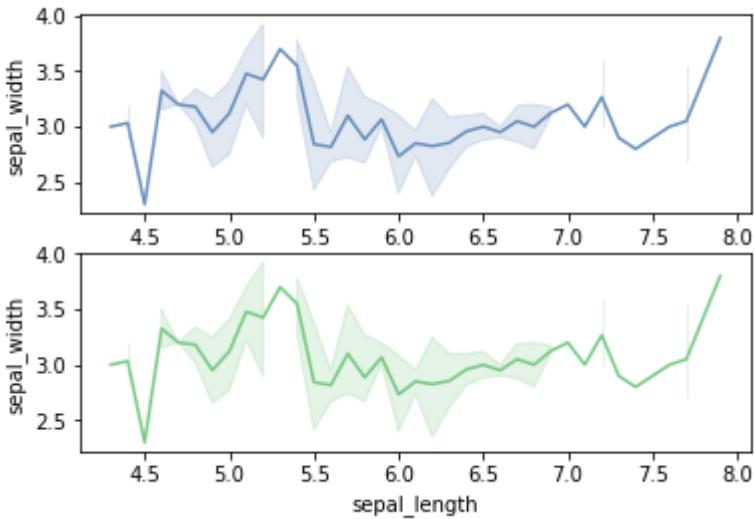
    sns.set_palette('Accent')

    plt.subplot(212)

    plot()
```

```
plt.show()
```

Output:



Refer to the below article to get detailed information about the color palette.

- [Seaborn – Color Palette](#)

Multiple plots with Seaborn

You might have seen multiple plots in the above examples and some of you might have got confused. Don't worry we will cover multiple plots in this section. Multiple plots in Seaborn can also be created using the Matplotlib as well as Seaborn also provides some functions for the same.

Using Matplotlib

Matplotlib provides various functions for plotting subplots. Some of them are [add_axes\(\)](#), [subplot\(\)](#), and [subplot2grid\(\)](#). Let's see an example of each function for better understanding.

Example 1: Using [add_axes\(\)](#) method

- Python3

```
# importing packages

import seaborn as sns

import matplotlib.pyplot as plt


# loading dataset

data = sns.load_dataset("iris")


def graph():

    sns.lineplot(x="sepal_length", y="sepal_width", data=data)


    # Creating a new figure with width = 5 inches

    # and height = 4 inches

    fig = plt.figure(figsize =(5, 4))




    # Creating first axes for the figure

    ax1 = fig.add_axes([0.1, 0.1, 0.8, 0.8])


    # plotting the graph

    graph()


    # Creating second axes for the figure

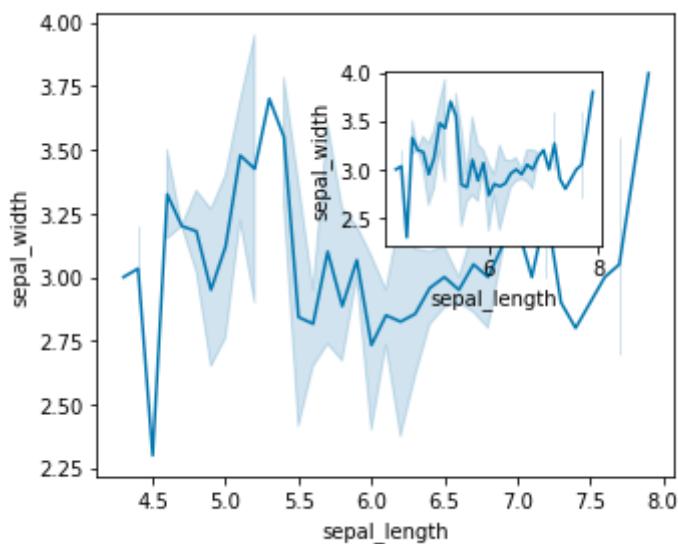
    ax2 = fig.add_axes([0.5, 0.5, 0.3, 0.3])


    # plotting the graph
```

```
graph()
```

```
plt.show()
```

Output:



Example 2: Using [subplot\(\)](#) method

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")
```

```

def graph():

    sns.lineplot(x="sepal_length", y="sepal_width", data=data)

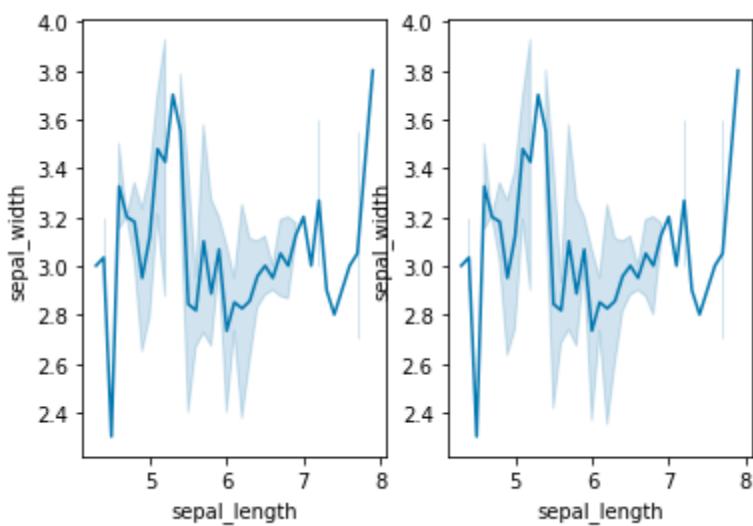
    # Adding the subplot at the specified
    # grid position
    plt.subplot(121)
    graph()

    # Adding the subplot at the specified
    # grid position
    plt.subplot(122)
    graph()

plt.show()

```

Output:



Example 3: Using [subplot2grid\(\)](#) method

- Python3

```
# importing packages

import seaborn as sns

import matplotlib.pyplot as plt


# loading dataset

data = sns.load_dataset("iris")


def graph():

    sns.lineplot(x="sepal_length", y="sepal_width", data=data)


    # adding the subplots

    axes1 = plt.subplot2grid (

        (7, 1), (0, 0), rowspan = 2, colspan = 1)

    graph()

    axes2 = plt.subplot2grid (

        (7, 1), (2, 0), rowspan = 2, colspan = 1)

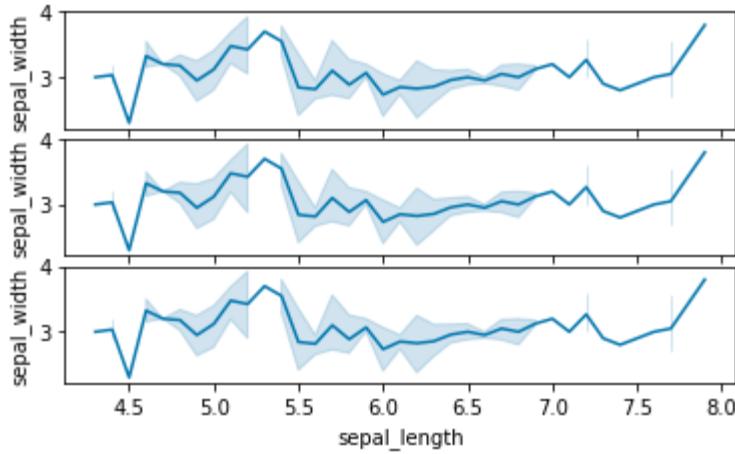
    graph()

    axes3 = plt.subplot2grid (

        (7, 1), (4, 0), rowspan = 2, colspan = 1)

    graph()
```

Output:



Using Seaborn

Seaborn also provides some functions for plotting multiple plots. Let's see them in detail

Method 1: Using [FacetGrid\(\)](#) method

- FacetGrid class helps in visualizing distribution of one variable as well as the relationship between multiple variables separately within subsets of your dataset using multiple panels.
- A FacetGrid can be drawn with up to three dimensions ? row, col, and hue. The first two have obvious correspondence with the resulting array of axes; think of the hue variable as a third dimension along a depth axis, where different levels are plotted with different colors.
- FacetGrid object takes a dataframe as input and the names of the variables that will form the row, column, or hue dimensions of the grid. The variables should be categorical and the data at each level of the variable will be used for a facet along that axis.

Syntax:

```
seaborn.FacetGrid( data, \*|\*kwargs)
```

Example:

- Python3

```
# importing packages

import seaborn as sns

import matplotlib.pyplot as plt

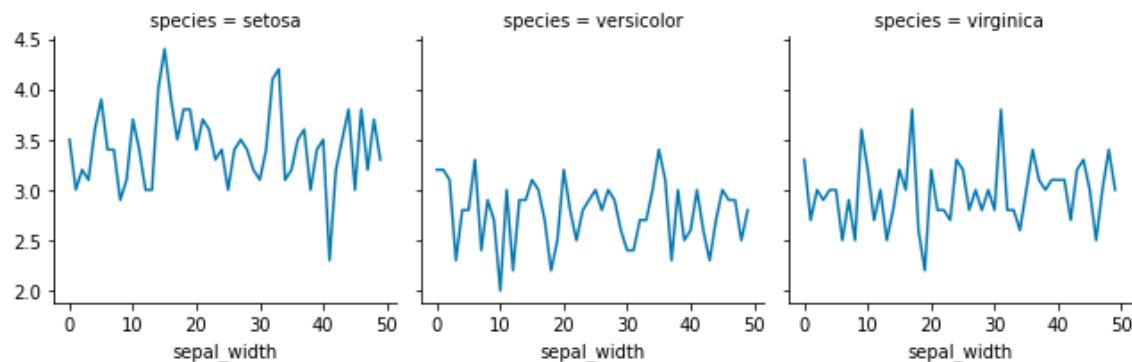

# loading dataset

data = sns.load_dataset("iris")

plot = sns.FacetGrid(data, col="species")
plot.map(plt.plot, "sepal_width")

plt.show()
```

Output:



Method 2: Using [PairGrid\(\)](#) method

- Subplot grid for plotting pairwise relationships in a dataset.
- This class maps each variable in a dataset onto a column and row in a grid of multiple axes. Different axes-level plotting functions can be used to draw bivariate plots in the upper and lower triangles, and the marginal distribution of each variable can be shown on the diagonal.

- It can also represent an additional level of conventionalization with the hue parameter, which plots different subsets of data in different colors. This uses color to resolve elements on a third dimension, but only draws subsets on top of each other and will not tailor the hue parameter for the specific visualization the way that axes-level functions that accept hue will.

Syntax:

```
seaborn.PairGrid( data, \*|\*kwargs)
```

Example:

- Python3

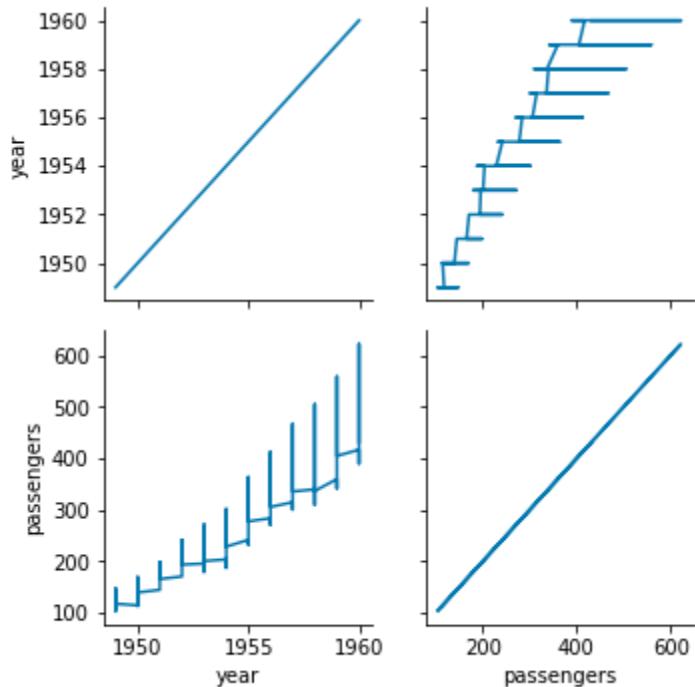
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("flights")

plot = sns.PairGrid(data)
plot.map(plt.plot)

plt.show()
```

Output:



Refer to the below articles to get detailed information about the multiple plots

- [Python – seaborn.FacetGrid\(\) method](#)
- [Python – seaborn.PairGrid\(\) method](#)

Creating Different Types of Plots

Relational Plots

Relational plots are used for visualizing the statistical relationship between the data points. Visualization is necessary because it allows the human to see trends and patterns in the data. The process of understanding how the variables in the dataset relate each other and their relationships are termed as Statistical analysis. Refer to the below articles for detailed information.

- [Relational plots in Seaborn – Part I](#)
- [Relational plots in Seaborn – Part II](#)

There are different types of Relational Plots. We will discuss each of them in detail –

Relplot()

This function provides us the access to some other different axes-level functions which shows the relationships between two variables with semantic mappings of subsets. It is plotted using the **relplot()** method.

Syntax:

```
seaborn.relplot(x=None, y=None, data=None, **kwargs)
```

Example:

- Python3

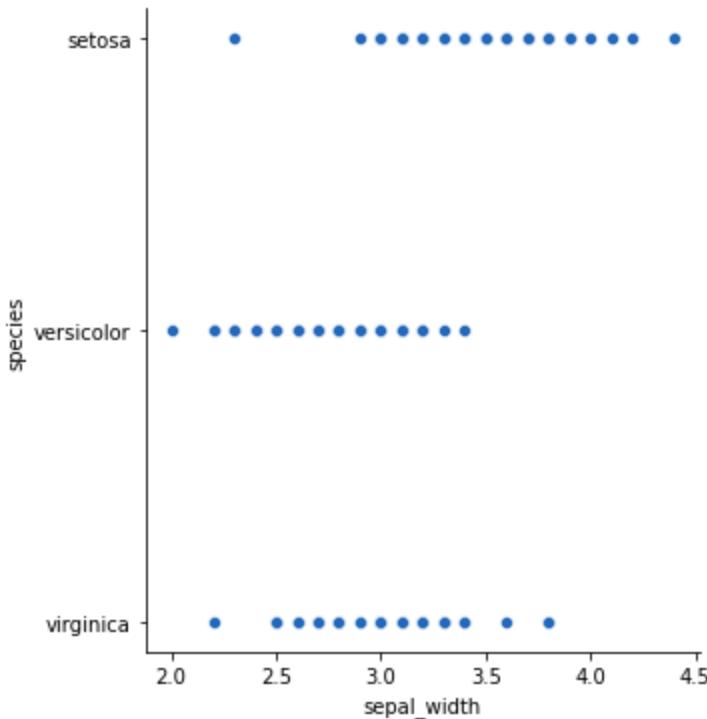
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

# creating the relplot
sns.relplot(x='sepal_width', y='species', data=data)

plt.show()
```

Output:



Scatter Plot

The **scatter plot** is a mainstay of statistical visualization. It depicts the joint distribution of two variables using a cloud of points, where each point represents an observation in the dataset. This depiction allows the eye to infer a substantial amount of information about whether there is any meaningful relationship between them. It is plotted using the **scatterplot()** method.

Syntax:

```
seaborn.scatterplot(x=None, y=None, data=None, **kwargs)
```

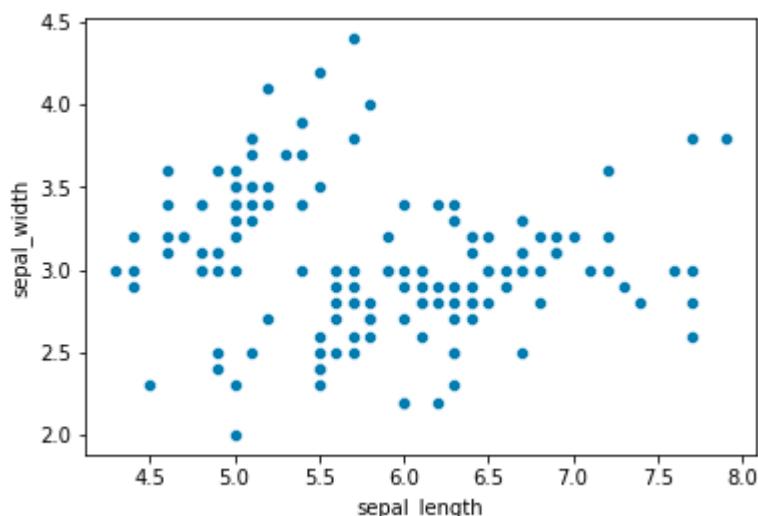
Example:

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt
```

```
# loading dataset  
  
data = sns.load_dataset("iris")  
  
sns.scatterplot(x='sepal_length', y='sepal_width', data=data)  
plt.show()
```

Output:



Refer to the below articles to get detailed information about Scatter plot.

- [Scatterplot using Seaborn in Python](#)
- [Visualizing Relationship between variables with scatter plots in Seaborn](#)
- [How To Make Scatter Plot with Regression Line using Seaborn in Python?](#)
- [Scatter Plot with Marginal Histograms in Python with Seaborn](#)

Line Plot

For certain datasets, you may want to consider changes as a function of time in one variable, or as a similarly continuous variable. In this case, drawing a line-plot is a better option. It is plotted using the [lineplot\(\)](#) method.

Syntax:

```
seaborn.lineplot(x=None, y=None, data=None, **kwargs)
```

Example:

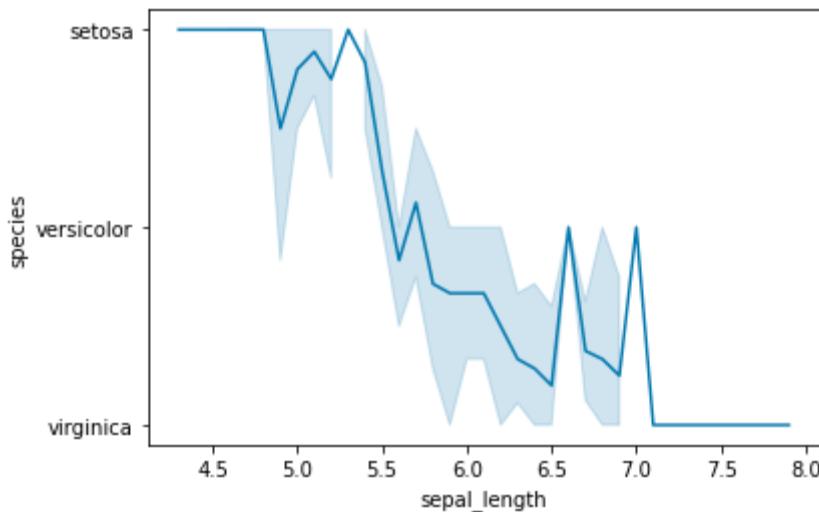
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.lineplot(x='sepal_length', y='species', data=data)
plt.show()
```

Output:



Refer to the below articles to get detailed information about line plot.

- [seaborn.lineplot\(\) method in Python](#)
- [Data Visualization with Seaborn Line Plot](#)
- [Creating A Time Series Plot With Seaborn And Pandas](#)
- [How to Make a Time Series Plot with Rolling Average in Python?](#)

Categorical Plots

Categorical Plots are used where we have to visualize relationship between two numerical values. A more specialized approach can be used if one of the main variable is **categorical** which means such variables that take on a fixed and limited number of possible values.

Refer to the below articles to get detailed information.

- [Categorical Plots](#)

There are various types of categorical plots let's discuss each one them in detail.

Bar Plot

A **barplot** is basically used to aggregate the categorical data according to some methods and by default its the mean. It can also be understood as a visualization of the group by action. To use this plot we choose a categorical column for the x axis and a numerical column for the y axis and we see that it creates a plot taking a mean per categorical column. It can be created using the [barplot\(\)](#) method.

Syntax:

barplot([x, y, hue, data, order, hue_order, ...])

Example:

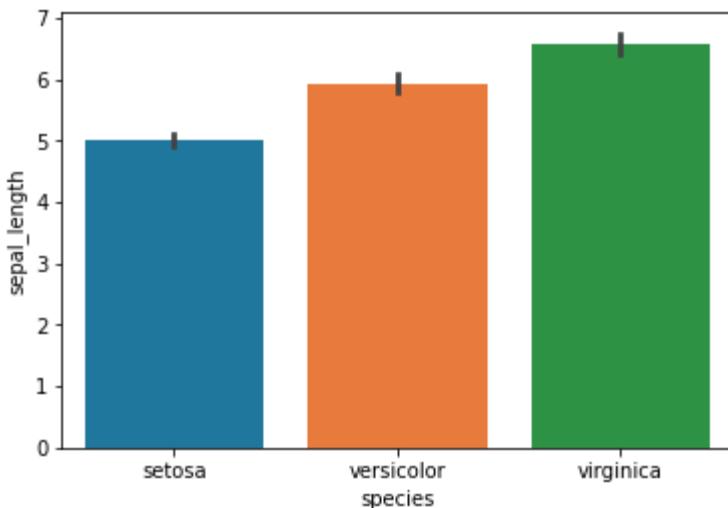
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.barplot(x='species', y='sepal_length', data=data)
plt.show()
```

Output:



Refer to the below article to get detailed information about the topic.

- [Seaborn.barplot\(\) method in Python](#)
- [Barplot using seaborn in Python](#)
- [Seaborn – Sort Bars in Barplot](#)

Count Plot

A **countplot** basically counts the categories and returns a count of their occurrences. It is one of the most simple plots provided by the seaborn library. It can be created using the **countplot()** method.

Syntax:

```
countplot([x, y, hue, data, order, ...])
```

Example:

- Python3

```
# importing packages  
import seaborn as sns
```

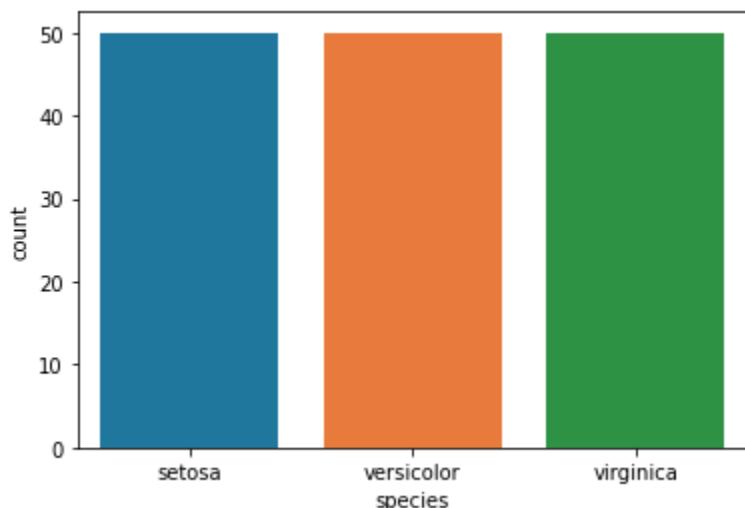
```
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.countplot(x='species', data=data)

plt.show()
```

Output:



Refer to the below articles to get detailed information about the count plot.

- [Countplot using seaborn in Python](#)

Box Plot

A **boxplot** is sometimes known as the box and whisker plot. It shows the distribution of the quantitative data that represents the comparisons between variables. boxplot shows the quartiles of the dataset while the whiskers extend

to show the rest of the distribution i.e. the dots indicating the presence of outliers. It is created using the **boxplot()** method.

Syntax:

boxplot([x, y, hue, data, order, hue_order, ...])

Example:

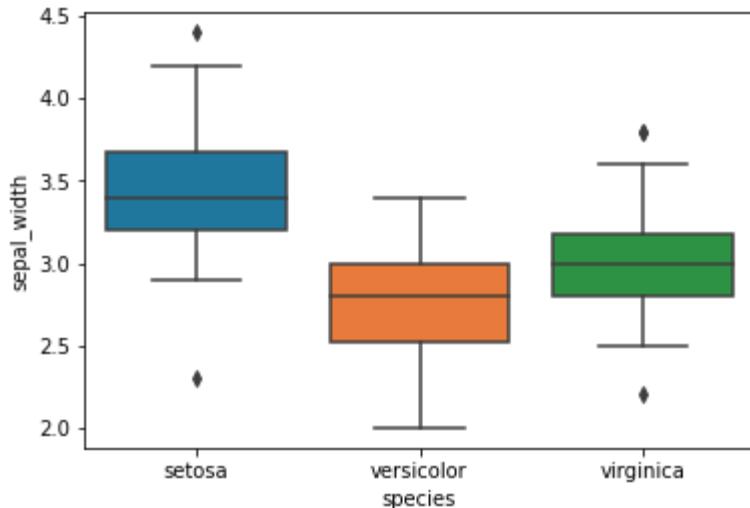
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.boxplot(x='species', y='sepal_width', data=data)
plt.show()
```

Output:



Refer to the below articles to get detailed information about box plot.

- [Boxplot using Seaborn in Python](#)
- [Horizontal Boxplots with Seaborn in Python](#)
- [How To Use Seaborn Color Palette to Color Boxplot?](#)
- [Seaborn – Coloring Boxplots with Palettes](#)
- [How to Show Mean on Boxplot using Seaborn in Python?](#)
- [Sort Boxplot by Mean with Seaborn in Python](#)
- [How To Manually Order Boxplot in Seaborn?](#)
- [Grouped Boxplots in Python with Seaborn](#)
- [Horizontal Boxplots with Points using Seaborn in Python](#)
- [How to Make Boxplots with Data Points using Seaborn in Python?](#)
- [Box plot visualization with Pandas and Seaborn](#)

Violinplot

It is similar to the boxplot except that it provides a higher, more advanced visualization and uses the kernel density estimation to give a better description about the data distribution. It is created using the **violinplot()** method.

Syntax:

```
violinplot([x, y, hue, data, order, ...])
```

Example:

- Python3

```

# importing packages

import seaborn as sns

import matplotlib.pyplot as plt


# loading dataset

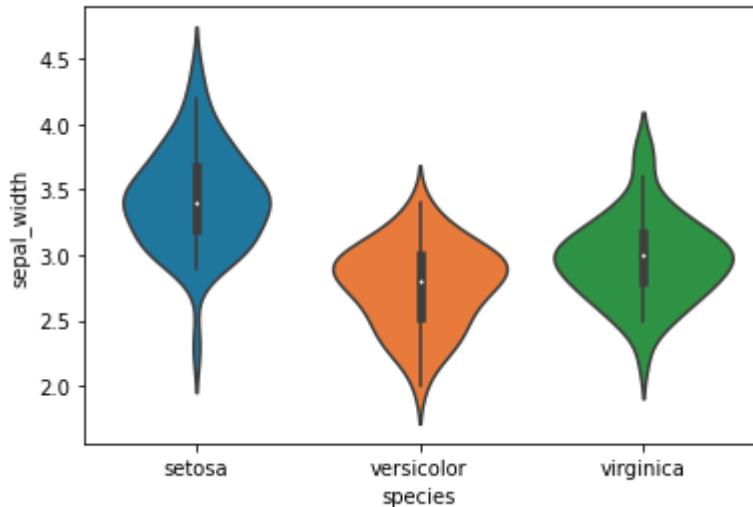
data = sns.load_dataset("iris")

sns.violinplot(x='species', y='sepal_width', data=data)

plt.show()

```

Output:



Refer to the below articles to get detailed information about violin plot.

- [Violinplot using Seaborn in Python](#)
- [How to Make Horizontal Violin Plot with Seaborn in Python?](#)
- [Make Violinplot with data points using Seaborn](#)
- [How To Make Violinpot with data points in Seaborn?](#)
- [How to Make Grouped Violinplot with Seaborn in Python?](#)

Stripplot

It basically creates a scatter plot based on the category. It is created using the **stripplot()** method.

Syntax:

```
stripplot([x, y, hue, data, order, ...])
```

Example:

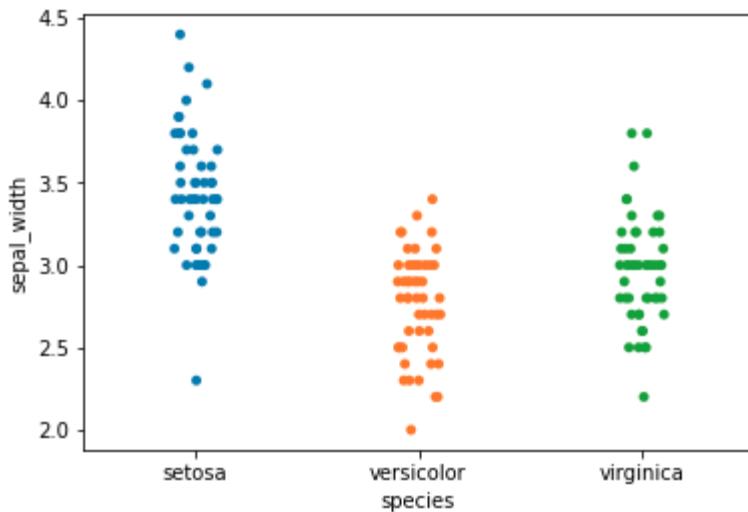
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.stripplot(x='species', y='sepal_width', data=data)
plt.show()
```

Output:



Refer to the below articles to detailed information about strip plot.

- [Stripplot using Seaborn in Python](#)

Swarmplot

Swarmplot is very similar to the stripplot except the fact that the points are adjusted so that they do not overlap. Some people also like combining the idea of a violin plot and a stripplot to form this plot. One drawback to using swarmplot is that sometimes they dont scale well to really large numbers and takes a lot of computation to arrange them. So in case we want to visualize a swarmplot properly we can plot it on top of a violinplot. It is plotted using the [`swarmplot\(\)`](#) method.

Syntax:

```
swarmplot([x, y, hue, data, order, ...])
```

Example:

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt
```

```

# loading dataset

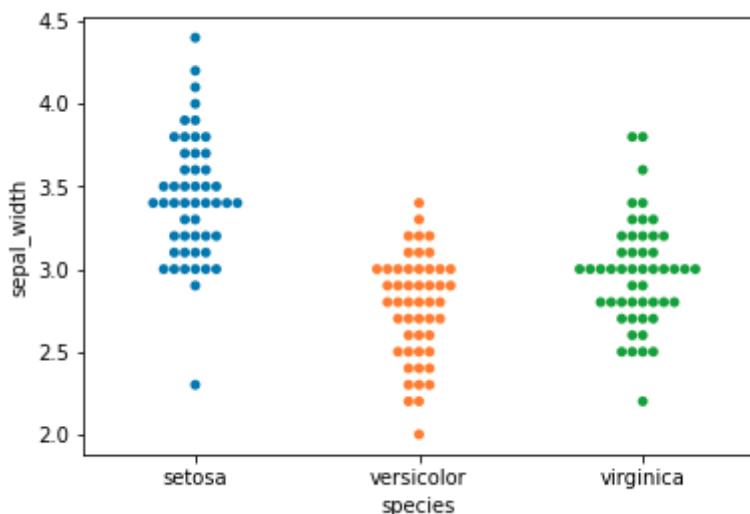
data = sns.load_dataset("iris")

sns.swarmplot(x='species', y='sepal_width', data=data)

plt.show()

```

Output:



Refer to the below articles to get detailed information about swarmplot.

- [Python – seaborn.swarmplot\(\) method](#)
- [Swarmplot using Seaborn in Python](#)

Factorplot

Factorplot is the most general of all these plots and provides a parameter called kind to choose the kind of plot we want thus saving us from the trouble of writing these plots separately. The kind parameter can be bar, violin, swarm etc. It is plotted using the [factorplot\(\)](#) method.

Syntax:

sns.factorplot([x, y, hue, data, row, col, ...])

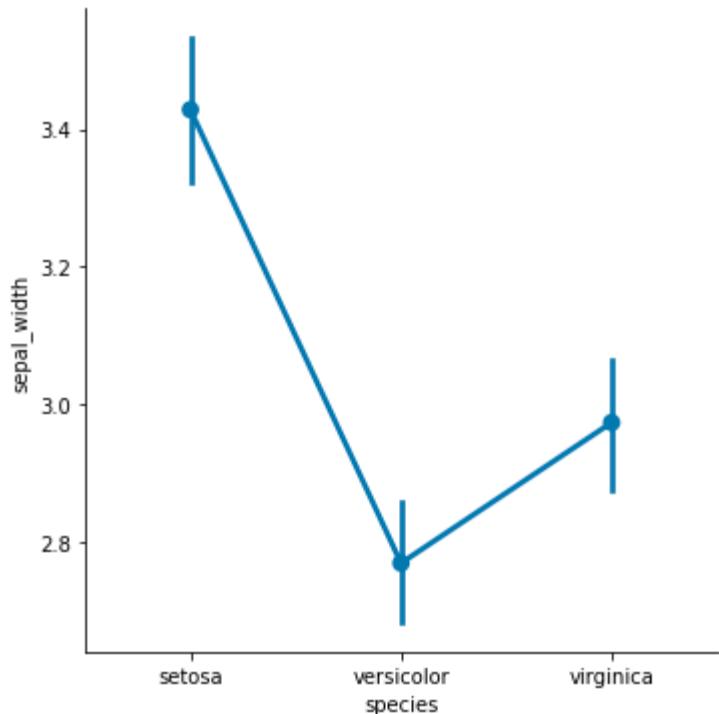
Example:

- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.factorplot(x='species', y='sepal_width', data=data)
plt.show()
```



Refer to the below articles to get detailed information about the factor plot.

- [Python – seaborn.factorplot\(\) method](#)
- [Plotting different types of plots using Factor plot in seaborn](#)

Distribution Plots

Distribution Plots are used for examining univariate and bivariate distributions meaning such distributions that involve one variable or two discrete variables.

Refer to the below article to get detailed information about the distribution plots.

- [Distribution Plots](#)

There are various types of distribution plots let's discuss each one them in detail.

Histogram

A histogram is basically used to represent data provided in a form of some groups. It is accurate method for the graphical representation of numerical data distribution. It can be plotted using the **histplot()** function.

Syntax:

```
histplot(data=None, *, x=None, y=None, hue=None, **kwargs)
```

Example:

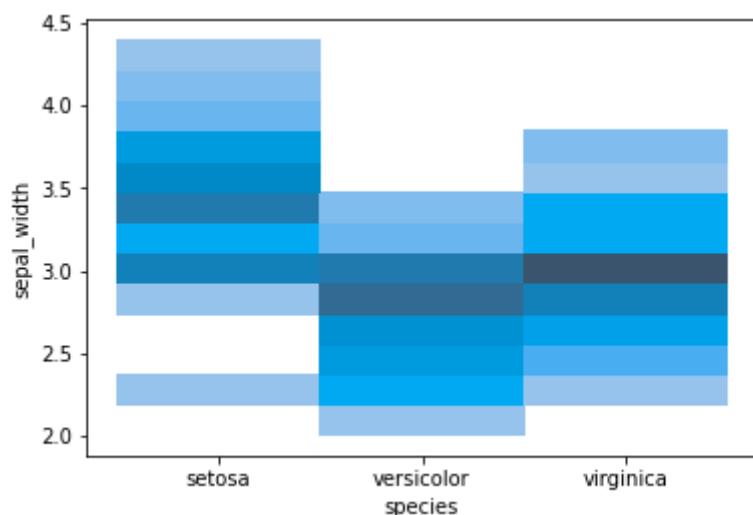
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.histplot(x='species', y='sepal_width', data=data)
plt.show()
```

Output:



Refer to the below articles to get detailed information about histplot.

- [How to Make Histograms with Density Plots with Seaborn histplot?](#)
- [How to Add Outline or Edge Color to Histogram in Seaborn?](#)
- [Scatter Plot with Marginal Histograms in Python with Seaborn](#)

Distplot

Distplot is used basically for univariant set of observations and visualizes it through a histogram i.e. only one observation and hence we choose one particular column of the dataset. It is potted using the **distplot()** method.

Syntax:

```
distplot(a[, bins, hist, kde, rug, fit, ...])
```

Example:

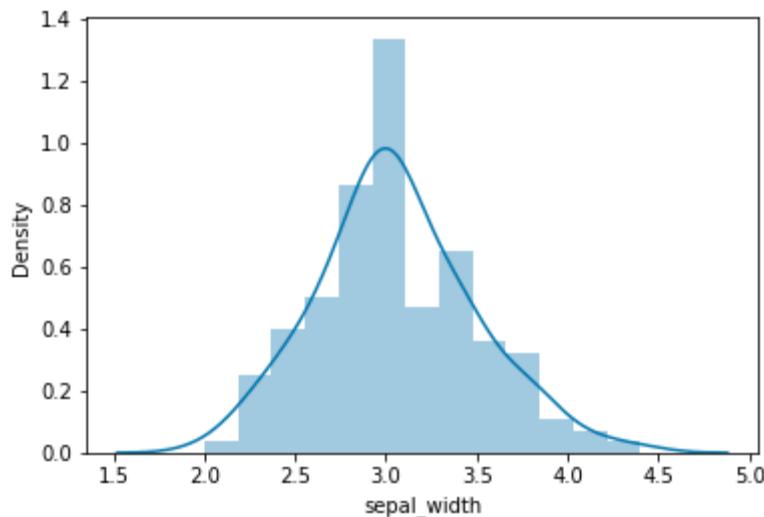
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.distplot(data['sepal_width'])
plt.show()
```

Output:



Jointplot

Jointplot is used to draw a plot of two variables with bivariate and univariate graphs. It basically combines two different plots. It is plotted using the [`jointplot\(\)`](#) method.

Syntax:

```
jointplot(x, y[, data, kind, stat_func, ...])
```

Example:

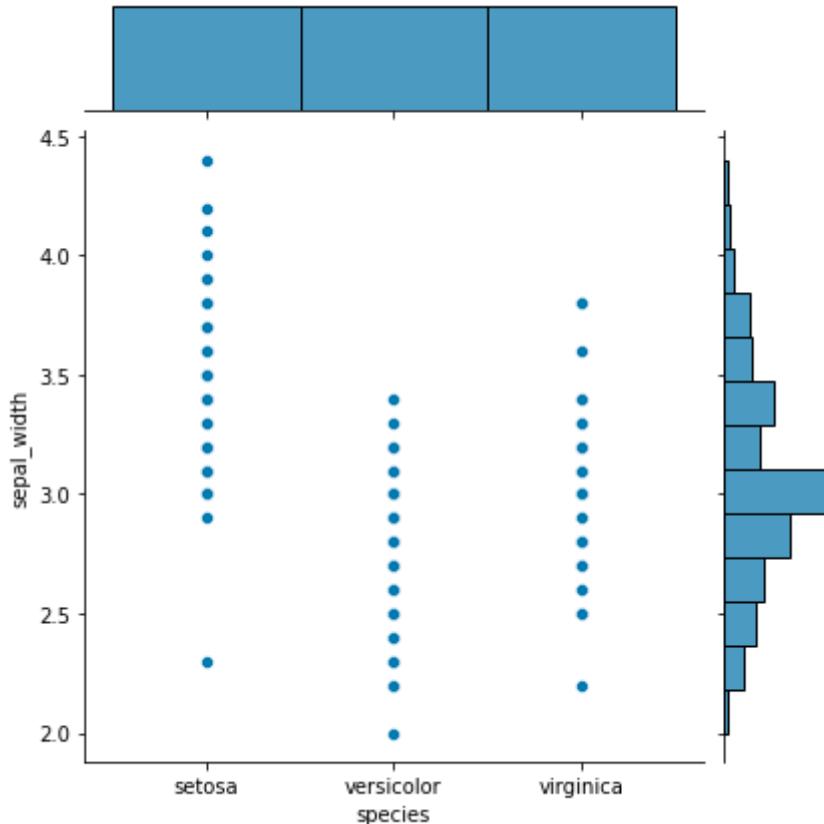
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.jointplot(x='species', y='sepal_width', data=data)
plt.show()
```

Output:



Refer to the below articles to get detailed information about the topic.

- [Python – seaborn.jointplot\(\) method](#)

Pairplot

Pairplot represents pairwise relation across the entire dataframe and supports an additional argument called hue for categorical separation. What it does basically is create a jointplot between every possible numerical column and takes a while if the dataframe is really huge. It is plotted using the [pairplot\(\)](#) method.

Syntax:

```
pairplot(data[, hue, hue_order, palette, ...])
```

Example:

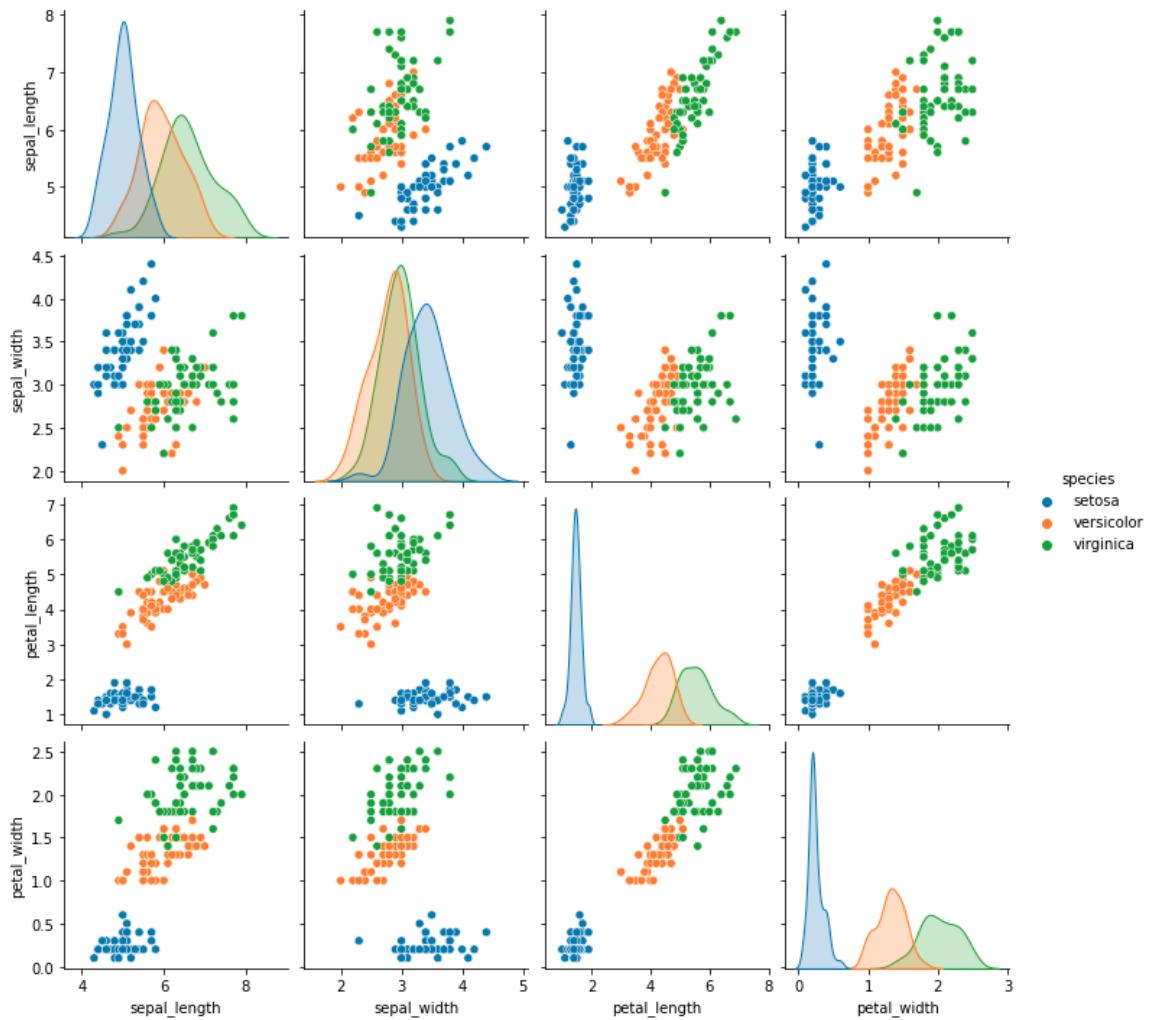
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.pairplot(data=data, hue='species')
plt.show()
```

Output:



Refer to the below articles to get detailed information about the pairplot.

- [Python – seaborn.pairplot\(\) method](#)
- [Data visualization with Pairplot Seaborn and Pandas](#)

Rugplot

Rugplot plots datapoints in an array as sticks on an axis. Just like a distplot it takes a single column. Instead of drawing a histogram it creates dashes all across the plot. If you compare it with the jointplot you can see that what a jointplot does is that it counts the dashes and shows it as bins. It is plotted using the **rugplot()** method.

Syntax:

```
rugplot(a[, height, axis, ax])
```

Example:

- Python3

```
# importing packages

import seaborn as sns

import matplotlib.pyplot as plt

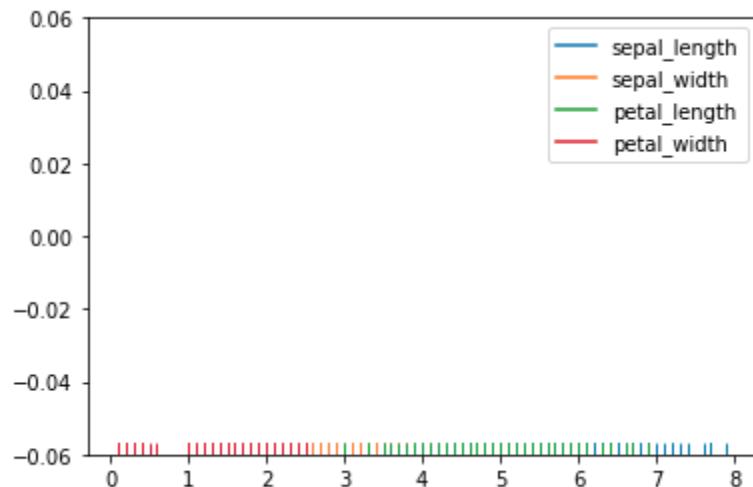

# loading dataset

data = sns.load_dataset("iris")

sns.rugplot(data=data)

plt.show()
```

Output:



KDE Plot

KDE Plot described as **Kernel Density Estimate** is used for visualizing the Probability Density of a continuous variable. It depicts the probability density at different values in a continuous variable. We can also plot a single graph for multiple samples which helps in more efficient data visualization.

Syntax:

```
seaborn.kdeplot(x=None, *, y=None, vertical=False, palette=None, **kwargs)
```

Example:

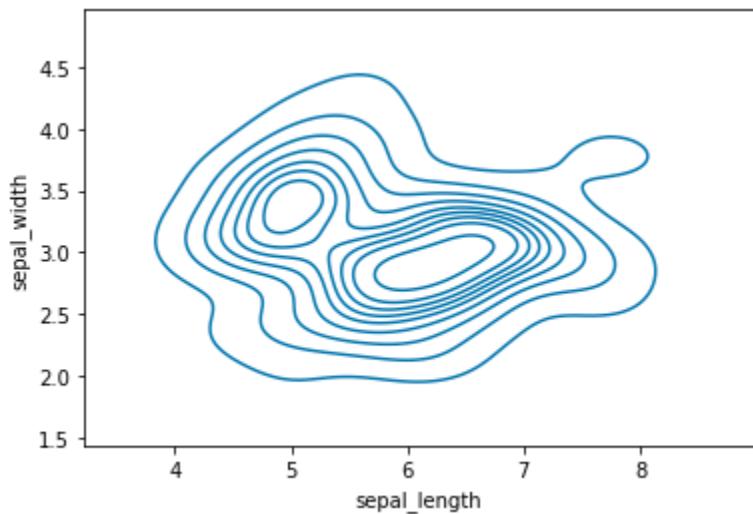
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("iris")

sns.kdeplot(x='sepal_length', y='sepal_width', data=data)
plt.show()
```

Output:



Refer to the below articles to get detailed information about the topic.

- [Seaborn Kdeplot – A Comprehensive Guide](#)
- [KDE Plot Visualization with Pandas and Seaborn](#)

Regression Plots

The **regression plots** are primarily intended to add a visual guide that helps to emphasize patterns in a dataset during exploratory data analyses. Regression plots as the name suggests creates a regression line between two parameters and helps to visualize their linear relationships.

Refer to the below article to get detailed information about the regression plots.

- [Regression Plots](#)

there are two main functions that are used to draw linear regression models. These functions are lmplot(), and regplot(), are closely related to each other. They even share their core functionality.

lmplot

lmplot() method can be understood as a function that basically creates a linear model plot. It creates a scatter plot with a linear fit on top of it.

Syntax:

```
seaborn.lmplot(x, y, data, hue=None, col=None, row=None, **kwargs)
```

Example:

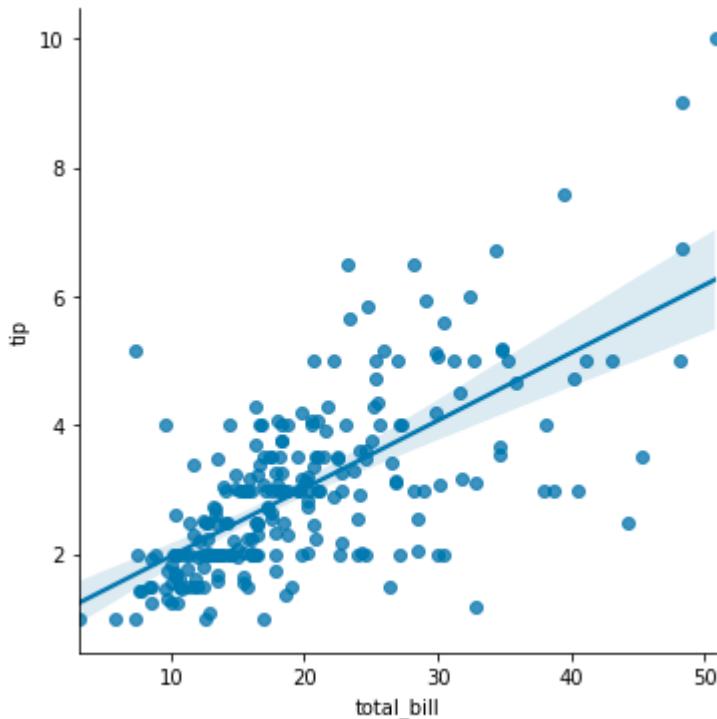
- Python3

```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("tips")

sns.lmplot(x='total_bill', y='tip', data=data)
plt.show()
```

Output:



Refer to the below articles to get detailed information about the lmplot.

- [Python – seaborn.lmplot\(\) method](#)

Regplot

[regplot\(\)](#) method is also similar to lmplot which creates linear regression model.

Syntax:

```
seaborn.regplot(x, y, data=None, x_estimator=None, **kwargs)
```

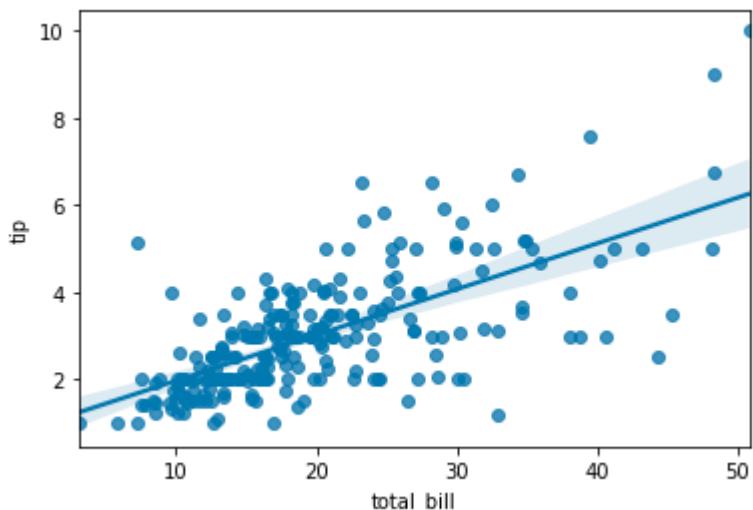
Example:

- Python3

```
# importing packages  
import seaborn as sns  
import matplotlib.pyplot as plt
```

```
# loading dataset  
data = sns.load_dataset("tips")  
  
sns.regplot(x='total_bill', y='tip', data=data)  
plt.show()
```

Output:



Refer to the below articles to get detailed information about regplot.

- [Python – seaborn.regplot\(\) method](#)

Note: The difference between both the function is that regplot accepts the x, y variables in different format including NumPy arrays, Pandas objects, whereas, the lmplot only accepts the value as strings.

Matrix Plots

A **matrix plot** means plotting matrix data where color coded diagrams shows rows data, column data and values. It can be shown using the heatmap and clustermap.

Refer to the below articles to get detailed information about the matrix plots.

- [Matrix plots](#)

Heatmap

Heatmap is defined as a graphical representation of data using colors to visualize the value of the matrix. In this, to represent more common values or higher activities brighter colors basically reddish colors are used and to represent less common or activity values, darker colors are preferred. It can be plotted using the **heatmap()** function.

Syntax:

```
seaborn.heatmap(data, *, vmin=None, vmax=None, cmap=None, center=None,  
annot_kws=None, linewidths=0, linecolor='white', cbar=True, **kwargs)
```

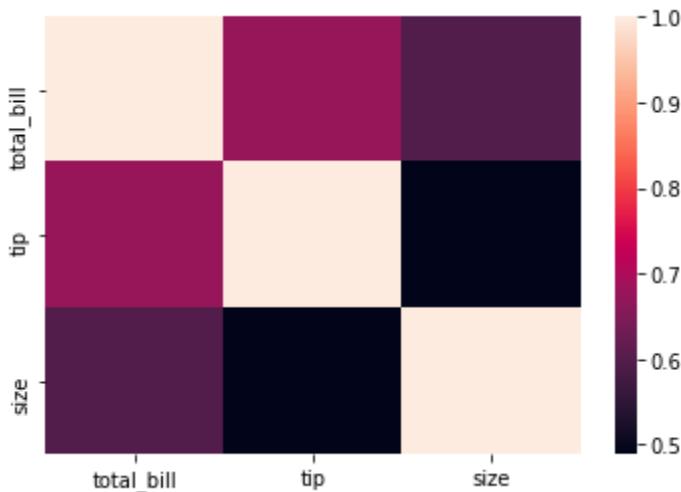
Example:

- Python3

```
# importing packages  
  
import seaborn as sns  
  
import matplotlib.pyplot as plt  
  
  
# loading dataset  
  
data = sns.load_dataset("tips")  
  
  
# correlation between the different parameters  
  
tc = data.corr()
```

```
sns.heatmap(tc)  
plt.show()
```

Output:



Refer to the below articles to get detailed information about the heatmap.

- [Seaborn Heatmap – A comprehensive guide](#)
- [How to create a seaborn correlation heatmap in Python?](#)
- [How to create a Triangle Correlation Heatmap in seaborn – Python?](#)
- [ColorMaps in Seaborn HeatMaps](#)
- [How to change the colorbar size of a seaborn heatmap figure in Python?](#)
- [How to add a frame to a seaborn heatmap figure in Python?](#)
- [How to increase the size of the annotations of a seaborn heatmap in Python?](#)

Clustermap

The clustermap() function of seaborn plots the hierarchically-clustered heatmap of the given matrix dataset. [Clustering](#) simply means grouping data based on relationship among the variables in the data.

Syntax:

```
clustermap(data, *, pivot_kws=None, **kwargs)
```

Example:

- Python3

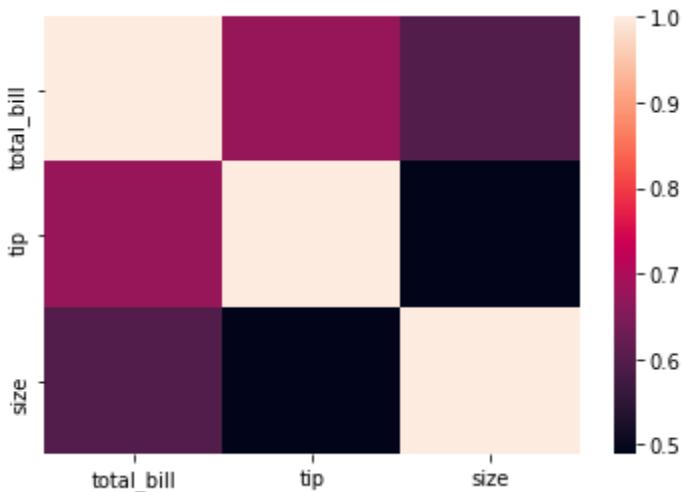
```
# importing packages
import seaborn as sns
import matplotlib.pyplot as plt

# loading dataset
data = sns.load_dataset("tips")

# correlation between the different parameters
tc = data.corr()

sns.clustermap(tc)
plt.show()
```

Output:



Web Scraping with BeautifulSoup

For More Practice

<https://www.w3resource.com/python-exercises/BeautifulSoup/index.php>

What Is Web Scraping?

Web scraping is the process of gathering information from the Internet. Even copying and pasting the lyrics of your favorite song is a form of web scraping! However, the words “web scraping” usually refer to a process that involves automation. [Some websites don’t like it when automatic scrapers gather their data](#), while others don’t mind.

If you’re scraping a page respectfully for educational purposes, then you’re unlikely to have any problems. Still, it’s a good idea to do some research on your own and make sure that you’re not [violating any Terms of Service](#) before you start a large-scale project.

Reasons for Web Scraping

Say you’re a surfer, both online and in real life, and you’re looking for employment. However, you’re not looking for just *any* job. With a surfer’s mindset, you’re waiting for the perfect opportunity to roll your way!

There’s a job site that offers precisely the kinds of jobs you want. Unfortunately, a new position only pops up once in a blue moon, and the site doesn’t provide an email notification service. You think about checking up on it every day, but that doesn’t sound like the most fun and productive way to spend your time.

Thankfully, the world offers other ways to apply that surfer’s mindset! Instead of looking at the job site every day, you can use Python to help automate your job search’s repetitive parts. **Automated web scraping** can be a solution to speed up the data collection process. You write your code once, and it will get the information you want many times and from many pages.

In contrast, when you try to get the information you want manually, you might spend a lot of time clicking, scrolling, and searching, especially if you need large amounts of data from websites that are regularly updated with new content. Manual web scraping can take a lot of time and repetition.

There's so much information on the Web, and new information is constantly added. You'll probably be interested in at least some of that data, and much of it is just out there for the taking. Whether you're actually on the job hunt or you want to download all the lyrics of your favorite artist, automated web scraping can help you accomplish your goals.

Challenges of Web Scraping

The Web has grown organically out of many sources. It combines many different technologies, styles, and personalities, and it continues to grow to this day. In other words, the Web is a hot mess! Because of this, you'll run into some challenges when scraping the Web:

- **Variety:** Every website is different. While you'll encounter general structures that repeat themselves, each website is unique and will need personal treatment if you want to extract the relevant information.
- **Durability:** Websites constantly change. Say you've built a shiny new web scraper that automatically cherry-picks what you want from your resource of interest. The first time you [run your script](#), it works flawlessly. But when you run the same script only a short while later, you run into a discouraging and lengthy stack of [tracebacks](#)!

Unstable scripts are a realistic scenario, as many websites are in active development. Once the site's structure has changed, your scraper might not be able to navigate the sitemap correctly or find the relevant information. The good news is that many changes to websites are small and incremental, so you'll likely be able to update your scraper with only minimal adjustments.

However, keep in mind that because the Internet is dynamic, the scrapers you'll build will probably require constant maintenance. You can set up [continuous integration](#) to run scraping tests periodically to ensure that your main script doesn't break without your knowledge.

An Alternative to Web Scraping: APIs

Some website providers offer [application programming interfaces \(APIs\)](#) that allow you to access their data in a predefined manner. With APIs, you can avoid parsing HTML. Instead, you can access the data directly using formats like [JSON](#) and XML. HTML is primarily a way to present content to users visually.

When you use an API, the process is generally more stable than gathering the data through web scraping. That's because developers create APIs to be consumed by programs rather than by human eyes.

The front-end presentation of a site might change often, but such a change in the website's design doesn't affect its API structure. The structure of an API is usually more permanent, which means it's a more reliable source of the site's data.

However, APIs *can* change as well. The challenges of both variety and durability apply to APIs just as they do to websites. Additionally, it's much harder to inspect the structure of an API by yourself if the provided documentation lacks quality.

The approach and tools you need to gather information using APIs are outside the scope of this tutorial. To learn more about it, check out [API Integration in Python](#).

Implementing Web Scraping in Python with BeautifulSoup

There are mainly two ways to extract data from a website:

- Use the API of the website (if it exists). For example, Facebook has the Facebook Graph API which allows retrieval of data posted on Facebook.
- Access the HTML of the webpage and extract useful information/data from it. This technique is called web scraping or web harvesting or web data extraction.

This article discusses the steps involved in web scraping using the implementation of a Web Scraping framework of Python called BeautifulSoup. **Steps involved in web scraping:**

1. Send an HTTP request to the URL of the webpage you want to access. The server responds to the request by returning the HTML content of the webpage. For this task, we will use a third-party HTTP library for python-requests.
2. Once we have accessed the HTML content, we are left with the task of parsing the data. Since most of the HTML data is nested, we cannot extract data simply through string processing. One needs a parser which can create a nested/tree structure of the HTML data. There are many HTML parser libraries available but the most advanced one is html5lib.
3. Now, all we need to do is navigating and searching the parse tree that we created, i.e. tree traversal. For this task, we will be using another third-party python library, [Beautiful Soup](#). It is a Python library for pulling data out of HTML and XML files.

Step 1: Installing the required third-party libraries

- Easiest way to install external libraries in python is to use pip. [pip](#) is a package management system used to install and manage software packages written in Python. All you need to do is:

```
pip install requests  
pip install html5lib  
pip install bs4  
pip3 install requests beautifulsoup4
```

- Another way is to download them manually from these links:
- [requests](#)
- [html5lib](#)
- [beautifulsoup4](#)

Step 2: Accessing the HTML content from webpage

- Python

```
import requests

URL = "https://www.aptechmeerut.com/contact-us/"

r = requests.get(URL)

#print(r.content)
print(r.text)
```

Let us try to understand this piece of code.

- First of all import the requests library.
- Then, specify the URL of the webpage you want to scrape.
- Send a HTTP request to the specified URL and save the response from server in a response object called r.
- Now, as print r.content to get the **raw HTML content** of the webpage. It is of 'string' type.

Note: Sometimes you may get error “Not accepted” so try adding a browser user agent like below. Find your user agent based on device and browser from here <https://deviceatlas.com/blog/list-of-user-agent-strings>

- Python3

```
headers = {'User-Agent': 'Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/42.0.2311.135 Safari/537.36 Edge/12.246'}

# Here the user agent is for Edge browser on windows 10. You can find your browser user agent from the above given link.

r = requests.get(url=URL, headers=headers)

print(r.content)
```

```
import requests
from bs4 import BeautifulSoup
URL = "https://realpython.github.io/fake-jobs/"
page = requests.get(URL)
```

```
soup = BeautifulSoup(page.content, "html.parser")
print(soup.prettify())
```

The element you're looking for is a `<div>` with an `id` attribute that has the value "ResultsContainer". It has some other attributes as well, but below is the gist of what you're looking for:

```
<div id="ResultsContainer">
    <!-- all the job listings -->
</div>
```

Beautiful Soup allows you to find that specific HTML element by its ID:

```
results = soup.find(id="ResultsContainer")
```

For easier viewing, you can prettify any Beautiful Soup object when you print it out. If you call `.prettify()` on the `results` variable that you just assigned above, then you'll see all the HTML contained within the `<div>`:

```
print(results.prettify())
```

When you use the element's ID, you can pick out one element from among the rest of the HTML. Now you can work with only this specific part of the page's HTML. It looks like the soup just got a little thinner! However, it's still quite dense.

Find Elements by HTML Class Name

You've seen that every job posting is wrapped in a `<div>` element with the class `card-content`. Now you can work with your new object called `results` and select only the job postings in it. These are, after all, the parts of the HTML that you're interested in! You can do this in one line of code:

```
job_elements = results.find_all("div", class_="card-content")
```

Here, you call `.find_all()` on a Beautiful Soup object, which returns an [iterable](#) containing all the HTML for all the job listings displayed on that page.

Take a look at all of them:

```
for job_element in job_elements:
    print(job_element, end="\n"*2)
```

That's already pretty neat, but there's still a lot of HTML! You saw earlier that your page has descriptive class names on some elements. You can pick out those child elements from each job posting with `.find()`:

```
for job_element in job_elements:
    title_element = job_element.find("h2", class_="title")
```

```
company_element = job_element.find("h3", class_="company")
location_element = job_element.find("p", class_="location")
print(title_element)
print(company_element)
print(location_element)
print()
```

Each `job_element` is another `BeautifulSoup()` object. Therefore, you can use the same methods on it as you did on its parent element, `results`.

With this code snippet, you're getting closer and closer to the data that you're actually interested in. Still, there's a lot going on with all those HTML tags and attributes floating around:

```
<h2 class="title is-5">Senior Python Developer</h2>
<h3 class="subtitle is-6 company">Payne, Roberts and Davis</h3>
<p class="location">Stewartbury, AA</p>
```

Next, you'll learn how to narrow down this output to access only the text content you're interested in.

Extract Text From HTML Elements

You only want to see the title, company, and location of each job posting. And behold! Beautiful Soup has got you covered. You can add `.text` to a Beautiful Soup object to return only the **text content** of the HTML elements that the object contains:

```
for job_element in job_elements:
    title_element = job_element.find("h2", class_="title")
    company_element = job_element.find("h3", class_="company")
    location_element = job_element.find("p", class_="location")
    print(title_element.text)
    print(company_element.text)
    print(location_element.text)
    print()
```

Run the above code snippet, and you'll see the text of each element displayed. However, it's possible that you'll also get some extra **whitespace**. Since you're now working with [Python strings](#), you can `.strip()` the superfluous whitespace. You can also apply any other familiar Python string methods to further clean up your text:

```
for job_element in job_elements:
    title_element = job_element.find("h2", class_="title")
    company_element = job_element.find("h3", class_="company")
    location_element = job_element.find("p", class_="location")
    print(title_element.text.strip())
```

```
print(company_element.text.strip())
print(location_element.text.strip())
print()
```

The results finally look much better:

```
Senior Python Developer
Payne, Roberts and Davis
Stewartbury, AA
```

```
Energy engineer
Vasquez-Davidson
Christopherville, AA
```

```
Legal executive
Jackson, Chambers and Levy
Port Ericaburgh, AA
```

That's a readable list of jobs that also includes the company name and each job's location. However, you're looking for a position as a software developer, and these results contain job postings in many other fields as well.

Find Elements by Class Name and Text Content

Not all of the job listings are developer jobs. Instead of printing out *all* the jobs listed on the website, you'll first filter them using keywords.

You know that job titles in the page are kept within `<h2>` elements. To filter for only specific jobs, you can use the [string argument](#):

```
python_jobs = results.find_all("h2", string="Python")
```

This code finds all `<h2>` elements where the contained string matches "Python" exactly. Note that you're directly calling the method on your first `results` variable. If you go ahead and `print()` the output of the above code snippet to your console, then you might be disappointed because it'll be empty:

```
>>>
```

```
>>> print(python_jobs)
```

```
[]
```

There *was* a Python job in the search results, so why is it not showing up?

When you use `string=` as you did above, your program looks for that string *exactly*. Any differences in the spelling, capitalization, or whitespace will prevent the element from matching. In the next section, you'll find a way to make your search string more general.

Pass a Function to a BeautifulSoup Method

In addition to strings, you can sometimes pass functions as arguments to BeautifulSoup methods. You can change the previous line of code to use a function instead:

```
python_jobs = results.find_all(  
    "h2", string=lambda text: "python" in text.lower()  
)
```

Now you're passing an **anonymous function** to the `string=` argument. The [lambda function](#) looks at the text of each `<h2>` element, converts it to lowercase, and checks whether the substring "python" is found anywhere. You can check whether you managed to identify all the Python jobs with this approach:

```
>>>
```

```
>>> print(len(python_jobs))  
10
```

Your program has found 10 matching job posts that include the word "python" in their job title!

Finding elements depending on their text content is a powerful way to filter your HTML response for specific information. BeautifulSoup allows you to use either exact strings or functions as arguments for filtering text in BeautifulSoup objects.

However, when you try to run your scraper to print out the information of the filtered Python jobs, you'll run into an error:

```
AttributeError: 'NoneType' object has no attribute 'text'
```

This message is a [common error](#) that you'll run into a lot when you're scraping information from the Internet. Inspect the HTML of an element in your `python_jobs` list. What does it look like? Where do you think the error is coming from?

Identify Error Conditions

When you look at a single element in `python_jobs`, you'll see that it consists of only the `<h2>` element that contains the job title:

```
<h2 class="title is-5">Senior Python Developer</h2>
```

When you revisit the code you used to select the items, you'll see that that's what you targeted. You filtered for only the `<h2>` title elements of the job postings that contain the word "python". As you can see, these elements don't include the rest of the information about the job.

The error message you received earlier was related to this:

```
AttributeError: 'NoneType' object has no attribute 'text'
```

You tried to find the job title, the company name, and the job's location in each element in `python_jobs`, but each element contains only the job title text.

Your diligent parsing library still looks for the other ones, too, and returns `None` because it can't find them. Then, `print()` fails with the shown error message when you try to extract the `.text` attribute from one of these `None` objects.

The text you're looking for is nested in sibling elements of the `<h2>` elements your filter returned. Beautiful Soup can help you to select sibling, child, and parent elements of each Beautiful Soup object.

Access Parent Elements

One way to get access to all the information you need is to step up in the hierarchy of the DOM starting from the `<h2>` elements that you identified. Take another look at the HTML of a single job posting. Find the `<h2>` element that contains the job title as well as its closest parent element that contains all the information that you're interested in:

```
<div class="card">
    <div class="card-content">
        <div class="media">
            <div class="media-left">
                <figure class="image is-48x48">
                    
                </figure>
            </div>
            <div class="media-content">
                <h2 class="title is-5">Senior Python Developer</h2>
                <h3 class="subtitle is-6 company">Payne, Roberts and Davis</h3>
            </div>
        </div>

        <div class="content">
            <p class="location">Stewartbury, AA</p>
            <p class="is-small has-text-grey">
                <time datetime="2021-04-08">2021-04-08</time>
            </p>
        </div>
        <footer class="card-footer">
            <a
                href="https://www.realpython.com"
            >
```

```
    target="_blank"
    class="card-footer-item"
    >Learn</a>
  >
  <a href="https://realpython.github.io/fake-jobs/jobs/senior-python-developer-0.html"
    target="_blank"
    class="card-footer-item"
    >Apply</a>
  >
</footer>
</div>
</div>
```

The `<div>` element with the `card-content` class contains all the information you want. It's a third-level parent of the `<h2>` title element that you found using your filter.

With this information in mind, you can now use the elements in `python_jobs` and fetch their great-grandparent elements instead to get access to all the information you want:

```
python_jobs = results.find_all(
    "h2", string=lambda text: "python" in text.lower()
)

python_job_elements = [
    h2_element.parent.parent.parent for h2_element in python_jobs
]
```

You added a [list comprehension](#) that operates on each of the `<h2>` title elements in `python_jobs` that you got by filtering with the lambda expression. You're selecting the parent element of the parent element of the parent element of each `<h2>` title element. That's three generations up!

When you were looking at the HTML of a single job posting, you identified that this specific parent element with the class name `card-content` contains all the information you need.

Now you can adapt the code in your [for loop](#) to iterate over the parent elements instead:

```
for job_element in python_job_elements:
    # -- snip --
```

When you run your script another time, you'll see that your code once again has access to all the relevant information. That's because you're now looping over the `<div class="card-content">` elements instead of just the `<h2>` title elements.

Using the `.parent` attribute that each BeautifulSoup object comes with gives you an intuitive way of stepping through your DOM structure and addressing the elements you need. You can also access child elements and sibling elements in a similar manner. Read up on [navigating the tree](#) for more information.

Extract Attributes From HTML Elements

At this point, your Python script already scrapes the site and filters its HTML for relevant job postings. Well done! However, what's still missing is the link to apply for a job.

While you were inspecting the page, you found two links at the bottom of each card. If you handle the link elements in the same way as you handled the other elements, you won't get the URLs that you're interested in:

```
for job_element in python_job_elements:  
    # -- snip --  
    links = job_element.find_all("a")  
    for link in links:  
        print(link.text.strip())
```

If you run this code snippet, then you'll get the link texts Learn and Apply instead of the associated URLs.

That's because the `.text` attribute leaves only the visible content of an HTML element. It strips away all HTML tags, including the HTML attributes containing the URL, and leaves you with just the link text. To get the URL instead, you need to extract the value of one of the HTML attributes instead of discarding it.

The URL of a link element is associated with the `href` attribute. The specific URL that you're looking for is the value of the `href` attribute of the second `<a>` tag at the bottom the HTML of a single job posting:

```
<!-- snip -->  
<footer class="card-footer">  
    <a href="https://www.realpython.com" target="_blank"  
       class="card-footer-item">Learn</a>  
    <a href="https://realpython.github.io/fake-jobs/jobs/senior-python-  
developer-0.html"  
       target="_blank"  
       class="card-footer-item">Apply</a>  
  </footer>  
</div>  
</div>
```

Step 3: Parsing the HTML content

- Python

```
#This will not run on online IDE

import requests

from bs4 import BeautifulSoup

URL = "http://www.values.com/inspirational-quotes"

r = requests.get(URL)

soup = BeautifulSoup(r.content, 'html5lib') # If this line causes an error,
# run 'pip install html5lib' or install html5lib

print(soup.prettify())
```

A really nice thing about the BeautifulSoup library is that it is built on the top of the HTML parsing libraries like html5lib, lxml, html.parser, etc. So BeautifulSoup object and specify the parser library can be created at the same time. In the example above,

```
soup = BeautifulSoup(r.content, 'html5lib')
```

We create a BeautifulSoup object by passing two arguments:

Scraping Covid-19 statistics using BeautifulSoup

Coronavirus, one of the biggest pandemic has brought all the world to Danger. Along with this, it is one of the trending News, everyone has this day. In this article, we will be scraping data and printing Covid-19 statistics in human-readable form. The data will be scraped from [this website](#)

Prerequisites:

- The libraries ‘requests’, ‘bs4’, and ‘texttable’ have to be installed –

```
pip install bs4
pip install requests
pip install texttable
```

Project : Let’s head over to code, create a file called run.py.

- Python3

```
# importing modules

import requests

from bs4 import BeautifulSoup


# URL for scrapping data

url = 'https://www.worldometers.info/coronavirus/countries-where-coronavirus-has-spread/'


# get URL html

page = requests.get(url)

soup = BeautifulSoup(page.text, 'html.parser')


data = []


# soup.find_all('td') will scrape every

# element in the url's table

data_iterator = iter(soup.find_all('td'))


# data_iterator is the iterator of the table

# This loop will keep repeating till there is

# data available in the iterator

while True:

    try:

        country = next(data_iterator).text

        confirmed = next(data_iterator).text

        deaths = next(data_iterator).text

        continent = next(data_iterator).text
```

```

# For 'confirmed' and 'deaths',
# make sure to remove the commas
# and convert to int

data.append((
    country,
    int(confirmed.replace(',', ' ', '')),
    int(deaths.replace(',', ' ', '')),
    continent
))

# StopIteration error is raised when
# there are no more elements left to
# iterate through

except StopIteration:

    break

# Sort the data by the number of confirmed cases

data.sort(key = lambda row: row[1], reverse = True)

```

To print the data in human-readable format, we will use the library '*texttable*'

- Python3

```

# create texttable object

import texttable as tt

table = tt.Texttable()

# Add an empty row at the beginning for the headers

table.add_rows([(None, None, None, None)] + data)

```

```

# 'l' denotes left, 'c' denotes center,
# and 'r' denotes right

table.set_cols_align(['c', 'c', 'c', 'c'])

table.header(['Country', 'Number of cases', 'Deaths', 'Continent'])

print(table.draw())

```

Output:

| Continent | Country | Number of cases | Deaths | Continent |
|---------------|---------------|-----------------|--------|-----------|
| North America | United States | 644348 | 28554 | |
| Europe | Spain | 180659 | 18812 | |
| Europe | Italy | 165155 | 21645 | |

Machine Learning with SciKit Learn

Scikit Learn - Introduction

What is Sklearn in Python?

We will learn about the sklearn library and how to use it to implement machine learning algorithms. In the real world, we don't want to construct a challenging algorithm each time we need to utilise it. Although creating an algorithm from the beginning is a terrific approach to grasping the underlying concepts behind how it operates, we might not achieve the efficiency or dependability we require.

A Python module called Scikit-learn offers a variety of supervised and unsupervised learning techniques. It is based on several technologies you may already be acquainted with, including NumPy, pandas, and Matplotlib.

What is Sklearn?

French research scientist David Cournapeau's scikits.learn is a Google Summer of Code venture where the scikit-learn project first began. Its name refers to the idea that it's a modification to SciPy called "SciKit" (SciPy Toolkit), which was independently created and published. Later, other programmers rewrote the core codebase.

The French Institute for Research in Computer Science and Automation at Rocquencourt, France, led the work in 2010 under the direction of Alexandre Gramfort, Gael Varoquaux, Vincent Michel, and Fabian Pedregosa. On February 1st of that year, the institution issued the project's first official release. In November 2012, scikit-learn and scikit-image were cited as examples of scikits that were "well-maintained and popular". One of the most widely used machine learning packages on GitHub is Python's scikit-learn.

Implementation of Sklearn

Scikit-learn is mainly coded in Python and heavily utilizes the NumPy library for highly efficient array and linear algebra computations. Some fundamental algorithms are also built in Cython to enhance the efficiency of this library. Support vector machines, logistic regression, and linear SVMs are performed using wrappers coded in Cython for LIBSVM and LIBLINEAR, respectively. Expanding these routines with Python might not be viable in such circumstances.

Scikit-learn works nicely with numerous other Python packages, including SciPy, Pandas data frames, NumPy for array vectorization, Matplotlib, seaborn and plotly for plotting graphs, and many more.

Key concepts and features include:

- Algorithms for making decisions, such as:

Data are identified and categorised by classification as per the patterns.

Regression is the process of forecasting or predicting data values using the historical and anticipated data average.

Clustering is the automatic collection of datasets with related data.

- Predictive analysis is supported by various algorithms, including neural networks for pattern recognition and straightforward linear regression.
- Compatibility with the libraries of NumPy, pandas, and matplotlib

A predictive model can be built or trained on input data by computers using machine learning (ML), eliminating the need for explicit programming. A subset of AI is machine learning (AI).

Let's examine its revision history-

- May 2019: scikit-learn 0.21.0
- March 2019: scikit-learn 0.20.3
- December 2018: scikit-learn 0.20.2
- November 2018: scikit-learn 0.20.1
- September 2018: scikit-learn 0.20.0
- July 2018: scikit-learn 0.19.2
- July 2017: scikit-learn 0.19.0
- September 2016: scikit-learn 0.18.0
- November 2015: scikit-learn 0.17.0
- March 2015: scikit-learn 0.16.0
- July 2014: scikit-learn 0.15.0
- August 2013: scikit-learn 0.14

The extensive community of open-source programs is one of the key justifications for using them, and Sklearn is comparable in this regard. There have been roughly 35 contributors to Python's scikit-learn library, with Andreas Mueller being the most noteworthy.

On the scikit learn the main page, many Organizations, including Evernote, Inria, and AWeber, are listed as customers. But the actual utilization is much higher than that.

Along with these groups, there are communities all around the world.

Scikit-learn's salient characteristics are:

- The package provides the functions for data mining and machine learning algorithms for data analysis that are easy to use and effective. Support vector machines, gradient boosting, random forests, k-means, and other regression, classification, and clustering algorithms are included.
- The package is open source, accessible to everyone and reusable in several contexts.
- It is built on top of SciPy, Matplotlib, and NumPy.
- The package has a commercially usable - BSD license.

Benefits of Using Scikit-Learn for Implementing Machine Learning Algorithms

You will discover that scikit-learn is well-documented and straightforward to understand, regardless of if you are seeking an overview of ML, wish to get up to speed quickly or seek the most recent ML learning tool. With the help of this high-level toolkit, you can quickly construct a predictive data analysis model and use it to fit the collected data. It is adaptable and works well alongside other Python libraries.

Installation of Sklearn on your System

Requirements to install Sklearn:

- Python (>=3.5)
- NumPy (>= 1.11.0)
- Scipy (>= 0.17.0)
- Joblib (>= 0.11)
- Matplotlib (>= 1.5.1) is required for Sklearn plotting capabilities.
- Pandas (>= 0.18.0) is required for some of the scikit-learn examples using data structure and analysis.

Make sure NumPy and SciPy libraries are installed in the system before installing the scikit-learn library. The simplest method for installing scikit-learn once NumPy and SciPy have been successfully installed is by using pip:

```
pip install -U scikit-learn
```

Essential Elements of Machine Learning

Let us first go through the basic terminology used in ML projects to use scikit-learn.

- **Accuracy Score-** The accuracy score tells the ratio of the predictions correctly predicted over the total sample size.
- In a classification problem involving multiple classes, the accuracy score is defined as follows:
Accuracy Score = Correctly Predicted Classes / Total Number of Samples Given for Prediction
- In a classification problem involving only two classes, the accuracy score is defined as follows:
Accuracy Score = (True Positive Samples + True Negative Samples) / Total Number of Samples Given for Prediction
- **Example Data-** These are the specific examples (features) of the data. Two types of data examples are available:
 - **Labelled Data-** This type of data includes the labels or the target values for the samples of the independent features. This is defined as:
{independent features, label}: (X, Y)
 - **Unlabelled Data-** This type of data only contains independent features and not labels or target values. This is defined as:
{independent features, Null}: (x, Null)
- **Feature-** These serve as input parameters, also called independent features. A feature is a quantifiable quality or attribute of the object under observation. Each ML project has a minimum of one feature.
- **Clustering-** Data points are grouped using a technique called clustering based on various metrics measuring similarity in samples. Each group is referred to as a Cluster.
- **K-Means Clustering-** It is an unsupervised machine learning strategy that locates the means (centroids) of a given number (k) clusters made out of the provided data points by placing them in the closest cluster.
- **Model-** A model defines the association between the independent features and the target label. For instance, a model for detecting rumours links specific characteristics to rumours.

- **Regression vs Classification-** Both regression and classification models let you construct forecasts that provide answers to questions like which party will dominate a particular election.
- The output of regression models is a number or continuum value.
- A discrete or categorical value as a prediction is provided by classification models.
- **Supervised Learning-** The system "learns" how to identify correct responses using a labelled dataset, which it may then deploy to the training dataset. The accuracy of the algorithm can then be assessed and improved. Supervised learning is used in the majority of machine learning projects.
- **Unsupervised Learning-** By "learning" traits and patterns entirely on its own, the algorithm seeks to interpret unlabelled data.
 - On the other hand, if NumPy and Scipy is not yet installed on your Python workstation then, you can install them by using either **pip** or **conda**.
 - Another option to use scikit-learn is to use Python distributions like **Canopy** and **Anaconda** because they both ship the latest version of scikit-learn.

Community & contributors

Scikit-learn is a community effort and anyone can contribute to it. This project is hosted on <https://github.com/scikit-learn/scikit-learn>. Following people are currently the core contributors to Sklearn's development and maintenance –

- Joris Van den Bossche (Data Scientist)
- Thomas J Fan (Software Developer)
- Alexandre Gramfort (Machine Learning Researcher)
- Olivier Grisel (Machine Learning Expert)
- Nicolas Hug (Associate Research Scientist)
- Andreas Mueller (Machine Learning Scientist)
- Hanmin Qin (Software Engineer)
- Adrin Jalali (Open Source Developer)
- Nelle Varoquaux (Data Science Researcher)
- Roman Yurchak (Data Scientist)

Various organisations like Booking.com, JP Morgan, Evernote, Inria, AWeber, Spotify and many more are using Sklearn.

Steps to Build a Model in Sklearn

Let us now learn the modelling process.

Step 1: Loading a Dataset

Simply put, a dataset is a collection of sample data points. A dataset typically consists of two primary parts:

Features: Features are essentially the variables in our dataset, often called predictors, data inputs, or attributes. A feature matrix, which is frequently symbolised by the letter "X," can be used to represent them since many of them may exist. The term "feature names" refers to a list of names of all the features.

Response: (sometimes referred to as the target feature, label, or output) Based on the variables feature, this variable is the output. In most cases, we only have one response column, which is depicted by a response column or vector (the letter 'y' is frequently used to denote a response vector). Target names refer to all the various values a response vector could take.

Step 2: Splitting the Dataset

The correctness of each machine learning model is a crucial consideration. Now, one may train a model with the provided dataset and then use that model to predict the target values for another set of the dataset to ascertain the correctness of the model.

To sum it up:

- Make a training dataset and a testing dataset out of the given dataset.
- On the practise set, train the model.
- Test the model using the testing dataset and assess its performance.

Step 3: Training the Model

It's time to use the training dataset to train the model, which will make predictions. A variety of machine learning techniques with an easy-to-use interface for fitting, prediction accuracy, etc., are offered by Scikit-learn.

Our classifier must now be tested using the testing dataset. For this, we can use the .predict() model class method, giving back the predicted values.

By comparing the actual values of the testing dataset and the predicted values, we can assess the model's performance with the help of sklearn methods. The accuracy_score function of the metrics package is used for this.

ML Algorithms

Algorithms are necessary for machines to learn without specific programming. Simply put, algorithms are just rules used in the calculation.

ML algorithm Fundamental Ideas

Representation - Data can be set up in a form to allow it to be analyzed. Examples include rules, model ensembles, decision trees, neural networks, SVM, graphical models, and more.

Evaluation - Evaluation is a method of determining the legitimacy of a hypothesis. Examples include accuracy score, squared error, prediction and recall, probability, cost, margin, and likelihood.

Optimization - By applying methods like combinatorial optimization, grid search, constrained optimization, etc., optimization is tuning an estimator's hyperparameters to reduce model errors.

Scikit-Learn ML Algorithms

Here is a list of several typical Scikit-learn algorithms and techniques, given in decreasing order of complexity:

Linear Regression Algorithm Example

The slope of a straight line is the projected output of the supervised machine learning process known as linear regression. It is only used to forecast values within a specific data point range.

Code

```
# Python program to show how to use sklearn to perform linear regression

# Importing the required modules and classes
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.linear_model import LinearRegression
from sklearn.datasets import load_iris

# Loading our load_iris dataset
X, Y = load_iris( return_X_y=True )
```

```

# Printing the shape of the complete dataset
print(X.shape)

# Splitting the dataset into the training and validating datasets
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.4, random_state = 10)

# Printing the shape of training and validation data
print(X_train.shape, Y_train.shape)
print(X_test.shape, Y_test.shape)

# Training the model using the training dataset
lreg = LinearRegression()
lreg.fit(X_train, Y_train)

# Printing the Coefficients of the linear Regression model
print("Coefficients of each feature: ", lreg.coef_)

# Printing the accuracy score of the trained model
score = lreg.score(X_test, Y_test)
print("Accuracy Score: ", score)

```

Output

```
(150, 4)
(90, 4) (90,)
(60, 4) (60,)
Coefficients of each feature: [-0.12949807  0.03421679  0.23781661
0.60472254]
Accuracy Score: 0.8885645804630061
```

Logistic Regression Algorithm Example

Logistic regression is the preferred approach for binary classification questions (such as the target values are 0 or 1). The results can then be evaluated using an equation resembling linear regression (e.g., how probable is it that a particular target value is 0 or 1?).

Code

```
# Python code to see how to perform Logistic Regression using sklearn.linear_model
```

```

# Importing the required modules and classes
from sklearn.datasets import load_iris
from sklearn.metrics import accuracy_score
from sklearn.linear_model import LogisticRegression
from sklearn.model_selection import train_test_split

# Loading our dataset
data = load_iris()

# Splitting the independent and dependent variables
X = data.data
Y = data.target
print("The size of the complete dataset is: ", len(X))

# Creating an instance of the LogisticRegression class for implementing logistic regression
log_reg = LogisticRegression()

# Segregating the training and testing dataset
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.3, random_state = 10)

# Performing the logistic regression on train dataset
log_reg.fit(X_train, Y_train)

# Printing the accuracy score
print("Accuracy score of the predictions made by the model: ", accuracy_score(log_reg.predict(X_test), Y_test))

```

Output

```
The size of the complete dataset is: 150
Accuracy score of the predictions made by the model: 1.0
```

Advanced Machine Learning Algorithms

Random Forest

A Random Forest algorithm is used in machine learning to perform ensemble learning. The ensemble learning system uses several Decision Trees and other machine learning algorithms to produce more outstanding predictive analyses than any one learning algorithm.

Code

```
# Python program to show how to Random Forest Algorithm

# importing the required libraries
from sklearn import datasets
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.ensemble import RandomForestClassifier

# Loading the dataset
X, Y = load_iris(return_X_y = True)

# Segregating the dataset into training and testing dataset
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size = 0.4)

# creating an object of the RF classifier class
rf = RandomForestClassifier(n_estimators = 100)

# Training the classifier model on the training dataset
rf.fit(X_train, Y_train)

# Predicting the values for the test dataset
Y_pred = rf.predict(X_test)

# using metrics module to calculate accuracy score
print("Accuracy score for the model is: ", accuracy_score(Y_test, Y_pred))

# predicting the type of flower
rf.predict([[5, 7, 3, 4]])
```

Output

```
Accuracy score for the model is:  0.95
```

```
array([1])
```

Decision Tree Algorithm

A node represents a feature (or property), a branch indicates a decision function, and every leaf node indicates the conclusion in a decision tree, which resembles a flowchart. The root node in a decision tree is the first node from the top. It gains the ability to divide data according to attribute values. Recursive partitioning is the process of repeatedly dividing a tree. This framework, which resembles a flowchart, aids in decision-making. It is a flowchart-like representation that perfectly replicates how people think. Decision trees are simple to grasp and interpret because of this.

Code

```
# Python program to perform classification using Decision Trees

# Importing the required libraries
import numpy as np
from sklearn.datasets import load_iris
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import cross_val_score, train_test_split

# Loading the dataset
X, Y = load_iris( return_X_y = True )

# Splitting the dataset in training and test data
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4, random_state=0)

# Creating an instance of the Decision Tree Classifier class
dtc = DecisionTreeClassifier(random_state = 0)
dtc.fit(X_train, Y_train)

# Calculating the accuracy score of the model using cross_val_score
score = cross_val_score(dtc, X, Y, cv = 10)

# Printing the scores
print("Accuracy scores: ", score)
print("Mean accuracy score: ", np.mean(score))
```

Output

```
Accuracy scores: [1.          0.93333333 1.          0.93333333 0.93333333  
0.86666667  
0.93333333 1.          1.          1.          ]  
Mean accuracy score: 0.96
```

Gradient Boosting

We might use a gradient boosting method when there are problems with regression and classification. It creates a predictive model based on many lesser prediction models, typically decision trees.

To work, the Gradient Boosting Classifier needs a loss function. In addition to handling custom loss functions, gradient boosting classifiers may take many standardised loss functions, and the loss function must, however, be differentiable.

Squared errors may be used in regression techniques, although logarithmic loss is typically used in classification algorithms. In gradient boosting systems, we don't need to explicitly derive a loss function for each incremental boosting step; instead, we can use any differentiable loss function.

Code

```
# Python program to perform classification using Gradient Boosting  
  
# Importing the required libraries  
from sklearn.datasets import make_hastie_10_2  
from sklearn.ensemble import GradientBoostingClassifier  
  
# Loading the dataset  
X, Y = make_hastie_10_2(random_state = 10)  
  
# Splitting the dataset in training and test data  
X_train, X_test, Y_train, Y_test = train_test_split(X, Y, test_size=0.4, random_state=0)  
  
# Creating an instance of the Gradient Boosting Classifier class  
gbc = GradientBoostingClassifier(n_estimators = 100, learning_rate = 1.0, max_depth = 1, ra  
ndom_state = 0)  
gbc.fit(X_train, Y_train)  
  
# Calculating the accuracy score of the model using cross_val_score  
score = gbc.score(X_test, Y_test)
```

```
# Printing the scores
print("Accuracy scores: ", score)
```

Output

```
Accuracy scores: 0.9185416666666667
```

Scikit Learn - Modelling Process

This chapter deals with the modelling process involved in Sklearn. Let us understand about the same in detail and begin with dataset loading.

Dataset Loading

A collection of data is called dataset. It is having the following two components –

Features – The variables of data are called its features. They are also known as predictors, inputs or attributes.

- **Feature matrix** – It is the collection of features, in case there are more than one.
- **Feature Names** – It is the list of all the names of the features.

Response – It is the output variable that basically depends upon the feature variables. They are also known as target, label or output.

- **Response Vector** – It is used to represent response column. Generally, we have just one response column.
- **Target Names** – It represent the possible values taken by a response vector.

Scikit-learn have few example datasets like **iris** and **digits** for classification and the **Boston house prices** for regression.

Example

Following is an example to load **iris** dataset –

```
from sklearn.datasets import load_iris
iris = load_iris()
X = iris.data
y = iris.target
feature_names = iris.feature_names
target_names = iris.target_names
```

```
print("Feature names:", feature_names)
print("Target names:", target_names)
print("\nFirst 10 rows of X:\n", X[:10])
```

Output

```
Feature names: ['sepal length (cm)', 'sepal width (cm)', 'petal length (cm)', 'petal width (cm)']
Target names: ['setosa' 'versicolor' 'virginica']
First 10 rows of X:
[
[5.1 3.5 1.4 0.2]
[4.9 3. 1.4 0.2]
[4.7 3.2 1.3 0.2]
[4.6 3.1 1.5 0.2]
[5. 3.6 1.4 0.2]
[5.4 3.9 1.7 0.4]
[4.6 3.4 1.4 0.3]
[5. 3.4 1.5 0.2]
[4.4 2.9 1.4 0.2]
[4.9 3.1 1.5 0.1]
]
```

Splitting the dataset

To check the accuracy of our model, we can split the dataset into two pieces-a **training set** and a **testing set**. Use the training set to train the model and testing set to test the model. After that, we can evaluate how well our model did.

Example

The following example will split the data into 70:30 ratio, i.e. 70% data will be used as training data and 30% will be used as testing data. The dataset is iris dataset as in above example.

```
from sklearn.datasets import load_iris
iris = load_iris()

X = iris.data
y = iris.target

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size = 0.3, random_state = 1
)

print(X_train.shape)
print(X_test.shape)

print(y_train.shape)
print(y_test.shape)
```

Output

```
(105, 4)
(45, 4)
(105,)
(45,)
```

As seen in the example above, it uses `train_test_split()` function of scikit-learn to split the dataset. This function has the following arguments –

- **X, y** – Here, **X** is the **feature matrix** and **y** is the **response vector**, which need to be split.
- **test_size** – This represents the ratio of test data to the total given data. As in the above example, we are setting **test_data = 0.3** for 150 rows of **X**. It will produce test data of $150 \times 0.3 = 45$ rows.
- **random_size** – It is used to guarantee that the split will always be the same. This is useful in the situations where you want reproducible results.

Binarisation

This preprocessing technique is used when we need to convert our numerical values into Boolean values.

Example

```
import numpy as np
from sklearn import preprocessing
Input_data = np.array([
    [2.1, -1.9, 5.5],
    [-1.5, 2.4, 3.5],
    [0.5, -7.9, 5.6],
    [5.9, 2.3, -5.8]])
data_binarized = preprocessing.Binarizer(threshold=0.5).transform(input_data)
print("\nBinarized data:\n", data_binarized)
```

In the above example, we used **threshold value = 0.5** and that is why, all the values above 0.5 would be converted to 1, and all the values below 0.5 would be converted to 0.

Output

Binarized data:

```
[
    [ 1. 0. 1.]
    [ 0. 1. 1.]
    [ 0. 0. 1.]
    [ 1. 1. 0.]
]
```

Mean Removal

This technique is used to eliminate the mean from feature vector so that every feature centered on zero.

Example

```
import numpy as np
from sklearn import preprocessing
Input_data = np.array([
    [2.1, -1.9, 5.5],
    [-1.5, 2.4, 3.5],
    [0.5, -7.9, 5.6],
    [5.9, 2.3, -5.8]])
#displaying the mean and the standard deviation of the input data
print("Mean =", input_data.mean(axis=0))
print("Stddeviation =", input_data.std(axis=0))
#Removing the mean and the standard deviation of the input data

data_scaled = preprocessing.scale(input_data)
print("Mean_removed =", data_scaled.mean(axis=0))
print("Stddeviation_removed =", data_scaled.std(axis=0))
```

Output

```
Mean = [ 1.75 -1.275 2.2 ]
Stddeviation = [ 2.71431391 4.20022321 4.69414529]
Mean_removed = [ 1.11022302e-16 0.00000000e+00 0.00000000e+00]
Stddeviation_removed = [ 1. 1. 1.]
```

Normalisation

We use this preprocessing technique for modifying the feature vectors. Normalisation of feature vectors is necessary so that the feature vectors can be measured at common scale. There are two types of normalisation as follows –

L1 Normalisation

It is also called Least Absolute Deviations. It modifies the value in such a manner that the sum of the absolute values remains always up to 1 in each row. Following example shows the implementation of L1 normalisation on input data.

Example

```
import numpy as np
from sklearn import preprocessing
Input_data = np.array([
    [
        [2.1, -1.9, 5.5],
        [-1.5, 2.4, 3.5],
```

```

        [0.5, -7.9, 5.6],
        [5.9, 2.3, -5.8]
    ]
)
data_normalized_l1 = preprocessing.normalize(input_data, norm='l1')
print("\nL1 normalized data:\n", data_normalized_l1)

```

Output

L1 normalized data:

```

[
[ 0.22105263 -0.2 0.57894737]
[-0.2027027 0.32432432 0.47297297]
[ 0.03571429 -0.56428571 0.4 ]
[ 0.42142857 0.16428571 -0.41428571]
]
```

L2 Normalisation

Also called Least Squares. It modifies the value in such a manner that the sum of the squares remains always up to 1 in each row. Following example shows the implementation of L2 normalisation on input data.

Example

```

import numpy as np
from sklearn import preprocessing
Input_data = np.array(
[
    [2.1, -1.9, 5.5],
    [-1.5, 2.4, 3.5],
    [0.5, -7.9, 5.6],
    [5.9, 2.3, -5.8]
])
data_normalized_l2 = preprocessing.normalize(input_data, norm='l2')
print("\nL2 normalized data:\n", data_normalized_l2)

```

Output

L2 normalized data:

```

[
[ 0.33946114 -0.30713151 0.88906489]
[-0.33325106 0.53320169 0.7775858 ]
[ 0.05156558 -0.81473612 0.57753446]
[ 0.68706914 0.26784051 -0.6754239 ]
]
```

Scikit Learn - Data Representation

As we know that machine learning is about to create model from data. For this purpose, computer must understand the data first. Next, we are going to discuss various ways to represent the data in order to be understood by computer –

Data as table

The best way to represent data in Scikit-learn is in the form of tables. A table represents a 2-D grid of data where rows represent the individual elements of the dataset and the columns represents the quantities related to those individual elements.

Example

With the example given below, we can download *iris dataset* in the form of a Pandas DataFrame with the help of python **seaborn** library.

```
import seaborn as sns
iris = sns.load_dataset('iris')
iris.head()
```

Output

```
sepal_length sepal_width petal_length petal_width species
0      5.1      3.5       1.4      0.2  setosa
1      4.9      3.0       1.4      0.2  setosa
2      4.7      3.2       1.3      0.2  setosa
3      4.6      3.1       1.5      0.2  setosa
4      5.0      3.6       1.4      0.2  setosa
```

From above output, we can see that each row of the data represents a single observed flower and the number of rows represents the total number of flowers in the dataset. Generally, we refer the rows of the matrix as samples.

On the other hand, each column of the data represents a quantitative information describing each sample. Generally, we refer the columns of the matrix as features.

Data as Feature Matrix

Features matrix may be defined as the table layout where information can be thought of as a 2-D matrix. It is stored in a variable named **X** and assumed to be two dimensional with shape [n_samples, n_features]. Mostly, it is contained in a NumPy array or a Pandas DataFrame. As told earlier, the samples always represent the individual objects described by the dataset and the features represents the distinct observations that describe each sample in a quantitative manner.

Data as Target array

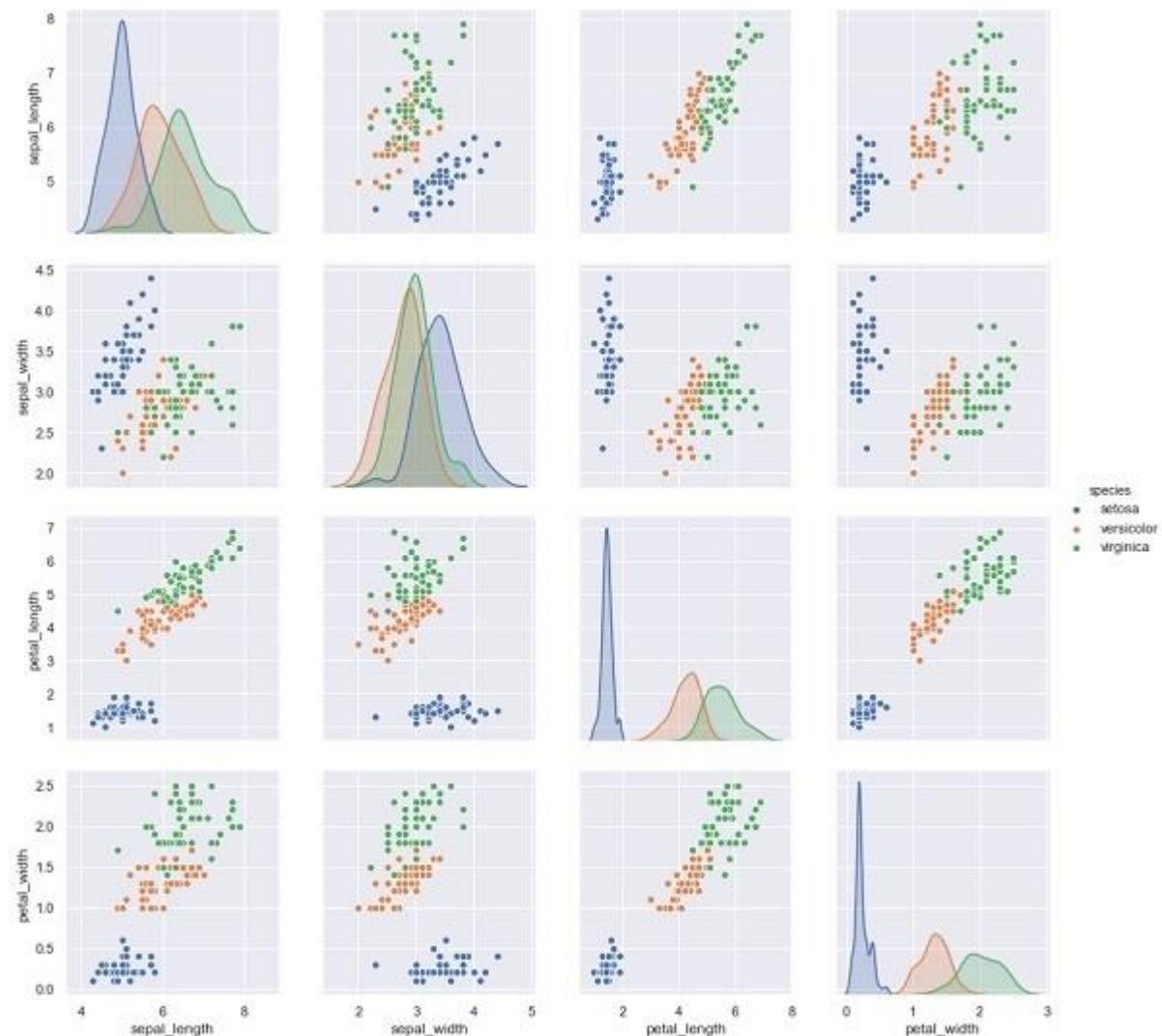
Along with Features matrix, denoted by X, we also have target array. It is also called label. It is denoted by y. The label or target array is usually one-dimensional having length n_samples. It is generally contained in NumPy **array** or Pandas **Series**. Target array may have both the values, continuous numerical values and discrete values.

Example

In the example below, from iris dataset we predict the species of flower based on the other measurements. In this case, the Species column would be considered as the feature.

```
import seaborn as sns
iris = sns.load_dataset('iris')
%matplotlib inline
import seaborn as sns; sns.set()
sns.pairplot(iris, hue='species', height=3);
```

Output



Hadoop with Python

Hadoop Distributed File System (HDFS) The Hadoop Distributed File System (HDFS) is a Java-based distributed, scalable, and portable filesystem designed to span large clusters of commodity servers. The design of HDFS is based on GFS, the Google File System, which is described in a paper published by Google. Like many other distributed filesystems, HDFS holds a large amount of data and provides transparent access to many clients distributed across a network. Where HDFS excels is in its ability to store very large files in a reliable and scalable manner. HDFS is designed to store a lot of information, typically petabytes (for very large files), gigabytes, and terabytes. This is accomplished by using a block-structured filesystem. Individual files are split into fixed-size blocks that are stored on machines across the cluster. Files made of several blocks generally do not have all of their blocks stored on a single machine. HDFS ensures reliability by replicating blocks and distributing the replicas across the cluster. The default replication factor is three, meaning that each block exists three times on the cluster. Block-level replication enables data availability even when machines fail. This chapter begins by introducing the core concepts of HDFS and explains how to interact with the filesystem using the native built-in commands. After a few examples, a Python client library is introduced that enables HDFS to be accessed programmatically from within Python applications.

Hadoop Distributed File System (HDFS) The Hadoop Distributed File System (HDFS) is a Java-based distributed, scalable, and portable filesystem designed to span large clusters of commodity servers. The design of HDFS is based on GFS, the Google File System, which is described in a paper published by Google. Like many other distributed filesystems, HDFS holds a large amount of data and provides transparent access to many clients distributed across a network. Where HDFS excels is in its ability to store very large files in a reliable and scalable manner. HDFS is designed to store a lot of information, typically petabytes (for very large files), gigabytes, and terabytes. This is accomplished by using a block-structured filesystem. Individual files are split into fixed-size blocks that are stored on machines across the cluster. Files made of several blocks generally do not have all of their blocks stored on a single machine. HDFS ensures reliability by replicating blocks and distributing the replicas across the cluster. The default replication factor is three, meaning that each block exists three times on the cluster. Block-level replication enables data availability even when machines fail. This chapter begins by introducing the core concepts of HDFS and explains how to interact with the filesystem using the native built-in commands. After a few examples, a Python client library is introduced that enables HDFS to be accessed programmatically from within Python applications.

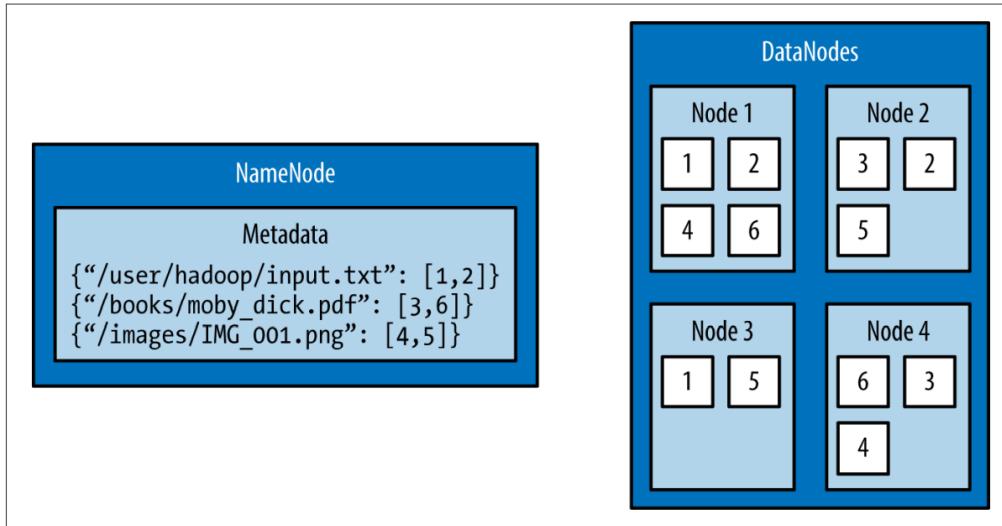


Figure 1-1. An HDFS cluster with a replication factor of two; the NameNode contains the mapping of files to blocks, and the DataNodes store the blocks and their replicas

Snakebite Snakebite is a Python package, created by Spotify, that provides a Python client library, allowing HDFS to be accessed programmatically from Python applications. The client library uses protobuf messages to communicate directly with the NameNode. The Snakebite package also includes a command-line interface for HDFS that is based on the client library. This section describes how to install and configure the Snakebite package. Snakebite's client library is explained in detail with multiple examples, and Snakebite's built-in CLI is introduced as a Python alternative to the hdfs dfs command. Installation Snakebite requires Python 2 and python-protobuf 2.4.1 or higher. Python 3 is currently not supported. Snakebite is distributed through PyPI and can be installed using pip: \$ pip install snakebite.

MapReduce with Python

MapReduce is a programming model that enables large volumes of data to be processed and generated by dividing work into independent tasks and executing the tasks in parallel across a cluster of machines. The MapReduce programming style was inspired by the functional programming constructs map and reduce, which are commonly used to process lists of data. At a high level, every MapReduce program transforms a list of input data elements into a list of output data elements twice, once in the map phase and once in the reduce phase. This chapter begins by introducing the MapReduce programming model and describing how data flows through the different phases of the model. Examples then show how MapReduce jobs can be written in Python.

Data Flow

The MapReduce framework is composed of three major phases: map, shuffle and sort, and reduce. This section describes each phase in detail.

Map

The first phase of a MapReduce application is the map phase. Within the map phase, a function (called the mapper) processes a series of key-value pairs. The mapper sequentially processes each 15 key-value pair individually, producing zero or more output keyvalue pairs

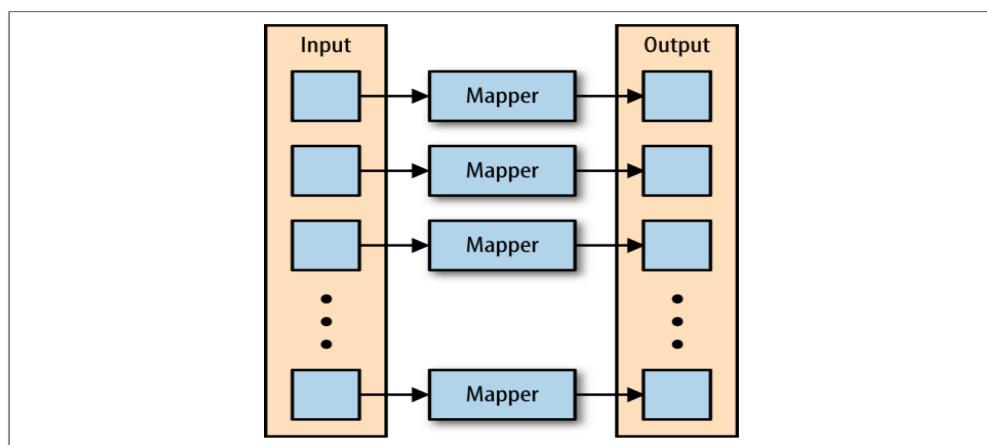


Figure 2-1. The mapper is applied to each input key-value pair, producing an output key-value pair

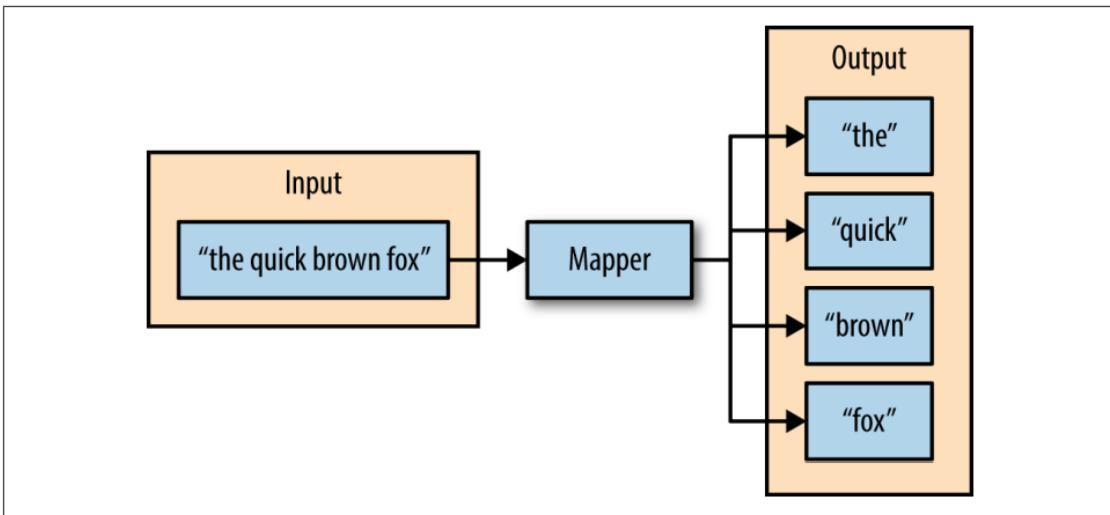


Figure 2-2. The input of the mapper is a string, and the function of the mapper is to split the input on spaces; the resulting output is the individual words from the mapper's input

Shuffle and Sort

The second phase of MapReduce is the shuffle and sort. As the mappers begin completing, the intermediate outputs from the map phase are moved to the reducers. This process of moving output from the mappers to the reducers is known as shuffling. Shuffling is handled by a partition function, known as the partitioner. The partitioner is used to control the flow of key-value pairs from mappers to reducers. The partitioner is given the mapper's output key and the number of reducers, and returns the index of the intended reducer. The partitioner ensures that all of the values for the same key are sent to the same reducer. The default partitioner is hash-based. It computes a hash value of the mapper's output key and assigns a partition based on this result. The final stage before the reducers start processing data is the sorting process. The intermediate keys and values for each partition are sorted by the Hadoop framework before being presented to the reducer.

Reduce

The third phase of MapReduce is the reduce phase. Within the reducer phase, an iterator of values is provided to a function known as the reducer. The iterator of values is a nonunique set of values for each unique key from the output of the map phase. The reducer aggregates the values for each unique key and produces zero or more output key-value pairs

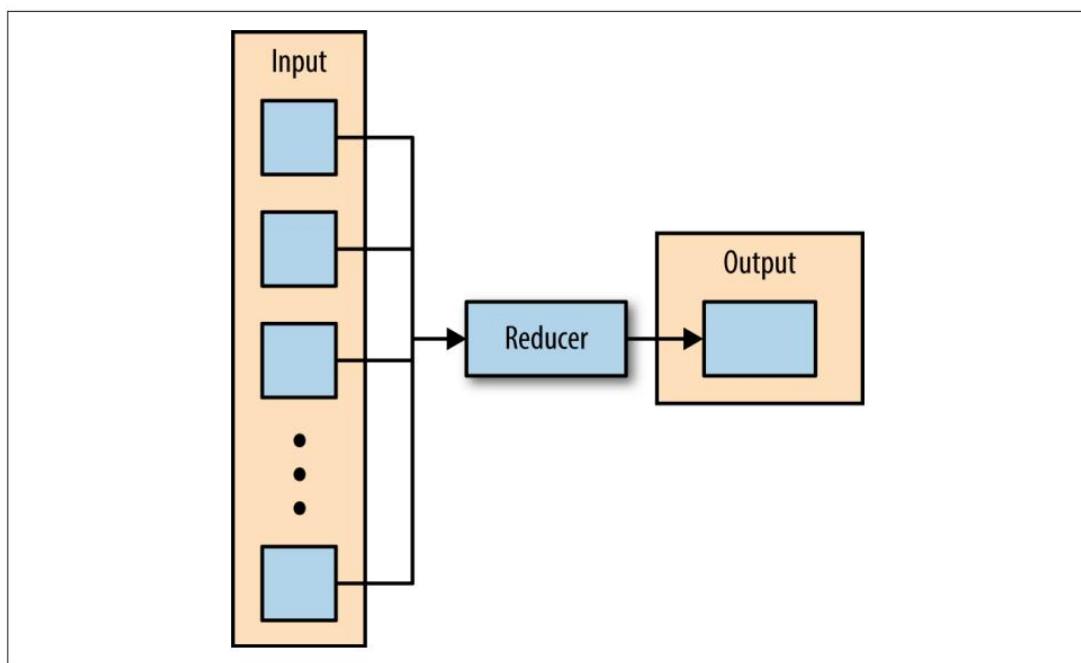


Figure 2-3. The reducer iterates over the input values, producing an output key-value pair

As an example, consider a reducer whose purpose is to sum all of the values for a key. The input to this reducer is an iterator of all of the values for a key, and the reducer sums all of the values. The reducer then outputs a key-value pair that contains the input key and the sum of the input key values (Figure 2-4).

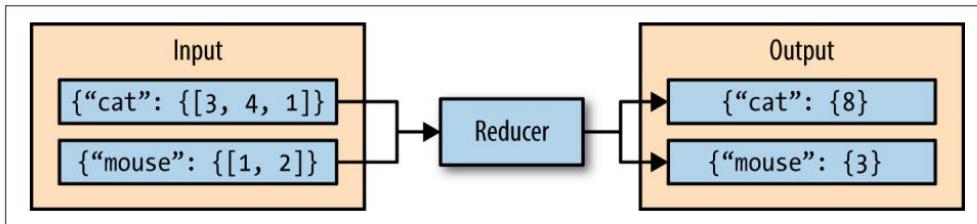


Figure 2-4. This reducer sums the values for the keys “cat” and “mouse”

The next section describes a simple MapReduce application and its implementation in Python.

How It Works The mapper and reducer are both executables that read input, line by line, from the standard input (stdin), and write output to the standard output (stdout). The Hadoop streaming utility creates a MapReduce job, submits the job to the cluster, and monitors its progress until it is complete. When the mapper is initialized, each map task launches the specified executable as a separate process. The mapper reads the input file and presents each line to the executable via stdin. After the executable processes each line of input, the mapper collects the output from stdout and converts each line to a key-value pair. The key consists of the part of the line before the first tab character, and the value consists of the part of the line after the first tab character. If a line contains no tab character, the entire line is considered the key and the value is null. When the reducer is initialized, each reduce task launches the specified executable as a separate process. The reducer converts the input key-value pair to lines that are presented to the executable via stdin. The reducer collects the executable's result from stdout and converts each line to a key-value pair. Similar to the mapper, the executable specifies key-value pairs by separating the key and value by a tab character.

Pig and Python

Pig is composed of two major parts: a high-level data flow language called Pig Latin, and an engine that parses, optimizes, and executes the Pig Latin scripts as a series of MapReduce jobs that are run on a Hadoop cluster. Compared to Java MapReduce, Pig is easier to write, understand, and maintain because it is a data transformation language that allows the processing of data to be described as a sequence of transformations. Pig is also highly extensible through the use of the User Defined Functions (UDFs) which allow custom processing to be written in many languages, such as Python. An example of a Pig application is the Extract, Transform, Load (ETL) process that describes how an application extracts data from a data source, transforms the data for querying and analysis purposes, and loads the result onto a target data store. Once Pig loads the data, it can perform projections, iterations, and other transformations. UDFs enable more complex algorithms to be applied during the transformation phase. After the data is done being processed by Pig, it can be stored back in HDFS. This chapter begins with an example Pig script. Pig and Pig Latin are then introduced and described in detail with examples. The chapter concludes with an explanation of how Pig's core features can be extended through the use of Python.

Spark with Python

Spark is a cluster computing framework that uses in-memory primitives to enable programs to run up to a hundred times faster than Hadoop MapReduce applications. Spark applications consist of a driver program that controls the execution of parallel operations across a cluster. The main programming abstraction provided by Spark is known as Resilient Distributed Datasets (RDDs). RDDs are collections of elements partitioned across the nodes of the cluster that can be operated on in parallel. Spark was created to run on many platforms and be developed in many languages. Currently, Spark can run on Hadoop 1.0, Hadoop 2.0, Apache Mesos, or a standalone Spark cluster. Spark also natively supports Scala, Java, Python, and R. In addition to these features, Spark can be used interactively from a command-line shell. This chapter begins with an example Spark script. PySpark is then introduced, and RDDs are described in detail with examples. The chapter concludes with example Spark programs written in Python

Workflow Management with Python

The most popular workflow scheduler to manage Hadoop jobs is arguably Apache Oozie. Like many other Hadoop products, Oozie is written in Java, and is a server-based web application that runs workflow jobs that execute Hadoop MapReduce and Pig jobs. An Oozie workflow is a collection of actions arranged in a control dependency directed acyclic graph (DAG) specified in an XML document. While Oozie has a lot of support in the Hadoop community, configuring workflows and jobs through XML attributes has a steep learning curve. Luigi is a Python alternative, created by Spotify, that enables complex pipelines of batch jobs to be built and configured. It handles dependency resolution, workflow management, visualization, and much more. It also has a large community and supports many Hadoop technologies. This chapter begins with the installation of Luigi and a detailed description of a workflow. Multiple examples then show how Luigi can be used to control MapReduce and Pig jobs. Installation Luigi is distributed through PyPI and can be installed using pip: \$ pip install luigi 53 Or it can be installed from source: \$ git clone <https://github.com/spotify/luigi> \$ python setup.py install Workflows Within Luigi, a workflow consists of a pipeline of actions, called tasks. Luigi tasks are nonspecific, that is, they can be anything that can be written in Python. The locations of input and output data for a task are known as targets. Targets typically correspond to locations of files on disk, on HDFS, or in a database. In addition to tasks and targets, Luigi utilizes parameters to customize how tasks are executed. Tasks are the sequences of actions that comprise a Luigi workflow. Each task declares its dependencies on targets created by other tasks. This enables Luigi to create dependency chains that ensure a task will

not be executed until all of the dependent tasks and all of the dependencies for those tasks are satisfied.

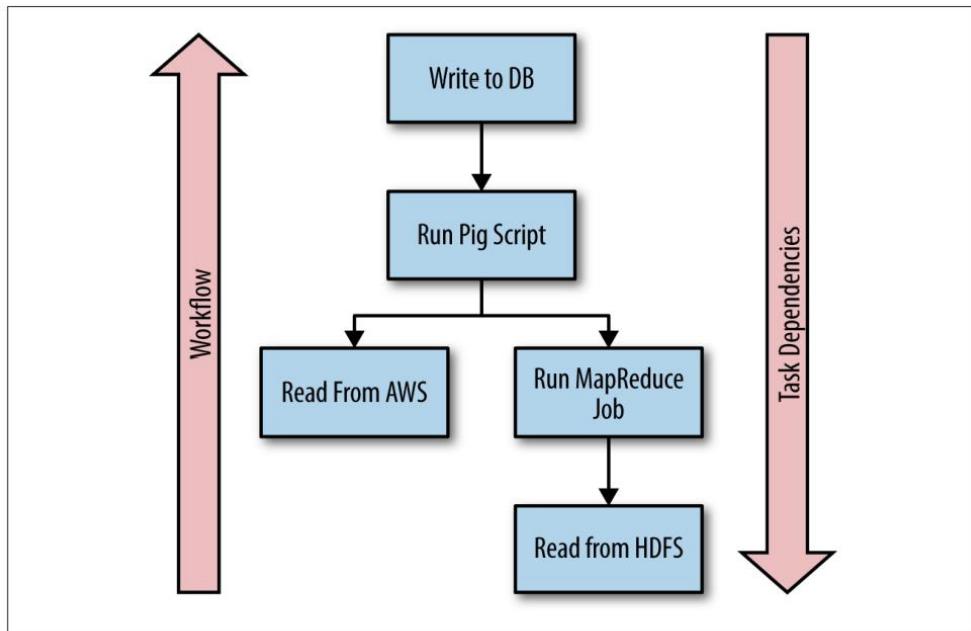


Figure 5-1. A Luigi task dependency diagram illustrates the flow of work up a pipeline and the dependencies between tasks

Read data from SQL SERVER using Pandas in python

Change database type from SQL Server to some other database, such as MySQL or Postgres
Use named parameters in my queries Avoid the use of raw string queries to protect against SQL injection attacks without having to deal with the complexities of changing hard-coded database connection settings and dialect-specific nuances? A quick note about using query parameters in PyODBC: - PyODBC supports the use of parameter markers using a question mark as placeholder in the SQL query. However, it does not natively support the use of named parameters in SQL queries.

Using a database abstraction layer such as SQLAlchemy would be useful in handling the nuances associated with different SQL dialects, as well as providing some protection against SQL injection through the use of built queries.

```
import pyodbc #open database driver
import pandas as pd
# Some other example server values are
# server = 'localhost\sqlexpress' # for a named instance
# server = 'myserver,port' # to specify an alternate port
server = 'DESKTOP-PC9C3P6'
database = 'AdventureWorks2012'
username = 'sa'
password = 'sqlserver'
cnxn = pyodbc.connect('DRIVER={SQL Server};SERVER='+server+';DATABASE='+database+';UID='+use
rname+';PWD=' + password)
cursor = cnxn.cursor()
# select 26 rows from SQL table to insert in dataframe.
query = "select * from humanresources.employee;"
df = pd.read_sql(query, cnxn)
print(df.head(26))
```

Read data from SQL SERVER using SQL Alchemy in python

PyODBC connections According to SQLAlchemy's documentation, an exact PyODBC connection string can be sent in pyodbc's format directly using the parameter odbc_connect. As the delimiters need to be URL-encoded (especially the Driver), urllib.parse.quote_plus is used to encode the PyODBC connection string.

```
import urllib
from sqlalchemy import create_engine

server = 'DESKTOP-PC9C3P6' # to specify an alternate port
database = 'AdventureWorks2012'
username = 'sa'
password = 'sqlserver'
```

```
params = urllib.parse.quote_plus("DRIVER={ODBC Driver 17 for SQL Server};SERVER='+server+';DATA  
BASE='+database+';  
        UID='+username+';PWD='+ password")  
engine = create_engine("mssql+pyodbc:///?odbc_connect=%s" % params)  
print(engine)
```