## Module 1: An Introduction to Python:

### What is Python?

Python is a popular programming language. It was created by Guido van Rossum, and released in 1991.

### It is used for:

- Web development (server-side),
- Software development,
- Mathematics,
- System scripting.

### What can Python do?

- Python can be used on a server to create web applications.
- Python can be used alongside software to create workflows.
- Python can connect to database systems. It can also read and modify files.
- Python can be used to handle big data and perform complex mathematics.
- Python can be used for rapid prototyping, or for production-ready software development.

### 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 can be treated in a procedural way, an object-orientated way or a functional way.

- The most recent major version of Python is Python 3.

- Python will be written in a text editor. It is possible to write Python in an Integrated Development Environment, such as Thonny, Pycharm, Netbeans or Eclipse which are particularly useful when managing larger collections of Python files.

- Python uses new lines to complete a command, as opposed to other programming languages which often use semicolons or parentheses.

- Python relies on indentation, using whitespace, to define scope; such as the scope of loops, functions and classes. Other programming languages often use curly-brackets for this purpose.

## Module 2: Beginning Python Basics:

### Comments

- Python has commenting capability for the purpose of in-code documentation.
- Comments start with a #, and Python will commit the rest of the line as a comment.

### Example

```
#This is a comment.
print("Hello, World!")
```

- Comments can be used to explain Python code.
- Comments can be used to make the code more readable.
- Comments can be used to prevent execution when testing code.

you can add a multiline comment in your code by triple quotes

### Example:

```
"""
This is a comment written in more than just one line
"""
print("Hello, World!")
```



### Python Indentations:

- In other programming languages the indentation in code is for readability only, but in Python the indentation is very important.
- Python uses indentation to indicate a block of code.

Example:

```
if 5 > 2:

print("Five is greater than two!")
```

O/P in error:

```
if 5 > 2:

    print("Five is greater than two!")
```

## Python Variables:

In Python variables are created the moment you assign a value to it:

Example:

```
x = 5

y = "Hello, World!"
```

Python has no command for declaring a variable.

## Rules for Python variables:

- A variable can have a short name (like x and y) or a more descriptive name (age,carname, total_volume).
- A variable name must start with a letter or the underscore character.
- A variable name cannot start with a number.
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _ ).
- Variable names are case-sensitive (age, Age and AGE are three different variables).

## Creating Variables:

- Variables are containers for storing data values.
- Unlike other programming languages, Python has no command for declaring a variable.

- A variable is created the moment you first assign a value to it.

Example:

```
x = 5
y = "John"
print(x)
print(y)
```

- Variables do not need to be declared with any particular type and can even change type after they have been set.

Example:

```
x = 4 # x is of type int
x = "Sally" # x is now of type str
print(x)
```

- String variables can be declared either by using single or double quotes:

Example:

```
x = "John"
# is the same as
x = 'John'
```

**Assign Values to Multiple Variables:**

- Python allows you to assign values to multiple variables in one line.

Example:

```
x, y, z = "Orange", "Banana", "Cherry"
print(x)
print(y)
```

```
print(z)
```

- And you can assign the same value to multiple variables in one line

Example:

```
x = y = z = "Orange"
print(x)
print(y)
print(z)
```

**Output Variables:**

- The Python print statement is often used to output variables.

To combine both text and a variable, Python uses the + character:

Example:

```
x = "awesome"
print("Python is " + x)
```

- You can also use the + character to add a variable to another variable.

Example:

```
x = "Python is "
y = "awesome"
z = x + y
print(z)
```

- For numbers, the + character works as a mathematical operator.

Example:

```
x = 5
```

```
y = 10

print(x + y)
```

- If you try to combine a string and a number, Python will give you an error.

Example:

```
x = 5

y = "John"

print(x + y)
```

**Python Numbers:**

There are three numeric types in Python:

- int
- float
- Complex

Variables of numeric types are created when you assign a value to them:

Example:

```
x = 1 # int

y = 2.8 # float

z = 1j # complex
```

To verify the type of any object in Python, use the type() function:

Example:

```
print(type(x))

print(type(y))

print(type(z))
```

Int , or integer, is a whole number, positive or negative, without decimals, of unlimited length.

Example:

```
x = 1
y = 35656222554887711
z = -3255522
print(type(x))
print(type(y))
print(type(z))
```

Float, or "floating point number" is a number, positive or negative, containing one or more decimals.

Example:

```
x = 1.10
y = 1.0
z = -35.59
print(type(x))
print(type(y))
print(type(z))
```

Float can also be scientific numbers with an "e" to indicate the power of 10.

Example:

```
x = 35e3
y = 12E4
z = -87.7e100
```

```
print(type(x))
print(type(y))
print(type(z))
```

Complex numbers are written with a "j" as the imaginary part:

Example:

```
x = 3+5j
y = 5j
z = -5j
print(type(x))
print(type(y))
print(type(z))
```

Note: You cannot convert complex numbers into another number type.

**Random Number:**

Python does not have a random() function to make a random number, but Python has a built-in module called random that can be used to make random numbers.

Import the random module, and display a random number between 1 and 9:

Example:

```
import random
x=random.randrange(1,10))
```

**Type Conversion:**

You can convert from one type to another with the int(), float(), and complex() methods:

Example:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex
#convert from int to float:
a = float(x)
#convert from float to int:
b = int(y)
#convert from int to complex:
c = complex(x)
print(a)
print(b)
print(c)
print(type(a))
print(type(b))
print(type(c))
```



**String Literals:**

- String literals in python are surrounded by either single quotation marks, or double quotation marks.
- 'hello' is the same as "hello".

**Assign String to a Variable:**

Assigning a string to a variable is done with the variable name followed by an equal sign and the string

Eg:

```
a = "Hello"
print(a)
```

**Multiline Strings:**

You can assign a multiline string to a variable by using three quotes

Eg:

```
a = """Lorem ipsum dolor sit amet,
consectetur adipiscing elit,
sed do eiusmod tempor incididunt
ut labore et dolore magna aliqua."""
print(a)
```

**Strings are Arrays:**

Python does not have a character data type, a single character is simply a string with a length of 1. Square brackets can be used to access elements of the string.

Get the character at position 1 (remember that the first character has the position 0):

Eg1:

```
a = "Hello, World!"
print(a[1])
```

Substring:

Get the characters from position 2 to position 5 .

Eg2:

```
b = "Hello, World!"
print(b[2:5])
```

The strip() method removes any whitespace from the beginning or the end:

Eg:

```
a = " Hello, World! "
print(a.strip()) # returns "Hello, World!“
```

The len() method returns the length of a string:

Eg:

```
a = "Hello, World!"
print(len(a))
```

The lower() method returns the string in lower case:

Eg:

```
a = "Hello, World!"
print(a.lower())
```

The upper() method returns the string in upper case:

Eg:

```
a = "Hello, World!"
print(a.upper())
```

The split() method splits the string into substrings if it finds instances of the separator:

Eg:

```
a = "Hello, World!"
print(a.split(",")) # returns ['Hello', ' World!']
```

**String Format:**

we cannot combine strings and numbers.

Eg:

```
age = 36
txt = "My name is John, I am " + age
print(txt)
```

o/p: gave error

But we can combine strings and numbers by using the format() method

Eg:

```
age = 36
txt = "My name is John, and I am {}"
print(txt.format(age))
```

Eg2:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want {} pieces of item {} for {} dollars."
print(myorder.format(quantity, itemno, price))
```

You can use index numbers {0} to be sure the arguments are placedin the correct placeholders.

Eg:

```
quantity = 3
itemno = 567
price = 49.95
myorder = "I want to pay {2} dollars for {0}
pieces of item {1}."
print(myorder.format(quantity, itemno, price))
```

# Operators

**Operators**:

Operators are special symbols in Python that carry out arithmetic or logical computation. The value that the operator operates on is called the operand. For example: >>> 2+3 5. Here, + is the operator that performs addition. 2 and 3 are the operands and 5 is the output of the operation.

**Types:**

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Identity operators
- Membership operators
- Bitwise operators

Arithmetic Operators:

Arithmetic operators are used to performing mathematical operations like
addition, subtraction, multiplication, and division.

| Operator | Description | Syntax |
|---|---|---|
| + | Addition: adds two operands | x + y |
| – | Subtraction: subtracts two operands | x – y |
| * | Multiplication: multiplies two operands | x * y |
| / | Division (float): divides the first operand by the second | x / y |
| // | Division (floor): divides the first operand by the second | x // y |
| % | Modulus: returns the remainder when the first operand is divided by the second | x % y |
| ** | Power: Returns first raised to power second | x ** y |

Assignment Operators:

Assignment operators are used to assign values to the variables

.

| Operator | Description | Syntax |
|---|---|---|
| = | Assign value of right side of expression to left side operand | x = y + z |
| += | Add AND: Add right-side operand with left side operand and then assign to left operand | a+=b a=a+b |
| -= | Subtract AND: Subtract right operand from left operand and then assign to left operand | a-=b<br>a=a-b |
| *= | Multiply AND: Multiply right operand with left operand and then assign to left operand | a*=b<br>a=a*b |
| /= | Divide AND: Divide left operand with right | a/=b |

| Operator | Description | Syntax |
|---|---|---|
| | operand and then assign to left operand | a=a/b |
| %= | Modulus AND: Takes modulus using left and right operands and assign the result to left operand | a%=b a=a%b |
| //= | Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand | a//=b<br>a=a//b |
| **= | Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand | a**=b a=a**b |
| &= | Performs Bitwise AND on operands and assign value to left operand | a&=b a=a&b |
| \|= | Performs Bitwise OR on operands and assign value to left operand | a\|=b<br>a=a\|b |
| ^= | Performs Bitwise xOR on operands and assign value to left operand | a^=b<br>a=a^b |
| >>= | Performs Bitwise right shift on operands and assign value to left operand | a>>=b a=a>>b |
| <<= | Performs Bitwise left shift on operands and assign value to left operand | a <<= b<br>a= a << b |

Comparison Operators:

Comparison of Relational operators compares the values. It either returns True or False according to the condition.

| Operator | Description | Syntax |
|---|---|---|

| Operator | Description | Syntax |
|---|---|---|
| > | Greater than: True if the left operand is greater than the right | x > y |
| < | Less than: True if the left operand is less than the right | x < y |
| == | Equal to: True if both operands are equal | x == y |
| != | Not equal to – True if operands are not equal | x != y |
| >= | Greater than or equal to True if the left operand is greater than or equal to the right | x >= y |
| <= | Less than or equal to True if the left operand is less than or equal to the right | x <= y |
| is | x is the same as y | x is y |
| is not | x is not the same as y | x is not y |

= is an assignment operator and == comparison operator.

Logical Operators:

Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements.

| operator | Description | Syntax |
|---|---|---|
| and | Logical AND: True if both the operands are true | x and y |
| or | Logical OR: True if either of the operands is true | x or y |
| not | Logical NOT: True if the operand is false | not x |

Identity Operators:

is and is not are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

| | |
|---|---|
| is | True if the operands are identical |
| is not | True if the operands are not identical |

Membership Operators:

in and not in are the membership operators; used to test whether a value or variable is in a sequence.

| | |
|---|---|
| in | True if value is found in the sequence |
| not in | True if value is not found in the sequence |

Bitwise Operators:

Bitwise operators act on bits and perform the bit-by-bit operations. These are used to operate on binary numbers.

| Operator | Description | Syntax |
|---|---|---|
| & | Bitwise AND | x & y |
| \| | Bitwise OR | x \| y |
| ~ | Bitwise NOT | ~x |
| ^ | Bitwise XOR | x ^ y |
| >> | Bitwise right shift | x>> |
| << | Bitwise left shift | x<< |

# Module 3: Python Program Flow:

## Indentation:

- Indentation refers to the spaces at the beginning of a code line.
- Where in other programming languages the indentation in code is for readability only, the indentation in Python is very important.
- Python uses indentation to indicate a block of code.

**Eg:**

```
if 5 > 2:
    print("Five is greater than two!")
```

## Decision making statements:

If else statements are also called decision making statements which are used to execute a block of statement on certain conditions. There are 4 types of decision making statements in python.

- If Statement
- If - else Statement
- Elif Statement

## If Statement:

If statement is the simplest of decision making statements. It contains a condition and a block of statements.

If the condition is True, interpreter executes the block of statements, else not.

**Syntax:**

```
if expression:
    statement
```

**Eg:**

```
a = 5

if  a > 0 :

    print('a is positive.')
```

**If-Else Statement:**

If-else statement contains a condition and two block of statements: if-block and else-block.

If the condition is True, interpreter executes if-block, or if the condition is False, interpreter executes else-block.

**Syntax:**

```
if condition:
    #block of statement
else:
    #another block of statement(else-block)
```

**Eg:**

.

```
a = -5
if a > 0 :
   print('a is positive.')
else :
   print('a is not positive')
```

**The elif Statement:**

The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them.

**Syntax:**

```
if exp1:
    #block of statement
elif exp2:
    #block of statement
elif exp3:
    #block of statement
```

```
else:
    #block of statement
```

**Eg:**

```
a = 5
if a == 0 :
   print('a is zero.')
elif a > 0:
   print('a is positive.')
elif a < 0:
   print('a is negative.')
```

## Loops:

It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times.

## For loop:

The for loop in Python is used to iterate over a sequence (list, tuple, string) or other iterable objects. Iterating over a sequence is called traversal.

**Syntax :**

```
For val in sequence:
    Loop body
```

**Eg:**

```
# Program to find the sum of all numbers stored in a list

# List of numbers

numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]

# variable to store the sum

sum = 0
```

```
# iterate over the list

for val in numbers:

    sum = sum+val

print("The sum is", sum)
```

**Nested for loop:**

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":

**Syntax:**

```
for iterative –var1 in sequence:

        for iterative –var2 in sequence:

                #block of statement

        #other statement
```

**Eg:**

```
adj = ["red", "big", "tasty"]
fruits = ["apple", "banana", "cherry"]

for x in adj:
  for y in fruits:
    print(x, y)
```

**The while Loop:**

With the while loop we can execute a set of statements as long as a condition is true.

**syntax :**

```
while expression:
```

statements

**Eg:**

```
i = 1
while i < 6:
        print(i)
        i += 1
```

**The break Statement:**

With the break statement we can stop the loop even if the while condition is true:

**Eg:**

```
i = 1
while i < 6:
  print(i)
  if i == 3:
        break
  i += 1
```

**The continue Statement:**

With the continue statement we can stop the current iteration, and continue with the next:

**Eg:**

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

**Pass Statement:**

The pass statement is a null operation since nothing happens when it is executed.

**Example:**

```
for letter in 'Welcome to Home':
    pass
print("Last letter is: ",letter)
```

# Arrays

Arrays are used to store multiple values in one single variable. An array is a special variable, which can hold more than one value at a time.

**Example**:

```
cars = ["Ford", "Volvo", "BMW"]
```

**Looping Array Elements:**

```
cars = ["Ford", "Volvo", "BMW"]
for x in cars:
    print(x)
```

**Output:**

['Ford', 'Volvo', 'BMW']



**Adding Array Elements: (append)**

```
cars = ["Ford", "Volvo", "BMW"]
cars.append("Honda")
print(cars)
```

Output:

['Ford', 'Volvo', 'BMW','Honda']

**Removing Array Elements:**

Delete the second element of the cars array.

Eg:

```
cars = ["Ford", "Volvo", "BMW"]
cars.pop(1)
print(cars)
```

Output:

['Ford', 'BMW']

**Delete the element that has the value "Volvo":**

```
cars = ["Ford", "Volvo", "BMW"]
cars.remove("Volvo")
print(cars)
```

Output:

['Ford', 'BMW']

Eg:1

```
cars = ["Ford", "Volvo", "BMW"]

cars.remove()

print(cars)
```

Output:

Type error:

**Access the Elements of an Array:**

```
cars = ["Ford", "Volvo", "BMW"]

x = cars[0]

print(x)
```

Output:

Ford

**The Length of an Array:**

```
cars = ["Ford", "Volvo", "BMW"]

x = len(cars)

print(x)
```

Output:

3

**Use indices to access elements of an array:**

```
import array as a
b=a.array('i',[2,3,4,5])
print("first element:",b[0])
print("second element:",b[1])
print("second last element:",b[-1])
```

Output:

```
('first element:', 2)
('second element:', 3)
('second last element:', 5)
```

**1D Array:**

```
s=5
arr=[0]*s
print(arr)
```

Output:

```
[0, 0, 0, 0, 0]
```

**Second method for 1D array:**

```
s=5
arr=[1 for i in range(s)]
print(arr)
```

Output:

```
[1, 1, 1, 1, 1]
```

**Multi Dimensional Array:**

```
rows,cols=(5,5)
arr=[[0]*cols]*rows
arr[0][0]=1
for row in arr:
    print(row)
```

Output:

```
[1, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
[1, 0, 0, 0, 0]
```

**We can concatenate two arrays using + operator:**

```
import array as arr

odd=arr.array('i',[1,3,5])

even=arr.array('i',[2,4,6])

nums=arr.array('i')

nums=odd+even

print(nums)
```

Output:

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

## Python Matrix

Python doesn't have a built-in type for matrices. However, we can treat list of a list as a matrix.

A matrix is a two-dimensional data structure where numbers are arranged into rows and columns.

**Program to add two matrices using nested loop:**

```
x=[[12,7,3],
   [4,5,6],
   [7,8,9]]

y=[[5,8,1],
   [6,7,3],
   [4,5,9]]

result=[[0,0,0],
```

```
        [0,0,0],
        [0,0,0]]
for i in range(len(x)):
    for j in range(len(x[0])):
        result[i][j]=x[i][j]+y[i][j]
for r in result:
    print(r)
```

Output:

```
[17, 15, 4]
[10, 12, 9]
[11, 13, 18]
```

**Matrix Multiplication using Nested Loop:**

```
x=[[12,7,3],
   [4,5,6],
   [7,8,9]]
y=[[5,8,1,2],
   [6,7,3,0],
   [4,5,9,1]]
result=[[0,0,0,0],
        [0,0,0,0],
        [0,0,0,0]]
for i in range(len(x)):
```

```
    for j in range(len(y[0])):
        for k in range(len(y)):
            result[i][j]+=x[i][k]+y[k][j]
for r in result:
    print(r)
```

Output:

```
[37, 42, 35, 25]
[30, 35, 28, 18]
[39, 44, 37, 27]
```

**Array - Collection**

- **List** is a collection which is ordered and changeable. Allows duplicate members.
- **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
- **Set** is a collection which is unordered and unindexed. No duplicate members.
- **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

| | | | | |
|---|---|---|---|---|
| list | ordered | Changeable | - | Allow duplicate members |
| tuple | ordered | unchangeable | - | Allow duplicate members |
| set | unordered | Changeable | unindexed | not allow Duplicate member |
| Dictionary | unordered | Changeable | indexed | not allow Duplicate member |

## List

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

print(thislist)

Output:

[‘apple’, ‘banana’, ‘cherry’]

**Access Items:**

Eg:

thislist = [“apple”,”banana”, ”cherry”]

print(thislist[1])

Output:

banana

**Change Item Value:**

To change the value of a specific item, refer to the index number.

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

thislist[1] = ”blackcurrant”

print(thislist)

Output:

[‘apple’, ‘blackcurrant’, ‘cherry’]

**Loop Through a List:**

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

for x in thislist:

    print(x)

Output:

apple

banana

cherry

**Check if Item Exists:**

Check if "apple" is present in the list:

Eg:

```
thislist = ["apple", "banana", "cherry"]
if "apple" in thislist:
    print("Yes, 'apple' is in the fruits list")
```

Output:

Yes, 'apple' is in the fruits list

**List Length:**

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

Output:

3

**Add Items:**

Using the append() method to append an item.

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

Output:

['apple', 'banana', 'cherry', 'orange']

To add an item at the specified index,use the insert() method:

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

thislist.insert(1, ”orange”)

print(thislist)

Output:

[‘apple’, ‘orange’, ‘banana’, ‘cherry’]

**Remove Item:**

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

thislist.remove(“banana”)

print(thislist)

Output:

[‘apple’, ‘cherry’]



The pop() method removes the specified index, (or the last item if index is not specified):

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

thislist.pop()

print(thislist)

Output:

[‘apple’, ‘banana’]

**The del keyword removes the specified index:**

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

del thislist[0]

print(thislist)

Output:

[‘banana’, ‘cherry’]

The del keyword can also delete the list completely.

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

del thislist

Output:

**********(“thislist” doesn’t exist)

**Copy a List:**

To make a copy is to use the built-in method list().

Eg:

thislist = [“apple”, ”banana”, ”cherry”]

mylist = list(thislist)

print(mylist)

Output:

[‘apple’, ‘banana’, ‘cherry’]

**Tuple**

Python Tuple is used to store the sequence of immutable python objects. Tuple is similar to lists since the value of the items stored in the list can be changed whereas the tuple is immutable and the value of the items stored in the tuple cannot be changed.

Ex:

x= (“apple”, “banana”, “cherry”)

print(x)

Output:

(‘apple’, ‘banana’, ‘cherry’)



**Access Tuple Items:**

t= (“apple”, “banana”, “cherry”)

print(t[1])

Output:

banana

**Set**

The set in python can be defined as the unordered collection of various items enclosed within the curly braces. The elements of the set cannot be duplicate.

Ex:

t = {“apple”, “banana”, “cherry”}

print(t)

**Output:**

set([‘cherry’, ‘banana’, ‘apple’])

**Add items:**

s= {“apple”, “banana”, “cherry”}

s.add(“orange”)

print(s)

Output:

set([‘orange’, ‘cherry’, ‘banana’, ‘apple’])

**Update items:**

n= {“apple”, “banana”, “cherry”}

s= {“pineapple”, “mango”, “papaya”}

n.update(s)

print(n)

Output:

set([‘mango’, ‘papaya’, ‘apple’, ‘pineapple’, ‘cherry’, ‘banana’])

**Remove() item:**

thisset = {“apple”, “banana”, “cherry”}

thisset.remove(“banana”)

print(thisset)

Output:

set([‘cherry’, ‘apple’])

**Discard() item:**

thisset = {“apple”, “banana”, “cherry”}

thisset.discard(“banana”)

print(thisset)

Output:

set([‘cherry’, ‘apple’])



**Frozen Set:**

The frozenset() function returns an unchangeable frozenset object (which is like a set object, only unchangeable)

syntax :

frozenset([iterable])

(An iterable object, like list, set, tuple etc..)

Ex:

mylist = [‘apple’, ‘banana’, ‘cherry’]

x = frozenset(mylist)

print(x)

Output:

frozenset([‘cherry’, ‘apple’, ‘banana’])

**Dictionary**

The dictionary can be created by using multiple key-value pairs enclosed with the small brackets () and separated by the colon (:). The collections of the key-value pairs are enclosed within the curly braces {}.

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.

Ex:

studs={“raja”:”id04”,

“sabari”:”id07”,

“dharsan”:”id09”}

print(studs)

Output:

{‘raja’: ‘id04’, ‘sabari’: ‘id07’, ‘dharsan’: ‘id09’}

**#for loop to print the items of the dictionary set**

```
emp={“name”:”john”,”age”:”24”,”salary”:”25000”,”company”:”hcl”}
for x in emp.items():
        print(x)
```

Output:

(‘salary’, ‘25000’)

(‘age’, ‘24’)

(‘company’, ‘hcl’)

(‘name’, ‘john’)

**Iterating Dictionary:**

```
studs={“raja”:”id04”,
“sabari”:”id07”,
“dharsan”:”id09”}
print(studs.keys())
```

Output:

[‘raja’, ‘sabari’, ‘dharsan’]



Ex:

```
studs={“raja”:”id04”,
“sabari”:”id07”,
“dharsan”:”id09”}
print(studs.values())
```

Output:

[‘id04’, ‘id07’, ‘id09’]

Add items of the dictionary set:

```
studs={“raja”:”id04”,
“sabari”:”id07”,
“dharsan”:”id09”}
studs[“aswin”]=“id02”
print(studs)
```

Output:

{‘raja’: ‘id04’, ‘sabari’: ‘id07’, ‘dharsan’: ‘id09’, ‘aswin’: ‘id02’}

# Module 4: Functions& Modules:

## Functions:

- 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:

**Eg:**

```
def my_function():
    print("Hello from a function")
```

## Calling a Function:

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

**Eg:**

```
def my_function():
    print("Hello from a function")
my_function()
```

## Arguments:

- 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.

**Eg:**

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

```
my_function("Emil")
my_function("Tobias")
my_function("Linus")
```

**Scope:**

A variable is only available from inside the region it is created. This is called **scope**.

- Local Scope
- Global Scope

**Local Scope:**

A variable created inside a function belongs to the *local scope* of that function, and can only be used inside that function.

**Eg:**

```
def myfunc():
    x = 300

    print(x)
myfunc()
```

**Global Scope:**

- A variable created in the main body of the Python code is a global variable and belongs to the global scope.
- Global variables are available from within any scope, global and local.

**Eg:**

```
x = 300

def myfunc():
    print(x)
```

myfunc()

print(x)

## Lambda Functions:

- In Python, anonymous function means that a function is without a name. As we already know that def keyword is used to define the normal functions and the lambda keyword is used to create anonymous functions.
- Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

**syntax :**

*lambda arguments : expression*

**Use of lambda() with filter():**

This offers an elegant way to filter out all the elements of a sequence, for which the function returns True.

Lambda eg:

```
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(filter(lambda x: (x%2 != 0) , li))
print(final_list)
```

Function Eg:

```
ages = [5, 12, 17, 18, 24, 32]

def myFunc(x):
  if x < 18:
    return False
  else:
    return True

adults = filter(myFunc, ages)

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

## Converting map to list, tuple, set

```
def to_upper_case(x):
    y=x.upper()
    return y
```

```
map_iterator = map(to_upper_case, ['a', 'b', 'c'])
my_list = list(map_iterator)
print(my_list)

map_iterator = map(to_upper_case, ['a', 'b', 'c'])
my_set = set(map_iterator)
print(my_set)

map_iterator = map(to_upper_case, ['a', 'b', 'c'])
my_tuple = tuple(map_iterator)
print(my_tuple)
```

Output:

```
['A', 'B', 'C']
{'C', 'B', 'A'}
('A', 'B', 'C')
```

## Use of lambda() with map()

The function is called with a lambda function and a list and a new list is returned which contains all the lambda modified items returned by that function for each item in the list.

```
# to get double of a list.
li = [5, 7, 22, 97, 54, 62, 77, 23, 73, 61]
final_list = list(map(lambda x: x*2 , li))
print(final_list)
```

Output:

```
[10, 14, 44, 194, 108, 124, 154, 46, 146, 122]
```

```
# python code to demonstrate working of reduce()
# using operator functions

# importing functools for reduce()
import functools

# importing operator for operator functions
import operator

# initializing list
lis = [ 1 , 3, 5, 6, 2, ]

# using reduce to compute sum of list
# using operator functions
print ("The sum of the list elements is : ",end="")
print (functools.reduce(operator.add,lis))

# using reduce to compute product
# using operator functions
print ("The product of list elements is : ",end="")
print (functools.reduce(operator.mul,lis))

# using reduce to concatenate string
print ("The concatenated product is : ",end="")
print (functools.reduce(operator.add,["geeks","for","geeks"]))
```

Output

```
The sum of the list elements is : 17
The product of list elements is : 180
The concatenated product is : geeksforgeeks
```

## Use of lambda() with reduce()

- The function is called with a lambda function and a list and a new reduced result is returned. This performs a repetitive operation over the pairs of the list. This is a part of functools module.

```
# to get sum of a list
from functools import reduce
li = [5, 8, 10, 20, 50, 100]
sum = reduce((lambda x, y: x + y), li)
print (sum)
```

## What is a Module?

Consider a module to be the same as a code library. A file containing a set of functions you want to include in your application.

**Syntax**:

module_name.function_name.

## Create a Module

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named mymodule.py

Eg:

```
def greeting(name):
    print("Hello, " + name)
```

**Use a Module**

Now we can use the module we just created, by using the import statement:

Example

Import the module named mymodule, and call the greeting function:

Eg:

```
import mymodule

mymodule.greeting("Jonathan")
```

**Output**:

Hello, Jonathan

Hello, Jonathan

**Variables in Module**

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

**Ex**:

```
person1={
"name":"Chandru",
"age":"27",
"country":"Norway"
}
import mymodule
a=mymodule.person1["name"]
print(a)
```

**Output**:

Chandru

Chandru

# Module 5: Exceptions Handling

## Exception Handling:

An exception can be defined as an abnormal condition in a program resulting in the disruption in the flow of the program.

If we do not handle the exception, the interpreter doesn't execute all the code that exists after the that

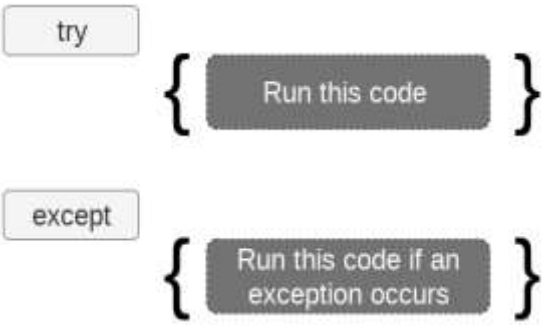
- In the **try** clause, all statements are executed until an exception is encountered.
- **except** is used to catch and handle the exception(s) that are encountered in the try clause.
- **raise** allows you to throw an exception at any time.
- **else** lets you code sections that should run only when no exceptions are encountered in the try clause.
- **finally** enables you to execute sections of code that should always run, with or without any previously encountered exceptions.

**Common Exceptions:**

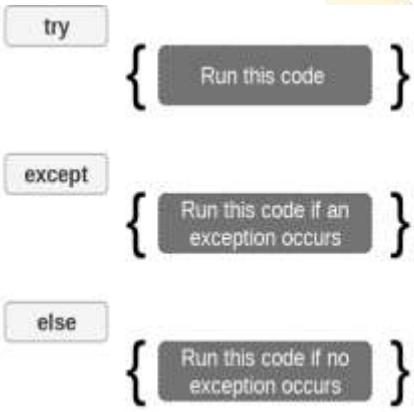
- **ZeroDivisionError:** Occurs when a number is divided by zero.
- **NameError:** It occurs when a name is not found. It may be local or global.
- **IndentationError:** If incorrect indentation is given.
- **IOError:** It occurs when Input Output operation fails.
- **EOFError:** It occurs when the end of the file is reached, and yet operations are being performed.

**Syntax**:

```
try:
    #block of code

except Exception1:
    #block of code

except Exception2:
    #block of code

#other code
```




- The try-except statement :




**Built-in Exceptions:**

Python (interpreter) raises exceptions when it encounter errors. For example: divided by zero,IOError etc.

```
try:
    #this will throw an exception if the file doesn't exist.
    fileptr = open("file.txt","r")
except IOError:
    print("File not found")
else:
    print("The file opened successfully")
    fileptr.close()
```

```
try:
   fh = open("testfile", "r")
   fh.write("This is my test file for exception handling!!")
except IOError:
   print "Error: can\'t find file or read data"
else:
   print "Written content in the file successfully"
```

This produces the following result –

```
Error: can't find file or read data
```

---

```
try:
    a=5
    b='0'
    print (a+b)
except TypeError:
    print('Unsupported operation')
print ("Out of try except blocks")
```

---

## Catching Exceptions :

```
# import module sys to get the type of exception
import sys

randomList = ['a', 0, 2]

for entry in randomList:
    try:
        print("The entry is", entry)
        r = 1/int(entry)
        break
    except:
        print("Oops!",sys.exc_info()[0],"occured.")
        print("Next entry.")
        print()
print("The reciprocal of",entry,"is",r)
```

```
try:
    print("try block")
    x=int(input('Enter a number: '))
    y=int(input('Enter another number: '))
    z=x/y
except ZeroDivisionError:
    print("except ZeroDivisionError block")
    print("Division by 0 not accepted")
else:
    print("else block")
    print("Division = ", z)
finally:
    print("finally block")
    x=0
    y=0
print ("Out of try, except, else and finally blocks." )
```

## Output

```
try block
Enter a number: 10
Enter another number: 0
except ZeroDivisionError block
Division by 0 not accepted
finally block
Out of try, except, else and finally blocks.
```

## User-Defined Exception( Raising Exceptions):

Python also provides the raise keyword to be used in the context of exception handling. It causes an exception to be generated explicitly. Built-in errors are raised implicitly. However, a built-in or custom exception can be forced during execution.

```
try:
    a = int(input("Enter a positive integer: "))
    if a <= 0:
        raise ValueError("That is not a positive number!")
except ValueError as ve:
    print(ve)
```

```
# define Python user-defined exceptions
class Error(Exception):
   """Base class for other exceptions"""
   pass
class ValueTooSmallError(Error):
   """Raised when the input value is too small"""
   pass
class ValueTooLargeError(Error):
   """Raised when the input value is too large"""
   pass
# our main program
# user guesses a number until he/she gets it right
# you need to guess this number
number = 10

while True:
   try:
      i_num = int(input("Enter a number: "))
      if i_num < number:
         raise ValueTooSmallError
      elif i_num > number:
         raise ValueTooLargeError
      break
   except ValueTooSmallError:
      print("This value is too small, try again!")
      print()
   except ValueTooLargeError:
      print("This value is too large, try again!")
      print()
print("Congratulations! You guessed it correctly.")
```

Output:

```
Enter a number: 12
This value is too large, try again!

Enter a number: 0
This value is too small, try again!

Enter a number: 8
This value is too small, try again!

Enter a number: 10
Congratulations! You guessed it correctly.
```

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
except ValueError:
    print(x, "is out of allowed range")
else:
    print(x, "is within the allowed range")
```

**Result:**

```
Enter a number upto 100: 200
200 is out of allowed range
Enter a number upto 100: 50
50 is within the allowed range
```

## Assert:

The assert statement is used to continue the execute if the given condition evaluates to True.

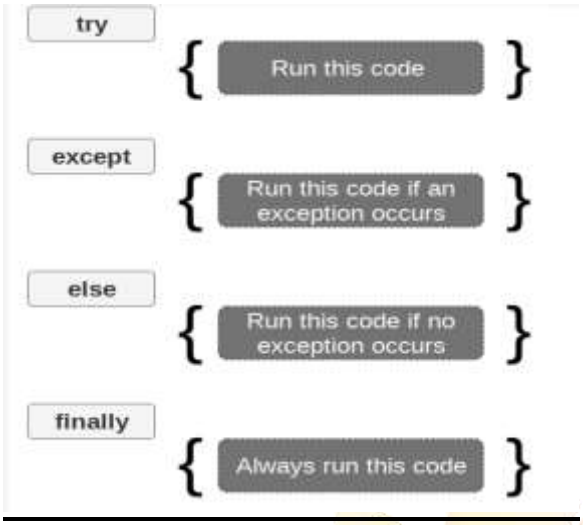
If the assert condition evaluates to False, then it raises the AssertionError exception with the specified error message.

**Eg:**

```
x = "hello"
#if condition returns False, AssertionError is raised:
assert x == "goodbye", "x should be 'hello'"
```

## The finally block:

We can use the finally block with the try block in which, we can place the important code which must be executed before the try statement throws an exception.




```
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"
```

## Python program to handle simple runtime error:

```
a = [1, 2, 3]
try:
    print "Second element = %d" %(a[1])

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

except IndexError:
    print "An error occurred"
```

## Program to handle multiple errors with one except statement:

```
try :
    a = 3
    if a < 4 :

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

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

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

```
while True:
    try:
        x = int(input("Please enter a number: "))
        break
    except ValueError:
        print("Oops!  That was no valid number.  Try again...")
```

```
def this_fails():
    x = 1/0

try:
    this_fails()
except ZeroDivisionError as err:
    print('Handling run-time error:', err)
```

```
Handling run-time error: division by zero
```

```
try:
    with open('file.log') as file:
        read_data = file.read()
except FileNotFoundError as fnf_error:
    print(fnf_error)
```

```
try:
    f = open("test.txt",encoding = 'utf-8')
    # perform file operations
finally:
    f.close()
```

This type of construct makes sure the file is closed even if an exception occurs.

# Module 6: File Handling:

## File Handling

File handling in python including, creating a file, opening a file, closing a file, writing and appending the file, etc.

### Opening a file:example

```
#opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")
```

The close() method:

### Example

```
# opens the file file.txt in read mode
fileptr = open("file.txt","r")

if fileptr:
    print("file is opened successfully")

#closes the opened file
fileptr.close()
```

### Read Lines of the file:

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file.txt","r");

#stores all the data of the file into the variable content
content = fileptr.readline();

# prints the type of the data stored in the file
print(type(content))

#prints the content of the file
print(content)

#closes the opened file
fileptr.close()
```

## Looping through the file:

```
#open the file.txt in read mode. causes an error if no such file exists.
fileptr = open("file.txt","r");

#running a for loop
for i in fileptr:
   print(i) # i contains each line of the file
```

## Writing the file:

To write some text to a file, we need to open the file using the open method with one of the following access modes.

- **a**: It will append the existing file. The file pointer is at the end of the file. It creates a new file if no file exists.
- **w**: It will overwrite the file if any file exists. The file pointer is at the beginning of the file.

## Creating a new file:

- The new file can be created by using one of the following access modes with the function open(). **x:** it creates a new file with the specified name. It causes an error a file exists with the same name

```
#open the file.txt in read mode. causes error if no such file exists.
fileptr = open("file2.txt","x");

print(fileptr)

if fileptr:
   print("File created successfully");
```

## Using with statement with files

It is always suggestible to use the with statement in the case of file s because, if the break, return, or exception occurs in the nested block of code then it automatically closes the file. It doesn't let the file to be corrupted.

```
with open("file.txt",'r') as f:
    content = f.read();
    print(content)
```

## File Pointer positions

Python provides the tell() method which is used to print the byte number at which the file pointer exists.

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#reading the content of the file
content = fileptr.read();

#after the read operation file pointer modifies. tell() returns the location of the fileptr.

print("After reading, the filepointer is at:",fileptr.tell())
```

## Modifying file pointer position:

The python provides us the seek() method which enables us to modify the file pointer position externally.

```
# open the file file2.txt in read mode
fileptr = open("file2.txt","r")

#initially the filepointer is at 0
print("The filepointer is at byte :",fileptr.tell())

#changing the file pointer location to 10.
fileptr.seek(10);

#tell() returns the location of the fileptr.
print("After reading, the filepointer is at:",fileptr.tell())
```

**Output:**

```
The filepointer is at byte : 0
After reading, the filepointer is at 10
```

## os module:

The os module provides us the functions that are involved in file processing operations like renaming, deleting, etc.

Renaming the file:

Syntax and eg:

```
rename(?current-name?, ?new-name?)
```

Example

```
import os;

#rename file2.txt to file3.txt
os.rename("file2.txt","file3.txt")
```

Removing the file:

```
remove(?file-name?)
```

Example

```
import os;

#deleting the file named file3.txt
os.remove("file3.txt")
```

## Creating the new directory:

The mkdir() method is used to create the directories in the current working directory.

```
mkdir(?directory name?)
```

Example

```
import os;

#creating a new directory with the name new
os.mkdir("new")
```

**Changing the current working directory**

The chdir() method is used to change the current working directory to a specified directory.

```
chdir("new-directory")
```

Example

```
import os;

#changing the current working directory to new

os.chdir("new")
```

The getcwd() method:

This method returns the current working directory.

```
os.getcwd()
```

Example

```
import os;

#printing the current working directory
print(os.getcwd())
```

Deleting directory:

The rmdir() method is used to delete the specified directory.

```
os.rmdir(?directory name?)
```

Example

```
import os;

#removing the new directory
os.rmdir("new")
```

## OOPs:

Object-oriented programming (OOP) is **a method of structuring a program by bundling related properties and behaviors into individual objects**. We'll learn the basics of object-oriented programming in Python. Conceptually, objects are like the components of a system.

Oops stands for Object Oriented programming system.

- Class & object
- Inheritance
- Polymorphism
- Data abstraction
- Encapsulation

## Class:

A class is a code template for creating objects. Objects have member variables and have behaviour associated with them. In python a class is created by the keyword class.

### Class and Objects:

- Suppose a class is a prototype of a building. A building contains all the details about the floor, doors, windows, etc. we can make as many buildings

as we want, based on these details. Hence, the building can be seen as a class, and we can create as many objects of this class.

- On the other hand, the object is the instance of a class. The process of creating an object can be called as instantiation

**Creating classes in Python:**

In Python, a class can be created by using the keyword class, followed by the class name. The syntax to create a class is given below.

**Syntax:**

class ClassName:

#statement_suite

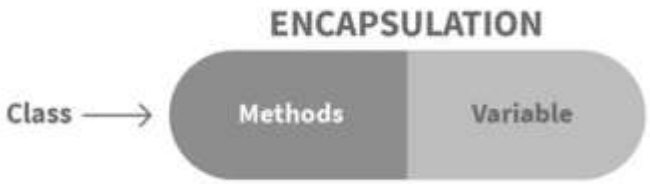
Example

```
class Employee:
        id = 10
        name = "Devansh"
        def display (self):
                print(self.id,self.name)
```

**Encapsulation:**

Encapsulation in Python describes the concept of bundling data and methods within a single unit. So, for example, when you create a class, it means you are implementing encapsulation. A class is an example of encapsulation as it binds all the data members (instance variables) and methods into a single unit.

The concept of encapsulation is the same in all object-oriented programming languages. The difference is seen when the concepts are applied to particular languages.




Check the below demonstration of how variables can easily be accessed.

```
class Person:
    def __init__(self, name, age=0):
        self.name = name
        self.age = age

    def display(self):
        print(self.name)
        print(self.age)

person = Person('Dev', 30)
#accessing using class method
person.display()
#accessing directly from outside
print(person.name)
print(person.age)
```

**Output**

```
Dev
30
Dev
30
```

**Access Modifiers**

Compared to languages like Java that offer access modifiers (public or private) for variables and methods, Python provides access to all the variables and methods globally.

**Access Modifiers in Python encapsulation**

Sometimes there might be a need to restrict or limit access to certain variables or functions while programming. That is where access modifiers come into the picture.

Now when we are talking about access, 3 kinds of access specifiers can be used while performing Encapsulation in Python. They are as follows:

- Public Members
- Private Members
- Protected Members

| Class member access specifier | Access from own class | Accessible from derived class | Accessible from object |
|---|---|---|---|
| Private member | Yes | No | No |
| Protected member | Yes | Yes | No |
| Public member | Yes | Yes | Yes |

Encapsulation in Python using **public** members

```
# illustrating public members & public access modifier
class pub_mod:
    # constructor
    def __init__(self, name, age):
        self.name = name;
        self.age = age;

    def Age(self):
        # accessing public data member
        print("Age: ", self.age)
# creating object
obj = pub_mod("Jason", 35);
# accessing public data member
print("Name: ", obj.name)
# calling public member function of the class
```

```
obj.Age()
```

**Output**

```
Name: Jason
Age: 35
```

Encapsulation in Python using **private** members

```
# illustrating private members & private access modifier
class Rectangle:
        __length = 0 #private variable
        __breadth = 0#private variable
        def __init__(self):
                 #constructor
                self.__length = 5
                self.__breadth = 3
                #printing values of the private variable within the class
                print(self.__length)
                print(self.__breadth)

rect = Rectangle() #object created
#printing values of the private variable outside the class
print(rect.length)
print(rect.breadth)
```

**Output:**

```
5
3
Traceback (most recent call last) :
  File "main.py", line 14, in <module>
    print(rect.length)
AttributeError: 'Rectangle' object has no attribute 'length'
```

Encapsulation in Python using **protected** members

```
# illustrating protected members & protected access modifier
class details:
```

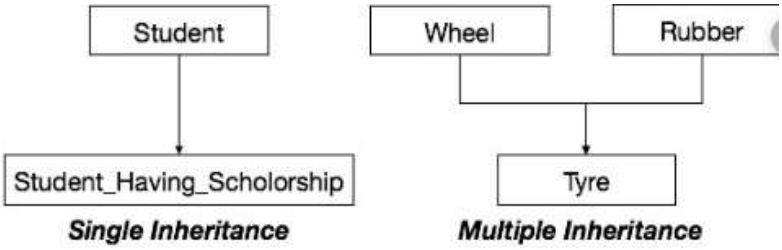
```
    _name="Jason"
    _age=35
    _job="Developer"
class pro_mod(details):
        def __init__(self):
                print(self._name)
                print(self._age)
                print(self._job)

# creating object of the class
obj = pro_mod()
# direct access of protected member
print("Name:",obj.name)
print("Age:",obj.age)
```

**Output:**

```
Jason
35
Developer
Traceback (most recent call last):
  File "main.py", line 15, in <module>
    print("Name:",obj.name)
AttributeError: 'pro_mod' object has no attribute 'name'
```

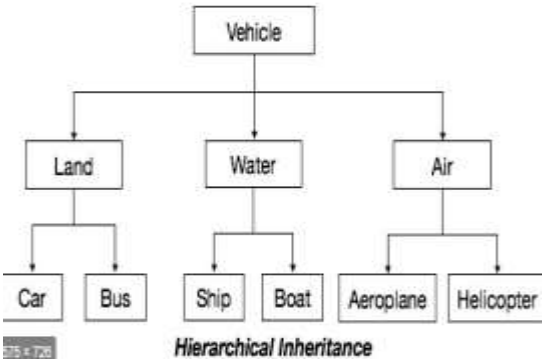
**Inheritance:**

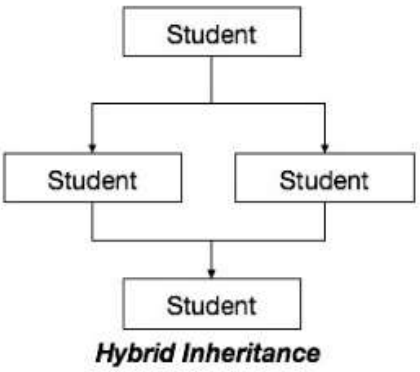
Inheritance allows us to define a class that inherits all the methods and properties from another class.

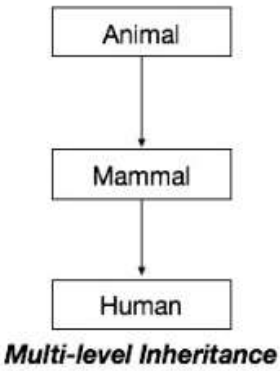




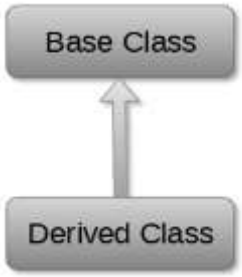





Single Inheritance:




## Syntax

```
class derived-class(base class):
    <class-suite>
```

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
d = Dog()
d.bark()
d.speak()
```

**Output:**

```
dog barking
Animal Speaking
```

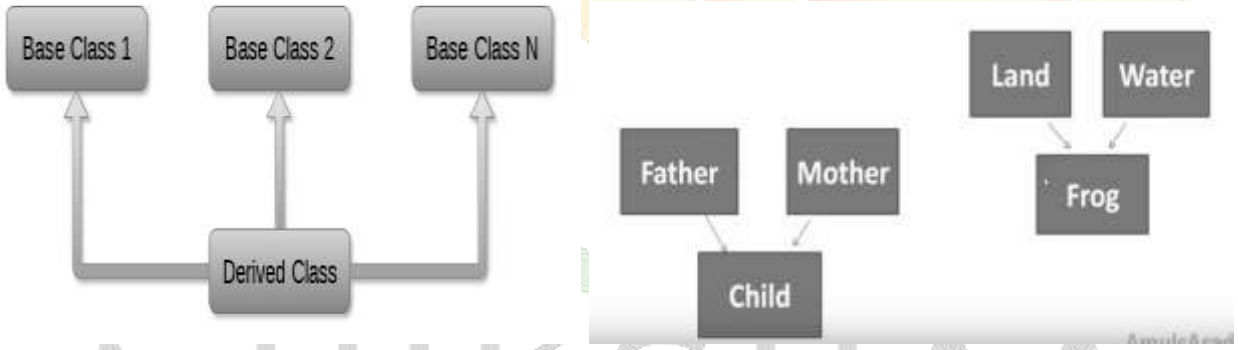
Multi-Level inheritance:

```
class Animal:
    def speak(self):
        print("Animal Speaking")
#The child class Dog inherits the base class Animal
class Dog(Animal):
    def bark(self):
        print("dog barking")
#The child class Dogchild inherits another child class Dog
class DogChild(Dog):
    def eat(self):
        print("Eating bread...")
d = DogChild()
d.bark()
d.speak()
d.eat()
```

**Output:**

```
dog barking
Animal Speaking
Eating bread...
```

Multiple inheritance:




Syntax

```
class Base1:
    <class-suite>

class Base2:
    <class-suite>
.
.
.
class BaseN:
    <class-suite>

class Derived(Base1, Base2, ..... BaseN):
    <class-suite>
```

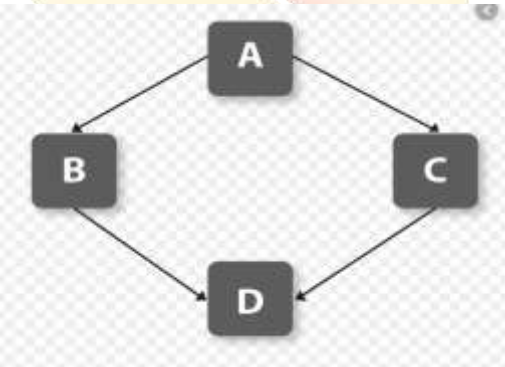
```
class Calculation1:
    def Summation(self,a,b):
        return a+b;
class Calculation2:
    def Multiplication(self,a,b):
        return a*b;
class Derived(Calculation1,Calculation2):
    def Divide(self,a,b):
        return a/b;
d = Derived()
print(d.Summation(10,20))
print(d.Multiplication(10,20))
print(d.Divide(10,20))
```

**Output:**

```
30
200
0.5
```

Hybrid:

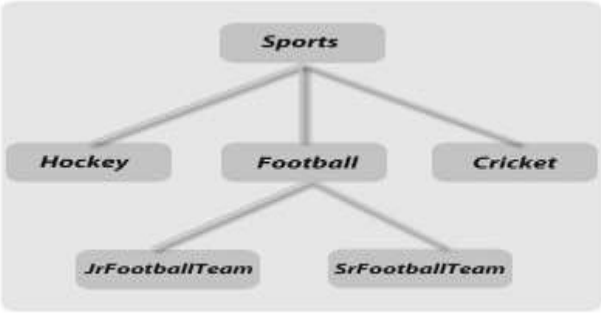
**Hybrid inheritance** is a combination of multiple **inheritance** and multilevel **inheritance**. A class is derived from two classes as in multiple **inheritance**. However, one of the parent classes is not a base class.



```
class A:
    def m(self):
        print("m() from class A...")

class B(A):
    def m(self):
        print("m() from class B...")
        super().m()

class C(A):
    def m(self):
        print("m() from class C ...")
        super().m()

class D(B,C):
    def m(self):
        print("m() from class D...")
        super().m()

obj1=D()
obj1.m()
```

**Hierarchical Inheritance:**

When a class has more than one child classes (sub classes) or in other words more than one child classes have the same parent class then this type of **inheritance** is known as **hierarchical inheritance**.




Hierarchical Inheritance

```
class Details:
    def __init__(self):
        self.__id="<No Id>"
        self.__name="<No Name>"
        self.__gender="<No Gender>"
    def setData(self,id,name,gender):
        self.__id=id
        self.__name=name
        self.__gender=gender
    def showData(self):
        print("Id: ",self.__id)
        print("Name: ", self.__name)
        print("Gender: ", self.__gender)

class Employee(Details): #Inheritance
    def __init__(self):
        self.__company="<No Company>"
        self.__dept="<No Dept>"
    def setEmployee(self,id,name,gender,comp,dept):
        self.setData(id,name,gender)
        self.__company=comp
        self.__dept=dept
    def showEmployee(self):
        self.showData()
        print("Company: ", self.__company)
        print("Department: ", self.__dept)
```

```
class Doctor(Details): #Inheritance
    def __init__(self):
        self.__hospital="<No Hospital>"
        self.__dept="<No Dept>"
    def setEmployee(self,id,name,gender,hos,dept):
        self.setData(id,name,gender)
        self.__hospital=hos
        self.__dept=dept
    def showEmployee(self):
        self.showData()
        print("Hospital: ", self.__hospital)
        print("Department: ", self.__dept)

def main():
    print("Employee Object")
    e=Employee()
    e.setEmployee(1,"Prem Sharma","Male","gmr","excavation")
    e.showEmployee()
    print("\nDoctor Object")
    d = Doctor()
    d.setEmployee(1, "pankaj", "male", "aiims", "eyes")
    d.showEmployee()

if __name__=="__main__":
    main()
```

**The issubclass(sub,sup) method:**

- The issubclass(sub, sup) method is used to check the relationships between the specified classes. It returns true if the first class is the subclass of the second class, and false otherwise**.**

```
class Calculation1:
   def Summation(self,a,b):
      return a+b;
class Calculation2:
   def Multiplication(self,a,b):
      return a*b;
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
d = Derived()
print(issubclass(Derived,Calculation2))
print(issubclass(Calculation1,Calculation2))
```

**Output:**

```
True
False
```

## The isinstance (obj, class) method:

The isinstance() method is used to check the relationship between the objects and classes. It returns true if the first parameter, i.e., obj is the instance of the second parameter, i.e., class.

```
class Calculation1:
   def Summation(self,a,b):
      return a+b;
class Calculation2:
   def Multiplication(self,a,b):
      return a*b;
class Derived(Calculation1,Calculation2):
   def Divide(self,a,b):
      return a/b;
d = Derived()
print(isinstance(d,Derived))
```

**Output:**

```
True
```

## Polymorphism:

The word **polymorphism** means having many forms. Typically, **polymorphism** occurs when there is a hierarchy of classes and they are related by inheritance.

Method overloading:

- Method overloading in Python is achieved by using one method with different number of arguments.
- In other programming languages, we have more than one method definition for the same method name to achieve method overloading.

```
class Compute:

    # area method
    def area(self, x = None, y = None):

        if x != None and y != None:
            return x * y

        elif x != None:
            return x * x

        else:
            return 0
```

```
# object
obj = Compute()

# zero argument
print("Area:", obj.area())

# one argument
print("Area:", obj.area(2))

# two argument
print("Area:", obj.area(4, 5))
```

Output:

Area: 0

Area: 4

Area: 20



**Overloading the * Operator:**

Operator Overloading:

Operator Overloading means giving extended meaning beyond their predefined operational meaning.

To overload the * sign, we will need to implement __mul__() function in the class.

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)

    def __mul__(self,other):
        x = self.x *other.x
        y = self.y * other.y
        return Point(x,y)

p1 = Point(2,3)
p2 = Point(-1,2)
print(p1 * p2)
```

Operator Overloading Special Functions:

| Operator | Expression | Internally |
|---|---|---|
| Addition | p1 + p2 | p1.__add__(p2) |
| Subtraction | p1 - p2 | p1.__sub__(p2) |
| Multiplication | p1 * p2 | p1.__mul__(p2) |
| Power | p1 ** p2 | p1.__pow__(p2) |
| Division | p1 / p2 | p1.__truediv__(p2) |
| Floor Division | p1 // p2 | p1.__floordiv__(p2) |
| Remainder (modulo) | p1 % p2 | p1.__mod__(p2) |

**Overriding**:

**Override** means having two methods with the same name but doing different tasks. It means that one of the methods overrides the other.

Following conditions must be met for overriding a function:

- **Inheritance** should be there. Function overriding cannot be done within a class. We need to derive a child class from a parent class.
- The function that is redefined in the child class should have the same signature as in the parent class i.e. same **number of parameters**.

Polymorphism, Method Overriding:

```
class Bank:
    def getroi(self):
        return 10;
class SBI(Bank):
    def getroi(self):
        return 7;

class ICICI(Bank):
    def getroi(self):
        return 8;
b1 = Bank()
b2 = SBI()
b3 = ICICI()
print("Bank Rate of interest:",b1.getroi());
print("SBI Rate of interest:",b2.getroi());
print("ICICI Rate of interest:",b3.getroi());
```

**Output:**

```
Bank Rate of interest: 10
SBI Rate of interest: 7
ICICI Rate of interest: 8
```

## Data abstraction:

Data abstraction refers to providing only essential information about the data to the outside world, hiding the background details or implementation.

We can also perform data hiding by adding the double underscore (___) as a prefix to the attribute which is to be hidden. After this, the attribute will not be visible outside of the class through the object.

```
class Employee:
   __count = 0;
   def __init__(self):
      Employee.__count = Employee.__count+1
   def display(self):
      print("The number of employees",Employee.__count)
emp = Employee()
emp2 = Employee()
try:
   print(emp.__count)
finally:
   emp.display()
```

**Output:**

```
The number of employees 2
AttributeError: 'Employee' object has no attribute '__count'
```

## Generators and iterators:

To create iterations easily using Python generators, how it is different from iterators and normal functions, and why you should use it.

### Generators in Python

There is a lot of work in building an iterator in Python. We have to implement a class with __iter__() and __next__() method, keep track of internal states, and raise StopIteration when there are no values to be returned.

This is both lengthy and counterintuitive. Generator comes to the rescue in such situations.

Python generators are a simple way of creating iterators. All the work we mentioned above are automatically handled by generators in Python.

Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

### Create Generators in Python

It is fairly simple to create a generator in Python. It is as easy as defining a normal function, but with a yield statement instead of a return statement.

If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.

The difference is that while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

**Differences between Generator function and Normal function**

- Here is how a generator function differs from a normal function.
- Generator function contains one or more yield statements.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.
- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.
- Here is an example to illustrate all of the points stated above. We have a generator function named my_gen() with several yield statements.

```
# A simple generator function
defmy_gen():
    n = 1
print('This is printed first')
    # Generator function contains yield statements
yield n

    n += 1
print('This is printed second')
yield n

    n += 1
print('This is printed at last')
yield n
```

An interactive run in the interpreter is given below. Run these in the Python shell to see the output.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>>next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is transferred to the caller.

>>> # Local variables and theirs states are remembered between successive calls.
>>>next(a)
This is printed second
2

>>>next(a)
This is printed at last
3

>>> # Finally, when the function terminates, StopIteration is raised automatically on further calls.
>>>next(a)
Traceback (most recent call last):
...
StopIteration
>>>next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that the value of variable n is remembered between each call.

Unlike normal functions, the local variables are not destroyed when the function yields. Furthermore, the generator object can be iterated only once.

To restart the process we need to create another generator object using something like a = my_gen().

One final thing to note is that we can use generators with for loops directly.

This is because a for loop takes an iterator and iterates over it using next() function. It automatically ends when StopIteration is raised. Check here to know how a for loop is actually implemented in Python.

```
# A simple generator function
defmy_gen():
    n = 1
print('This is printed first')
    # Generator function contains yield statements
yield n

    n += 1
print('This is printed second')
yield n

    n += 1
print('This is printed at last')
yield n


# Using for loop
for item in my_gen():
print(item)
```

When you run the program, the output will be:

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Python Generators with a Loop

The above example is of less use and we studied it just to get an idea of what was happening in the background.

Normally, generator functions are implemented with a loop having a suitable terminating condition.

Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1, -1, -1):
        yieldmy_str[i]


# For loop to reverse the string
    for char in rev_str("hello"):
        print(char)
```

Output

```
o
l
l
e
h
```

In this example, we have used the range() function to get the index in reverse order using the for loop.

**Note: This generator function not only works with strings, but also with other kinds of iterables like list, tuple, etc.**

**Python Generator Expression**

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Similar to the lambda functions which create anonymous functions, generator expressions create anonymous generator functions.

The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.

The major difference between a list comprehension and a generator expression is that a list comprehension produces the entire list while the generator expression produces one item at a time.

They have lazy execution ( producing items only when asked for ). For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
list_ = [x**2 for x in my_list]

# same thing can be done using a generator expression
# generator expressions are surrounded by parenthesis ()
generator = (x**2 for x in my_list)

print(list_)
print(generator)
```

Output

```
[1, 9, 36, 100]
<generator object <genexpr> at 0x7f5d4eb4bf50>
```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object, which produces items only on demand.

Here is how we can start getting items from the generator:

```
# Initialize the list
my_list = [1, 3, 6, 10]

a = (x**2 for x in my_list)
print(next(a))
```

```
print(next(a))

print(next(a))

print(next(a))

next(a)
```

When we run the above program, we get the following output:

```
1
9
36
100
Traceback (most recent call last):
  File "<string>", line 15, in <module>
StopIteration
```

Generator expressions can be used as function arguments. When used in such a way, the round parentheses can be dropped.

```
>>>sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

**Use of Python Generators**

There are several reasons that make generators a powerful implementation.

**1. Easy to Implement**

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2 using an iterator class.

```
Class PowTwo:

    def __init__(self, max=0):
```

```
        self.n = 0
        self.max = max

    def __iter__(self):
        return self

    def __next__(self):
        ifself.n>self.max:
        raiseStopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

The above program was lengthy and confusing. Now, let's do the same using a generator function.

```
defPowTwoGen(max=0):
    n = 0
while n < max:
        yield 2 ** n
        n += 1
```

Since generators keep track of details automatically, the implementation was concise and much cleaner.

**2. Memory Efficient**

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill, if the number of items in the sequence is very large.

Generator implementation of such sequences is memory friendly and is preferred since it only produces one item at a time.

**3. Represent Infinite Stream**

Generators are excellent mediums to represent an infinite stream of data. Infinite streams cannot be stored in memory, and since generators produce only one item at a time, they can represent an infinite stream of data.

The following generator function can generate all the even numbers (at least in theory).

```
defall_even():
    n = 0
    while True:
        yield n
    n += 2
```

4. Pipelining Generators

Multiple generators can be used to pipeline a series of operations. This is best illustrated using an example.

Suppose we have a generator that produces the numbers in the Fibonacci series. And we have another generator for squaring numbers.

If we want to find out the sum of squares of numbers in the Fibonacci series, we can do it in the following way by pipelining the output of generator functions together.

```
def fibonacci_numbers(nums):

    x, y = 0, 1
    for _ in range(nums):
        x, y = y, x+y
        yield x
```

```
def square(nums):
        for num in nums:
                yieldnum**2

print(sum(square(fibonacci_numbers(10))))
```

Output

4895

This pipelining is efficient and easy to read (and yes, a lot cooler!).

# ***Thank You…***