Python

Introduction

Python was introduced by **Guido van Rossum in the year of 1991**.

Before that he already worked in **ABC Programming language**.

It was named as python because he was inspired by **Monty Python's Flying Circus BBC Comedy show.**

Currently we have used Version - **Python 3.14.**

Python language is being used by almost all tech-giant companies like – **Google, Amazon, Face book, Instagram, Drop box, Uber… You Tube** etc. .

Features of Python:

- less code do more
- Object Oriented Programming Language
- Interpreted Programming language
- Multi - Purpose Programming Language
- Multi - Paradigm (procedural, functional, Oops)
- Open source & Wide range of libraries available
- works on all platform (windows, mac, Linux, raspberry, pi)
- Simple syntax similar to English
- High level programming language

Uses of Python:

Software development,

web development,

Artificial Intelligence,etc. .

Installation:

IDLE - Integrated Development Learning Environment

Basic Print Function:

Print(“WELCOME TO SDLC”)

Two modes in Python:

Shell mode - to check output

Script mode - to write code

Variables:

containers for storing data values / information.

Python has no command for declaring a variable.

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

Python is a case sensitive language.

Eg:

a = 10

A = 15

two different variables, A will not overwrite a

Rules for variables:

Can have short name(x & y) or descriptive name(age & mark)

Must start with alphabets / underscore.

Cannot start with number.

Variable names are case sensitive(name, Name, NAME).

A variable name cannot be python keywords.

Can have numbers.

Comments:

Used to explain py code.

Used to make the code more readable.

And used to explain the code and convey some information about the code to the developer.

- single line comment

no spcl syntax for multi line comments, but we can use triple single / double quotes(when not assigned to a variable).

Note:

In other programming Language indentation is for readability, but in python it is very important for the execution of the code.

Data Types:

Integer – It contains the whole number 0 to etc.. .

e.g.: 55,-7,7845121,0

Float – contains negative or positive values with one or more decimal / pointed values

e.g.: z = 1.0, w = -35.59

String - sequence of characters enclosed within either single / double quotes

we use triple single / double quotes for multi line variables

“Karthi”

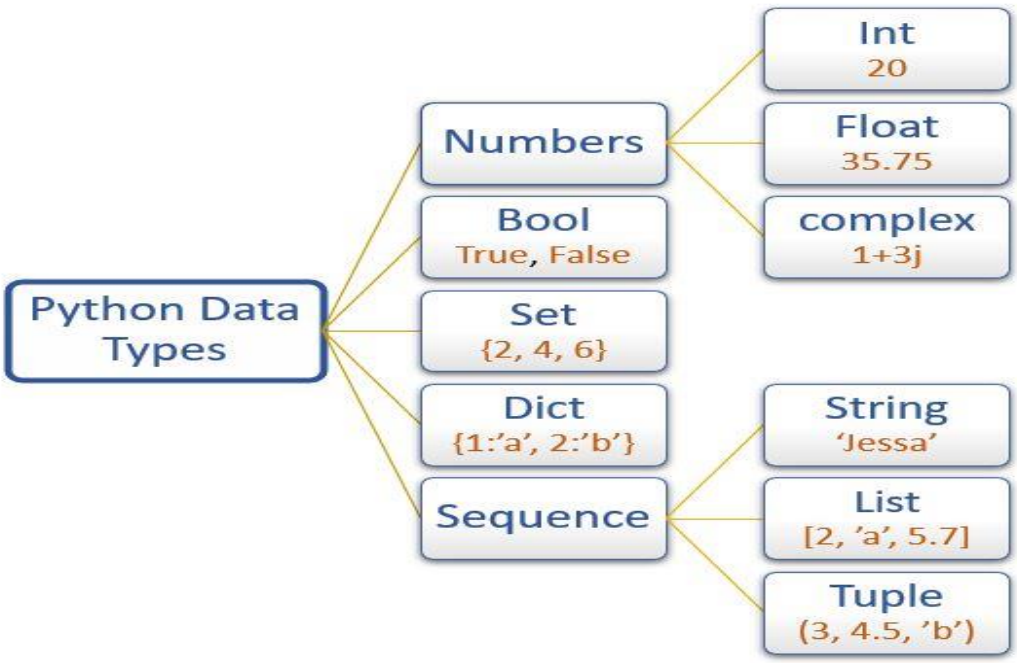
Boolean - True - 1, False - 0

print(10 > 9)

Any value is evaluated to True if it has some sort of content.

It checks the value and returns in Boolean Values.

Find the datatype -> we use type(variable)



DataType Category	DataType	Stores	
Text-Type	str	strings (can be declared using single quotes or double quotes)	name =
Numeric-Type	int	integers	val = 1
	float	floating point decimals	val = 2.3 val = 0.4
	complex	complex numbers	a = 2 b = 3 c = com => c = 2
Sequence-Type	list	* similar / different elements in between [] * can have repetitive elements	x = ['ap
	tuple	* ordered collection of elements stored in () * cannot contain repetitive elements	x = (1, 2
	range	range of values	x = rang => x = ra
Mapping-Type	dict	values in the form of key-value pairs	dict = {'
Set-Type	set	unordered collection of elements stored in ()	x = set(
	frozenset	immutable sets	x = fron
Boolean-Type	bool	boolean values of True / False	x = True
Binary-Type	bytes	* immutable sequence of small integers in the range 0 <= x < 256 * printed as ASCII characters when displayed * provide functions to encode and decode strings	x= b"He => b'He
	bytearray	* a mutable sequence of integers in the range 0 <= x < 256 * provide functions to encode and decode strings	x = byte => byte
	memoryview	allow Python code to access the internal data of an object that supports the buffer protocol without copying	x = mer => <me

Operators:

Operators are used to **perform operations on variables and values**

1)Arithmetic operators +, -, *, /, %,

Arithmetic Operator or used to Perform Mathematical Operations in between the values and the variables

2)Assignment operators =, +=, -=, *=, /=, %=

Assignment operator are used to assign the value to the variables

```
a, b = 4, 3
c = a + b
a += 2
print(c)
```

3)Comparison operators

Comparison Operator are used to compare the two or more values or values and the variables.

If the values are compared it will return the output in Boolean values.

== ---- Equal to

!= ---- Not Equal To

>-------Greater than

< -------Lesser Than

>= ---- Greater than Or Equal To

<= Less Than or Equal to

4)Logical operators

and - Returns true if both operands are true

e.g.: x = (5 > 3 and 2 < 4) # True

T T = T

T F = F

F T = F

F F = F

or - Returns True if at least one of the operands is true

e.g.: x = (5 > 3 or 2 > 4) # True

T T = T

T F = T

F T = T

F F = F

not - Reverses the result / boolean value of the operand

e.g.: x = not (5 > 3) # False

5)Identity operators

Used to compare the memory locations of two objects.

They determine whether two variables reference the same object in memory.

is e.g.: a = [1,2,3,4]

b = [1,2,3,4]

c = a

print(a is c) # True

print(a is b) # False

is not

6) Membership operators

used to test if a value is a member of a sequence

in e.g.: text = "Hello, World!"

print('H' in text) # True

print('hello' in text) # False (case-sensitive)

e.g.: numbers = [1, 2, 3, 4, 5]

print(3 in numbers) # True

print(6 in numbers) # False

not in e.g.: text = "Hello, World!"

print('H' not in text) # False

```
print('hello' not in text)  # True (case-sensitive)
```

7) Bit-wise operators

used to compare (binary) numbers

Bit - smallest storage storage unit in computer memory. 8 bit = 1 byte, KB, MB, GB

AND - &

OR - |

XOR - ^

NOT - ~ (unary operator, only one operand require to perform)

Left Shift - <<

Right shift - >>

```
a,b = 2,3
print(a & b)
print(a | b)
print(a ^ b)
print(~a)
print(a << 1)
Print(a >> 1)
```

print(a & b)

set it to 1 if both are 1, otherwise it is set to 0

```
0010
0011
-------
0010 - 2
```

print(a | b)

set it to 1 if one or both is 1, otherwise it is set to 0

```
0010
0011
```

0011 - 3

print(a ^ b)

^ - Sets each bit to 1 if only one of two bits is 1

C1C2R

0 0 0

1 1 0

0 1 1

1 0 1

0010

0011

0001 = 1

print(~3)

The ~ operator inverts each bit (0 becomes 1 and 1 becomes 0)

-4

print(2 << 1)

0010 = 2

0100 = 4

print(2 >> 1)

0010 = 2

0001 = 1

User Input

User input is used to give the input from the variable and to display the output for the given input.

User input is used to give the input from the user and used to display the output to the user.

Example

```
print("Welcome")
name=input("Enter Your Name:")
age=input("Enter Your Age:")
City=input("Enter Your City:")
print("Your Name is: ",name)
print("Your Age is: ",age)
print("Your City is: ",City)
print("Thanks for Using")
```

Type Casting:

Casting in Python, also known as type conversion, is the process of converting one data type into another data type.

User Input:

```
a = input()
print(a)
a = input()
b = input()
print(a + b)
```

e.g.: 10, 20 = 1020

Task - BMI Calculator

```
weight = int(input("Enter Your Weight in Kg: "))
height = float(input("Enter Your Height in Meter: "))
BMI = weight/(height * height)
print(BMI)
```

Task

You have to get the user input for the Name, Age, School or College, Mark1 to Mark5 and find the total mark and Average for the student.

Control Flow Statements:

Python has six conditional statements that are used to make decisions based on variable values or comparison results:

If

Executes a block of code if a condition is true. The syntax is if keyword (condition). If will be executed as the given condition is true and if you already know the condition is true you can use the if statement you need to check the output based on the condition and you don’t know what the condition is you can use else statement.

Else

Executes a different block of code if the if condition is false.

Elif

Checks for multiple conditions and executes the code block if any of the conditions are true. The keyword "elif" stands for "else if". For example, if the first condition is false, it moves on to the next "elif" statement to check if that condition is true.

If-elif-else ladder

An order of if statements connected by elif statements that allows you to check for numerous conditions and run separate code blocks based on which condition is met.

Conditional expression

Also known as a "ternary operator", this is a simplified, single-line version of an if-else statement.

1) Conditional Branching

    simple if statement

    If else statements

    elif statements

    nested if statements

2) Conditional Looping

    For Loop

While Loop

If else Task:

Find the number odd or even

Get input for variable age

get input for variable income

1 – 18 => not eligible

18 – 25 => eligible , 5000 – 10000 => eligible for loan 10500

26 – 30 => eligible , 10001 – 25000 => eligible for loan 26500

25001 – 40000 => eligible for loan 50000

40000 above not eligible

50 age above

Check the person for the loan approval

Write a program that checks if a person is eligible to vote (age 18 or older).

Write a program that assigns a grade based on a score out of 100.

score > 70, get input for name, dept, location, print "you are eligible", not eligible

ELIF: TASK

make a mini calculator

Write a program that checks if a number is positive, negative or zero

AND:

Check the number if it is divisible by 3 & 5

NESTED IF:

Nested means within. Nested if condition means if-within-if. Nested if condition comes under decision-making statement in Java. There could be infinite if conditions inside an if condition. The below syntax represents the Nested if condition

score<35 - poor, score>35 & <70 - average, score > 70 - good

OR & Nested if:Task

Get input for salary & age. If salary >= 30k or age <= 25, get input for required loan amount.

if not print your are not eligible loan <= 50k, you are eligible for the loan, loan> 50k, max loan amt. is 50k

Write a program that takes a person's age as input and prints the age category:

Child: 0-12

Teenager: 13-19

Adult: 20-64

Senior: 65 and above

Take age as input from the user

age = int(input("Enter your age: "))

Determine the age category using nested if statements

FOR LOOP:

A for loop in Python is used to iterate over a sequence (such as a list, tuple, dictionary, set, or string).A for loop is a control flow statement in programming that iterates a specific number of times. It allows you to execute a block of code repeatedly, each time with a different value known as the loop variable. The for loop is particularly useful when you know before how many times you need to execute a block of code.It is usually used when the number of iteration is known.

Syntax:

for variable in iterable:

Looping through a string:

```
for letter in 'Python':
    print(letter)
```

Printing using End

```
for letter in 'Python':
    print(letter,end="")
```

Using range function:

```
for i in range(0,30,2):
    print(i)
```

Breaking out of a Loop:

```
for i in range(10):
    if i == 5:
        break
    print(i)
```

Skipping an Iteration:

```
for i in range(10):
    if i == 5:
        continue
    print(i)
```

Pass Keyword:

```
for z in "apple":
  Pass
```

Task:

1. Print 2 table using for loop
2. print even numbers between 1 to 10
3. Get input for a & b variables. Print the number between a & b
4. print odd numbers between 1 - 10
5. Count the number of even number between 1 to 10

```
for i in range(1,11):
   if(i%2==0):
     e_count = e_count+1
   else:
     o_count = o_count+1
print(e_count)
print(o_count)
```

NESTED FOR LOOP:

Python programming language there are two types of loops which are for loop and while loop. Using these loops we can create nested loops in Python. Nested loops mean loops inside a loop. For example, while loop inside the for loop, for loop inside the for loop

Program

Square pattern:

```
n = 5
for i in range(n):
    for j in range(n):
        print("*", end=" ")
    print()
```

While Loop:

A while loop is another type of control flow statement that repeatedly executes a block of code as long as a specified condition is true.

Syntax:

```
//initialization
While(condition):
    #block of code
    //increment / decrement
end
```

while condition:

Code block to be repeated.This block will continue to execute as long as the condition is true.

Write a Python program using a while loop to print numbers from 0 to 5 sequentially

```
i = 0
while(i<=5):
    print(i)
    i += 1
```

Write a Python program using a while loop to print multiples of 5 starting from 0 up to 100. Your program should incrementally print each number until reaching 100

```
i = 0
while(i<=100):
    print(i)
    i += 5
```

Write a program to print first 10 natural numbers in reverse order

```
i = 10
while(i>0):
    print(i)
    i = i - 1
```

Using else with while Loop:

The else block is executed when the loop condition becomes False, unless the loop is terminated by a break statement.

Example using else with while loop

```
i = 1
while i <= 5:
    print(i)
    i += 1
else:
    print("Loop completed successfully!")
```

Loop Control Statements (continue and break):

Continue skips the code in the current iteration and moves to the next iteration.

Break terminates the loop prematurely.

Example using continue and break

```
i = 0
while i < 10:
    i += 1
    if i % 2 == 0:
        continue  # Skip even numbers
    if i == 7:
        break  # Stop loop at 7
    print(i)
```

Functions:

A function in Python is a block of organized, reusable code that performs a specific task. Functions help break our program into smaller and modular chunks, making it easier to manage and reuse code. Functions can take inputs, called arguments, and can return outputs.

Syntax:

```
def function_name():
```

Components of a Function

Function Name: A unique name that identifies the function. Function names should follow the same rules as variable names.

Parameters: Variables that the function uses to receive inputs.

Function Body: The block of code that runs when the function is called. This part can contain any valid Python code.

Return Statement: The return statement is used to send a value back to the caller. If no return statement is used, the function returns None by default.

Types of Functions

Built-in Functions: Python provides many built-in functions like print(), Len(), type() etc. .

User-Defined Functions: Functions that users create to perform specific tasks.

Lambda Functions: Type of function that doesn't have any function name.

```
def my_function():
    print("Hello Python")
my_function()
def addition(a, b):
    return a + b
def subtraction(a, b):
    return a - b
print("Addition Result:", addition(10, 5))
print("Subtraction Result:", subtraction(10, 5))
```

If else in Function:

```
a = int(input("Enter a number: "))
def numcheck():
    if a % 2 == 0:
        print("Even")
    else:
        print("Odd")
numcheck()
```

```
mark = int(input("Enter your score: "))
def check():
    if(mark>=35):
        print("You are pass")
    else:
        print("You are fail")
check()
```

Looping in function:

```
def range_function(a, b):
    for i in range(a, b):
        print(i)
a_input = int(input("Enter a number: "))
b_input = int(input("Enter another number: "))
range_function(a_input, b_input)
```

Types of Arguments in python:

Default

```
def myfun(name=""):
    print(name)
myfun()
myfun("SDLC")
```

Positional

Keyword **arg , arg[“fname”]

Arbitrary *arg , arg[0]

Default Arguments:

Function with default values for its parameters. This means if you don't provide a value for a parameter when calling the function, the default value will be used.

```
def greet(name, age=25):
```

```
    print(f"Hello, my name is {name} and I am {age} years old.")
greet("Alice")
greet("Bob", 30)
#Positional Arguments:
def hello (name,subject,depth="Cs"):
  print("Hi",name)
  print("do you teach", subject)
  print("are you from",depth)
hello("Tharun","Python")
#Positional Arguments:
a1 = input("Enter your name: ")
a2 = input("Enter your subject: ")
def hello(name,subject,depth="CS"):
  print(f"Hi, {name}")
  print(f"Do you teach {subject} Language")
  ans = input("Enter your answer?")
  if(ans=="yes"):
    print("You can join session from tomorrow...")
  else:
    print("You may exit now")
  print(f"Are you from {depth}")
  ans2 = input("Enter your answer: ")
  if(ans2 == "yes"):
    print("Okay ")
  else:
    print("you need to attend Cs basics class")
hello(a1,a2)
```

```
#Keyword Arguments
def student_info(name, age, grade, school="High School"):
    print(f"Name: {name}")
    print(f"Age: {age}")
    print(f"Grade: {grade}")
    print(f"School: {school}")
# Using keyword arguments
student_info(name="Alice", age=16, grade="10th")
student_info(name="Bob", age=17, grade="11th", school="Senior High School")
```

Arbitrary Arguments:

Arbitrary arguments allow a function to accept an indefinite number of arguments. This is useful when you don't know in advance how many arguments will be passed to the function.

```
def function_name(*students):
    print("The first rank student is " + students[2])  # Accessing the third student (index 2)
def add(*numbers):
    c = 0
    for i in numbers:
        c = c + i
    print(f"Sum is {c}")  # Print the result inside the function
function_name("Tharun", "Shasti", "Perarasu")
add(5, 7, 9)
add(1, 2, 3, 4, 5, 6, 7, 8, 9)
```

****kwargs (Keyword arguments):**

```
# Function to display the last name
def my_function(**kid):
    print("His last name is " + kid["l_name"])
```

```
my_function(f_name="Tharun", l_name="Hair")
def person(name, **data):
    print(name)
    for i, j in data.items():
        print(i, j)
person("Tharun", age=17, city="NKL", mbl=785491230)
```

Swapping 2 variables in Python:

```
a = 5
b = 6
print(a)  # 5
print(b)  # 6
a = b    # a becomes 6
b = a    # b remains 6, as `a` is already updated to 6
print(a)  # 6
print(b)  # 6
a = 5
b = 6
print(a)  # 5
print(b)  # 6
temp = a  # temp stores 5
a = b     # a becomes 6
b = temp  # b becomes 5 (original value of `a`)
print(a)  # 6
print(b)  # 5
a = 5
b = 6
a = a + b  # a becomes 11
```

```
b = a - b  # b becomes 5 (11 - 6)
a = a - b  # a becomes 6 (11 - 5)
print(a)  # 6
print(b)  # 5
```

Return Keyword in Python:

The return keyword is used in functions to send a value back to the caller.When encountered a return keyword, it exits the function immediately and returns the specified value or None if omitted.

```
def add_numbers(a, b):
    result = a + b
    print(result)  # Print the sum
    return result  # Return the sum
result = add_numbers(3, 5)
print("Sum:", result)  # Output: Sum: 8
```

Lambda Functions:

Lambda functions in Python, also known as anonymous functions, are small, unnamed functions defined using the lambda keyword.

They are often used for short, simple operations that are not complex enough to require a full function definition.

Syntax:

```
lambda arguments: expression
```

```
def square(a):
    return a * a
result = square(5)
print(result)
f = lambda a: a*a
result = f(5)
print(result)
```

```
add = lambda x, y: x + y
print(add(2, 3))  # Output: 5
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Math Functions in Python:

```
import math
print(math.ceil(4.7))       # Output: 5
print(math.floor(4.7))      # Output: 4
print(math.factorial(5))    # Output: 120
print(math.pow(2, 3))       # Output: 8.0
print(math.sqrt(16))        # Output: 4.0
print(math.cbrt(8))         # output: 2.0
print(math.pi)              # output:3.14
import math
help(math)
import keyword
# List all Python keywords
print(keyword.kwlist)
# Check if a string is a keyword
print(keyword.iskeyword('if'))  # Output: True
print(keyword.iskeyword('hello'))  # Output: False
```

```
   print(result)
fact(5)
```

Data Structures:

Data structures are a way of organizing and storing data so that it can be accessed and modified efficiently.

List[], Tuple(), Set{}, Dictionary{}

What is a List in Python?

A list in Python is a **collection of items** that can be of different types (e.g., **integers, strings, or even other lists***)*. Lists are ordered, changeable (mutable), and allow duplicate values.

Ordered - If you add new items to a list, the new items will be placed at the end of the list.

Changeable - we can change, add, and remove items in a list after it has been created.

list - [] ,changeable, ordered, allows duplicate, indexed

creating a list:

```
numbers = [1, 2, 3, 4, 5]
fruits = ["apple", "banana", "cherry"]
mixed = [1, "hello", 3.14, True]
```

Accessing List Elements:

```
print(numbers[0])
```

Modifying List Elements:

```
numbers[0] = 10
print(numbers)
```

Adding Elements to a List:

```
numbers.append(6)
print(numbers)
numbers.insert(1, 20)
print(numbers)
numbers.extend([7, 8, 9])
print(numbers)
```

Removing Elements from a List:

```
numbers.remove(20)
print(numbers)
```

```
numbers.pop(0)
print(numbers)
del numbers[0]
print(numbers)
del numbers
print(numbers)
```

Slicing Lists:

```
print(numbers[1:4])
```

len(list) : Returns the number of elements in the list.

list.sort(): Sorts the list in ascending order.

list.reverse(): Reverses the elements of the list.

list.index(element): Returns the index of the first occurrence of the element.

list.count(element): Returns the number of times the element appears in the list

unpacking a list:

```
my_list = [1, 2, 3]
a, b, c = my_list
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
my_list = [1, 2, 3]
a = my_list[0]
b = my_list[1]
c = my_list[2]
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

Tuple:

It is an ordered collection of elements, which can be of different types, and is immutable. This means that once a tuple is created, its contents cannot be changed. Tuples are defined by placing elements inside parentheses (), separated by commas.

tuple = (), unchangeable, ordered, allows duplicate, indexed

Creating a tuple:

```
this_tuple = ("apple", "banana", "cherry")
print(len(this_tuple))
print(type(this_tuple))
```

Accessing Tuple Elements:

```
my_tuple = (10, 20, 30, 40)
print(my_tuple[0])  # Output: 10
print(my_tuple[2])  # Output: 30
```

Tuple Unpacking:

```
my_tuple = (1, 2, 3)
a, b, c = my_tuple
print(a)  # Output: 1
print(b)  # Output: 2
print(c)  # Output: 3
```

Tuple Methods:

```
my_tuple = (1, 2, 3, 2, 2)
print(my_tuple.count(2))  # Output: 3
print(my_tuple.index(3))  # Output: 2
```

Dictionary:

A dictionary in Python is an ordered collection of items. Each item is a key-value pair. Dictionaries are defined by enclosing a comma-separated list of key-value pairs in curly braces {}.

dict = {},changeable, ordered, allow duplicate values(*but not keys),non - indexed

Creating a Dictionary

```
my_dict = {
   "name": "John",
   "age": 25,
   "city": "New York"
}
```

Accessing a value

```
print(my_dict["name"])
```

Using get() method

```
print(my_dict.get("name"))
```

Modifying an existing key-value pair

```
my_dict["age"] = 26
print(my_dict)
```

Adding a new key-value pair

```
my_dict["email"] = "John@example.com"
print(my_dict)
```

File Handling in Python:

File handling in Python refers to the process of interacting with files (such as reading and writing the files) using Python's built-in functions and methods

Basic Operations in File Handling:

Opening a File: Using the open() function to access a file.

Reading from a File: Using methods like read(), read line(), or read lines() to read data.

Writing to a File: Using methods like write() or write lines() to write data.

Closing a File: Using the close() method to release the file resource.

Using Context Managers: Using the with statement to handle files which ensures that the file is properly closed after its suite finishes, even if an exception is raised.

Practical Applications

Storing User Data: Saving user preferences, game scores, or any persistent user-specific data.

Processing Large Data Sets: Reading data from files for processing (e.g., large datasets in data science).

Automation Scripts: Storing logs and outputs of automation scripts.

Web Development: Managing content files, configuration files, or templates in web applications.

Opening a file:

'''

There are two path one is absolute and relative path relative path we are using

'''

```
file = open('Karthi.txt','r')#path of the file
print(file.read())
file.close
```

Read mode

write mode

File Reading

```
# reading Program
'''
There are two path one is absolute and relative path, here we are using relative path.
'''
# Open the file in read mode
file = open('Karthi.txt', 'r')  # path of the file
print(file.read())  # Reading the entire content of the file
```

```
file.close()  # Closing the file
# Read line by line using read line() method
file = open('Karthi.txt', 'r')  # Reopen the file in read mode
print(file.readline())  # Read one line at a time
file.close()  # Close the file
file = open('Karthi.txt', 'r')  # Reopen the file
print(file.readlines())  # Read all lines as a list
file.close()  # Close the file
file = open('Karthi.txt', 'r')  # Reopen the file
print(file.read(5))  # Read first 5 characters
file.close()  # Close the file
```

Writing in a file:

```
f = open("doc.txt", "w")
f.write("world")  # Write "world" in the file
f.close()  # Close the file
f = open("doc.txt", "w")
f.write("world")
f.write("python")  # Append "python" after "world" (but it overwrites content)
f.close()  # Close the file
```

Writing lines with newline characters

```
f = open("text/doc.txt", "w")
f.write("world\n")  # Writing "world" with a newline after it
f.write("python")  # Writing "python" after the new line
f.close()  # Close the file
```

Reading and writing to the same file

```
f = open("doc.txt", "r+")
print(f.read())  # Read the content of the file
```

```
f.close()  # Close the file
```

OOPS in python

In Python, an object is a fundamental concept representing a collection of data (attributes) and methods (functions) that operate on that data. Everything in Python is an object, including basic data types like integers, strings, lists, and more complex user-defined classes.

Key Concepts:

Attributes: These are variables that belong to an object. They store information about the object.

Methods: These are functions that belong to an object and can operate on its attributes or perform actions.

Class: A class is a blueprint for creating objects. It defines a set of attributes and methods that the created objects will have.

Instance: An instance is a specific object created from a class. It has its own attributes and methods defined by the class.

Class

```
# Define a class named 'car'
class Car:
    pass
# Variable and object creation
a = 10
swift = Car()
# Check instance types
print(isinstance(swift, Car))  # True
print(isinstance(a, int))      # True
print(type(swift))             # <class '__main__.Car'>
# Define a class named 'Student'
class Student:
```

```
    name = "Karthi"
    age = 26
# Using getattr to access attributes
print(getattr(Student, 'name'))# Karthi
print(getattr(Student, 'age')) # 26
print(getattr(Student, 'gender', 'No such attribute found'))

# Access attributes using dot notation
print(Student.name)  # Karthi
print(Student.age)   # 26
# Set attributes using setattr
setattr(Student, 'name', 'Karthikeyan')
print(Student.name)  # Karthikeyan
# Add a new attribute if it doesn't exist
setattr(Student, 'gender', 'Male')
print(Student.gender)  # Male
# Delete an attribute using del
delattr(Student, 'gender')
# Accessing 'gender' now will raise an Attribute Error:
print(Student.gender)
```

We can also delete a attribute using class also using

del(student.gender)

Python is an object oriented programming language.Almost everything in Python is an object, with its properties and methods.A class in Python is like a blue print for creating objects.

```
print(o1.course)
```

Class Method

The class method() is an inbuilt function in Python, which returns a class method for a given function. This means that class method() is a built-in Python function that transforms a regular method into a class method. When a method is defined using the @class method decorator (which internally calls class method()), the method is bound to the class and not to an instance of the class. As a result, the method receives the class (cls) as its first argument, rather than an instance (self).

Without any object we can call

```
# Class Method Example

class Students:

    name = "Karthi"

    age = 26

    @static method  # Decorator for static methods (no self parameter required)

    def print_all():

        print("Name : ", Students.name)

        print("Age  : ", Students.age)

Students.print_all()

print(Students.__dict__)

Students.__dict__['print_all']()
```

Instant Method in Python

```
class Students:

    name = "Karthi SK"

    age = 26

    def print_all(self, gender):

        print("Name : ", Students.name)

        print("Age  : ", Students.age)

        print("Gender:", gender)

o = Students()

o.print_all("Male")
```

Another Example

```
class Car:
    # Correcting the __init__ method (Constructor)
    def __init__(self, make, model, year):
        self.make = make        # Instance attribute
        self.model = model      # Instance attribute
        self.year = year        # Instance attribute
    def display_info(self):
        print(f"{self.year} {self.make} {self.model}")
my_car = Car("Toyota", "Corolla", 2020)
print(my_car.make)   # Output: Toyota
print(my_car.model)  # Output: Corolla
print(my_car.year)   # Output: 2020
my_car.display_info()  # Output: 2020 Toyota Corolla
```

in it Method here we call in it it is known as Constructor

Program

```
class User:
    def __init__(self, name):
        # Initialize the instance with the name parameter
        self.name = name
    def print(self):
        # Print the name attribute
        print("Name:", self.name)
# Create an instance of the User class with the name parameter
o1 = User("Karthikeyan")
# Call the method to print the name
o1.print()
```

car:

property -

4 tyres

20km / L

behaviour:

accelerate

brake

clutch

reverse

by using procedural / functional

we need to create lots of variable and functions(turn left(), reverse())

Self Keyword:

The self keyword in Python is a reference to the current instance of the class. It is used within a class to access the instance's attributes and methods.

What is a Class?

A class is a blueprint for creating objects (instances). A class defines a set of attributes and methods that the objects created from the class can have.

What is an Object?

An object is an instance of a class. When you create an object, you instantiate a class, meaning you create an instance of that class with specific data.

Encapsulation:

Encapsulation is a fundamental concept in object-oriented programming (OOP) that refers to the **binding of data (attributes) and methods (functions)** that operate on that data within **a single unit or class**. In Python, encapsulation helps to achieve two primary goals:

Data Hiding: Encapsulation allows for restricting access to certain components of an object, thereby protecting the internal state of the object from unintended interference and misuse.

Controlled Access: By exposing **only the necessary parts** of an object (through public methods), encapsulation allows for controlled access to the object's data.In simple words, Encapsulation is the concept of **restricting** access to certain attributes or methods within a class.

Inheritance:

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows a class (called the child or subclass) to inherit attributes and methods from another class (called the parent or superclass). This promotes code reusability and establishes a relationship between classes.

Superclass (Parent Class) or (Base Class)

The class whose properties and methods are inherited.

Subclass (Child Class): or (Derived class)

The class that inherits properties and methods from the superclass.

Types of Inheritance-

- Single Inheritance.
- Multiple Inheritance.
- Multilevel Inheritance.
- Hierarchical Inheritance.
- Hybrid inheritance.

Single inheritance

Single inheritance enables a derived class to inherit properties from a single parent class, thus enabling code reusability and the addition of new features to existing code.

Program

```
class Parent:
    def func1(self):
        print("This function is in parent class.")
# Derived class
class Child(Parent):
    def func2(self):
```

```
        print("This function is in child class.")
# Driver's code
object = Child()
object.func1()
object.func2()
```

Multiple Inheritance

When a class can be derived from more than one base class this type of inheritance is called multiple inheritances. In multiple inheritances, all the features of the base classes are inherited into the derived class.

Program

```
# Base class1
class Mother:
    def __init__(self):
        self.mother_name = ""
    def mother(self):
        print("Mother's name:", self.mother_name)
# Base class2
class Father:
    def __init__(self):
        self.father_name = ""
    def father(self):
        print("Father's name:", self.father_name)
# Derived class
class Son(Mother, Father):
    def __init__(self):
        # Initialize parent classes
        Mother.__init__(self)
        Father.__init__(self)
```

```
    def parents(self):
        print("Father:", self.father_name)
        print("Mother:", self.mother_name)
# Driver's code
s1 = Son()
s1.father_name = "RAM"
s1.mother_name = "SITHA"
s1.parents()
```

Multilevel Inheritance

In multilevel inheritance, features of the base class and the derived class are further inherited into the new derived class.

Program

```
# Base class
class Grandfather:
    def __init__(self, grandfather_name):
        self.grandfather_name = grandfather_name
# Intermediate class inheriting from Grandfather
class Father(Grandfather):
    def __init__(self, father_name, grandfather_name):
        self.father_name = father_name
        # invoking constructor of Grandfather class
        super().__init__(grandfather_name)
# Derived class inheriting from Father
class Son(Father):
    def __init__(self, son_name, father_name, grandfather_name):
        self.son_name = son_name
        # invoking constructor of Father class
        super().__init__(father_name, grandfather_name)
    def print_name(self):
        print('Grandfather name:', self.grandfather_name)
        print("Father name:", self.father_name)
        print("Son name:", self.son_name)
# Driver's code
s1 = Son('Prince', 'Rampal', 'Lal Mani')
print(s1.grandfather_name)
s1.print_name()
```

Hierarchical Inheritance:

When more than one derived class are created from a single base this type of inheritance is called hierarchical inheritance. In this program, we have a parent (base) class and two child (derived) classes.

Program

```
# Base class
class Parent:
    def func1(self):
        print("This function is in the parent class.")
# Derived class1
class Child1(Parent):
    def func2(self):
        print("This function is in child 1.")
# Derived class2
class Child2(Parent):
    def func3(self):
        print("This function is in child 2.")
# Driver's code
object1 = Child1()
object2 = Child2()
object1.func1()
object1.func2()
object2.func1()
object2.func3()
```

Hybrid Inheritance

Inheritance consisting of multiple types of inheritance is called hybrid inheritance.

Program

```
# Base class
class School:
```

```
    def func1(self):
        print("This function is in school.")
# Derived class 1
class Student1(School):
    def func2(self):
        print("This function is in student 1.")
# Derived class 2
class Student2(School):
    def func3(self):
        print("This function is in student 2.")
# Derived class 3 with multiple inheritance
class Student3(Student1):
    def func4(self):
        print("This function is in student 3.")
# Driver's code
obj = Student3()
obj.func1()
obj.func2()
obj.func4()
```

Polymorphism:

Polymorphism is a key concept in object-oriented programming that allows objects of different classes to be treated as objects of a common super class. It provides a way to perform a single action in different forms.

Method Overriding:

This occurs when a subclass provides a specific implementation of a method that is already defined in its superclass.When a method in a subclass has the same name, parameters, and return type as a method in its superclass, the subclass method overrides the superclass method.

Method Overloading:

Python doesn't support method overloading by default (i.e., defining multiple methods with the same name in the same class but with different parameters), but you can achieve similar behaviour by handling different types of arguments in the same method.Python does not support method overloading like some other languages (e.g., Java). However, you can emulate it by using default parameters or variable-length arguments.

Program

```
class Animal:
    def speak(self):
        raise NotImplementedError("Subclasses must implement this method")
class Dog(Animal):
    def speak(self):
        return "Woof!"
class Cat(Animal):
    def speak(self):
        return "Meow!"
class Cow(Animal):
    def speak(self):
        return "Moo!"
# List of animals
animals = [Dog(), Cat(), Cow()]
# Demonstrating polymorphism
for animal in animals:
    print(animal.speak())
```

Abstraction:

Abstraction in Python is a concept in object-oriented programming (OOP) that **involves hiding the implementation details of a function or class and showing only the essential features**.

Program

```
from abc import ABC, abstractmethod
```

```
# Abstract base class
class BankAccount(ABC):
    @abstractmethod
    def deposit(self, amount):
        pass
    @abstractmethod
    def withdraw(self, amount):
        pass
    @abstractmethod
    def get_balance(self):
        pass
# Concrete class for Savings Account
class SavingsAccount(BankAccount):
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited: {amount}")
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrawn: {amount}")
        else:
            print("Insufficient funds!")
    def get_balance(self):
        return self.balance
# Concrete class for Checking Account
```

```
class CheckingAccount(BankAccount):
    def __init__(self):
        self.balance = 0
    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited: {amount}")
    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
            print(f"Withdrawn: {amount}")
        else:
            print("Insufficient funds!")
    def get_balance(self):
        return self.balance
# Using the classes
accounts = [SavingsAccount(), CheckingAccount()]
# Performing operations
for account in accounts:
    account.deposit(100)
    print(f"Balance: {account.get_balance()}")
    account.withdraw(50)
    print(f"Balance: {account.get_balance()}")
    account.withdraw(100)  # Attempting to withdraw more than the balance
    print()
```