What is Python?

Python is a high-level, general-purpose programming language with an elegant syntax that allows programmers to focus more on problem-solving than on syntax errors. One of the primary goals of Python Developers is to keep it fun to use. Python has become a big buzz in the field of modern software development, infrastructure management, and especially in Data Science and Artificial Intelligence. It was created by Guido van Rossum, and released in 1991.

Why Python?

- Python works on different platforms (Windows, Mac, Linux, Raspberry Pi, etc).
- Python has a simple syntax similar to the English language.
- Python has syntax that allows developers to write programs with fewer lines than some other programming languages.
- Python runs on an interpreter system, meaning that code can be executed as soon as it is written. This means that prototyping can be very quick.
- Python can be treated in a procedural way, an object-oriented way or a functional way.

Why Is Python Popular for Data Science?

Data science is applicable in different industries, and it's helping to solve problems and discover more about the universe. In the health industry, data science helps doctors to make use of past data in making decisions, for example, diagnosis, or the right treatment for a disease. The education sector is not left out, you can now predict students dropping out of school, all thanks to data science.

- Python Has a Simple Syntax
- Wide Community as more people prefer working on python rather than R
- Python Offers All the Libraries including Numpy, Pandas, Matplotlib, Seaborn,etc.

Learn to write your `first program` in Python

In [ ]:

```
print("Hello World")
```

```
Hello World
```

We can see that it's a cake walk, being a dynamic programming language ,Python is good option for beginners to start their coding journey.

What are Identifiers in Python ?

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers.Python is a case sensitive programming language. Thus, Manpower and manpower are two different identifiers in Python.

Valid identifiers:

- ab10c: contains only letters and numbers
- abc_DE: contains all the valid characters
- _: surprisingly but Yes, underscore is a valid identifier
- _abc: identifier can start with an underscore

Invalid identifers:

- 99: identifier can’t be only digits
- 9abc: identifier can’t start with number
- x+y: the only special character allowed is an underscore
- for: it’s a reserved keyword

Worried about to confirm whether an identifier is ok or not??

There's a built in method to confirm that.

Let's see

In [ ]:

```
# Run this cell and see the answer in boolean way(True, False) and don't worry if you don't understand th
# Just keep in mind, there's a way to cross check the identifier.
print("abc".isidentifier())  # True
print("99a".isidentifier())  # False
print("_".isidentifier())  # True
print("for".isidentifier())  # True - wrong output
```

```
True
False
True
True
```

Reserved words in Python

Reserved words (also called keywords) are defined with predefined meaning and syntax in the language. These keywords have to be used to develop programming instructions. Reserved words can't be used as identifiers for other programming elements like name of variable, function etc.

Following is the list of reserved keywords in Python 3

and, except, lambda, with, as, finally, nonlocal, while, assert, false, None, yield, break, for, not, class, from, or, continue ,global, pass, def, if, raise, del, import, return, elif, in, True, else, is, try

To see what each keyword means here, you can refer this linkClick here

There's another way to check reserved keywords in Python

In [ ]:

```
# Run this command and you will see reserved keywords in a list(list is a data structure you will see la
import keyword
keyword.kwlist
```

Out[ ]:

```
['False',
 'None',
 'True',
 'and',
 'as',
 'assert',
 'async',
 'await',
 'break',
 'class',
 'continue',
 'def',
 'del',
 'elif',
 'else',
 'except',
 'finally',
 'for',
 'from',
 'global',
 'if',
 'import',
 'in',
 'is',
 'lambda',
 'nonlocal',
 'not',
 'or',
 'pass',
 'raise',
 'return',
 'try',
 'while',
 'with',
 'yield']
```

Comments in Python

As programs get bigger and more complicated, they get more difficult to read. Formal languages are dense, and it is often difficult to look at a piece of code and figure out what it is doing, or why.

For this reason, it is a good idea to add notes to your programs to explain in natural language what the program is doing. These notes are called comments, and they are marked with the # symbol:

compute the percentage of the hour that has elapsed percentage = (minute * 100) / 60

In this case, the comment appears on a line by itself. You can also put comments at the end of a line:

In [ ]:

```
minute =13
percentage = (minute * 100) / 60     # caution: integer division
```

Everything from the # to the end of the line is ignored it has no effect on the program. The message is intended for the programmer or for future programmers who might use this code. In this case, it reminds the reader about the ever-surprising behavior of integer division.

This sort of comment is less necessary if you use the integer division operation, //. It has the same effect as the division operator * Note, but it signals that the effect is deliberate.

percentage = (minute * 100) // 60

The integer division operator is like a comment that says, "I know this is integer division, and I like it that way!"

Variables in Python

Variables are containers for storing data values. Python has no command for declaring a variable.A variable is created the moment you first assign a value to it.

In [ ]:

```
# example
x=5
s='CloudyML'
print(x)
print(s)
```

```
5
CloudyML
```

Variables do not need to be declared with any particular type, and can even change type after they have been set. If you want to specify the data type of a variable, this can be done with **casting**

In [ ]:

```
# example
x=int(6)
y=str(6)
z=float(6)
print(x)
print(y)
print(z)
```

```
6
6
6.0
```

In the above example, the interpreter is explicitly type casting the variables and showing the necessary outputs.

Values and types

A value is one of the fundamental things like a letter or a number that a program manipulates.

For example, 2 is an integer, and 'Hello, World!' is a string, because it contains a "string" of letters. You (and the interpreter) can identify strings because they are enclosed in quotation marks.

If you are not sure what type a value has, the interpreter can tell you.

In [ ]:

```
print(type('Hello, World!'))
```

```
print(type(17))
```

```
<class 'str'>
<class 'int'>
```

Not surprisingly, strings belong to the type str and integers belong to the type int. Less obviously, numbers with a decimal point belong to a type called float, because these numbers are represented in a format called floating-point.

In [ ]:

```
print(type(3.2))
```

```
<class 'float'>
```

What about values like '17' and '3.2'? They look like numbers, but they are in quotation marks like strings.

In [ ]:

```
print(type('17'))
print(type('3.2'))
```

```
<class 'str'>
<class 'str'>
```

They're strings.

When you type a large integer, you might be tempted to use commas between groups of three digits, as in 1,000,000. This is not a legal integer in Python, but it is a legal expression:

In [ ]:

```
print (1,000,000100)
```

```
  File "<ipython-input-9-620cc0d67dc4>", line 1
    print (1,000,000100)
                      ^
SyntaxError: invalid token
```

Well, that's not what we expected at all! Python interprets 1,000,000 as a comma-separated list of three integers, which it prints consecutively. This is the first example we have seen of a semantic error: the code runs without producing an error message, but it doesn't do the "right" thing.

Python operators

Python operators allow us to do common processing on variables. We will look into different types of operators with examples and also operator precedence. They are the special symbols that can manipulate the values of one or more operands.

List of Python Operators Python operators can be classified into several categories.

- Assignment Operators
- Arithmetic Operators
- Comparison Operators
- Bitwise Operators
- Logical Operators

Assignment Operators

Assignment operators include the basic assignment operator equal to sign (=).

But to simplify code, and reduce redundancy, Python also includes arithmetic assignment operators.

This includes the += operator in Python used for addition assignment, //= floor division assignment operator, and others.

Here's a list of all the arithmetic assignment operators in Python.

In [ ]:

```
# take two variable, assign values with assignment operators
a=3
b=4

print("a: "+str(a))
print("b: "+str(b))

# it is equivalent to a=a+b
a+=b

print("a: "+str(a))
print("b: "+str(b))
```

```
# it is equivalent to a=a*b
a*=b
print("a: "+str(a))
print("b: "+str(b))

# it is equivalent to a=a/b
a/=b
print("a: "+str(a))
print("b: "+str(b))

# it is equivalent to a=a%b
a%=b
print("a: "+str(a))
print("b: "+str(b))

# it is equivalent to a=a**b ( exponent operator)
a**=b
print("a: "+str(a))
print("b: "+str(b))

# it is equivalent to a=a//b ( floor division)
a//=b
print("a: "+str(a))
print("b: "+str(b))
```

```
a: 3
b: 4
a: 7
b: 4
a: 28
b: 4
a: 7.0
b: 4
a: 3.0
b: 4
a: 81.0
b: 4
a: 20.0
b: 4
```

Arithmetic Operators

'+' is used to add two numbers sum = a + b

'–' is used for subtraction difference = a – b

'*' used to multiply two numbers. If a string and int is multiplied then the string is repeated the int times.

'/' used to divide two numbers div = b/a

'%' modulus operator, returns the remainder of division mod = a%b

'' exponent operator, return a rasied to the power of b**

In [ ]:

```
#create two variables
a=100
b=2

# addition (+) operator
print(a+b)

# subtraction (-) operator
print(a-b)

# multiplication (*) operator
print(a*b)

# division (/) operator
print(b/a)

# modulus (%) operator
print(a%b) # prints the remainder of a/b
```

```
# exponent (**) operator
print(a**b) #prints a^b
```

```
102
98
200
0.02
0
10000
```

Comparison Operators

'==' returns True if two operands are equal, otherwise False.

'!=' returns True if two operands are not equal, otherwise False.

'>' returns True if left operand is greater than the right operand, otherwise False.
'<' returns True if left operand is smaller than the right operand, otherwise False.

'>=' returns True if left operand is greater than or equal to the right operand, otherwise False.

'<=' returns True if left operand is smaller than or equal to the right operand, otherwise False.

In [ ]:

```
# create two variables
a=100
b=200

# (==) operator, checks if two operands are equal or not
print(a==b)

# (!=) operator, checks if two operands are not equal
print(a!=b)

# (>) operator, checks left operand is greater than right operand or not
print(a>b)

# (<) operator, checks left operand is less than right operand or not
print(a<b)
#(>=) operator, checks left operand is greater than or equal to right operand or not
print(a>=b)

# (<=) operator, checks left operand is less than or equal to right operand or not
print(a<=b)
```

```
False
True
False
True
False
True
```

Conditional Statements

These are the conditions used basically in if, elif, and else statement

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

In [ ]:

```
# simple if statement
a = 100
b = 101
if b > a:  # observe the ':' after if block
  print("b is greater than a")  # Observe the print statment line indentation(4 spaces gap from the line
```

```
b is greater than a
```

In [ ]:

```
a = 100
b = 50
if b > a:
  print("b is greater than a")
elif a == b:                    # elif keyword is pythons way of saying "if the previous
  print("a and b are equal")    # conditions were not true, then try this condition".

else:        # else keyword catches anything which isn't caught by the preceding conditions
  print("a is greater than b")
```

```
a is greater than b
```

Explanation

1. First if condition is checked, if it meets the requirement then code under if block is executed.
2. If it fails the requirement of first if condition, then it goes to next condition which is elif.
3. If it meets the requirement of elif condition, it executes code inside elif, otherwise it goes to final else block.

Logical Operators

'and' is Logical AND Operator

'or' is Logical OR Operator 'not' is Logical NOT Operator

In [ ]:

```
#take user input as int
a=int(input())

# logical AND operation

if a%4==0 and a%3==0:
    print("divided by both 4 and 3")

# logical OR operation
if a%4==0 or a%3==0:
    print("either divided by 4 or 3")

# logical NOT operation
if not(a%4==0 or a%3==0):
    print("neither divided by 4 nor 3")
```

```
47
neither divided by 4 nor 3
```

Python Operators Precedence

Precedence of these operators means the priority level of operators. This becomes vital when an expression has multiple operators in it. Below is a list of operators indicating the precedence level. It's in descending order. That means the upper group has more precedence than that of the lower group.

Parenthesis – ()

Exponentiation – **

Compliment, unary plus and minus – ~, +, -

Multiply, Divide, modulo – *, /, %

Addition and Subtraction – +, -

Right and Left Shift – >>, <<

Bitwise AND – &

Bitwise OR and XOR – |, ^

Comparison Operators – ==, !=, >, <, >=, <=

Assignment Operator- =

In [ ]:

```
print(2**9>>7//8-52+61)
```

1

Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

is Returns True if both variables are the same object

is not Returns True if both variables are not the same object

In [ ]:

```
a = 20
b = 20

if ( a is b ):
   print ("Line 1 - a and b have same identity")
else:
   print ("Line 1 - a and b do not have same identity")
b=30
if ( a is not b ):
   print ("Line 4 - a and b do not have same identity")
else:
   print ("Line 4 - a and b have same identity")
```

```
Line 1 - a and b have same identity
Line 4 - a and b do not have same identity
```

Membership Operators

Membership operators are used to test if a sequence is presented in an object:in Returns True if a sequence with the specified value is present in the object **not in** Returns True if a sequence with the specified value is not present in the object

In [ ]:

```
a = 10
b = 20
list = [1, 2, 3, 4, 5 ];

if ( a in list ):
   print ("Line 1 - a is available in the given list")
else:
   print ("Line 1 - a is not available in the given list")

if ( b not in list ):
   print ("Line 2 - b is not available in the given list")
else:
   print ("Line 2 - b is available in the given list")

a = 2
if ( a in list ):
   print ("Line 3 - a is available in the given list")
else:
   print ("Line 3 - a is not available in the given list")
```

```
Line 1 - a is not available in the given list
Line 2 - b is not available in the given list
Line 3 - a is available in the given list
```

Ternary Operator

Ternary operators in Python are terse conditional expressions. These are operators that test a condition and based on that, evaluate a value.

In [ ]:

```
a,b=2,3
print("a" if a>b else "b")
```

```
b
```

What is String in Python?

A string is a sequence of characters.

A character is simply a symbol. For example, the English language has 26 characters.

Computers do not deal with characters, they deal with numbers (binary). Even though you may see characters on your screen, internally it is stored and manipulated as a combination of 0s and 1s.

This conversion of character to a number is called encoding, and the reverse process is decoding. ASCII and Unicode are some of the popular encodings used.

In Python, a string is a sequence of Unicode characters. Unicode was introduced to include every character in all languages and bring uniformity in encoding. You can learn about Unicode from Python Unicode.

How to create a string in Python?

Strings can be created by enclosing characters inside a single quote or double-quotes. Even triple quotes can be used in Python but generally used to represent multiline strings and docstrings.

In [ ]:

```
# defining strings in Python
# all of the following are equivalent
my_string = 'Hello'
print(my_string)

my_string = "Hello"
print(my_string)

my_string = '''Hello'''
print(my_string)

# triple quotes string can extend multiple lines
my_string = """Hello, welcome to
           the world of Python"""
print(my_string)
```

```
Hello
Hello
Hello
Hello, welcome to
           the world of Python
```

How to access characters in a string?

We can access individual characters using indexing and a range of characters using slicing. Index starts from 0. Trying to access a character out of index range will raise an IndexError. The index must be an integer. We can't use floats or other types, this will result into TypeError.

Python allows negative indexing for its sequences.

The index of -1 refers to the last item, -2 to the second last item and so on. We can access a range of items in a string by using the slicing operator :(colon).

In [ ]:

```
#Accessing string characters in Python
str = 'CloudyML'
print('str = ', str)

#first character
print('str[0] = ', str[0])

#last character
print('str[-1] = ', str[-1])

#slicing 2nd to 5th character
print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character
print('str[5:-2] = ', str[5:-2])
```

```
str =  CloudyML
str[0] =  C
str[-1] =  L
str[1:5] =  loud
str[5:-2] =  y
```

Python String Methods

capitalize() Converts the first character to upper case

casefold() Converts string into lower case

center() Returns a centered string

count() Returns the number of times a specified value occurs in a string

encode() Returns an encoded version of the string

endswith() Returns true if the string ends with the specified value

expandtabs() Sets the tab size of the string

find() Searches the string for a specified value and returns the position of where it was found

format() Formats specified values in a string

format_map() Formats specified values in a string

index() Searches the string for a specified value and returns the position of where it was found

isalnum() Returns True if all characters in the string are alphanumeric

isalpha() Returns True if all characters in the string are in the alphabet

isascii() Returns True if all characters in the string are ascii characters

isdecimal() Returns True if all characters in the string are decimals

isdigit() Returns True if all characters in the string are digits

isidentifier() Returns True if the string is an identifier

islower() Returns True if all characters in the string are lower case

isnumeric() Returns True if all characters in the string are numeric

isprintable() Returns True if all characters in the string are printable

isspace() Returns True if all characters in the string are whitespaces

istitle() Returns True if the string follows the rules of a title

isupper() Returns True if all characters in the string are upper case

join() Converts the elements of an iterable into a string

ljust() Returns a left justified version of the string

lower() Converts a string into lower case

lstrip() Returns a left trim version of the string

maketrans() Returns a translation table to be used in translations

partition() Returns a tuple where the string is parted into three parts

replace() Returns a string where a specified value is replaced with a specified value

rfind() Searches the string for a specified value and returns the last position of where it was found

rindex() Searches the string for a specified value and returns the last position of where it was found

rjust() Returns a right justified version of the string

rpartition() Returns a tuple where the string is parted into three parts

rsplit() Splits the string at the specified separator, and returns a list

rstrip() Returns a right trim version of the string

split() Splits the string at the specified separator, and returns a list

splitlines() Splits the string at line breaks and returns a list

startswith() Returns true if the string starts with the specified value

strip() Returns a trimmed version of the string

swapcase() Swaps cases, lower case becomes upper case and vice versa

title() Converts the first character of each word to upper case

translate() Returns a translated string

upper() Converts a string into upper case

zfill() Fills the string with a specified number of 0 values at the beginning

In [ ]:

```
mystring = 'Hi there'
print(mystring.capitalize())      #converting the first character of the string into uppercase
print(mystring.casefold())        #converts the string into lowercase

print("------------")

s = "heLLo BuDdY"
s2 = s.lower()          #converts the string into lowercase
print(s2)

print("------------")

s = "heLLo BuDdY"
s2 = s.upper()            #converts the string into uppercase
print(s2)

print("------------")

s = "heLLo BuDdY"
s2 = s.title()          #converts first letter of each word of the string to uppercase
print(s2)

print("------------")

s = "heLLo BuDdY"
s2 = s.swapcase()    #swap uppercase to lowercase and vice versa
print(s2)

print("------------")

s = "Tech50"
print(s.isalnum())     #return True if all the characters of the string are alphanumeric
s = "Tech"
print(s.isalnum())
s = "678"
print(s.isalnum())

print("------------")

s1 = "Techvidvan"
s2 = "Tech50"
print(s1.isalpha())           #returns True if all characters in the string are alphabet
print(s2.isalpha())

print("------------")

s1 = "Tech50"
s2 = "56748"
print(s1.isdigit())
print(s2.isdigit())         #returns True if all characters in the string are digits

print("------------")

mystring = "Python"
print(mystring.find("P"))
print(mystring.find("on"))      #sQearches the string for THE specified value and returns the position o
```

```
Hi there
hi there
------------
hello buddy
------------
HELLO BUDDY
------------
Hello Buddy
------------
HEllO bUdDy
------------
True
True
True
------------
True
False
------------
False
True
------------
0
4
```

Escape Characters in Python

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

In [ ]:

```
txt = "We are the so-called \"Vikings\" from the north."
print(txt)
```

```
We are the so-called "Vikings" from the north.
```

Other escape characters used in Python:

\' -Single Quote
\ - Backslash
\n - New Line
\r - Carriage Return \t - Tab \b - Backspace
\f - Form Feed
\ooo - Octal value \xhh - Hex value

In [ ]:

```
# single quote escape character
x = 'hello\'world'
print(x)

print("------------")

# double quote escape character
x = "hello\"world"
print(x)

print("------------")

# backslash escape character
x = 'hello\\world'
print(x)

print("------------")

# newline escape character
x = 'hello\nworld'
print(x)

print("------------")

# carriage return escape character
x = 'hello\rworld'
print(x)
```

```
print("------------")

# tab escape character
x = 'hello\tworld'
print(x)

print("------------")

# backspace escape character
x = 'hello\bworld'
print(x)

print("------------")

# form feed escape character
x = 'hello\fworld'
print(x)

print("------------")

# octal value escape character
x = '\101\102\103'
print(x)

print("------------")

# hex value escape character
x = '\x41\x42\x43'
print(x)
```

```
hello'world
------------
hello"world
------------
hello\world
------------
hello
world
------------
world
------------
hello world
------------
hello world
------------
hello world
------------
ABC
------------
ABC
```

String characters & slicing

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string

Syntax: string[start:stop] - items start through stop-1

string[start:] - items start through the rest of the array

string[:stop] - items from the beginning through stop-1

string[:] - a copy of the whole array

string[start:stop:step] - start through not past stop, by step

In [ ]:

```
b = "Hello, World!"
print(b[1:8])

print("-----------")

# omit first or last index
print(b[:7])
```

```
print(b[5:])

print("------------")

# omit both index
print(b[:])

print("------------")

# using negative index
print(b[-5:-9])
print(b[::-5])

print("------------")

# slicing with step

print(b[0:8:3])
print(b[0::3])
print(b[:8:3])
print(b[::-1])
```

```
ello, W
------------
Hello,
, World!
------------
Hello, World!
------------

!Wl
------------
Hl
Hl r!
Hl
!dlroW ,olleH
```

Booleans in Python

The Python Boolean type is one of Python's built-in data types. It's used to represent the truth value of an expression. For example, the expression 1 <= 2 is True, while the expression 0 == 1 is False. Understanding how Python Boolean values behave is important to programming well in Python.

The Python Boolean type has only two possible values:

- **True**
- **False**

In [ ]:

```
print(type(False))
print(type(True))
```

```
<class 'bool'>
<class 'bool'>
```

Python Booleans as Keywords

Built-in names aren't keywords. As far as the Python language is concerned, they're regular variables. If you assign to them, then you'll override the built-in value.

In contrast, the names True and False are not built-ins. They're keywords. Unlike many other Python keywords, True and False are Python expressions. Since they're expressions, they can be used wherever other expressions, like 1 + 1, can be used.

It's possible to assign a Boolean value to variables, but it's not possible to assign a value to True:

In [ ]:

```
a_true_alias=True
print(a_true_alias)
```

```
True
```

But, True = 5 will throw an error like this:

SyntaxError: cannot assign to True

Because True is a keyword, you can't assign a value to it. The same rule applies to False

Python Booleans as Numbers

Booleans are considered a numeric type in Python. This means they're numbers for all intents and purposes. In other words, you can apply arithmetic operations to Booleans, and you can also compare them to numbers:

In [ ]:

```
print(True == 1)
print(False == 0)
print(True + False//True)
```

```
True
True
1
```

Number datatypes in Python

There are three distinct numeric types: **integers**, **floating point numbers**, and **complex numbers** The constructors **int()**, **float()**, and **complex()** can be used to produce numbers of a specific

Integers An integer is a whole number with no decimal places. For example, 1 is an integer, but 1.0 isn't. The name for the integer data type is int, which you can see with type() function.

You can create an integer by typing the desired number. For instance, the following assigns the integer 25 to the variable num.

You may already be familiar with how to convert a string containing an integer to a number using int(). For example, the following converts the string "25" to the integer 25

In [ ]:

```
print(type(1))
num=25
print(num)
print(int("25"))
```

```
<class 'int'>
25
25
```

When you write large numbers by hand, you typically group digits into groups of three separated by a comma or a decimal point. The number 1,000,000 is a lot easier to read than 1000000.

In Python, you can't use commas to group digits in integer literals, but you can use underscores (_). Both of the following are valid ways to represent the number one million as an integer literal.

In [ ]:

```
num=1000000
num1=1_000_000
print(num," ",num1)
```

```
1000000   1000000
```

Floating-Point Numbers A floating-point number, or float for short, is a number with a decimal place. 1.0 is a floating-point number, as is -2.75. The name of the floating-point data type is float.

Like integers, floats can be created from floating-point literals or by converting a string to a float with float().

There are three ways to represent a floating-point literal. Each of the following creates a floating-point literal with a value of one million.

The first two ways are similar to the two techniques for creating integer literals. The third approach uses E notation to create a float literal.

To write a float literal in E notation, type a number followed by the letter e and then another number. Python takes the number to the left of the e and multiplies it by 10 raised to the power of the number after the e. So 1e6 is equivalent to 1×10^6.

The float 200000000000000000.0 gets displayed as 2e+17. The + sign indicates that the exponent 17 is a positive number. You can also use negative numbers as the exponent.

In [ ]:

```
print(type(1.0))
```

```
print(float("1.25"))
f_num=1000000
f_num1=1_000_000
f_num2= 1e6        #positive number as exponent
f_num3 = 1e-4      #negative numbers as exponent
print(f_num)
print(f_num1)
print(f_num2)
```

```
<class 'float'>
1.25
1000000
1000000
1000000.0
```

Complex Numbers Python is one of the few programming languages that provides built-in support for complex numbers. While complex numbers don't often come up outside the domains of scientific computing and computer graphics, Python's support for them is one of its strengths.

If you've ever taken a precalculus or higher-level algebra math class, then you may remember that a complex number is a number with two distinct components: a real part and an imaginary part.

To create a complex number in Python, you simply write the real part, then a plus sign, then the imaginary part with the letter j at the end.

When you inspect the value of n, you'll notice that Python wraps the number with parentheses.

This convention helps eliminate any confusion that the displayed output may represent a string or a mathematical expression.

Imaginary numbers come with two properties, .real and .imag, that return the real and imaginary components of the number, respectively.

Notice that Python returns both the real and imaginary components as floats, even though they were specified as integers.

Complex numbers also have a .conjugate() method that returns the complex conjugate of the number.

For any complex number, its conjugate is the complex number with the same real part and an imaginary part that is the same in absolute value but with the opposite sign. So in this case, the complex conjugate of 1 + 2j is 1 - 2j.

Except for the floor division operator (//), all the arithmetic operators that work with floats and integers will also work with complex numbers. Instead, here are some examples of using complex numbers with arithmetic operators.

In [ ]:

```
n=1 +2j
print(n)

print(n.real)
print(n.imag)

print(n.conjugate)

a=1+2j
b=3-4j
print(a+b)
print(a-b)
print(a*b)
print(a**b)
print(a/b)
```

```
(1+2j)
1.0
2.0
<built-in method conjugate of complex object at 0x7f2f43b4b810>
(4-2j)
(-2+6j)
(11+2j)
(932.1391946432212+95.9465336603415j)
(-0.2+0.4j)
```

In [ ]:

```
print(a//b)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-35-1133a5709cb6> in <module>
----> 1 print(a//b)

TypeError: can't take floor of complex number.
```

In [ ]:

```
x=34
print(x.real)
print(x.imag)
print(x.conjugate)

y=3.14
print(y.real)
print(y.imag)
print(y.conjugate)
```

```
34
0
<built-in method conjugate of int object at 0xa9bbc0>
3.14
0.0
<built-in method conjugate of float object at 0x7f2f476e5c90>
```

Built-in Functions in Python

Python has a total of 65+ built-in functions. Link to the documentation-https://docs.python.org/3/library/functions.html

Commonly used built-in functions are given below:

- **Python abs()** returns absolute value of a number
- **Python any()** Checks if any Element of an Iterable is True
- **Python bool()** Converts a Value to Boolean
- **Python chr()** Returns a Character (a string) from an Integer
- **Python dict()** Creates a Dictionary
- **Python format()** returns formatted representation of a value
- **Python id()** Returns Identify of an Object
- **Python len()** Returns Length of an Object
- **Python max()** returns the largest item
- **Python min()** returns the smallest item
- **Python ord()** returns an integer of the Unicode character
- **Python pow()** returns the power of a number
- **Python reversed()** returns the reversed iterator of a sequence
- **Python round()** rounds a number to specified decimals
- **Python set()** constructs and returns a set
- **Python slice()** returns a slice object
- **Python sorted()** returns a sorted list from the given iterable
- **Python str()** returns the string version of the object
- **Python sum()** Adds items of an Iterable
- **Python tuple()** Returns a tuple
- **Python type()** Returns the type of the object

Taking User input

input() method is used to take user input in python

In [ ]:

```
username = input("Enter username:")
print("Username is: " + username)
```

```
Enter username:Taylor
Username is: Taylor
```

In [ ]:

```
# Python program showing use of input()
name = input("Enter your name: ")  # String Input
age = int(input("Enter your age: ")) # Integer Input
marks = float(input("Enter your marks: ")) # Float Input
print("The name is:", name)
print("The age is:", age)
print("The marks is:", marks)
#By default, the input() function takes input as a string so
```

```
# if we need to enter the integer or float type input then the input() function must be type casted.
```

```
Enter your name: Taylor
Enter your age: 56
Enter your marks: 23.65
The name is: Taylor
The age is: 56
The marks is: 23.65
```

Loops in python

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times.

Python programming language provides following types of loops to handle looping requirements.

- ***while loop*** repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.

In [ ]:

```
print("use of while loop")
count = 0
while (count < 3):
    count = count + 1
    print("CloudyML")
```

```
use of while loop
CloudyML
CloudyML
CloudyML
```

Range function in Python

Before learning FOR loop, we need to understand range() function in python and its significance in for loop.

The range() function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and stops before a specified number.

SYNTAX:

range(start, stop, step)

where

start is optional. It is an integer number specifying at which position to start. Default value is 0.

stop is required. It is an integer number specifying at which position to stop (not included).

step is also optional. It is an integer number specifying the incrementation. Default value is 1.

In [ ]:

```
x = range(3, 6)
for n in x:
  print(n)

print("---------------")

y = range(3, 20, 2)
for n in y:
  print(n)

print("---------------")

z = range(6)
for n in z:
  print(n)
```

```
3
4
5
---------------
3
5
7
9
11
13
15
17
19
---------------
0
1
2
3
4
5
```

- ***for loop*** executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.

In [ ]:

```
print("use of for loop")
for i in range(0,10,2):  #the for loop will go till (10-1)=9 with step 2
  print(i)
```

```
use of for loop
0
2
4
6
8
```

- ***nested loops*** can be used with one or more loop inside any another while, for or do..while loop.

A final note on loop nesting is that we can put any type of loop inside of any other type of loop. For example, a for loop can be inside a while loop or vice versa.

In [ ]:

```
print("Nested Loops for patterns")
for i in range(0,5):
  for j in range(0,i+1):
    print("*",end=" ")
  print("\r")
```

```
Nested Loops for patterns
*
* *
* * *
* * * *
* * * * *
```

Using else statement with while loops: We know, while loop executes the block until a condition is satisfied. When the condition becomes false, the statement immediately after the loop is executed. The else clause is only executed when your while condition becomes false. If you break out of the loop, or if an exception is raised, it won't be executed.

In [ ]:

```
print("Using else with while loop")
count = 4
while (count <7 ):
    count = count + 1
    print("Hello World")
else:
    print("Else executed")  #else statement gets executed after condition becomes false
```

```
Using else with while loop
Hello World
Hello World
Hello World
Else executed
```

using else statement with for loops: We can also combine else statement with for loop like in while loop. But as there is no condition in for loop based on which the execution will terminate so the else block will be executed immediately after for block finishes execution.

In [ ]:

```
print("Using else with for loop")
list = ["apple", "banana", "mango"]
for index in range(len(list)):
    print (list[index])
else:
    print ("Else executed")  # else statement gets executed after for loop gets over
```

```
Using else with for loop
apple
banana
mango
Else executed
```

Loop control Statements

Loop control statements change execution from their normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

- ***Continue Statement***: **It returns the control to the beginning of the loop.**
- ***Break Statement***: **It brings control out of the loop.**
- ***Pass Statement*** : **We use pass statement to write empty loops. Pass is also used for empty control statements, functions and classes.**

In [ ]:

```
string="It is a sunny day"
# use of continue statement with for loop
for char in string:
  if char=="a" or char=="i" or char=="u":
    continue
  else:
    print(char,end="")
print("\r")
# use of break statement with for loop
for char in string:
  if char=="a":
    break
  else:
    print(char,end="")
# use of pass statement with for loop
for char in string:
  pass
print("Last letter :",char)
```

```
It s  snny dy
It is Last letter : y
```

In [ ]:

```
#use of continue statement with while loop
num = 0
while num < 10:
    num += 1
    if num == 6:
        continue
    print(num)
# use of break statement with while loop
num = 1
odd_nums = []
while num:
    if num % 2 != 0:
        odd_nums.append(num)
    if num >=20:
        break
    num += 1
print("Odd numbers: ", odd_nums)
# use of pass statement with while loop
num = 1
while num <= 10:
    if num == 6:
        pass
    print(num)
    num += 1
```

```
1
2
3
4
5
7
8
9
10
Odd numbers:  [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
1
2
3
4
5
6
7
8
9
10
```

Functions in Python

A function is a block of code which only runs when it is called. You can pass data, known as parameters, into a function. A function can return data as a result.

Creating a Function: In Python a function is defined using the def keyword.

In [ ]:

```
def my_function():
  print("It is a function")
```

Calling a Function : To call a function, use the function name followed by parenthesis:

In [ ]:

```
def my_function():
  print("It is a function")

my_function()
```

```
It is a function
```

In the above example, there is no output because we have only created the function. But,in second code snippet we have called the function. The interpreter will call and then execute the function.

Arguements in Python Function

Information can be passed into functions as arguments.

Arguments are specified after the function name, inside the parentheses. You can add as many arguments as you want, just separate them with a comma.

The following example has a function with one argument (fname). When the function is called, we pass along a first name, which is used inside the function to print the full name

In [ ]:

```
def my_func(fname):
  print(fname + " Refsnes")

my_func("Emil")
my_func("Tobias")
my_func("Linus")
```

```
Emil Refsnes
Tobias Refsnes
Linus Refsnes
```

Types of Arguments in Python

There are two types of arguments: **positional arguments** and **keyword arguments**.

Positional arguments

Positional arguments are values that are passed into a function based on the order in which the parameters were listed during the function definition. Here, the order is especially important as values passed into these functions are assigned to corresponding parameters based on their position.

In [ ]:

```
def team(name, project):
    print(name, "is working on an", project)        #example A

team("John", "Titanic.csv")


def team(name, project):
    print(name, "is working on an", project)        #example B

team("Titanic.csv", "John")
```

```
John is working on an Titanic.csv
Titanic.csv is working on an John
```

In this example, we see that when the positions of the arguments are changed, the output produces different results. Though the code in example B isn't wrong, the values that have been passed into the function are in the wrong order; thus, producing a result that does not match our desired output. Titanic.csv is the name of the project that is being worked on by the person, John, not the other way around.

Keyword arguments

Keyword arguments (or named arguments) are values that, when passed into a function, are identifiable by specific parameter names. A keyword argument is preceded by a parameter and the assignment operator, = .

Keyword arguments can be likened to dictionaries in that they map a value to a keyword.

In [ ]:

```
def team(name, project):
    print(name, "is working on an", project)

team(project = "John", name = 'Titanic.csv')


def team(name, project):
    print(name, "is working on an", project)

team(name = "Titanic.csv", project = 'John')
```

```
Titanic.csv is working on an John
Titanic.csv is working on an John
```

As you can see, we had the same output from both codes although, when calling the function, the arguments in each code had different positions.

With keyword arguments, as long as you assign a value to the parameter, the positions of the arguments do not matter.

However, they do have to come after positional arguments and before default/optional arguments in a function call.

Lambda Function

In Python, an anonymous function is a function that is defined without a name.

While normal functions are defined using the def keyword in Python, anonymous functions are defined using the lambda keyword.

Hence, anonymous functions are also called lambda functions.

In [ ]:

```
# example of lambda function
y = lambda x: x * 2 + 5

print(y(5))
```

Variable Scope in Python

Since, python is not statically typed language,so we do not need to declare variables before using them or declare their type. A variable is created the moment we first assign a value to it.

Scope of Variable :The location where we can find a variable and also access it if required is called the scope of a variable.

Global and local variables Global variables are the ones that are defined and declared outside any function and are not specified to any function. They can be used by any part of the program.

In [ ]:

```
def func():
    print(string)

# Global scope
string = "Data Science is the future"
func()
```

```
Data Science is the future
```

In [ ]:

```
# This function has a variable with
# name same as x.
x = 300

def func():
  x = 200
  print(x)

func()

print(x)
```

```
200
300
```

Global Keyword

If you need to create a global variable, but are stuck in the local scope, you can use the global keyword. The global keyword makes the variable global.

In [ ]:

```
def func():
  global x
  x = 300

func()

print(x)
```

```
300
```

Also, use the global keyword if you want to make a change to a global variable inside a function.

In [ ]:

```
x = 300

def func():
  global x
  x = 200

func()

print(x)
```

```
200
```

Data structures in Python

Before starting with Python, we should be knowing what are data structures and why are they used.**Data structure** is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

Depending on your requirement and project, it is important to choose the right data structure for your project. For example, if you want to store data sequentially in the memory, then you can go for the Array data structure.

Python Specific Data Structures

These data structures are specific to python language and they give greater flexibility in storing different types of data and faster processing in python environment.

List – It is similar to array with the exception that the data elements can be of different data types. You can have both numeric and string data in a python list.

Tuple – Tuples are similar to lists but they are immutable which means the values in a tuple cannot be modified they can only be read.

Dictionary – The dictionary contains Key-value pairs as its data elements.

Set - A set is a collection which is unordered, unchangeable*, and unindexed.

Lists

- Lists are used to store multiple items in a single variable.
- Lists are created using square brackets
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- Since lists are indexed, lists can have items with the same value

example: listL1=["Apple", "Mango", "Banana", "Grapes"]

- List items can be of any data type: string, int and boolean data types.
- A list can contain different data types.

example: ListL2=[78, 'People', 812.369, 4+9j, True]

Access Items of list

List items are indexed and you can access them by referring to the index number

In [ ]:

```
listL1=["Apple", "Mango", "Banana", "Grapes"]
print(listL1[3])


#using negative indexing
print(listL1[-2])

#range of indexes
print(listL1[1:3])

#range of negtive indexes
print(listL1[-4:-2])
```

```
Grapes
Banana
['Mango', 'Banana']
['Apple', 'Mango']
```

Check if item exists in List

To determine if a specified item is present in a list use the***in*** keyword

In [ ]:

```
#check if cherry is present in the list
if "cherry" in listL1:
  print("yes")
else:
```

```
  print("no")
```

```
no
```

Change items in a list

In [ ]:

```
listL1[1] = "blackberries"
print(listL1)
```

```
['Apple', 'blackberries', 'Banana', 'Grapes']
```

Insert items in a list

To insert a new list item, without replacing any of the existing values, we can use the insert() method.

The insert() method inserts an item at the specified index.

In [ ]:

```
listL1.insert(2, "watermelon")
print(listL1)
```

```
['Apple', 'blackberries', 'watermelon', 'Banana', 'Grapes']
```

Append items to the list

To add an item to the end of the list, use the append() method.

In [ ]:

```
listL1.append("orange")
print(listL1)
```

```
['Apple', 'blackberries', 'watermelon', 'Banana', 'Grapes', 'orange']
```

Extend List

To append elements from another list to the current list, use the extend() method.

In [ ]:

```
tropical = ["mango", "pineapple", "papaya"]
listL1.extend(tropical)
print(listL1)
```

```
['Apple', 'blackberries', 'watermelon', 'Banana', 'Grapes', 'orange', 'mango', 'pineapple', 'papaya']
```

Remove list items

- **The remove() method removes the specified item**

In [ ]:

```
listL1.remove("Banana")
print(listL1)
```

```
['Apple', 'blackberries', 'watermelon', 'Grapes', 'orange', 'mango', 'pineapple', 'papaya']
```

- **The pop() method removes the specified index. If you do not specify the index, the pop() method removes the last item.**

In [ ]:

```
listL1.pop(1)
print(listL1)


listL1.pop()
print(listL1)
```

```
['Apple', 'watermelon', 'Grapes', 'orange', 'mango', 'pineapple', 'papaya']
['Apple', 'watermelon', 'Grapes', 'orange', 'mango', 'pineapple']
```

- **The del keyword also removes the specified index. It can also delete the list completely.**

In [ ]:

```
del listL1[0]
print(listL1)


thislist = ["apple", "banana", "cherry"]
del thislist
```

```
['watermelon', 'Grapes', 'orange', 'mango', 'pineapple']
```

- **The clear() method empties the list.The list still remains, but it has no content.**

In [ ]:

```
thislist = ["apple", "banana", "cherry"]
thislist.clear()
print(thislist)
```

```
[]
```

List comprehension

A Python list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element in the Python list.

Python List comprehension provides a much more short syntax for creating a new list based on the values of an existing list.

newList = [expression(element) for element in oldList if condition] #Syntax

In [ ]:

```
List1 = [character for character in [1, 2, 3]]
print(List1)



List2 = [i for i in range(11) if i % 2 == 0]
print(List2)
```

```
[1, 2, 3]
[0, 2, 4, 6, 8, 10]
```

In [ ]:

```
#taking another example, here we have to find names ending with letter 's' having length more than 5

names=['Lucas','Maximus','John',' Catilyn','Margaret','Ellis','Evans','Dua','Chris','Travis']

namess=[]
for name in names:
  if name.endswith('s') and len(name)>5:
      namess.append(name)       #append all the names that satisfy the condition into another new list ca

print(namess)  #prints the list namess
```

```
['Maximus', 'Travis']
```

In [ ]:

```
#writing same code using list comprehension
namess= [name for name in names if name.endswith('s') and len(name)>5]
print(namess)
```

```
['Maximus', 'Travis']
```

Hence, we can see that list comprehension does the work easier and reduced the no of lines of codes that we had to write earlier in the above example.

Taking list input in Python

In [ ]:

```
# creating an empty list
lst = []

# number of elements as input
n = int(input("Enter number of elements : "))

# iterating till the range
for i in range(0, n):
 ele = int(input())

 lst.append(ele) # adding the element

print(lst)
```

```
Enter number of elements : 2
1
2
[1, 2]
```

Using map() to take list input

In [ ]:

```
n = int(input("Enter the size of list : "))
print("\n")
l=map(list,input().split())
```

```
Enter the size of list : 3


0 1 2
```

Sorting Lists **List objects have a sort() method that will sort the list alphanumerically, ascending, by default.**

In [ ]:

```
#using example of names list used in list comprehension above
names=['Lucas','Maximus','John',' Catilyn','Margaret','Ellis','Evans','Dua','Chris','Travis']
names.sort()
print(names)
```

```
[' Catilyn', 'Chris', 'Dua', 'Ellis', 'Evans', 'John', 'Lucas', 'Margaret', 'Maximus', 'Travis']
```

Now sorting the lists in descending order, we have use sort() with condition reverse=True.

In [ ]:

```
# using the same example
names=['Lucas','Maximus','John',' Catilyn','Margaret','Ellis','Evans','Dua','Chris','Travis']
names.sort(reverse=True)
print(names)
```

```
['Travis', 'Maximus', 'Margaret', 'Lucas', 'John', 'Evans', 'Ellis', 'Dua', 'Chris', ' Catilyn']
```

The reverse() method reverses the current sorting order of the elements.

In [ ]:

```
#above we have sorted the list in descending order so using reverse() method will change the order in as
names.reverse()
print(names)
```

```
[' Catilyn', 'Chris', 'Dua', 'Ellis', 'Evans', 'John', 'Lucas', 'Margaret', 'Maximus', 'Travis']
```

Copy Lists

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

In [ ]:

```
names=['Lucas','Maximus','John',' Catilyn']
mylist=names.copy()
print(mylist)
```

```
['Lucas', 'Maximus', 'John', ' Catilyn']
```

In [ ]:

```
#Another way to make a copy is to use the built-in method list().

mylist2= list(names)
print(mylist2)
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
<ipython-input-233-30ef4c71904a> in <module>
      1 #Another way to make a copy is to use the built-in method list().
      2
----> 3 mylist2= list(names)
      4 print(mylist2)

TypeError: 'list' object is not callable
```

Join Lists

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

In [ ]:

```
list1=['dog','cat','parrot']
list2=[45,39,87]
list3=list1 + list2
print(list3)  #it is joining the lists and creating a new list called list3
```

```
print(list1 + list2)     #while this statement is just printing both the lists together without creating
#both the statements do the same work
```

```
['dog', 'cat', 'parrot', 45, 39, 87]
['dog', 'cat', 'parrot', 45, 39, 87]
```

In [ ]:

```
#Another way to join two lists is by appending all the items from list2 into list1, one by one using app
list1=['dog','cat','parrot']
list2=[45,39,87]
for x in list2:
  list1.append(x)

print(list1)
```

```
['dog', 'cat', 'parrot', 45, 39, 87]
```

In [ ]:

```
#Or you can use the extend() method, which purpose is to add elements from one list to another list
list1=['dog','cat','parrot']
list2=[45,39,87]
list1.extend(list2)
print(list1)
```

```
['dog', 'cat', 'parrot', 45, 39, 87]
```

Hence, using **append()** and **extend()** method we can join the lists without creating extra list.

Tuples

- Tuples are used to store multiple items in a single variable.
- Tuple is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Set, and Dictionary, all with different qualities and usage.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.

Example: Tuple1=('name','place','animal','thing')

- Tuple items are ordered, unchangeable, and allow duplicate values.
- Tuple items are indexed, the first item has index [0], the second item has index [1] etc.
- When we say that tuples are ordered, it means that the items have a defined order, and that order will not change.
- Tuples are unchangeable, meaning that we cannot change, add or remove items after the tuple has been created.
- Since tuples are indexed, they can have items with the same value.
- It is also possible to use the tuple() constructor to make a tuple.

In [ ]:

```
Tuple1=('name','place','animal','thing')
print(Tuple1)
```

```
('name', 'place', 'animal', 'thing')
```

We can find length of tuple using len() method.

In [ ]:

```
print(len(Tuple1))
```

```
4
```

Create tuple with one item

To create a tuple with only one item, you have to add a comma after the item, otherwise Python will not recognize it as a tuple.

In [ ]:

```
thistuple = ("Steve Jobs",)
print(type(thistuple))

#NOT a tuple
thistuple = ("Steve Jobs")
print(type(thistuple))
```

```
<class 'tuple'>
<class 'str'>
```

Tuple Items - Data Types

- Tuple items can be of any data type.
- A tuple can contain different data types.

In [ ]:

```
#example
my_tuple=("True",True,1,1-1j)
```

Access tuples items

- You can access tuple items by referring to the index number, inside square brackets.
- The first item has index 0.

In [ ]:

```
print(my_tuple[2])          #using index
print(my_tuple[-2])        #using negative indexing
print(my_tuple[1:3])       #using range of indexes
print(my_tuple[:2])         #By leaving out the start value, the range will start at the first item
print(my_tuple[2:])         #By leaving out the end value, the range will go on to the end of the list
print(my_tuple[-4:-1])       #for a range of negative indexes
```

```
1
1
(True, 1)
('True', True)
(1, (1-1j))
('True', True, 1)
```

Check if an item exists in tuple

To determine if a specified item is present in a tuple use the in keyword.

In [ ]:

```
print(True in my_tuple)      #return either true or false based on whether the item exists or not
```

```
True
```

Update Tuples

Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

In [ ]:

```
x=('cat','bat','rat')
z=[]
for each in x:
    z.append(each)
z[1]='hat'
x=tuple(z)
print(x)
```

```
('cat', 'hat', 'rat')
```

Add items to tuples

Since tuples are immutable, they do not have a build-in append() method, but there are other ways to add items to a tuple.

1. *Convert into a list*: Just like the workaround for changing a tuple, you can convert it into a list, add your item(s), and convert it back into a tuple.

In [ ]:

```
x=('cat','bat','rat')
z=[]
for each in x:
    z.append(each)
z.append('hat')
x=tuple(z)
print(x)
```

```
('cat', 'bat', 'rat', 'hat')
```

1. Add tuple to a tuple. You are allowed to add tuples to tuples, so if you want to add one item, (or many), create a new tuple with the item(s), and add it to the existing tuple.

In [ ]:

```
x = ("cat", "bat", "rat")
z = ("hat",)
x += z

print(x)
```

```
('cat', 'bat', 'rat', 'hat')
```

Remove items from tuple

Tuples are unchangeable, so you cannot remove items from it, but you can use the same workaround as we used for changing and adding tuple items.

In [ ]:

```
x=('cat','bat','rat')
p=[]
for each in x:
    p.append(each)
p.remove('cat')
x=tuple(p)
print(x)
```

```
('bat', 'rat')
```

Or you can delete the tuple completely.The del keyword can delete the tuple completely.

In [ ]:

```
x = ("cat", "bat", "rat")
del x
```

Unpack Tuples

When we create a tuple, we normally assign values to it. This is called "packing" a tuple. But, in Python, we are also allowed to extract the values back into variables. This is called "unpacking"

In [ ]:

```
animals=('lion','zebra','monkey')
(lion,zebra,monkey) = animals
print(lion)
print(zebra)
print(monkey)
```

```
lion
zebra
monkey
```

Using asterik '*' to unpack

If the number of variables is less than the number of values, you can add an * to the variable name and the values will be assigned to the variable as a list.

In [ ]:

```
#Assign the rest of the values as a list called "wild"
animals=('lion','zebra','monkey','tiger','bear')
(lion,zebra,*wild) = animals
print(lion)
print(zebra)
print(wild)
```

```
lion
zebra
['monkey', 'tiger', 'bear']
```

If the asterisk is added to another variable name than the last, Python will assign values to the variable until the number of values left matches the number of variables left.

In [ ]:

```
#Add a list of values the "tropic" variable
animals=('lion','zebra','monkey','tiger','bear')
(lion,*tropic,bear) = animals
print(lion)
print(tropic)
print(bear)
```

```
lion
['zebra', 'monkey', 'tiger']
bear
```

Join Tuples

To join two or more tuples you can use the + operator.

In [ ]:

```
t1=("S","K","Y")
t2=(87,39)
t3= t1 + t2
print(t3)    #creating a new tuple
print(t1+t2)    #printing both tuples together
```

```
('S', 'K', 'Y', 87, 39)
('S', 'K', 'Y', 87, 39)
```

If you want to multiply the content of a tuple a given number of times, you can use the * operator.

In [ ]:

```
t4=t1*2
print(t4)    #creating a new tuple
print(t2*2)         #just printing the specified tuple twice
```

```
('S', 'K', 'Y', 'S', 'K', 'Y')
(87, 39, 87, 39)
```

Sets

- Sets are used to store multiple items in a single variable.
- Set is one of 4 built-in data types in Python used to store collections of data, the other 3 are List, Tuple, and Dictionary, all with different qualities and usage.
- A set is a collection which is unordered and unindexed.
- Set items are unchangeable, but you can remove items and add new items.
- Sets are written with curly brackets.
- Once a set is created, you cannot change its items, but you can remove items and add new items.
- Sets cannot have two items with the same value.
- To determine how many items a set has, use the len() function.
- Set items can be of any data type.
- A set can contain different data types.
- It is also possible to use the set() constructor to make a set.

  example :animals={'lion','zebra','monkey','tiger','bear'}

  - Once a set is created, you cannot change its items, but you can add new items.

In [ ]:

```
animals={'lion','zebra','monkey','tiger','bear'}
print(type(animals))

print(len(animals))

my_set={"abc",False,39,87.9}
print(my_set)

thisset = set(("apple", "banana", "cherry")) # note the double round-brackets
print(thisset)
```

```
<class 'set'>
5
{False, 'abc', 87.9, 39}
{'banana', 'apple', 'cherry'}
```

Access items of a set

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the 'in' keyword.

In [ ]:

```
my_set={"abc",False,39,87.9}
for x in my_set:
  print(x)
```

```
False
abc
87.9
39
```

Add items in set

Once a set is created, you cannot change its items, but you can add new items.

To add one item to a set use the add() method.

In [ ]:

```
animals={'lion','zebra','monkey','tiger','bear'}
animals.add("giraffe")
print(animals)
```

```
{'bear', 'zebra', 'tiger', 'lion', 'monkey', 'giraffe'}
```

In [ ]:

```
#we can add two different sets as well using update() method
fruits={"apple","mango","banana","grapes","watermelon"}
animals.update(fruits)
print(animals)
```

```
{'bear', 'apple', 'zebra', 'tiger', 'watermelon', 'lion', 'banana', 'monkey', 'grapes', 'mango',
'giraffe'}
```

Remove items from set

To remove an item in a set, use the remove(), or the discard() method.

If the item to remove does not exist, remove() will raise an error.

In [ ]:

```
#remove giraffe from animals
animals.discard("giraffe")
print(animals)
```

```
{'bear', 'apple', 'zebra', 'tiger', 'watermelon', 'lion', 'banana', 'monkey', 'grapes', 'mango'}
```

If the item to remove does not exist, discard() will NOT raise an error.

You can also use the pop() method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

Sets are unordered, so when using the pop() method, you do not know which item that gets removed.

In [ ]:

```
x=animals.pop()
print(x)
```

```
bear
```

The clear() method empties the set.

In [ ]:

```
fruits.clear()
print(fruits)
```

```
set()
```

The del keyword will delete the set completely.

In [ ]:

```
del fruits
print(fruits)    #this will raise an error because the set no longer exists
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-267-045a5c3efd30> in <module>
      1 del fruits
----> 2 print(fruits)    #this will raise an error because the set no longer exists

NameError: name 'fruits' is not defined
```

Join Sets

There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from both sets, or the update() method that inserts all the items from one set into another.

The **union()** method returns a new set with all items from both sets.

The **update()** method inserts the items from one set to another.

Both union() and update() will exclude any duplicate items.

In [ ]:

```
set1 = {"S", "K" , "Y"}
set2 = {87, 39}

set3 = set1.union(set2)
print(set3)

set1.update(set2)
print(set1)
```

```
{'S', 87, 39, 'K', 'Y'}
{'S', 87, 39, 'K', 'Y'}
```

Dictionary

- Dictionaries are used to store data values in **key:value** pairs.
- A dictionary is a collection which is ordered*, changeable and do not allow duplicates.
- Dictionaries are written with curly brackets, and have keys and values.

example:

my_dict={"good":"bad","sharp":"blunt","new":"old","hard":"soft"}

- When we say that dictionaries are ordered, it means that the items have a defined order, and that order will not change.
- Unordered means that the items does not have a defined order, you cannot refer to an item by using an index.
- Dictionaries are changeable, meaning that we can change, add or remove items after the dictionary has been created.
- Dictionaries cannot have two items with the same key.
- To determine how many items a dictionary has, use the len() function.
- The values in dictionary items can be of any data type.
- From Python's perspective, dictionaries are defined as objects with the data type 'dict'.

In [ ]:

```
my_dict={"good":"bad","sharp":"blunt","new":"old","hard":"soft"}
print(my_dict)
```

```
{'good': 'bad', 'sharp': 'blunt', 'new': 'old', 'hard': 'soft'}
```

In [ ]:

```
my_dict={"good":"bad","sharp":"blunt","new":"old","hard":"soft","good":"wicked"}
print(my_dict)

print(len(my_dict))
```

```
{'good': 'wicked', 'sharp': 'blunt', 'new': 'old', 'hard': 'soft'}
4
```

Accessing items of Dictionary

- You can access the items of a dictionary by referring to its key name, inside square brackets.
- There is also a method called get() that will give you the same result.

In [ ]:

```
print(my_dict['new'])

x = my_dict.get("good")
print(x)


y = my_dict.keys()
print(y)
```

```
old
wicked
dict_keys(['good', 'sharp', 'new', 'hard'])
```

Change values

You can change the value of a specific item by referring to its key name.

In [ ]:

```
thisdict={"name":"abc","class":00,"roll no":87,"year":2022}
thisdict["year"] = 2018
print(thisdict)
```

```
{'name': 'abc', 'class': 0, 'roll no': 87, 'year': 2018}
```

The **update()** method will update the dictionary with the items from the given argument.

The argument must be a dictionary, or an iterable object with key:value pairs.

In [ ]:

```
thisdict.update({"year": 2020})
print(thisdict)
```

```
{'name': 'abc', 'class': 0, 'roll no': 87, 'year': 2020}
```

Adding items

Adding an item to the dictionary is done by using a new index key and assigning a value to it.

In [ ]:

```
thisdict["color"] = "red"
print(thisdict)
```

```
{'name': 'abc', 'class': 0, 'roll no': 87, 'year': 2020, 'color': 'red'}
```

Removing items

The **pop()** method removes the item with the specified key name.

In [ ]:

```
thisdict.pop("name")
print(thisdict)
```

```
{'class': 0, 'roll no': 87, 'year': 2020, 'color': 'red'}
```

The **popitem()** method removes the last inserted item (in versions before 3.7, a random item is removed instead.

In [ ]:

```
thisdict.popitem()
print(thisdict)
```

```
{'class': 0, 'roll no': 87, 'year': 2020}
```

The **clear()** method empties the dictionary.

In [ ]:

```
thisdict.clear()
print(thisdict)
```

```
{}
```

The **del** keyword removes the item with the specified key name.

In [ ]:

```
del thisdict
print(thisdict)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-278-ad193cd02359> in <module>
      1 del thisdict
----> 2 print(thisdict)

NameError: name 'thisdict' is not defined
```

Loop through Dictionary

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

In [ ]:

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

```
good
sharp
new
hard
```

Print all values in the dictionary, one by one.

In [ ]:

```
for x in my_dict:
  print(my_dict[x])
```

```
wicked
blunt
old
soft
```

You can also use the values() method to return values of a dictionary.

In [ ]:

```
for x in my_dict.values():
  print(x)
```

```
wicked
blunt
old
soft
```

You can use the keys() method to return the keys of a dictionary.

In [ ]:

```
for x in my_dict.keys():
  print(x)
```

```
good
sharp
new
hard
```

Loop through both keys and values, by using the items() method.

In [ ]:

```
for x, y in my_dict.items():
  print(x, y)
```

```
good wicked
sharp blunt
new old
hard soft
```

Copy a dictionary

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

In [ ]:

```
thisdict = my_dict.copy()
print(thisdict)
```

```
{'good': 'wicked', 'sharp': 'blunt', 'new': 'old', 'hard': 'soft'}
```

Another way to make a copy is to use the built-in function dict().

In [ ]:

```
thisdict = dict(my_dict)
print(thisdict)
```

```
{'good': 'wicked', 'sharp': 'blunt', 'new': 'old', 'hard': 'soft'}
```

Nested Dictionaries

A dictionary can contain dictionaries, this is called nested dictionaries.

In [ ]:

```
myfamily={
    "child1" : {
    "name" : "ABC",
    "year" : 2001
  },
  "child2" : {
    "name" : "DEF",
    "year" : 2003
  },
  "child3" : {
    "name" : "GHI",
    "year" : 2008
  }
}
```

Or, if you want to add three dictionaries into a new dictionary.

In [ ]:

```
child1 = {
    "name" : "ABC",
    "year" : 2001
  }
child2 = {
    "name" : "DEF",
    "year" : 2003
  }
child3 = {
    "name" : "GHI",
    "year" : 2008
  }

myfamily = {
  "child1" : child1,
  "child2" : child2,
  "child3" : child3
}

print(myfamily)
```

```
{'child1': {'name': 'ABC', 'year': 2001}, 'child2': {'name': 'DEF', 'year': 2003}, 'child3': {'name': 'G
HI', 'year': 2008}}
```

Exception Handling in Python

Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

In [ ]:

```
# initialize the amount variable
marks = 10000

# perform division with 0
a = marks / 0
print(a)
```

```
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-369-757d9509a65c> in <module>
      3
      4 # perform division with 0
----> 5 a = marks / 0
      6 print(a)

ZeroDivisionError: division by zero
```

In the above example raised the ZeroDivisionError as we are trying to divide a number by 0.

Try and Except Statement – Catching Exceptions

Try and except statements are used to catch and handle exceptions in Python. Statements that can raise exceptions are kept inside the try clause and the statements that handle the exception are written inside except clause.

Example: Let us try to access the array element whose index is out of bound and handle the corresponding exception.

In [ ]:

```
# Python program to handle simple runtime error
#Python 3

a = [87,39,108]
try:
    print ("Second element = %d" %(a[1]))

    # Throws error since there are only 3 elements in array
    print ("Fourth element = %d" %(a[3]))

except:
    print ("An error occurred")
```

```
Second element = 39
An error occurred
```

In the above example, the statements that can cause the error are placed inside the try statement (second print statement in our case). The second print statement tries to access the fourth element of the list which is not there and this throws an exception. This exception is then caught by the except statement.

Catching Specific Exception

A try statement can have more than one except clause, to specify handlers for different exceptions. Please note that at most one handler will be executed. For example, we can add IndexError in the above code. The general syntax for adding specific exceptions are –

In [ ]:

```
# Program to handle multiple errors with one
# except statement
# Python 3

def fun(a):
 if a < 4:

  # throws ZeroDivisionError for a = 3
  b = a/(a-3)

 # throws NameError if a >= 4
 print("Value of b = ", b)

try:
 fun(3)
 fun(5)

# note that braces () are necessary here for
# multiple exceptions
except ZeroDivisionError:
 print("ZeroDivisionError Occurred and Handled")
except NameError:
 print("NameError Occurred and Handled")
```

```
ZeroDivisionError Occurred and Handled
```

List of Standard Exceptions –

1. Exception

Base class for all exceptions

2. StopIteration

2. StopIteration

Raised when the next() method of an iterator does not point to any object.

3. SystemExit

Raised by the sys.exit() function.

4. StandardError

Base class for all built-in exceptions except StopIteration and SystemExit.

5 . ArithmeticError

Base class for all errors that occur for numeric calculation.

6 . OverflowError

Raised when a calculation exceeds maximum limit for a numeric type.

7 . FloatingPointError

Raised when a floating point calculation fails.

8 . ZeroDivisionError

Raised when division or modulo by zero takes place for all numeric types.

9 . AssertionError

Raised in case of failure of the Assert statement.

10 . AttributeError

Raised in case of failure of attribute reference or assignment.

11 . EOFError

Raised when there is no input from either the raw_input() or input() function and the end of file is reached.

12 . ImportError

Raised when an import statement fails.

13 . KeyboardInterrupt

Raised when the user interrupts program execution, usually by pressing Ctrl+c.

14 . LookupError

Base class for all lookup errors.

15 . IndexError

Raised when an index is not found in a sequence.

16 . KeyError

Raised when the specified key is not found in the dictionary.

17 . NameError

Raised when an identifier is not found in the local or global namespace.

18 . UnboundLocalError

Raised when trying to access a local variable in a function or method but no value has been assigned to it.

19 . EnvironmentError

Base class for all exceptions that occur outside the Python environment.

20 . IOError

Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.

21 . IOError

Raised for operating system-related errors.

22 . SyntaxError

Raised when there is an error in Python syntax.

23 . IndentationError

Raised when indentation is not specified properly.

24 . SystemError

Raised when the interpreter finds an internal problem, but when this error is encountered the Python interpreter does not exit.

25 . SystemExit

Raised when Python interpreter is quit by using the sys.exit() function. If not handled in the code, causes the interpreter to exit.

26 . TypeError

Raised when an operation or function is attempted that is invalid for the specified data type.

27 . ValueError

Raised when the built-in function for a data type has the valid type of arguments, but the arguments have invalid values specified.

28 . RuntimeError

Raised when a generated error does not fall into any category.

29 . NotImplementedError

Raised when an abstract method that needs to be implemented in an inherited class is not actually implemented.

The except Clause with No Exceptions

You can also use the except statement with no exceptions defined as follows –

```
#
try:
   You do your operations here;
   ......................
except:
   If there is any exception, then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

This kind of a**try-except** statement catches all the exceptions that occur. Using this kind of try-except statement is not considered a good programming practice though, because it catches all exceptions but does not make the programmer identify the root cause of the problem that may occur.

The except Clause with Multiple Exceptions

You can also use the same except statement to handle multiple exceptions as follows –

```
try:
   You do your operations here;
   ......................
except(Exception1[, Exception2[,...ExceptionN]]):
   If there is any exception from the given exception list,
   then execute this block.
   ......................
else:
   If there is no exception then execute this block.
```

The try-finally Clause

You can use a finally: block along with a try: block. The finally block is a place to put any code that must execute, whether the try-block raised an exception or not. The syntax of the try-finally statement is this

```
try:
   You do your operations here;
   ......................
   Due to any exception, this may be skipped.
finally:
   This would always be executed.
   ......................
```

In [ ]:

```
#!/usr/bin/python
```

```
try:
   fh = open("testfile", "w")
   try:
      fh.write("This is my test file for exception handling!!")
   finally:
      print ("Going to close the file")
      fh.close()
except IOError:
   print ("Error: can\'t find file or read data")
```

```
Going to close the file
```

When an exception is thrown in the try block, the execution immediately passes to the finally block. After all the statements in the finally block are executed, the exception is raised again and is handled in the except statements if present in the next higher layer of the try-except statement.

Argument of an Exception

An exception can have an argument, which is a value that gives additional information about the problem. The contents of the argument vary by exception. You capture an exception's argument by supplying a variable in the except clause as follows –

```
   try:
      You do your operations here;
      ......................
   except ExceptionType, Argument:
      You can print value of Argument here...
```

If you write the code to handle a single exception, you can have a variable follow the name of the exception in the except statement. If you are trapping multiple exceptions, you can have a variable follow the tuple of the exception.

This variable receives the value of the exception mostly containing the cause of the exception. The variable can receive a single value or multiple values in the form of a tuple. This tuple usually contains the error string, the error number, and an error location.

In [ ]:

```
def temp_convert(var):
   try:
      return int(var)
   except ValueError:
      print ("The argument does not contain numbers\n")

# Call above function here.
temp_convert("xyz");
```

```
The argument does not contain numbers
```

Raising an Exceptions

You can raise exceptions in several ways by using the raise statement. The general syntax for the raise statement is as follows.

Syntax-

```
raise [Exception [, args [, traceback]]] '
```

Here, Exception is the type of exception (for example, NameError) and argument is a value for the exception argument. The argument is optional; if not supplied, the exception argument is None.

The final argument, traceback, is also optional (and rarely used in practice), and if present, is the traceback object used for the exception.

Example-

An exception can be a string, a class or an object. Most of the exceptions that the Python core raises are classes, with an argument that is an instance of the class. Defining new exceptions is quite easy and can be done as follows –

```
def functionName( level ):
   if level < 1:
      raise "Invalid level!", level
      # The code below to this would not be executed
      # if we raise the exception
```

Note: In order to catch an exception, an "except" clause must refer to the same exception thrown either class object or simple string. For example, to capture above exception, we must write the except clause as follows –

```
try:
   Business Logic here...
except "Invalid level!":
   Exception handling here...
else:
   Rest of the code here...
```

File Handling in Python

To store data temporarily and permanently, we use files. A file is the collection of data stored on a disk in one unit identified by filename. File handling is an important part of any web application.

The concept of file handling has stretched over various other languages, but the implementation is either complicated or lengthy, but like other concepts of Python, this concept here is also easy and short. Python treats files differently as text or binary and this is important. Each line of code includes a sequence of characters and they form a text file. Each line of a file is terminated with a special character, called the EOL or End of Line characters like comma {,} or newline character. It ends the current line and tells the interpreter a new one has begun.

Types of Files

- **Text File**: Text file usually we use to store character data. For example, test.txt
- **Binary File** : The binary files are used to store binary data such as images, video files, audio files, etc.

File Path

- *Absolute path*: which always begins with the root folder.
- *Relative path*: which is relative to the program's current working directory.

File Access Modes

- **r**

It opens an existing file to read-only mode. The file pointer exists at the beginning.

- **rb**

It opens the file to read-only in binary format. The file pointer exists at the beginning.

- **r+**

It opens the file to read and write both. The file pointer exists at the beginning.

- **rb+**

It opens the file to read and write both in binary format. The file pointer exists at the beginning of the file.

- **w**

It opens the file to write only. It overwrites the file if previously exists or creates a new one if no file exists with the same name.

- **wb**

It opens the file to write only in binary format. It overwrites the file if it exists previously or creates a new one if no file exists.

- **w+**

It opens the file to write and read data. It will override existing data. wb+ It opens the file to write and read both in binary format.

- **a**

It opens the file in the append mode. It will not override existing data. It creates a new file if no file exists with the same name.

- **ab**

It opens the file in the append mode in binary format.

- **a+**

It opens a file to append and read both.

- **ab+**

It opens a file to append and read both in binary format.

Python File Methods

Python has various method available that we can use with the file object. The following table shows file method.

- **read()** Returns the file content.
- **readline()** Read single line
- **readlines()** Read file into a list
- **truncate(size)** Resizes the file to a specified size.
- **write()** Writes the specified string to the file.
- **writelines()** Writes a list of strings to the file.
- **close()** Closes the opened file.
- **seek()** Set file pointer position in a file
- **tell()** Returns the current file location.
- **fileno()** Returns a number that represents the stream, from the operating system's perspective.
- **flush()** Flushes the internal buffer.

Working of open() function

Before performing any operation on the file like reading or writing, first, we have to open that file. For this, we should use Python's inbuilt function **open()** but at the time of opening, we have to specify the mode, which represents the purpose of the opening file.

```
f = open(filename, mode)
```

where mode is the file access modes.

Working of read() mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use file.read().

Working of rename()

In Python, the os module provides the functions for file processing operations such as renaming, deleting the file, etc. The os module enables interaction with the operating system.

The os module provides rename() method to rename the specified file name to the new name. The syntax of rename() method is shown below.

Example

```
import os

# Absolute path of a file
old_name = r"E:\demos\files\reports\details.txt"
new_name = r"E:\demos\files\reports\new_details.txt"

# Renaming the file
os.rename(old_name, new_name)
```

Delete Files

In Python, the os module provides the remove() function to remove or delete file path.

Example:

```
import os

# remove file with absolute path
os.remove(r"E:\demos\files\sales_2.txt")
```

Python Libraries

1. ***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.
- NumPy is a Python library and is written partially in Python, but most of the parts that require fast computation are written in C or C++.
- The 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.

Getting started with NumPy

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

Install it using this command:

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

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

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

```
import numpy #Now NumPy is imported and ready to use.
```

In [ ]:

```
#Example
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.

In [ ]:

```
#Example
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.

In [ ]:

```
import numpy as np

print(np.__version__)
```

```
1.21.6
```

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.

In [ ]:

```
#Example
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'>
```

Dimensions in Arrays

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

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

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.

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**.

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**.

In [ ]:

```
from numpy.core.arrayprint import array2string
import numpy as np

arr0 = np.array(87)         #0-D array
print(arr0)

arr1 = np.array([1, 2, 3, 4, 5])         #1-D array
print(arr1)

arr2 = np.array([[1, 2, 3], [4, 5, 6]])         #2-D array
print(arr2)

arr3 = np.array([[[1, 2, 3], [4, 5, 6]], [[1, 2, 3], [4, 5, 6]]])         #3-D array
```

```
print(arr3)
```

```
87
[1 2 3 4 5]
[[1 2 3]
 [4 5 6]]
[[[1 2 3]
  [4 5 6]]

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

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

In [ ]:

```
print(arr0.ndim)
print(arr1.ndim)
print(arr2.ndim)
print(arr3.ndim)
```

```
0
1
2
3
```

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.

In [ ]:

```
import numpy as np

arr = np.array([110, 12, 3, 14])          #1D-array

print(arr[0])
print(arr[2] + arr[3])

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

print('2nd element on 1st row: ', arr[0, 1])
print('5th element on 2nd row: ', arr[1, 4])

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

print(arr[0, 1, 2])

print('Last element from 2nd dim: ', arr[1, -1])          #negative indexing
```

```
110
17
2nd element on 1st row:  2
5th element on 2nd row:  10
6
Last element from 2nd dim:  [10 11 12]
```

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]`.

In [ ]:

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

print(arr[1:5])
print(arr[4:])
print(arr[-3:-1])     #negative indexing
print(arr[1:5:2])   #with step
print(arr[::2])

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

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

```
[2 3 4 5]
[5 6 7]
[5 6]
[2 4]
[1 3 5 7]
[7 8 9]
[3 8]
[[2 3 4]
 [7 8 9]]
```

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.

In [ ]:

```
#Get the data type of an array object
arr = np.array([1, 2, 3, 4])
print(arr.dtype)
```

```
int64
```

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.

In [ ]:

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

print(lis)
print(lis.dtype)
```

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

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.

In [ ]:

```
l1 = np.array([1.1, 2.1, 3.1])

newl1 = l1.astype('i')

print(newl1)
print(newl1.dtype)
```

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

Copy and View in NumPy

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.

In [ ]:

```
x = l1.copy()
l1[0] = 42

print(l1)
print(x)
```

```
[42.   2.1  3.1]
[1.1 2.1 3.1]
```

In [ ]:

```
y = l1.view()
l1[0] = 42

print(l1)
print(y)
```

```
[42.   2.1  3.1]
[42.   2.1  3.1]
```

Shape of Array

The shape of an array is the number of elements in each dimension.NumPy arrays have an attribute called shape that returns a tuple with each index having the number of corresponding elements.

In [ ]:

```
l2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])
print(l2.shape)
```

```
(2, 4)
```

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.

In [ ]:

```
l3 = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
new13 = l3.reshape(3, 4)         #from 1d to 2d
print(new13)

print("----------------")

l4 = l3.reshape(2, 2, 3)
print(l4)
```

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

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

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.

In [ ]:

```
#using l1 from above
for i in l1:
  print(i)
```

```
42.0
2.1
3.1
```

In [ ]:

```
#using l2 from above example
for i in l2:
  print(i)

#with another way
for i in (l2):
  for j in i:
    print(j)
```

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

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.

In [ ]:

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

for x in np.nditer(array):
  print(x)
```

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

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.

In [ ]:

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

for idx, x in np.ndenumerate(arr1):   #here idx is giving the index of the array
  print(idx, x)
```

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

In [ ]:

```
arr2 = np.array([[1, 2, 3, 4], [5, 6, 7, 8]])

for idx, x in np.ndenumerate(arr2):   #here idx is giving the index of the array
  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
```

Joining NumPy Arrays

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

- 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.

In [ ]:

```
arr3 = np.array([1, 2, 3])
arr4 = np.array([4, 5, 6])
newarr = np.concatenate((arr3, arr4))
print(newarr)
```

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

- 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.

In [ ]:

```
newarr2 = np.stack((arr3, arr4), axis=1)
print(newarr2)
```

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

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

In [ ]:

```
newarr3 = np.hstack((arr3, arr4))
print(newarr3)
```

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

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

In [ ]:

```
newarr4 = np.vstack((arr3, arr4))
print(newarr4)
```

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

- **dstack()** to stack along height, which is the same as depth.

In [ ]:

```
newarr5 = np.dstack((arr3, arr4))
print(newarr5)
```

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

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.

In [ ]:

```
arr6 = np.array([39, 82, 40, 12, 25, 0])
print(np.sort(arr6))
```

```
[ 0 12 25 39 40 82]
```

In [ ]:

```
arr7=np.array(["Alex","Advik","Ayaan","Andrew","Asher","Ariel"])
print(np.sort(arr7))
```

```
['Advik' 'Alex' 'Andrew' 'Ariel' 'Asher' 'Ayaan']
```

In [ ]:

```
arr8 = np.array([[31, 22, 84], [95, 10, 12]])
print(np.sort(arr8))
```

```
[[22 31 84]
 [10 12 95]]
```

What is a Random Number?

Random number does NOT mean a different number every time. Random means something that can not be predicted logically.

Generate Random Number

NumPy offers the random module to work with random numbers.

In [ ]:

```
from numpy import random
a = random.randint(450)
print(a)
```

```
228
```

The random module's **rand()** method returns a random float between 0 and 1.

In [ ]:

```
b = random.rand()
print(b)
```

```
0.904184270818259
```

The **randint()** method takes a size parameter where you can specify the shape of an array.

In [ ]:

```
c=random.randint(1200, size=(10))
print(c)
```

```
[1043  490 1092  668  927 1042  111   57 1129  248]
```

In [ ]:

```
d = random.randint(780, size=(4, 6))
print(d)
```

```
[[330 684 511 429 515 149]
 [449 164 323 467 506  52]
 [668 625 768 526 700 750]
 [435 266 597 406 200 357]]
```

The **rand()** method also allows you to specify the shape of the array.

In [ ]:

```
# Generate a 1-D array containing 5 random floats
x = random.rand(5)
print(x)
```

```
[0.05578433 0.36038776 0.48380388 0.46783925 0.77475563]
```

arange() function

The arange() function is used to get evenly spaced values within a given interval. Values are generated within the half-open interval [start, stop]. For integer arguments the function is equivalent to the Python built-in range function, but returns an ndarray rather than a list. When using a non-integer step, such as 0.1, the results will often not be consistent. It is better to use linspace for these cases.

Syntax:

```
numpy.arange([start, ]stop, [step, ]dtype=None)
```

Return value:

arange : ndarray - Array of evenly spaced values.

For floating point arguments, the length of the result is ceil((stop - start)/step). Because of floating point overflow, this rule may result in the last element of out being greater than stop.

In [ ]:

```
import numpy as np
print("\n", np.arange(4).reshape(2, 2), "\n")
print("\n", np.arange(4, 10), "\n")
print("\n", np.arange(4, 20, 3), "\n")

# Printing all numbers from 1 to
# 2 in steps of 0.1
print(np.arange(1, 2, 0.1))
```

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


 [4 5 6 7 8 9]


 [ 4  7 10 13 16 19]

[1.  1.1 1.2 1.3 1.4 1.5 1.6 1.7 1.8 1.9]
```

We can also find product of all the elements of an array using prod() function.

In [ ]:

```
example=[12,5,0,45]
product=np.prod(example)
print(product)
```

```
0
```

In [ ]:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

x = np.prod([arr1, arr2])    #finding product of two arrays

print(x)
```

```
40320
```

In [ ]:

```
arr1 = np.array([1, 2, 3, 4])
arr2 = np.array([5, 6, 7, 8])

newarr = np.prod([arr1, arr2], axis=1)            #axis=1 means y axis

print(newarr)
```

```
[  24 1680]
```

Cummulative Product

Cummulative product means taking the product partially.

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

Perfom partial sum with the cumprod() function.

In [ ]:

```
arr=[1,4,7,8,9,6]
newarr = np.cumprod(arr)
print(newarr)
```

```
[    1     4    28   224  2016 12096]
```

numpy.multiply()

numpy.multiply() function is used when we want to compute the multiplication of two array. It returns the product of arr1 and arr2, element-wise.

In [ ]:

```
# Python program explaining
# numpy.multiply() function

import numpy as np

in_arr1 = np.array([[2, -7, 5], [-6, 2, 0]])
in_arr2 = np.array([[0, -7, 8], [5, -2, 9]])

print ("1st Input array : ", in_arr1)
print ("2nd Input array : ", in_arr2)


out_arr = np.multiply(in_arr1, in_arr2)
print ("Resultant output array: ", out_arr)
```

```
1st Input array :  [[ 2 -7  5]
 [-6  2  0]]
2nd Input array :  [[ 0 -7  8]
 [ 5 -2  9]]
Resultant output array:  [[  0  49  40]
 [-30  -4   0]]
```

Cross Product

Cross product of two vectors yield a vector that is perpendicular to the plane formed by the input vectors and its magnitude is proportional to the area spanned by the parallelogram formed by these input vectors.

We can compute cross product using Numpy**cross()** function.

In [ ]:

```
A = np.array([2, 3])
B = np.array([1, 7])

#compute cross product
output = np.cross(A, B)

print(output)
```

```
11
```

In [ ]:

```
C = np.array([2, 7, 4])
D = np.array([3, 9, 8])

#compute cross product
output = np.cross(C, D)

print(output)
```

```
[20 -4 -3]
```

A **discrete difference** means subtracting two successive elements.

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

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

In [ ]:

```
X = np.array([10, 15, 25, 5])

newX = np.diff(X)

print(newX)
```

```
[  5  10 -20]
```

1. **Pandas**

Pandas is an open-source library that is made mainly for working with relational or labeled data both easily and intuitively. It provides various data structures and operations for manipulating numerical data and time series. This library is built on top of the NumPy library. Pandas is fast and it has high performance & productivity for users.

Getting started with Pandas

The first step of working in pandas is to ensure whether it is installed in the Python folder or not. If not then we need to install it in our system using pip command. Type cmd command in the search box and locate the folder using cd command where python-pip file has been installed. After locating it, type the command:

```
pip install pandas
```

After the pandas have been installed into the system, you need to import the library. This module is generally imported as:

```
import pandas as pd
```

Here, pd is referred to as an alias to the Pandas.

Pandas generally provide two data structures for manipulating data, They are:

- **Series:**

Pandas Series is a one-dimensional labeled array capable of holding data of any type (integer, string, float, python objects, etc.). The axis labels are collectively called indexes. Pandas Series is nothing but a column in an excel sheet. Labels need not be unique but must be a hashable type. The object supports both integer and label-based indexing and provides a host of methods for performing operations involving the index.

Creating a Series

In the real world, a Pandas Series will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, an Excel file. Pandas Series can be created from the lists, dictionary, and from a scalar value etc.

- **DataFrame:**

Pandas DataFrame is a two-dimensional size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). A Data frame is a two-dimensional data structure, i.e., data is aligned in a tabular fashion in rows and columns. Pandas DataFrame consists of three principal components, the data, rows, and columns.

Creating a DataFrame:

In the real world, a Pandas DataFrame will be created by loading the datasets from existing storage, storage can be SQL Database, CSV file, an Excel file. Pandas DataFrame can be created from the lists, dictionary, and from a list of dictionaries, etc.

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.

Load the CSV into a DataFrame:

```
import pandas as pd

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

print(df.to_string())
```

In [ ]:

```
#creating a dataframe using dictionary
import pandas as pd
import numpy as np
df=pd.DataFrame({'Name':['Alex','Benjamin','Divyanshi','Asif','Yashi','Archie','Chris'],
      'Computer Science':[12,58,78,96,32,np.nan,np.nan],
      'Maths':[87,45, np.nan,25,75,20,3],
      'Physics':[np.nan,np.nan ,67,90,11,8,57]},index=[1,2,3,4,5,6,7])
df
```

Out[ ]:

	Name	Computer Science	Maths	Physics
1	Alex	12.0	87.0	NaN
2	Benjamin	58.0	45.0	NaN
3	Divyanshi	78.0	NaN	67.0
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0
6	Archie	NaN	20.0	8.0
7	Chris	NaN	3.0	57.0

Analyzing data using Pandas

- 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.
- 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.

In [ ]:

```
df.head()
```

```
df.head()
```

Out[ ]:

	Name	Computer Science	Maths	Physics
1	Alex	12.0	87.0	NaN
2	Benjamin	58.0	45.0	NaN
3	Divyanshi	78.0	NaN	67.0
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0

In [ ]:

```
df.tail()
```

Out[ ]:

	Name	Computer Science	Maths	Physics
3	Divyanshi	78.0	NaN	67.0
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0
6	Archie	NaN	20.0	8.0
7	Chris	NaN	3.0	57.0

Information About the Data

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

In [ ]:

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 7 entries, 1 to 7
Data columns (total 4 columns):
 #   Column            Non-Null Count  Dtype  
---  ------            --------------  -----  
 0   Name              7 non-null      object 
 1   Computer Science  5 non-null      float64
 2   Maths             6 non-null      float64
 3   Physics           5 non-null      float64
dtypes: float64(3), object(1)
memory usage: 280.0+ bytes
```

- **df.shape** gives you the shape of the complete dataset ,i.e., number of rows and columns respectively. While,**df.shape[0]** tells you the no. of rows and **df.shape[1]** gives you the no. of columns .

In [ ]:

```
df.shape
```

Out[ ]:

```
(7, 4)
```

In [ ]:

```
df.shape[0]
```

Out[ ]:

```
7
```

In [ ]:

```
df.shape[1]
```

Out[ ]:

```
4
```

- **columns** gives us the list of columns in the dataframe

In [ ]:

```
df.columns
```

Out[ ]:

```
Index(['Name', 'Computer Science', 'Maths', 'Physics'], dtype='object')
```

- **describe( )** method gives the basic statistical summaries of all numerical attributes in the dataframe.

In [ ]:

```
df.describe()
```

Out[ ]:

	Computer Science	Maths	Physics
count	5.000000	6.000000	5.000000
mean	55.200000	42.500000	46.600000
std	33.899853	32.910485	35.934663
min	12.000000	3.000000	8.000000
25%	32.000000	21.250000	11.000000
50%	58.000000	35.000000	57.000000
75%	78.000000	67.500000	67.000000
max	96.000000	87.000000	90.000000

Data Cleaning

Data cleaning means fixing bad data in your data set.

Bad data could be:

1. Empty cells
2. Data in wrong format
3. Wrong data
4. Duplicates

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.

By default, the **dropna()** method returns a new DataFrame, and will not change the original. The dropna(inplace = True) will NOT return a new DataFrame, but it will remove all rows containing NULL values from the original DataFrame.

In [ ]:

```
new_df=df.dropna()
new_df
```

Out[ ]:

	Name	Computer Science	Maths	Physics
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0

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

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.

In [ ]:

```
x = df["Computer Science"].mean()
y = df["Computer Science"].median()
z = df["Computer Science"].mode()
print(x)
print(y)
print(z)
```

```
55.2
58.0
0    12.0
1    32.0
2    58.0
3    78.0
4    96.0
dtype: float64
```

In [ ]:

```
df['Computer Science'].fillna(y,inplace=True)
df
```

Out[ ]:

	Name	Computer Science	Maths	Physics
1	Alex	12.0	87.0	NaN
2	Benjamin	58.0	45.0	NaN
3	Divyanshi	78.0	NaN	67.0
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0
6	Archie	58.0	20.0	8.0
7	Chris	58.0	3.0	57.0

In [ ]:

```
a=df['Maths'].mean()
b=df['Maths'].median()
c=df['Maths'].mode()
print(a)
print(b)
print(c)
```

```
42.5
35.0
0     3.0
1    20.0
2    25.0
3    45.0
4    75.0
5    87.0
dtype: float64
```

In [ ]:

```
df['Maths'].fillna(b,inplace=True)
df
```

Out[ ]:

	Name	Computer Science	Maths	Physics
1	Alex	12.0	87.0	NaN
2	Benjamin	58.0	45.0	NaN
3	Divyanshi	78.0	35.0	67.0
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0
6	Archie	58.0	20.0	8.0
7	Chris	58.0	3.0	57.0

In [ ]:

```
m=df['Physics'].mean()
n=df['Physics'].mode()
p=df['Physics'].median()
print(m)
print(n)
print(p)
```

```
46.6
0     8.0
1    11.0
2    57.0
3    67.0
4    90.0
dtype: float64
57.0
```

In [ ]:

```
df['Physics'].fillna(m,inplace=True)
df
```

Out[ ]:

	Name	Computer Science	Maths	Physics
1	Alex	12.0	87.0	46.6
2	Benjamin	58.0	45.0	46.6
3	Divyanshi	78.0	35.0	67.0
4	Asif	96.0	25.0	90.0
5	Yashi	32.0	75.0	11.0
6	Archie	58.0	20.0	8.0
7	Chris	58.0	3.0	57.0

Removing Duplicates

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

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

In [ ]:

```
df.duplicated()
```

Out[ ]:

```
1    False
2    False
3    False
4    False
5    False
6    False
7    False
dtype: bool
```

Removing Duplicates

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

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

In [ ]:

```
df.drop_duplicates(inplace = True)
```

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.

In [ ]:

```
df.corr()
```

Out[ ]:

	Computer Science	Maths	Physics
Computer Science	1.000000	-0.741782	0.611398
Maths	-0.741782	1.000000	-0.303406
Physics	0.611398	-0.303406	1.000000

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

Plotting using Pandas

Pandas plotting methods can be used to plot styles other than the default Line Plot. These methods can be provided as the 'kind' keyword argument to plot(). The available options are:

- Line plot
- Bar plot
- Box plot
- Pie plot
- Scatter plot
- Histogram, etc.

1. *Line Plot*

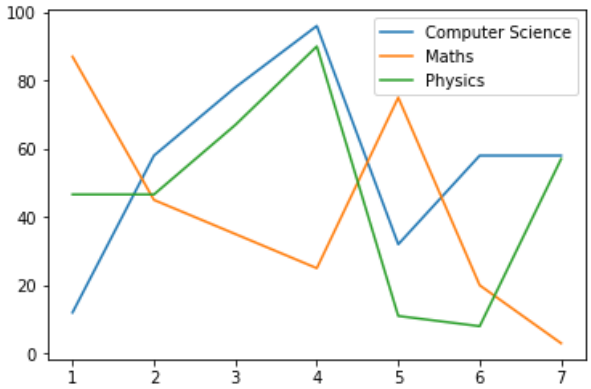
TRhe most basic form of plotting is a line plot.Here, we plot a line using Dataframe's values as coordinates. Below is the implementation to plot a basic line plot using the pandas plotting function.

In [ ]:

```
df.plot()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f29d27e90>
```



1. *Area Plot*

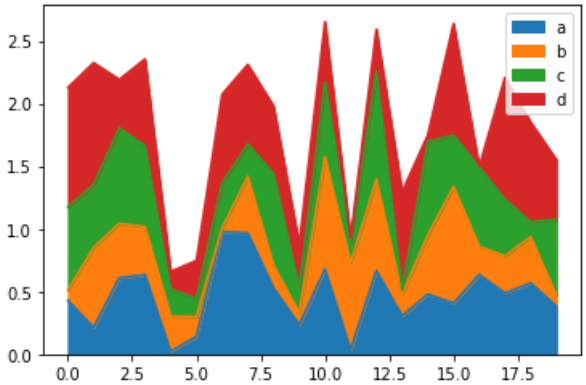
This is an extended version of the line plot. In the area plot, instead of leaving the plot to connected data points, the area under the line is filled with colors. This is helpful in cases when you want to show the proportions of values captured by a particular value. As you can plot multiple variables,this can give you insights about when the variables are overlapping.

In [ ]:

```
Df=pd.DataFrame(np.random.rand(20,4),
                columns=['a','b','c','d'])
Df.plot.area()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f29e8ca10>
```



1. *Bar and Barh Plot*

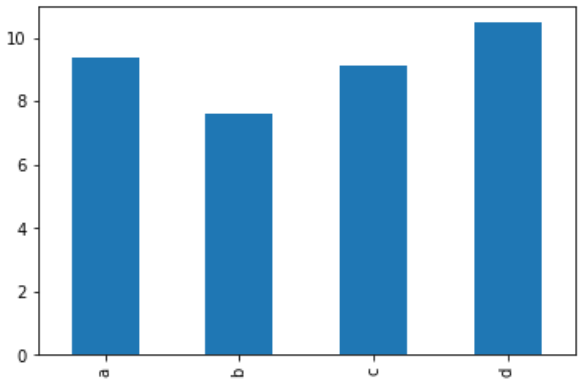
Bar plots are used to represent the values repressed by categorical values. This may include the count of a particular categoryu, any statistic, or other value defined. These are useful in cases when ou want to compare one category with another.

In [ ]:

```
Df.sum().plot.bar()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f29f718d0>
```

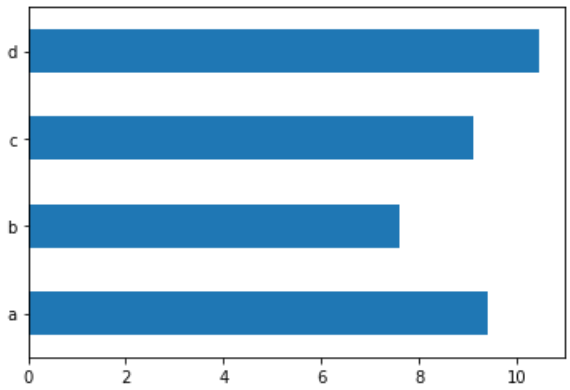


In [ ]:

```
Df.sum().plot.barh()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f30ef5390>
```



1. *Density Plot*

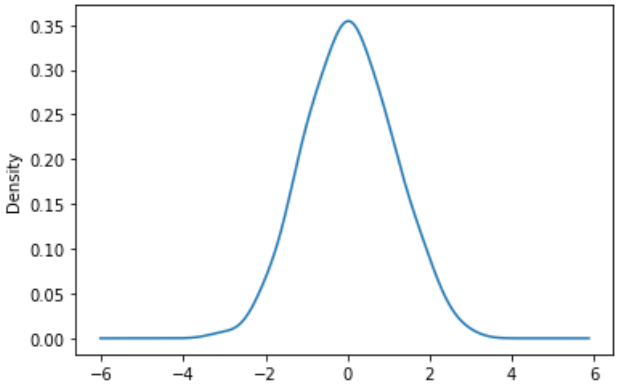
This plot visualizes the **probability density function** of a continuous random variable. This doesn't directly tell the probability of the value taken by a random variable. In fact, for a continuous random variable, the probability of any value is zero and we are only concerned about the probabilities of a range of values. It only gives how many data points may be present around specified values.

In [ ]:

```
DF=pd.Series(np.random.randn(200))
DF.plot.kde()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f2aac7cd0>
```



1. *Histogram Plot*

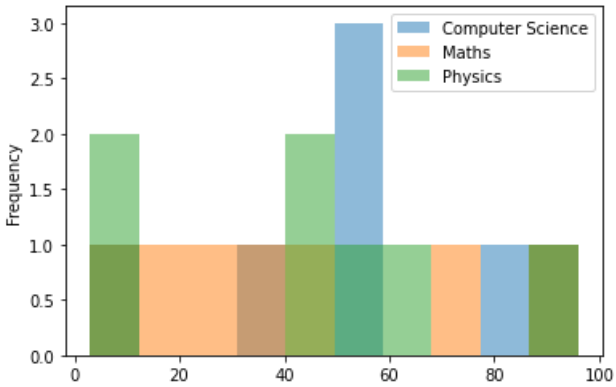
Histograms are used to represent the frequency of numerical variables. These are subversions of bar plots with the changes that in the histogram, we talk about numerical values. There are no categories but the numeric data is divided among small buckets called bins. These bins take in the number of values that fall in the range of the bin. Histograms are also quoted as frequency polygons when the bars are replaced by connecting lines from the midpoint of the bars.

In [ ]:

```
df.plot.hist(alpha=0.5)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f2a225c90>
```



Here, the "alpha" parameter defines the transparency of colors for the histograms.

1. *Scatter Plot*

A scatter plot is also called a scatter chart, scattergram, or scatter plot, XY graph. The scatter diagram graphs numerical data pairs, with one variable on each axis, show their relationship. Now the question comes for everyone: when to use a scatter plot?

Scatter plots are used in either of the following situations.

- When we have paired numerical data.
- When there are multiple values of the dependent variable for a unique value of an independent variable.
- In determining the relationship between variables in some scenarios, such as identifying potential root causes of problems, checking whether two products that appear to be related both occur with the exact cause and so on.

Scatter Plot Uses and Examples Scatter plots instantly report a large volume of data. It is beneficial in the following situations –

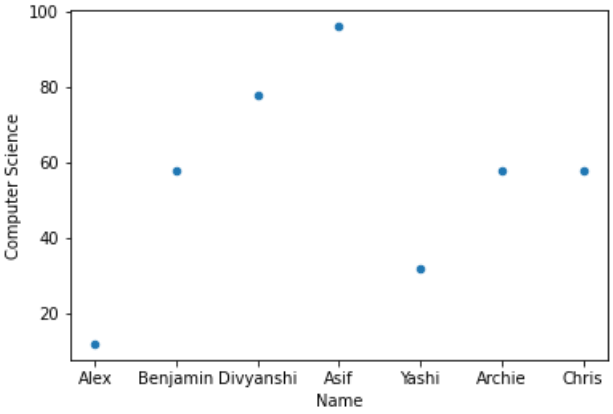
- For a large set of data points given
- Each set comprises a pair of values
- The given data is in numeric form.

In [ ]:

```
df.plot.scatter(x ='Name', y ='Computer Science')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f2a66d2d0>
```



1. *Box Plot*

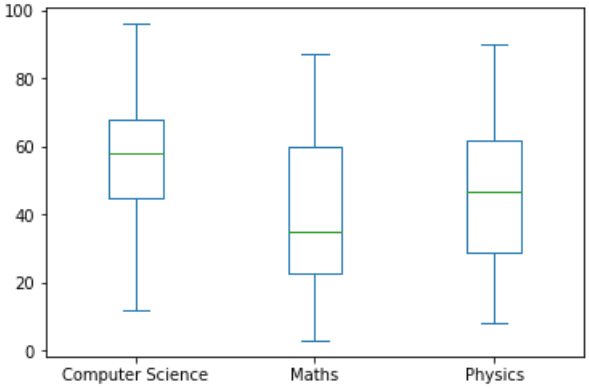
Box plots are very crucial and important plots. These plots help in understanding the overall trend and spread of a feature of a dataset. For numerical data, it represents where 50% of data lies, where the median of the data lies, and it also specifies the boundary conditions in the form of whiskers.

In [ ]:

```
df.plot.box()
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f2a01fcd0>
```



1. *Pie Plot*

Pie plots are used to represent the portion of a value as compared to others in a whole. They represent how much percent of the circular area a value is contributing and fill it with color. They are widely used in every project and at the same time, they are discouraged too!

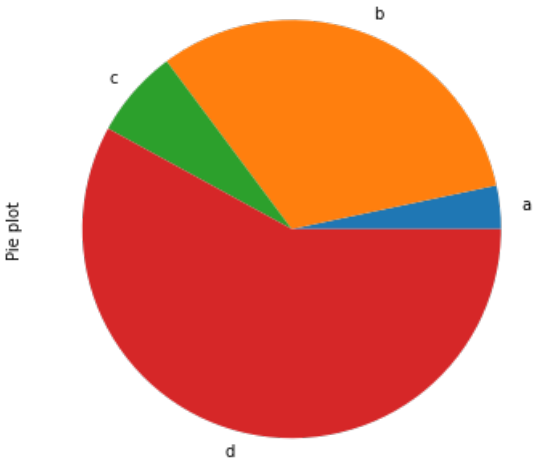
Because to represent a portion numeric value, a large amount of space is allocated and colors are wasted to fill the area. Instead, donut plots are highly encouraged as they need less color quantity plus they can be made as nested pie charts which convey more information than regular pie charts.

In [ ]:

```
series = pd.Series(3 * np.random.rand(4), index=["a", "b", "c", "d"], name="Pie plot")
series.plot.pie(figsize=(6, 6))
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f2a0a7850>
```



1. **Matplotlib**

Matplotlib is a low level graph plotting library in python that serves as a visualization utility.It was created by John D. Hunter. It is open source and we can use it freely. It is mostly written in python, a few segments are written in C, Objective-C and Javascript for Platform compatibility.

Getting started with 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
```

Import Matplotlib

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

In [ ]:

```
import matplotlib
```

Checking Matplotlib Version

The version string is stored under **version** attribute.

In [ ]:

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

```
3.2.2
```

Pyplot

Most of the Matplotlib utilities lies under the pyplot submodule, and are usually imported under the plt alias.

In [ ]:

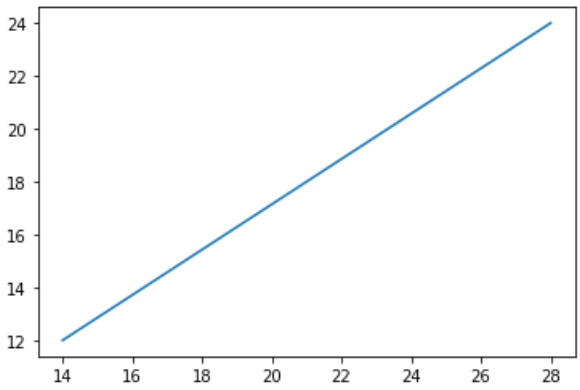
```
import matplotlib.pyplot as plt
```

Plotting using Matplotlib

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 (14, 28) to (12, 24), we have to pass two arrays [14, 28] and [12, 24] to the plot function.

In [ ]:

```
xpoint=np.array([14,28])
ypoint=np.array([12,24])
plt.plot(xpoint,ypoint)
plt.show()
```

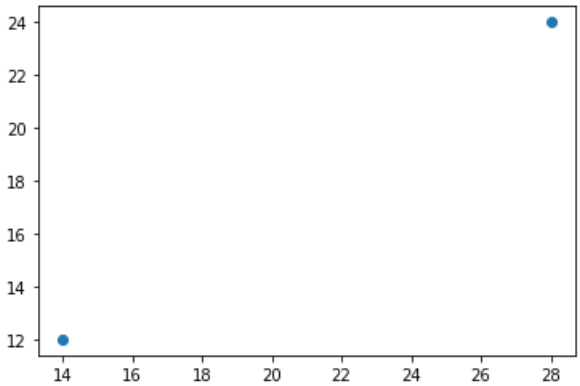


Plotting without Line

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

In [ ]:

```
plt.plot(xpoint,ypoint,'o')
plt.show()
```

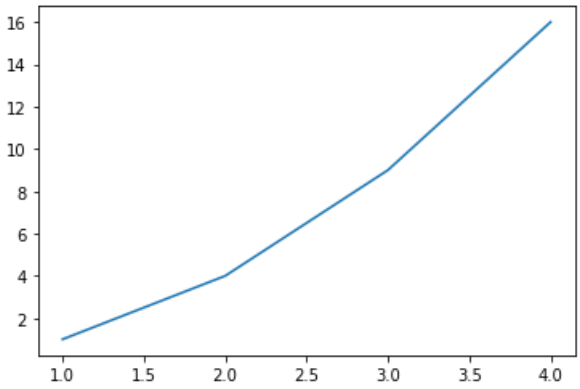


Plotting with multiple points

We can also plot as many po ints as we want, just make sure we have equal no of points on both x-axis and y-axis.

In [ ]:

```
xpoints=[1,2,3,4]
ypoints=[1,4,9,16]
plt.plot(xpoints,ypoints)
plt.show()
```



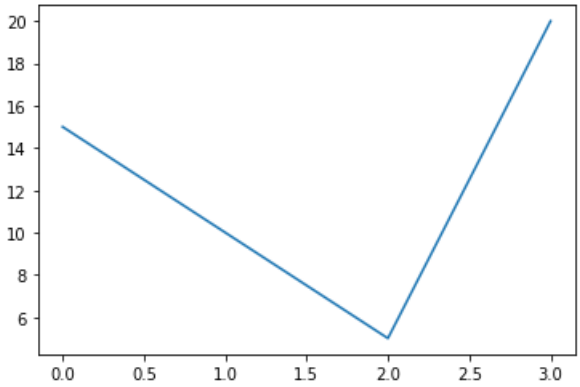
Default X-Points

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:

In [ ]:

```
ypoints=np.array([15,10,5,20])
plt.plot(ypoints)
plt.show()
```

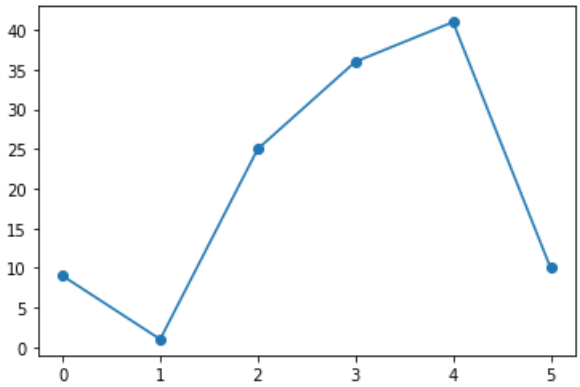


Markers in Matplotlib

We can use the keyword arguement marker to emphasize each point with a specified marker.

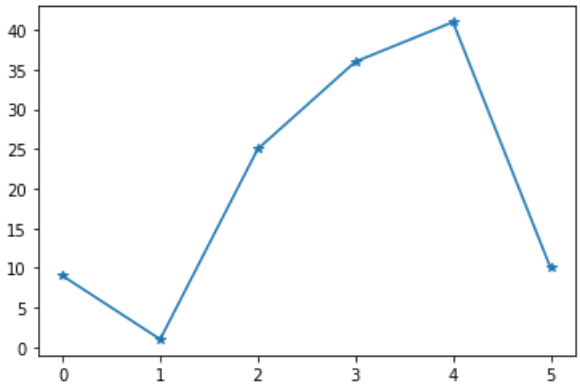
In [ ]:

```
#Let an example in which mark each point with a circle
y=np.array([9,1,25,36,41,10])
plt.plot(y,marker='o')
plt.show()
```



In [ ]:

```
#marking each point with a star
plt.plot(y,marker='*')
plt.show()
#similarly we can use point(.), diamond(d),square(s),pixel(,),etc to mark points.
```

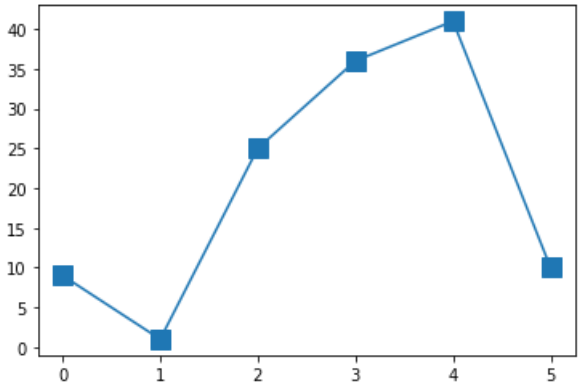


Markersize

You can use thekeyword arguement markersize or the shorter version ms to set the size of the markers.

In [ ]:

```
#set the sizeof the marker to 12
plt.plot(y,marker='s',ms=12)
plt.show()
```

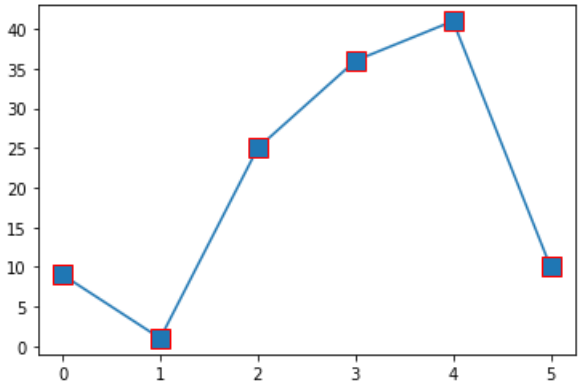


MarkerColor

We can use the keyword arguement markeredgecolor or the shorter version mec to set the color of the edge of the marker.

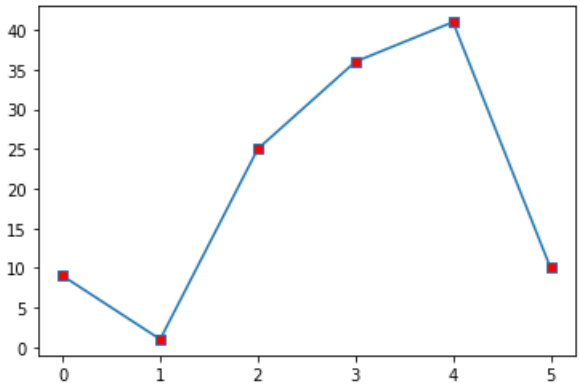
In [ ]:

```
plt.plot(y,marker='s',ms=12,mec='r') #set the red color
plt.show()
```



In [ ]:

```
plt.plot(y,marker='s',mfc='r') #set the markerface color to red
plt.show()
```



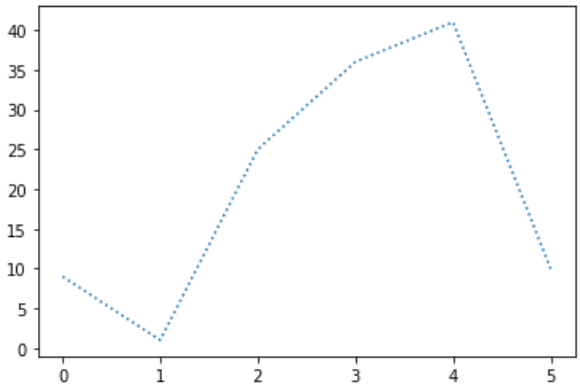
Matplotlib Line

LineStyle

You can use the word **linestyle**,or **ls** to change the style of the plotted line.

In [ ]:

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

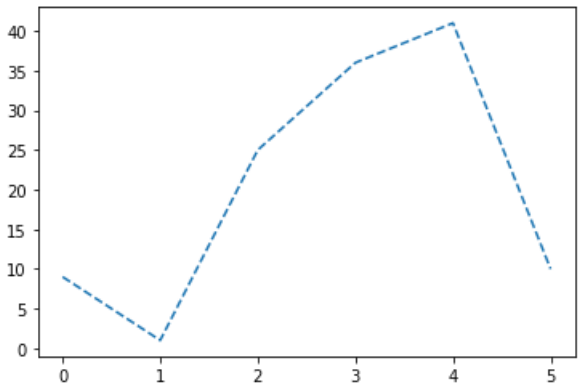


In [ ]:

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

Out[ ]:

```
[]
```

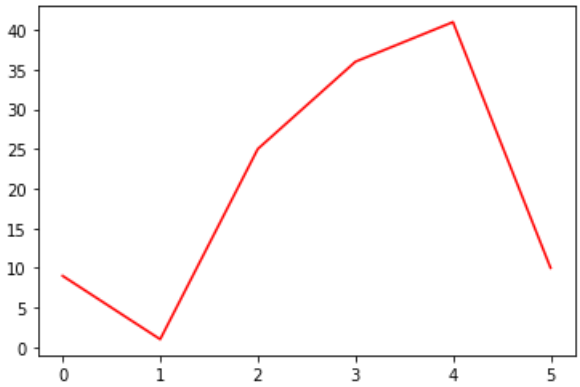


LineColor

You can use the keyword arguement **color** or shorter version **c** to set the color of the line.

In [ ]:

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



Matplotlib Labels and Title

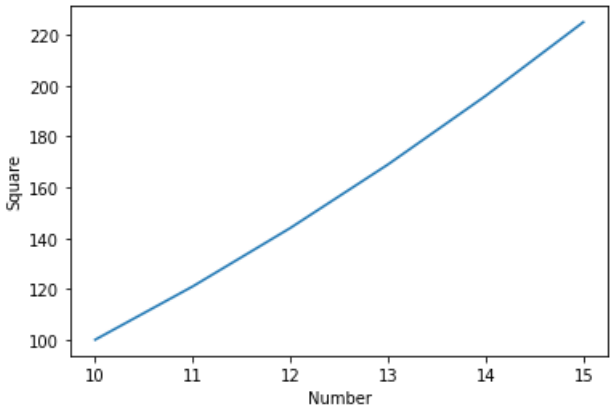
With pyplot, we can use the xlabel() and ylabel() functions to set a label for the x and y axis.

In [ ]:

```
a=np.array([10,11,12,13,14,15])
b=np.array([100,121,144,169,196,225])
plt.plot(a,b)
plt.xlabel('Number')
plt.ylabel('Square')
plt.plot()
```

Out[ ]:

```
[]
```



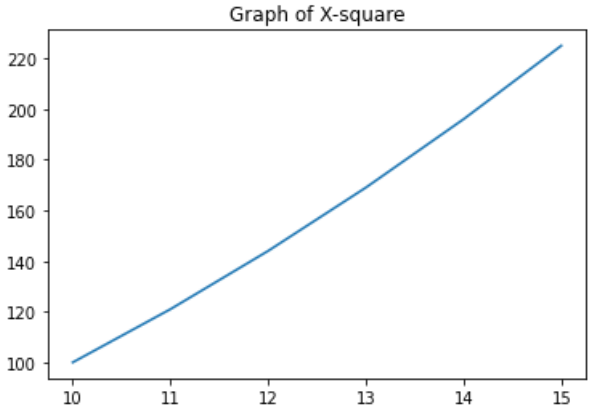
For writing a title, you can use the title() function to set a title for the plot.

In [ ]:

```
#writing title for the above plot
plt.title("Graph of X-square")
plt.plot(a,b)
plt.plot()
```

Out[ ]:

```
[]
```



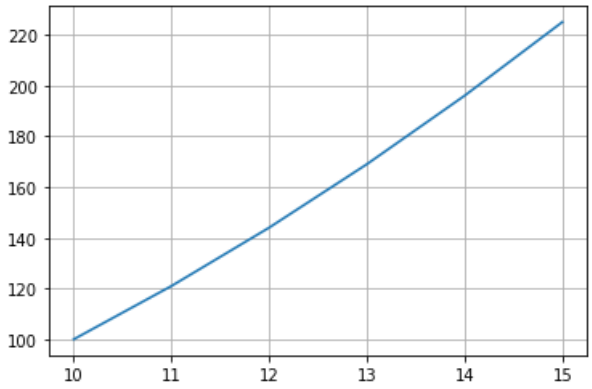
Add grid to the plot

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

In [ ]:

```
plt.plot(a,b)
```

```
plt.plot(a,b)
plt.grid()
plt.show()
```

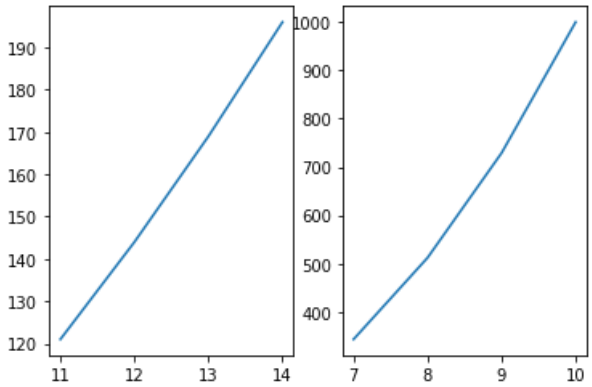


Display Multiple plots

With **subplot()** function, you can draw multiple plots in one figure. The subplot() function takes three arguements that describes the layout of the figure. The layout is organized in rows and columns, which are represented by the first and the second arguements. The third arguement represents the index of the current plot.

In [ ]:

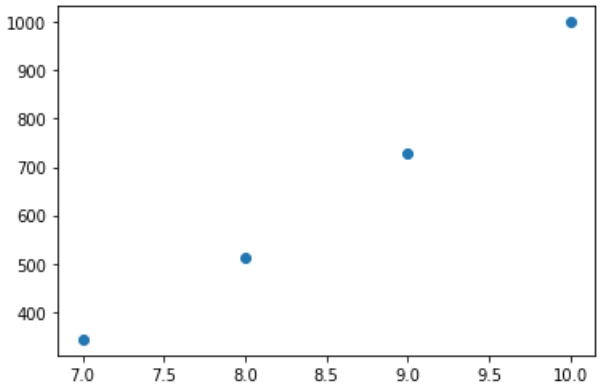
```
#plot1
a=np.array([11,12,13,14])
b=np.array([121,144,169,196])
plt.subplot(1,2,1)
plt.plot(a,b)

#plot2
x=np.array([7,8,9,10])
y=np.array([343,512,729,1000])
plt.subplot(1,2,2)
plt.plot(x,y)

plt.show()
```



Different Plots using Matplotlib

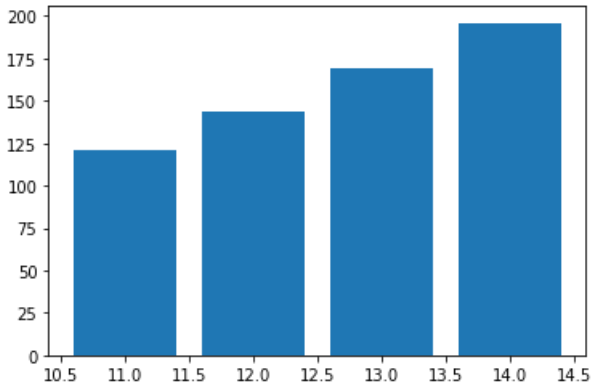
In [ ]:

```
#scatter plot using matplotlib
#using x and y from above
plt.scatter(x,y)
plt.show()
```



In [ ]:

```
#bar plot using matplotlib
#using a and b from above
plt.bar(a,b)
plt.show()
```

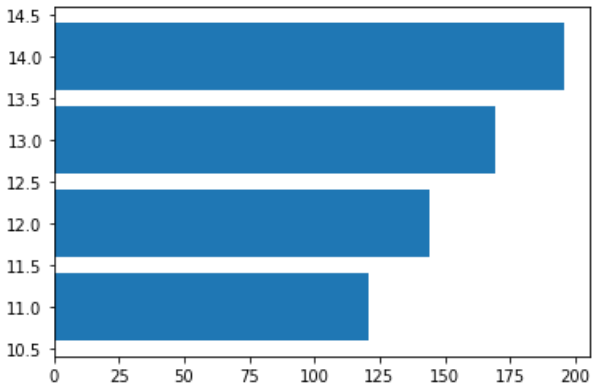


In [ ]:

```
#creatng horizontal bars using barh() function
plt.barh(a,b)
plt.plot()
```

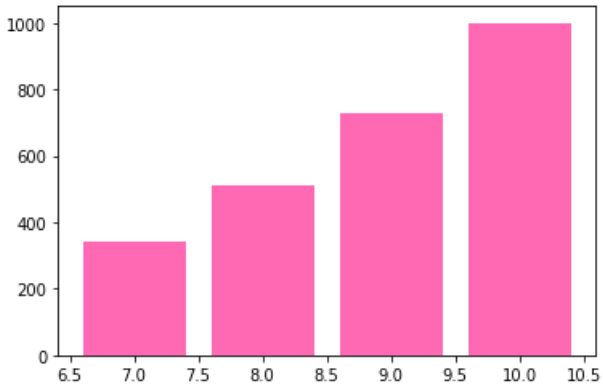
Out[ ]:

```
[]
```



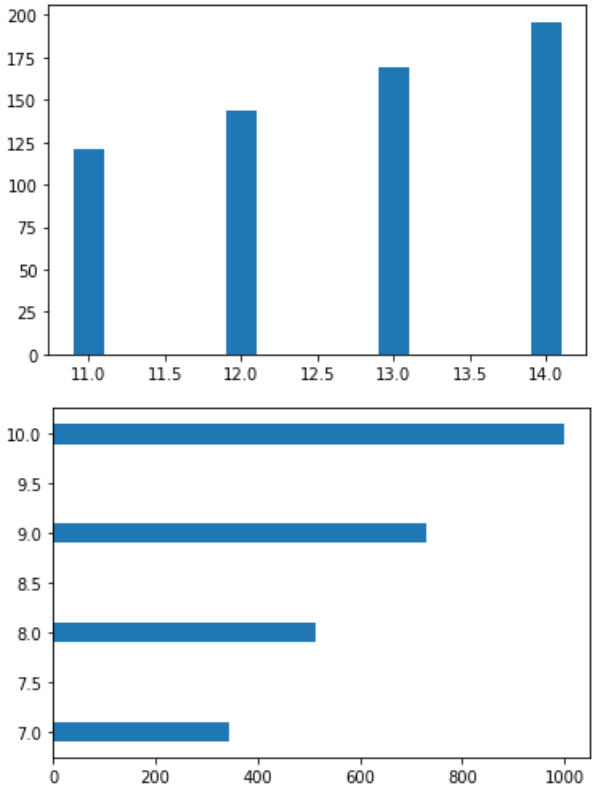
In [ ]:

```
#we can also add color to the bar plots
plt.bar(x,y,color='hotpink')  #since there are 140 supported colors,
plt.show()                      #so we can use any ot those colors.
```



In [ ]:

```
#we can also change bar width and bar height
plt.bar(a,b,width=0.2)
plt.show()

plt.barh(x,y,height=0.2)
plt.show()
```



- The default width value is **0.8.**
- The **barh()** takes the keyword arguement **height** to set the height of the bars.
- The default height value is **0.8.**

Histograms

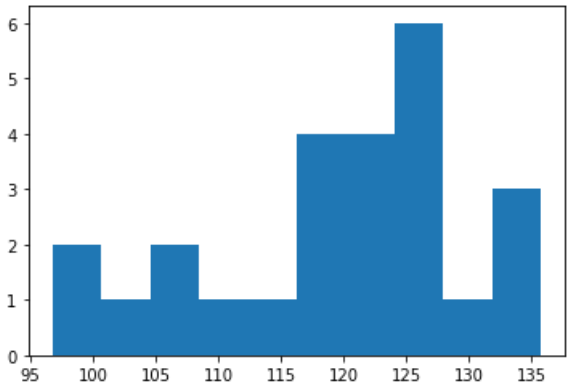
A histogram is a graph showing frequency distributions. It is a graph showing the number of observations within each given interval.

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.

In [ ]:

```
c=np.random.normal(120,10,25)
plt.hist(c)
plt.show()
```

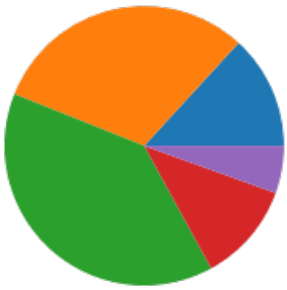


PieCharts

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

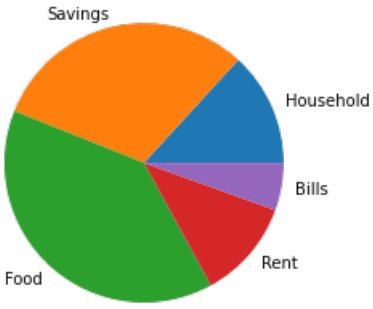
In [ ]:

```
z=np.array([24,56,71,21,10])
plt.pie(z)
plt.show()
```



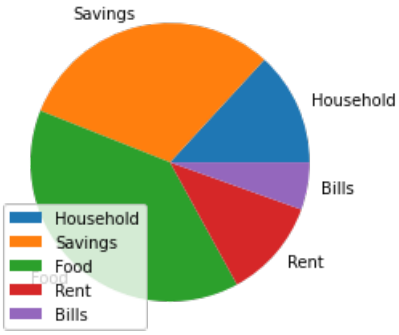
In [ ]:

```
#we can also add labels to the piechart
pielabels=['Household','Savings','Food','Rent','Bills']
plt.pie(z,labels=pielabels)
plt.show()
```



In [ ]:

```
#using legend
plt.pie(z,labels=pielabels)
plt.legend()
plt.show()
```



1. **Seaborn**

Seaborn is a library that uses Matplotlib underneath to plot graphs. It will be used to visualize random distributions.

Install Seaborn

If you have Python and PIP already installed on a system, install it using this command:

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

Import Seaborn

Import the Seaborn module in your code using the following statement.

```
import seaborn as sns
```

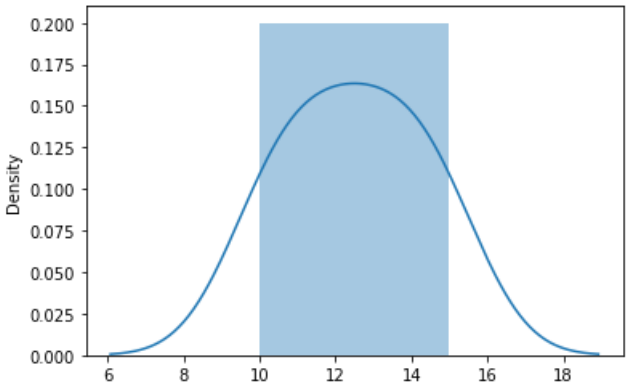
Distplots

Distplot stands for distribution plot, it takes as input an array and plots a curve corresponding to the distribution of points in the array.

In [ ]:

```
import seaborn as sns
sns.distplot([10, 11, 12, 13, 14, 15])
plt.show()
```

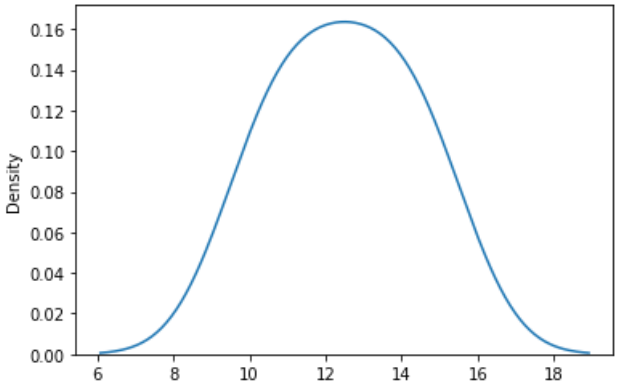
```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a
deprecated function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `histplot` (an axes-level function for h
istograms).
  warnings.warn(msg, FutureWarning)
```



In [ ]:

```
#Plotting a Distplot without the Histogram
sns.distplot([10, 11, 12, 13, 14, 15],hist=False)
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/distributions.py:2619: FutureWarning: `distplot` is a
deprecated function and will be removed in a future version. Please adapt your code to use either
`displot` (a figure-level function with similar flexibility) or `kdeplot` (an axes-level function for ke
rnel density plots).
  warnings.warn(msg, FutureWarning)
```



Different categories of plot in Seaborn

Plots are basically used for visualizing the relationship between variables. Those variables can be either be completely numerical or a category like a group, class or division. Seaborn divides plot into the below categories –

Relational plots: This plot is used to understand the relation between two variables.

Categorical plots: This plot deals with categorical variables and how they can be visualized.

Distribution plots: This plot is used for examining univariate and bivariate distributions

Regression plots: The regression plots in seaborn are primarily intended to add a visual guide that helps to emphasize patterns in a dataset during exploratory data analyses.

Matrix plots: A matrix plot is an array of scatterplots.

Multi-plot grids: It is an useful approach is to draw multiple instances of the same plot on different subsets of the dataset.

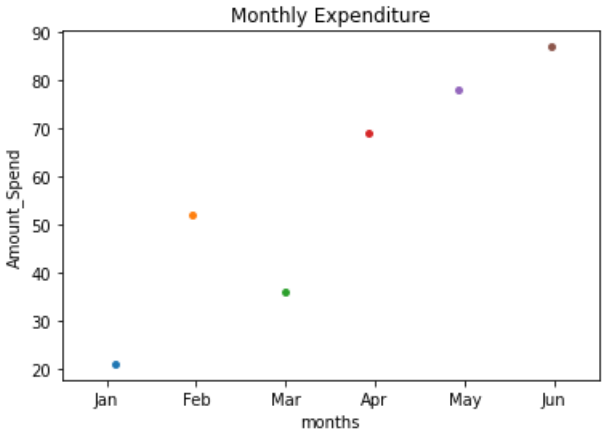
Plotting categorical scatter plots with Seaborn

- *StripPlot*

In [ ]:

```
a1=['Jan','Feb','Mar','Apr','May','Jun']
a2=[21,52,36,69,78,87]
ax=sns.stripplot(a1,a2)
ax.set(xlabel='months',ylabel='Amount_Spend')
plt.title('Monthly Expenditure')
plt.show()
```

```
/usr/local/lib/python3.7/dist-packages/seaborn/_decorators.py:43: FutureWarning: Pass the following
variables as keyword args: x, y. From version 0.12, the only valid positional argument will be `data`, an
d passing other arguments without an explicit keyword will result in an error or misinterpretation.
  FutureWarning
```



This is the one kind of scatter plot of categorical data with the help of seaborn.

Categorical data is represented on the x-axis and values correspond to them represented through the y-axis.

.striplot() function is used to define the type of the plot and to plot them on canvas using.

.set() function is used to set labels of x-axis and y-axis.

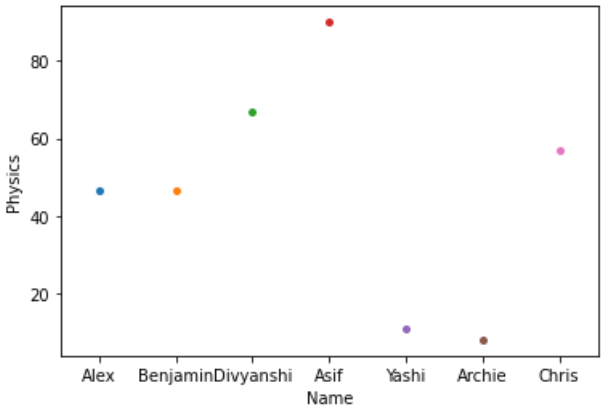
.title() function is used to give a title to the graph.

To view plot we use **.show()** function.

- ***SwarmPlot***

In [ ]:

```
#using dataframe that we  created earlier while demonstrating pandas
bx=sns.swarmplot(x='Name',y='Physics',data=df)
plt.show()
```



This is very much similar to stripplot but the only difference is that it does not allow overlapping of markers. It causes jittering in the markers of the plot so that graph can easily be read without information loss as seen in the above plot.

We use **.swarmplot()** function to plot swarm plot.

Another difference that we can notice in Seaborn and Matplotlib is that working with DataFrames doesn't go quite as smoothly with Matplotlib, which can be annoying if we doing exploratory analysis with Pandas. And that's exactly what Seaborn does easily, the plotting functions operate on DataFrames and arrays that contain a whole dataset.

If we want we can also change the representation of data on a particular axis.

- ***BarPlot***

A barplot is basically used to aggregate the categorical data according to some methods and by default it's 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.

Syntax:

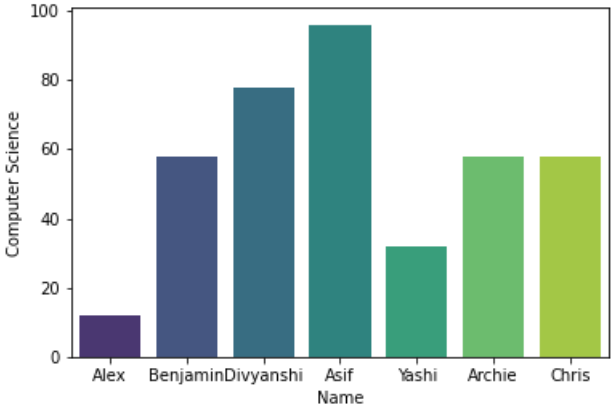
```
barplot([x, y, hue, data, order, hue_order, …])
```

In [ ]:

```
#using dataframe that we  created earlier while demonstrating pandas
sns.barplot(x='Name',y='Computer Science',data=df,palette='viridis')
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f29519910>
```



Palette is used to set the color of the plot.

The estimator is used as a statistical function for estimation within each categorical bin.

- ***CountPlot***

A countplot basically counts the categories and returns a count of their occurrences. It is one of the simplest plots provided by the seaborn library.

Syntax:

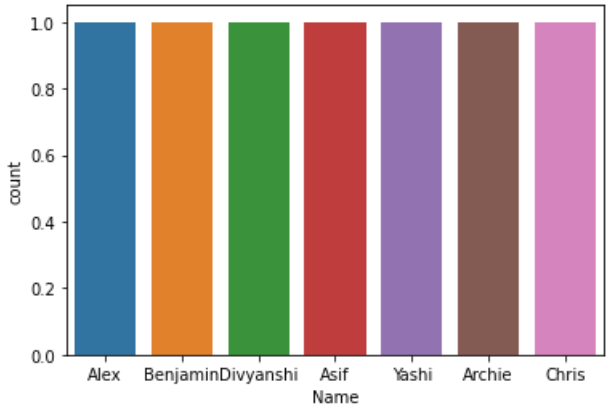
```
countplot([x, y, hue, data, order, …])
```

In [ ]:

```
#using dataframe that we  created earlier while demonstrating pandas
sns.countplot(x='Name',data=df)
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f29494050>
```



- ***BoxPlot***

Boxplot is a chart that is used to visualize how a given data (variable) is distributed using quartiles. It shows the minimum, maximum, median, first quartile and third quartile in the data set.

Syntax:

```
boxplot([x, y, hue, data, order, hue_order, …])
```

A box plot (or box-and-whisker plot) shows the distribution of quantitative data in a way that facilitates comparisons between variables or across levels of a categorical variable. The box shows the quartiles of the dataset while the whiskers extend to show the rest of the distribution, except for points that are determined to be "outliers" using a method that is a function of the inter-quartile range.

A box plot gives a five-number summary of a set of data which is-

Minimum – It is the minimum value in the dataset excluding the outliers

First Quartile (Q1) – 25% of the data lies below the First (lower) Quartile.

Median (Q2) – It is the mid-point of the dataset. Half of the values lie below it and half above.

Third Quartile (Q3) – 75% of the data lies below the Third (Upper) Quartile.

Maximum – It is the maximum value in the dataset excluding the outliers.

Note: The box plot shown in the above diagram is a perfect plot with no skewness. The plots can have skewness and the median might not be at the center of the box.

The area inside the box (50% of the data) is known as the Inter Quartile Range. The IQR is calculated as –

IQR = Q3-Q1

Outliers are the data points below and above the lower and upper limit. The lower and upper limit is calculated as –

Lower Limit = Q1 - 1.5*IQRUpper Limit = Q3 + 1.5*IQR

The values below and above these limits are considered outliers and the minimum and maximum values are calculated from the points which lie under the lower and upper limit.

How to interpret the box plot?

The bottom of the box is the 25% percentile and the top is the 75% percentile value of the data.

So, essentially the box represents the middle 50% of all the datapoints which represents the core region when the data is situated. The height of the boxplot is also called the Inter Quartile Range (IQR), which mathematically is the difference between the 75th and 25th percentile values of the data.

The thick line in the middle of the box represents the median. Whereas, the upper and lower whisker marks 1.5 times the IQR from the top (and bottom) of the box.

But, why whiskers matter?

Because, the points that lie outside the whiskers, that is,**(1.5 x IQR)** in both directions are generally considered as outliers.

Uses of a Box Plot

- Box plots provide a visual summary of the data with which we can quickly identify the average value of the data, how dispersed the data is, whether the data is skewed or not (skewness).
- The Median gives you the average value of the data.
- Box Plots shows Skewness of the data-

a) If the Median is at the center of the Box and the whiskers are almost the same on both the ends then the data is Normally Distributed.

b) If the Median lies closer to the First Quartile and if the whisker at the lower end is shorter (as in the above example) then it has a Positive Skew (Right Skew).

c) If the Median lies closer to the Third Quartile and if the whisker at the upper end is shorter then it has a Negative Skew (Left Skew).

- The dispersion or spread of data can be visualized by the minimum and maximum values which are found at the end of the whiskers.
- The Box plot gives us the idea of about the Outliers which are the points which are numerically distant from the rest of the data.

In [ ]:

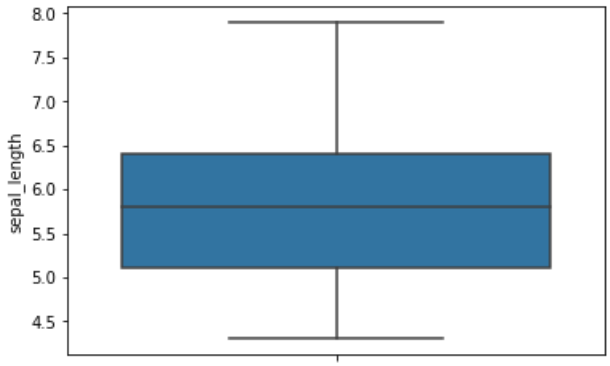
```
DF=sns.load_dataset('iris')
DF.head()
```

Out[ ]:

	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

In [ ]:

```
sns.boxplot( y=DF["sepal_length"] )
plt.show()
```



How to interpret the boxplot?

The bottom of the box is the 25% percentile and the top is the 75% percentile value of the data.

So, essentially the box represents the middle 50% of all the datapoints which represents the core region when the data is situated. The height of the boxplot is also called the **Inter Quartile Range (IQR)**, which mathematically is the difference between the 75th and 25th percentile values of the data.

The thick line in the middle of the box represents the median. Whereas, the upper and lower whisker marks 1.5 times the IQR from the top (and bottom) of the box.

But, why whiskers matter?

Because, the points that lie outside the whiskers, that is, (1.5 x IQR) in both directions are generally considered as outliers.

- ***ViolinPlot***

Violin Plot is a method to visualize the distribution of numerical data of different variables. It is similar to Box Plot but with a rotated plot on each side, giving more information about the density estimate on the y-axis. The density is mirrored and flipped over and the resulting shape is filled in, creating an image resembling a violin. The advantage of a violin plot is that it can show nuances in the distribution that aren't perceptible in a boxplot. On the other hand, the boxplot more clearly shows the outliers in the data.

Violin Plots hold more information than the box plots, they are less popular. Because of their unpopularity, their meaning can be harder to grasp for many readers not familiar with the violin plot representation.

Syntax:

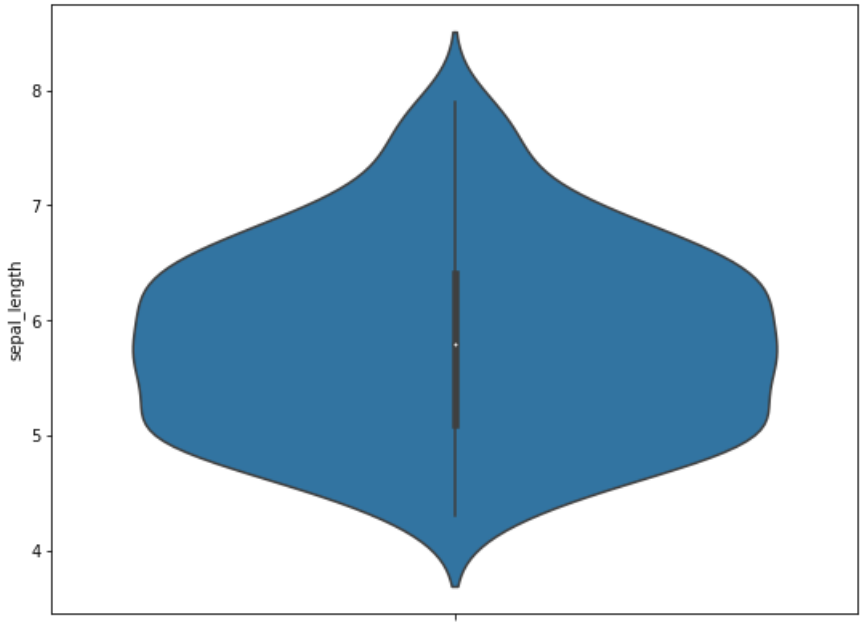
```
violinplot([x, y, hue, data, order, …])
```

In [ ]:

```
fig, ax = plt.subplots(figsize =(9, 7))
sns.violinplot( ax = ax, y = DF["sepal_length"] )
```

Out[ ]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2f29392610>
```



In [ ]:

```
sns.get_dataset_names()      #to see the list of available datasets
```

Out[ ]:

```
['anagrams',
 'anscombe',
 'attention',
 'brain_networks',
 'car_crashes',
 'diamonds',
 'dots',
 'dowjones',
 'exercise',
 'flights',
 'fmri',
 'geyser',
 'glue',
 'healthexp',
 'iris',
 'mpg',
 'penguins',
 'planets',
 'seaice',
 'taxis',
 'tips',
 'titanic']
```