PYTHON - ADVANCED

MODULE - 2

Since we have the essential basics of python we will see some advanced concepts like Comprehensions, File handling, Regular Expressions, Object-oriented Programming, Pickling and many more essential concepts.

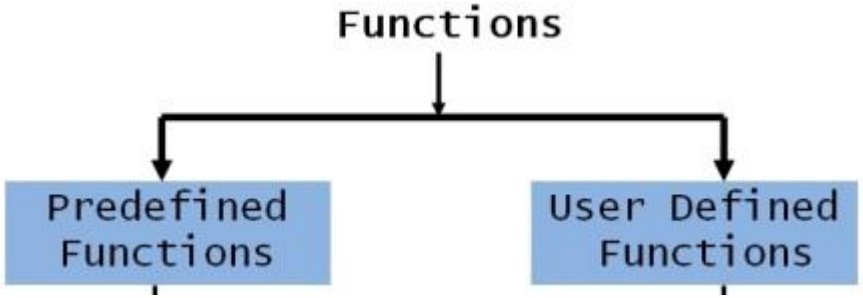
- Functions as Arguments
- List Comprehension
- File Handling
- Debugging in Python
- Class and Objects
- Lambda, Filters and Map

- Regular Expressions
- Python PIP
- Read Excel Data in Python
- Python MySQL
- Iterators
- Pickling
- Python JSON

WHY USE FUNCTIONS?

- **Functions serve two primary development roles**
  - Maximizing code reuse and minimizing redundancy
  - Procedural decomposition
- **Need for Functions**
  - Simplifies program development by making it easy to test separate functions.
  - Understanding programs becomes easier.
  - By dividing a large program into smaller functions, different programmers can work on different functions.
  - Users can create their own functions and use them in various locations in the main program.

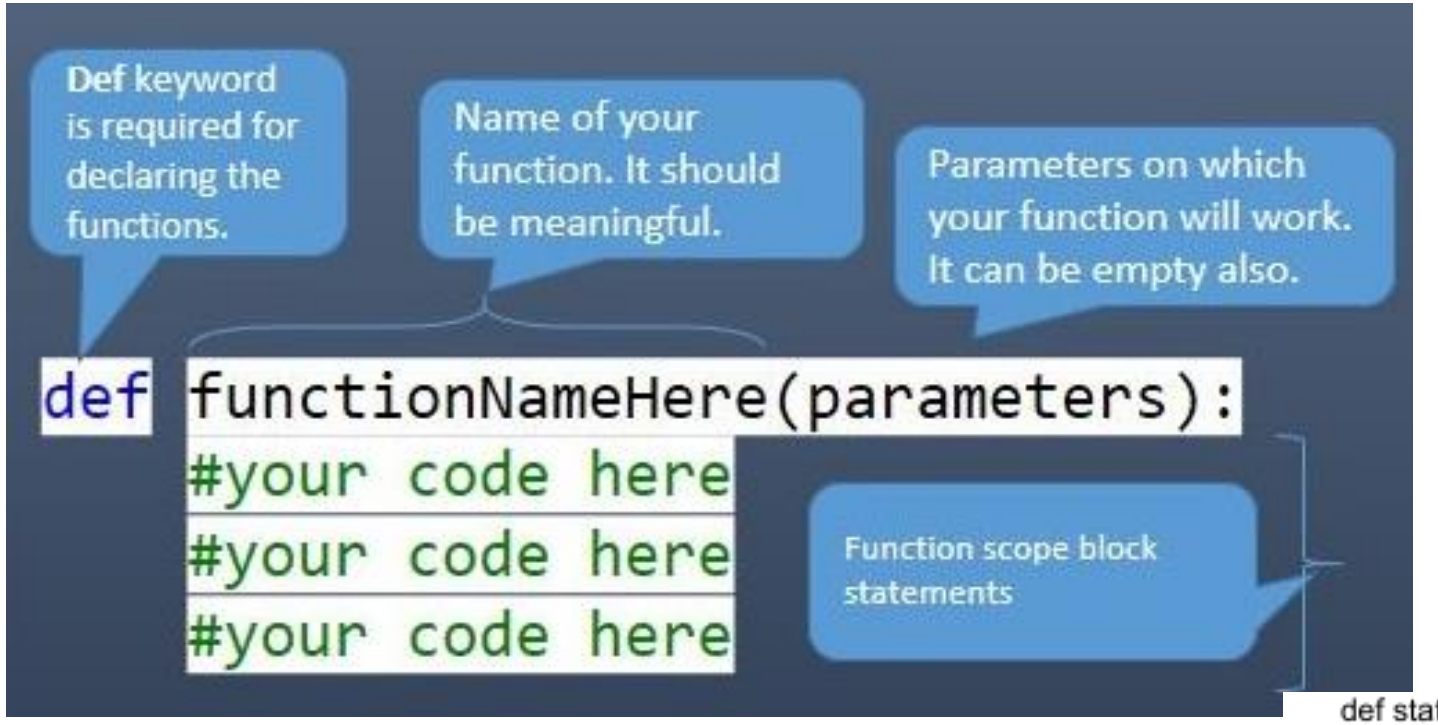
TYPES OF FUNCTIONS

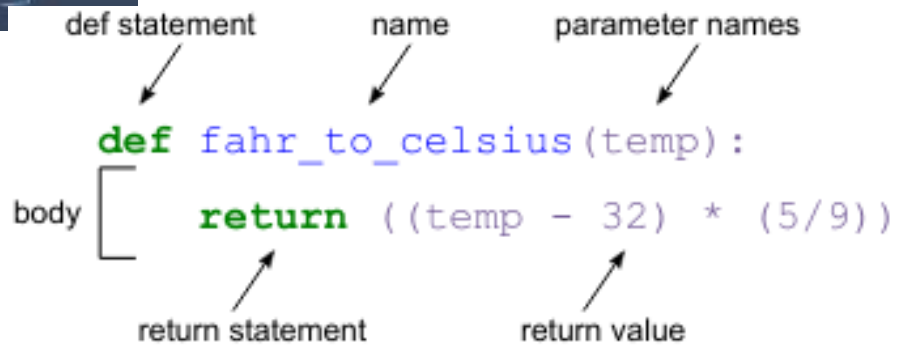


- Python support two types of functions
  1. Built-in function
  2. User-defined function

- Built-in function
  - The functions which come along with Python itself are called built-in functions or predefined functions.
  - eg: range(), id(), type(), input()
  - Refer slide …
- User-defined function
  - Functions which are created by programmer explicitly according to the requirement are called a user-defined function.

CREATING A FUNCTION





DEFINING A FUNCTION

- A function definition consists of a function header that identifies the function, followed by the body of the function.
- The body of a function contains the code that is to be executed when a function is called.
- To define a function, we have to remember following points:
  - Function definition starts with the keyword def
  - The keyword def is followed by the function name and parentheses.
  - After parentheses, a colon (:) should be placed.
  - Parameters or arguments that the function accepts should be placed inside the parentheses.
  - A function might have a return statement.
  - The function code should indent properly.

DEFINING A FUNCTION

- A function definition contains two parts:
  - Function header
  - Function body

- The syntax of a function definition:

```
def  function_name(arg1, arg2, ...):
     ["""documentation string"""]
     statement block
     return [expression]
```

- Example for defining a function

```
def  printstars():
     for i in range(1,101):
          print("*", end=' ')
```

CALLING FUNCTIONS

- Syntax for a function call

  function_name([arg1, arg2, ...])

- The arguments or parameters passed in a function call are called actual parameters.
- The arguments used in the function header are called formal parameters.
- Points to remember while calling
  - The function name and number of parameters must be same in the function call and function definition.
  - When the number parameters passed doesn't match with the parameters in the function definition, error is generated.
  - Names of arguments in the function call and function definition can be different.
  - Arguments can be passed as expressions. The expression will get executed first and then the value is passed to the formal parameter.
  - The parameter list must be separated by commas.
  - If the function returns a value it must be assigned to some variable in the calling function

FUNCTIONS RETURNING VALUE

- A function may or may not return a value.
- To return a value, we must use *return* statement in the function definition.
- Every function by default contains an implicit *return* statement as the last line which returns *None* object to the calling function.
- A return statement is used for two reasons:
  - Return a value to the caller.
  - To end the execution of a function and give control to caller.
- The syntax of *return* statement is as follows:

  return [expression]
- Example of a function returning a value is as follows:

```
def cube(x):
    return x*x*x
```

PASSING ARGUMENTS

- Key points in passing arguments to functions
  - Arguments are passed by automatically assigning objects to local variable names.
  - Assigning to argument names inside a function does not affect the caller
  - Changing a mutable object argument in a function may impact the caller.

```
>>> def f1(X):
        X=99


>>> K=100
>>> f1(K)
>>> K
100
```

```
>>> def f2(L):
        L.append(1)


>>> K=[1,2,3]
>>> f2(K)
>>> K
[1, 2, 3, 1]
```

KEYWORD ARGUMENTS

```
>>> def f(a,b,c):
        print(a,b,c)
```

```
>>> f(1,2,3)
1 2 3
```

- By default the arguments are positional….
- Keyword arguments are used to identify the arguments by parameter name
- Points to remember when using keyword arguments:
  - must match one of the parameters in the function definition.
  - Order of keyword arguments is not important.
  - A value should not be passed more than once to a keyword parameter.

```
>>> f(1,2,3)
1 2 3
>>> f(a=1, b=2, c=3)
1 2 3
>>> f(c=3, b=2, a=1)
1 2 3
>>> f(1, c=3, b=2)
1 2 3
>>> f(1,2,c=9)
1 2 9
>>> f(1,b=2)
Traceback (most recent call last):
  File "<pyshell#67>", line 1, in <module>
    f(1,b=2)
TypeError: f() missing 1 required positional argument: 'c'
```

DEFAULT ARGUMENTS

- The formal parameters in a function definition can be assigned a default value.
- parameters to which default values are assigned are called default arguments.
- allows a function call to pass less parameters than the number of parameters in the function definition
- Default value can be assigned to a parameter by using the assignment (=) operator
- should be at the end of the arguments list

```
>>> def f(a, b=2, c=3):
        print(a,b,c)


>>> f(1)
1 2 3
>>> f(a=1)
1 2 3
>>> f(1,3)
1 3 3
>>> f(1, c=6)
1 2 6
>>> f(2,3,4)
2 3 4
```

List Comprehensions

- Computed lists
- derive from set notations
- Coded in square brackets
- This method is used to create lists in concise way.

- Syntax:

  [expression for variable in sequence]

```
>>> L
['Strings', 'Lists', 'Python', 5.6, 123]
>>> [x for x in L]
['Strings', 'Lists', 'Python', 5.6, 123]
>>> [row[0] for row in M]
[1, 4, 7]
>>> #filter out odd items in a List
>>> L = list(range(1,11))
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> [x for x in L if x%2==1]
[1, 3, 5, 7, 9]
>>> #diagonal elements
>>> [M[i][i] for i in [0,1,2]]
[1, 5, 9]
>>> sum([M[i][i] for i in [0,1,2]])
15
```

Predict the output…

```
[2 ** x for x in range(10) if x > 5]
```

```
[x for x in range(20) if x % 2 == 1]
```

```
[x+y for x in ['Python ','C '] for y in
['Language','Programming']]
```

Built-in Functions

A
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

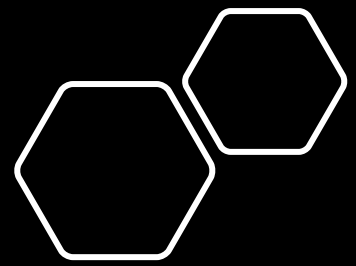
S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()



```
>>> L1 = [89, 34, 2, -1]
>>> L2 = ['Sai', 'Ram', 'Lakshmi', 1, 2, 3.4]
>>> len(L2)
6
>>> sum(L1)
124
>>> max(L1)
89
>>> min(L1)
-1
>>> sorted(L1)
[-1, 2, 34, 89]
>>> list("Bharat")
['B', 'h', 'a', 'r', 'a', 't']
```

```
>>> list(enumerate(L2))
[(0, 'Sai'), (1, 'Ram'), (2, 'Lakshmi'), (3, 1), (4, 2), (5, 3.4)]
>>> zip([1,2],['Ram','Laxman'])
<zip object at 0x000001B4945CD900>
>>> for item in zip([1,2],['Ram','Laxman']):
        print(item)


(1, 'Ram')
(2, 'Laxman')
```

ANONYMOUS FUNCTIONS: LAMBDA

- an expression form that generates function objects
- Because of its similarity to a tool in the Lisp language, it's called lambda
- Like def, this expression creates a function to be called later, but it returns the function instead of assigning it to a name.
- General form of lambda
- lambda argument1, argument2,... argumentN: expression using arguments

- Function objects returned by running lambda expressions work exactly the same as those created and assigned by defs, but there are a few differences that make lambdas useful in specialized roles
  - lambda is an expression, not a statement.
  - lambda's body is a single expression, not a block of statements
- Lambda is designed for coding simple functions, and def handles larger tasks.
- Consider a simple function

```
def func(x, y, z): return x + y + z
```

- Same effect can be achieved through a lambda expression by explicitly assigning its result to a name through which we can call the function later.

```
f = lambda x, y, z: x + y + z
f(2, 3, 4)
```

WHY USE LAMBDA?

- Lambdas are entirely optional
- a shorthand for def
- Allows you to embed a function's definition within the code that uses it
- Simpler coding constructs
- Lambda is also commonly used to code jump tables, which are lists or dictionaries of actions to be performed on demand.
- Lambdas can be nested within functions or lambdas

LAMBDA AS AN ARGUMENT TO AN HIGHER ORDER FUNCTION....

- Lambda functions are used along with built-in functions like filter(), map(), reduce()
- filter()
  - The filter() function in Python takes in a function and a list as arguments.
  - The function is called with all the items in the list and a new list is returned which contains items for which the function evaluates to True.

```
>>> # Program to filter out only the even items from a list
>>> L = [1,3,5,6,8,19]
>>> list(filter(lambda x:(x%2==0), L))
[6, 8]
```

- map()
  - The map() function in Python takes in a function and a list.
  - The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

```
>>> # Program to double each item in a list using map()
>>> L = [1, 5, 4, 6, 8, 11, 3, 12]
>>> list(map(lambda x:x*2, L))
[2, 10, 8, 12, 16, 22, 6, 24]
```

FILE HANDLING

FILES

- Files are named locations on disk to store related information. They are used to permanently store data in a non-volatile memory (e.g. hard disk).
  - What are the different types of files you see in your device?
- Files are divided into two categories
  - text files - contain simple text
  - binary files - contain binary data, which is only readable by computer
- Basic File Operations
  - Opening
  - Closing
  - Reading
  - Closing

CREATING AND READING OF TEXT DATA
OPENING OF A FILE

- Python built-in function **open()** is used to open a file.
- This function returns a file object (handle). It is used to read or modify the file.
- The syntax to open a file is
  filepointer = open(filename, mode)
- mode specifies the mode in which file is opened.
- The default mode is reading.

Mode	Description
r	Opens a file for reading (default)
w	opens a file for writing. Creates a new file if it does not exist or truncates the file if it exists.
x	Creates a file. returns error if the file already exists.
a	Opens a file for appending at the end of the file without truncating it. Creates a new file if it does not exist.
t	opens the file in text mode
b	opens the file in binary mode

Example:

```
f1 = open("test.txt")
f2 = open("test.txt", "w")
f3 = open("img.bmp", 'rb')
```

CREATING AND READING OF TEXT DATA
OPENING OF A FILE

- Files need to be closed properly after performing operations.
- closing a file will free up the resources that were tied with the file.
- close() method is used to close the file.

```
f = open("test.txt")
f.close()
```

- **with statement**
- file can also be opened with the help of with keyword.
- There is no need to explicitly close the file if it is opened with with keyword.
- It ensures that the file is closed when the block inside the with statement is exited.
- it internally calls close()

```
with open("test.txt") as f:
  perform operations
```

READ A TEXT FILE

- To read the content of a file in python, file must be opened in read (**r**) mode
- methods available for reading
  - read() - reads the entire content of the file into a string
  - read(size) - reads the size number of characters from the file
  - readline() - to read individual lines of a file
  - readlines() - to read all the lines into a list.
- other file methods
  - seek() - to change the current file cursor (position)
  - tell() - returns the current position

message.txt

```
I am Sai
I am Pursuing B.Tech. CSE at VITB
```

```
f = open("message.txt")

msg = f.read()
print(msg)

f.close()
```

```
I am Sai
I am Pursuing B.Tech. CSE at VITB
```

```
f = open("message.txt")

msg = f.read(2)
print(msg)

msg = f.read(2)
print(msg)

f.close()
```

```
I
am
```

```
f = open("message.txt")

msg = f.readlines()
print(msg)

f.close()
```

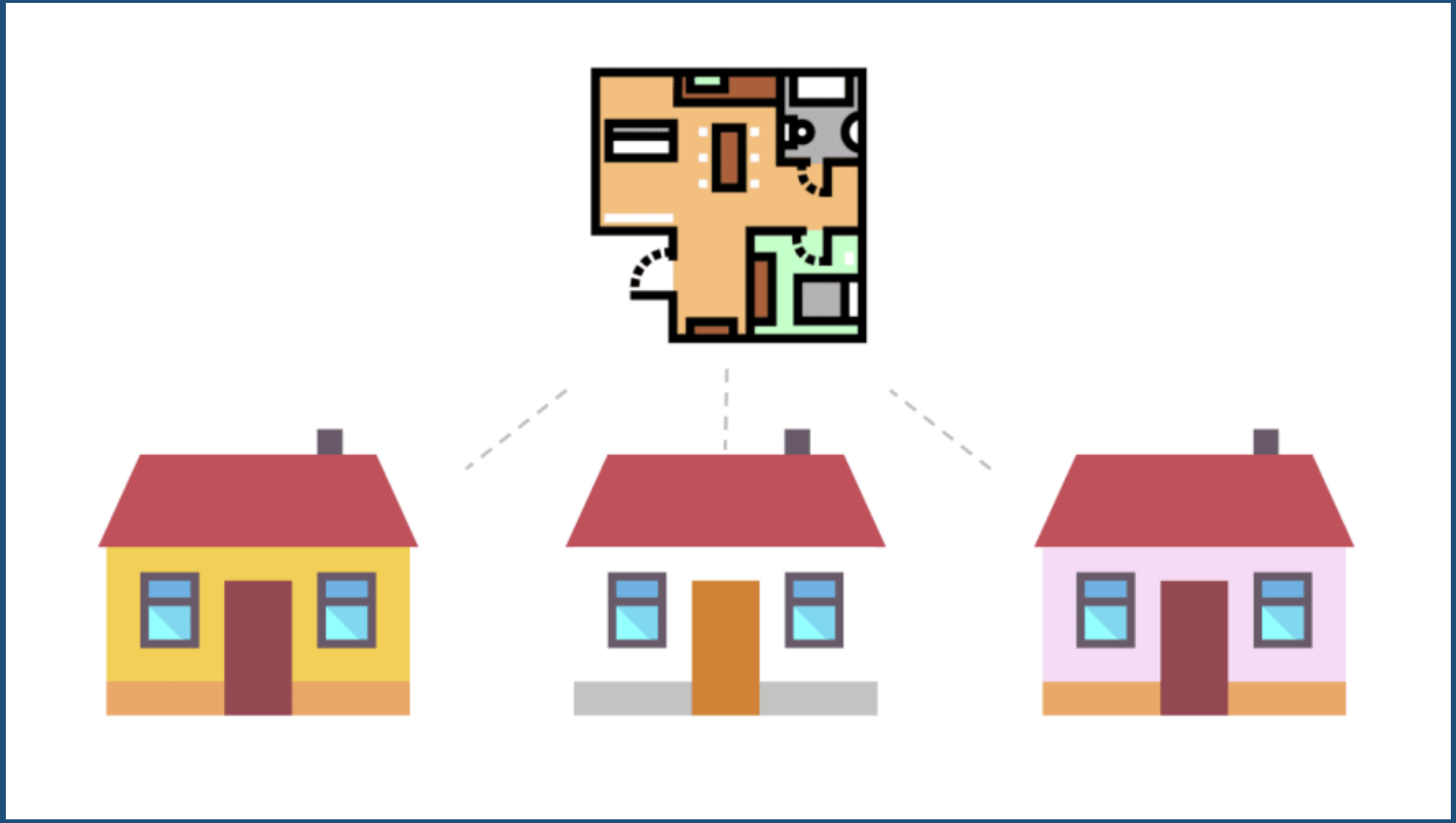
```
['I am Sai\n', 'I am Pursuing B.Tech. CSE at VITB']
```

WRITING TO FILES IN PYTHON

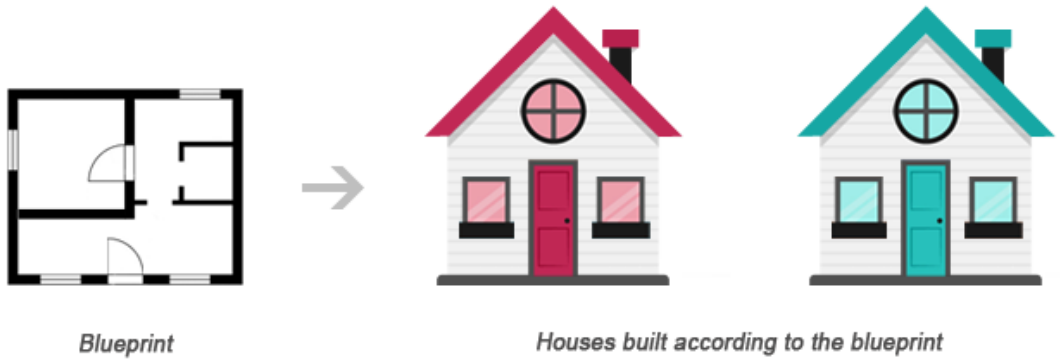
- To write content to a file, the file need to be opened in write (**w**) or append (**a**) mode.
- Opening a file in **w** mode creates a new file if the file does not exist otherwise it overwrites the content of the existing file.
- The methods used to write content
  - write() - Writes a string or sequence of bytes to a file. This method returns the number of characters written to the file.
  - writelines(lines) - writes a list of lines to the file.

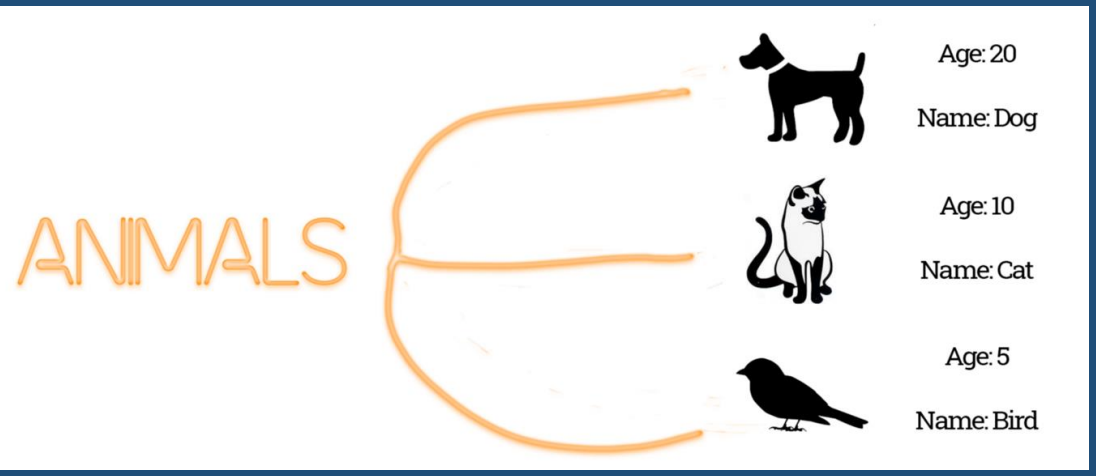
DEBUGGING

- A debugger is **a program that can help you find out what is going on in a computer program**.
- You can stop the execution at any prescribed line number, print out variables, continue execution, stop again, execute statements one by one, and repeat such actions until you have tracked down abnormal behavior and found bugs.
- IDEs will have inbuilt debugging tools
  - How to debug in VSCode - https://lightrun.com/debugging/debug-python-in-vscode/
- pdb module facilitates debugging in python. The major advantage of pdb is it runs purely in the command line.
- pdb supports
  - Setting breakpoints
  - Stepping through code
  - Source code listing
  - Viewing stack traces

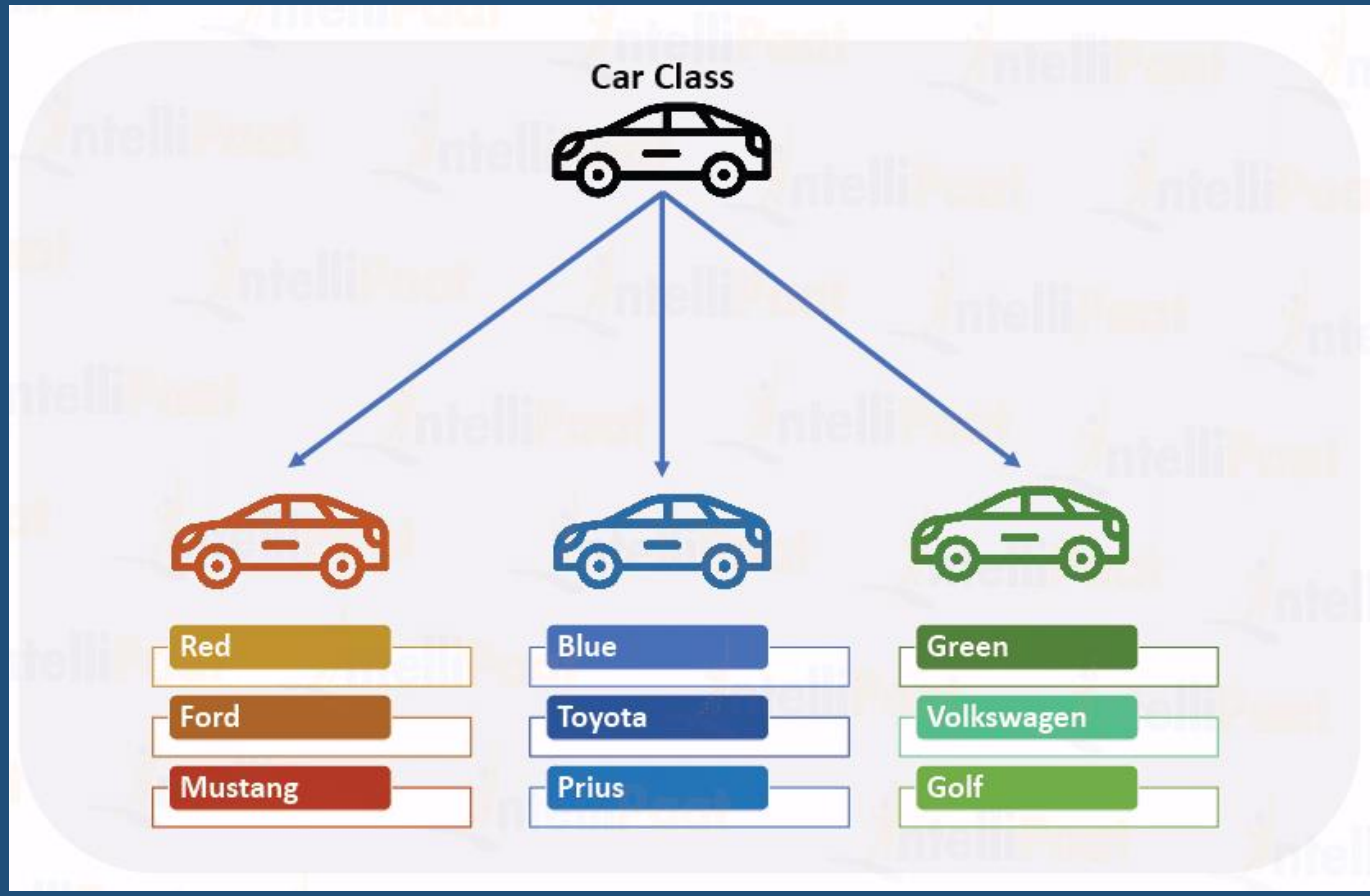


Object Oriented Programming


Blueprint
Houses built according to the blueprint


ANIMALS
Age: 20
Name: Dog
Age: 10
Name: Cat
Age: 5
Name: Bird


Objects
Class


Car Class
Red
Ford
Mustang
Blue
Toyota
Prius
Green
Volkswagen
Golf

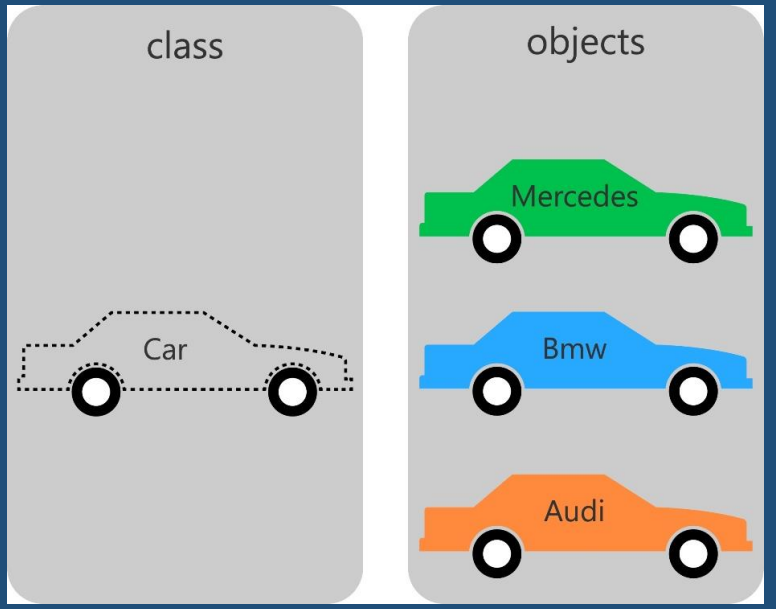
Object Oriented Programming

- Python is a multi-paradigm programming language. It supports different programming approaches.
- One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics:
  - attributes
  - behavior
- For example: A parrot is an object, as it has the following properties:
  - name, age, color as attributes
  - singing, dancing as behavior
- The concept of OOP in Python focuses on creating reusable code.

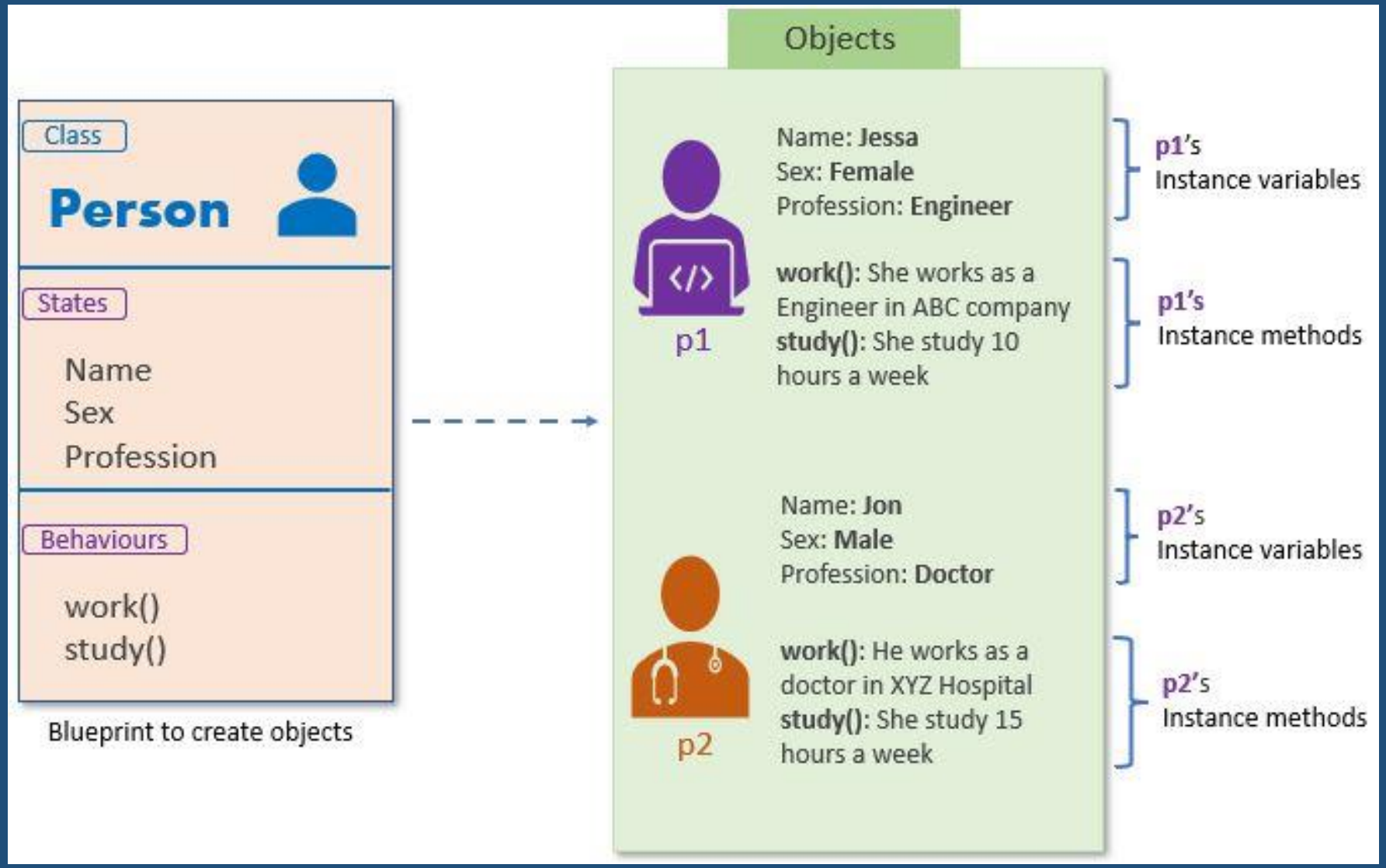
- In Python, the concept of OOP follows some basic principles

Classes and Objects

- Classes and objects
  - two basic concepts in object oriented programming.
- A class
  - is a blueprint or template for the object creation
  - Is a user-defined data structure that binds the data members and methods into a single unit.
  - When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.
- An object
  - (instance) is an instantiation of a class.
  - In Python everything is an object of some class.
- In Python, the base class for all classes is object.



An object is a real-life entity. It is the collection of various data and functions that operate on those data.



Creating a class

Syntax

```
class  class_name:
        <statement_1>
        <statement_2>
        ...
        ...
        <statement_n>
```

- The statements in the syntax can be
  - Attributes (variables)
  - Behavior (methods)
- The variables in a class are known as class variables.
- The functions defined in a class are known as class methods.
- Class variables and class methods are together called as class members.
- The class methods can access all class variables.
- The class members can be accessed outside the class by using the object of that class.
- A class definition creates a new namespace.

Creating Objects

- Once a class is defined, we can create objects for it. Creating an object for a class is known as instantiation.
- Syntax for creating an object :

  object_name = class_name(<arguments>)

- To access the class members use an object along with the dot (.) operator as follows:
  - object_name.variable_name
  - object_name.method_name(args_list)

```
#class
class Sample:
    pass

#object
s1 = Sample()
```

1. What is pass?
2. Which keyword is used to create a class?
3. How many objects are created here?
4. How to create one more object with the name S2?
5. How many objects I can create for a class?

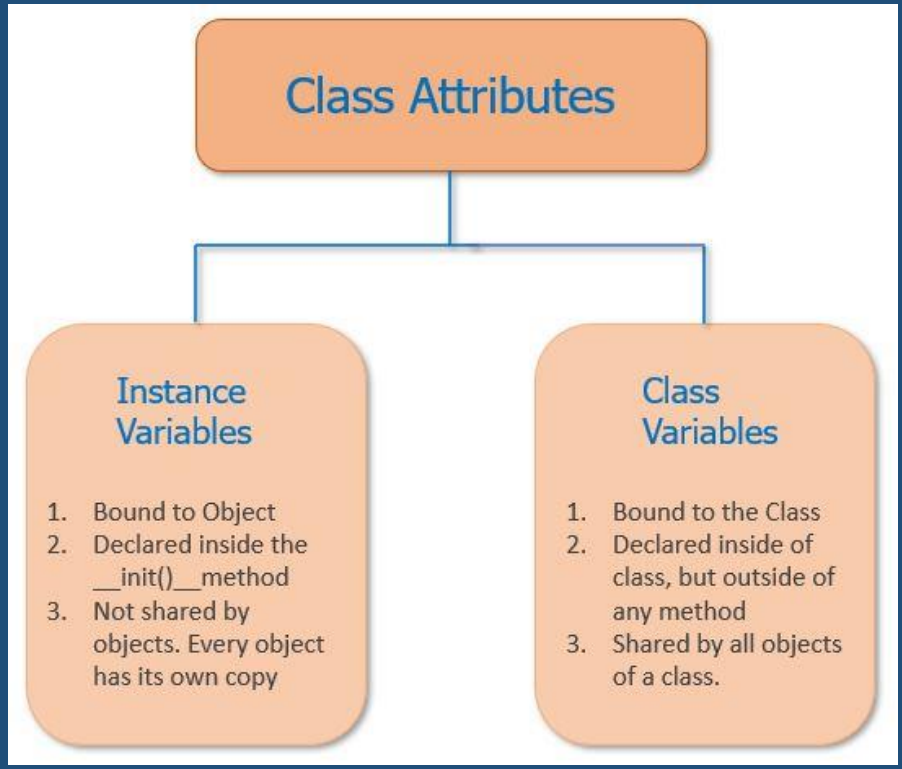
```python
class student:
    branch = 'CSE'      #class variable

    def set_details(self, rno, name):
        #instance variables
        self.rno = rno
        self.name = name

    def print_details(self):
        print("Rno :", self.rno)
        print("name :", self.name)
        print("branch :", student.branch)
```

```python
s1 = student()
s1.set_details('21PA1A0578', 'Sai')
s1.print_details()

s2 = student()
s2.set_details('21PA1A0577', 'Ram')
s2.print_details()
```

```
Rno : 21PA1A0578
name : Sai
branch : CSE
Rno : 21PA1A0577
name : Ram
branch : CSE
```

Class and Instance Variables

- Class Variables
  - Variables which are created inside a class and outside methods.
  - These are common for all objects. All the objects share the same value for each class variable.
  - Class variables are accessed as ClassName.variable_name.
- Instance variables
  - Variables created inside the class methods
  - accessed using the self variable
  - Instance variable value is unique to each object. The values of instance variables differ from one object to the other.



```python
class Faculty:
    #class variable
    college = "Vishnu Institute of Technology, Bhimavaram"

    #Constructor
    def __init__(self, name, designation, dept):
        self.name = name
        self.designation = designation
        self.dept = dept

    #Behavior (instance method)
    def worksin(self):
        print(self.name, "works in ", self.dept, " department")

    #Behavior (instance method)
    def worksas(self):
        print(self.name, " works as ", self.designation)

#object creation
f1 = Faculty('Sridevi', 'Assistant Professor', 'CSE')
f2 = Faculty('Mamata', 'Assistant Professor', 'EEE')

print("College Name : ", Faculty.college)
```

```python
f1.worksin()
f1.worksas()
print(f1.college)

f2.worksin()
f2.worksas()
print(f2.college)
```

```
College Name :  Vishnu Institute of Technology, Bhimavaram
Sridevi works in  CSE  department
Sridevi  works as  Assistant Professor
Vishnu Institute of Technology, Bhimavaram
Mamata works in  EEE  department
Mamata  works as  Assistant Professor
Vishnu Institute of Technology, Bhimavaram
```

self argument

- The self argument is used to refer the current object
  - (like this keyword in C++ and Java).
- All the class methods should have at least one argument
  - which is self argument.
- The self variable or argument is used to access the instance variables or object variables of an object.

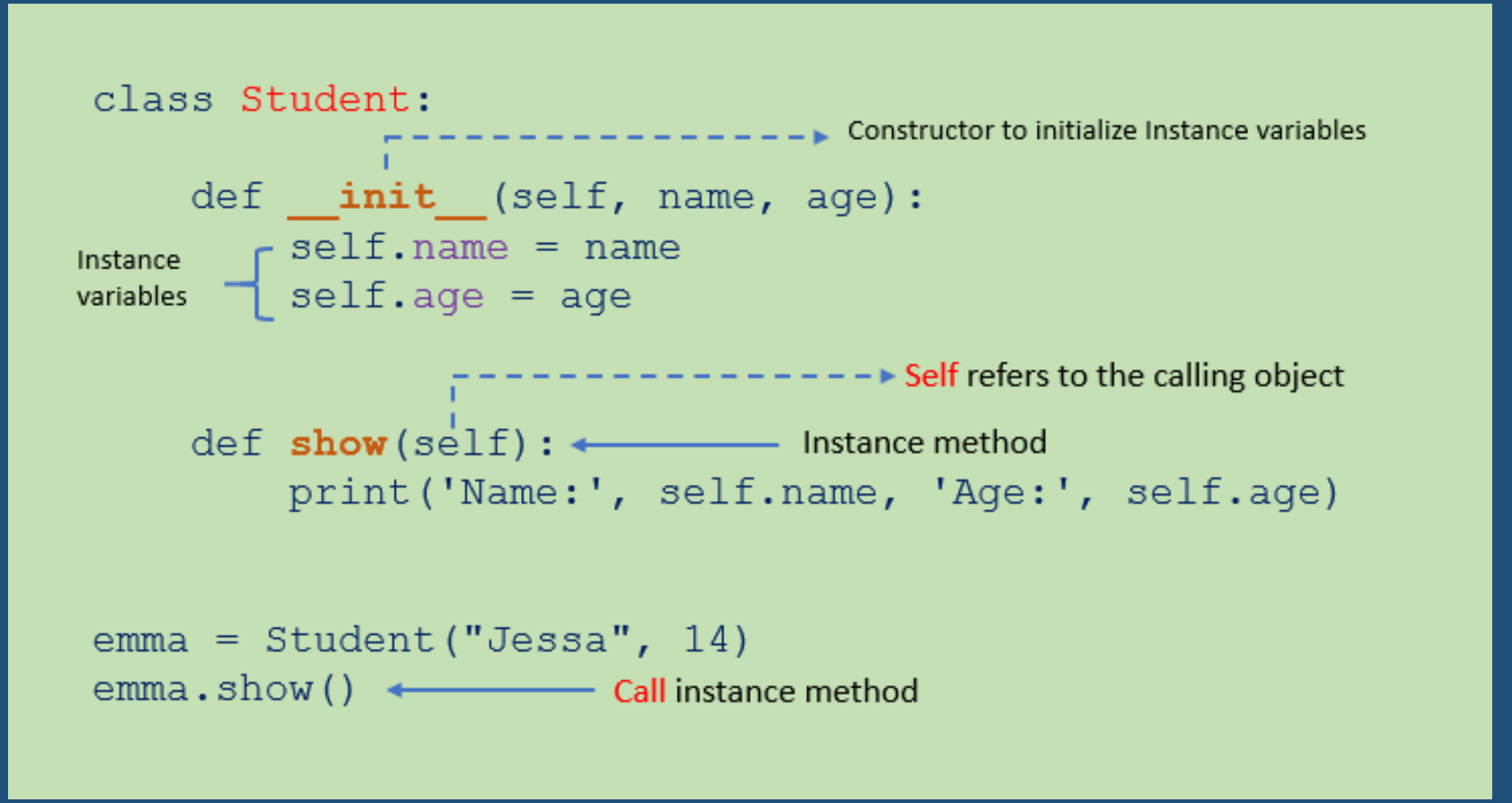
__init__() method / Constructor

- A constructor is a special method used to create and initialize an object of a class.
- This method is defined inside the class.
- constructor is implemented with the help of __init__() method.

- __init__() method
  - is a special method inside a class.
  - It serves as the constructor.
  - It is automatically executed when an object of the class is created.
  - This method is used generally to initialize the instance variables, although any code can be written inside it.

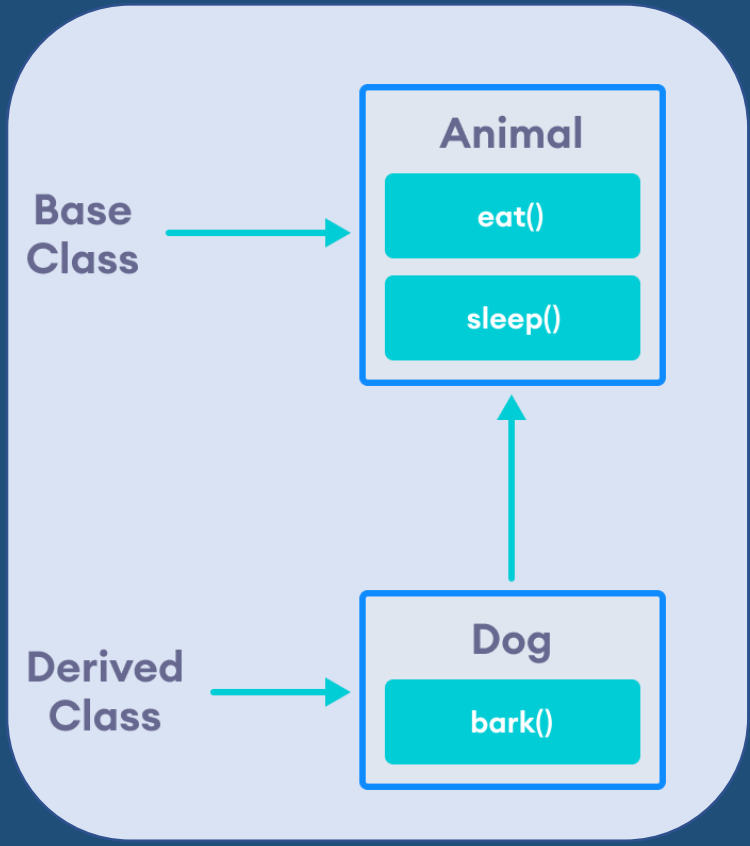
- Syntax of __init__() method is as follows:

```
def __init__(self, arg1, arg2, ...):
        statement1
        statement2
        ...
        statementN
```


class Student:
Constructor to initialize Instance variables
def __init__(self, name, age):
Instance variables
self.name = name
self.age = age
Self refers to the calling object
def show(self):
Instance method
print('Name:', self.name, 'Age:', self.age)
emma = Student("Jessa", 14)
emma.show()
Call instance method

Inheritance

- The process of acquiring properties of one class by another is called inheritance.
- It is the process of inheriting the properties of the parent class into a child class.
- The existing class is called a base class or parent class and the new class is called a subclass or child class or derived class.
- main benefit : Reusability of code
- Through inheritance, the child class acquires all the data members, functions from the parent class. A child class can also provide its specific implementation to the methods of the parent class.



Inheritance

Syntax

```
class BaseClass:
        Body of base class
class DerivedClass(BaseClass):
        Body of derived class
```

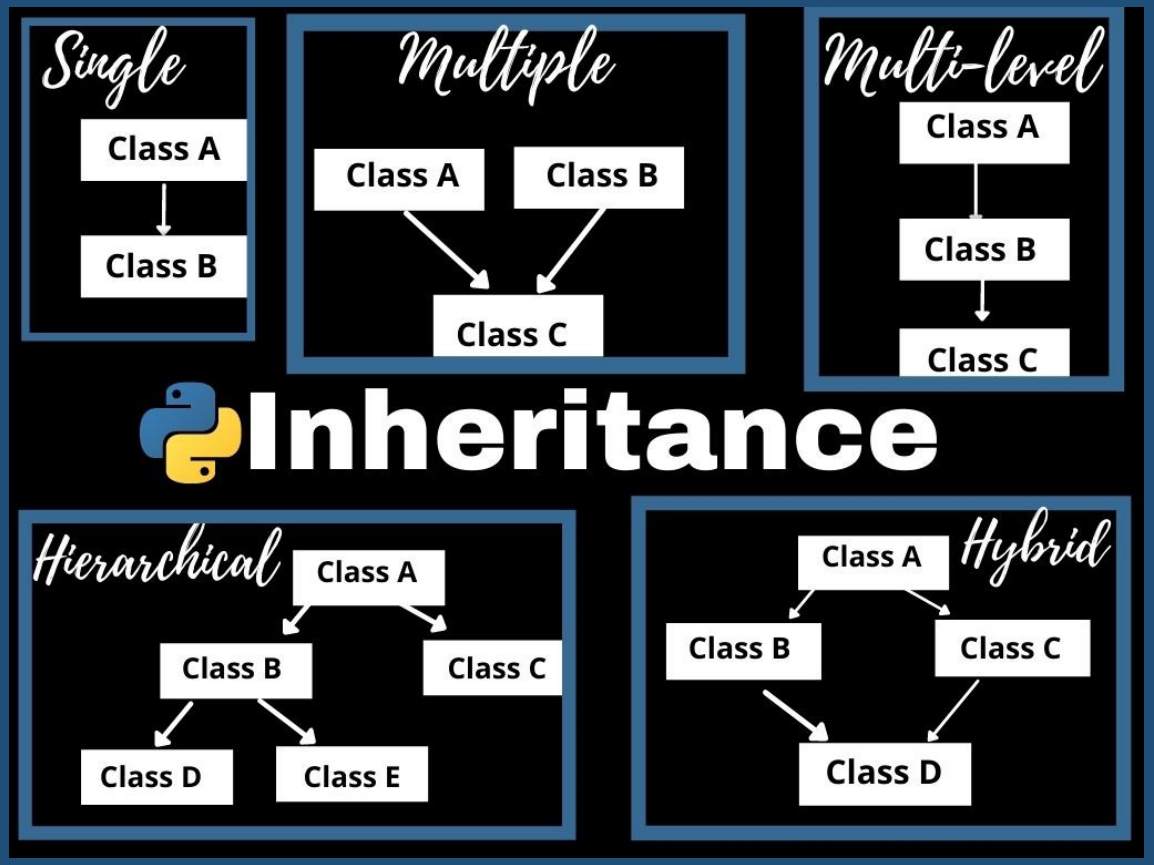
```
#Base class
class A:
    pass

#Child class
class B(A):
    pass
```

Inheritance

Types

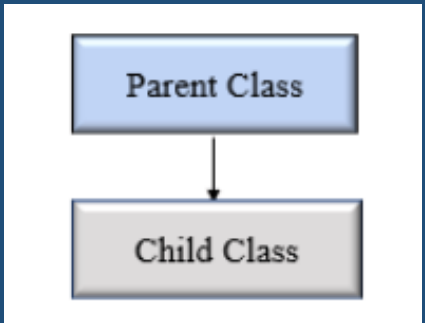
Based on the number of parent and child classes involved, there are five types of inheritance.

- Single Inheritance
- Multiple Inheritance
- Multilevel Inheritance
- Hierarchical Inheritance
- Hybrid Inheritance



Single Inheritance

- A child class inherits from a single parent class.
  - Parent class – one
  - Child class – one

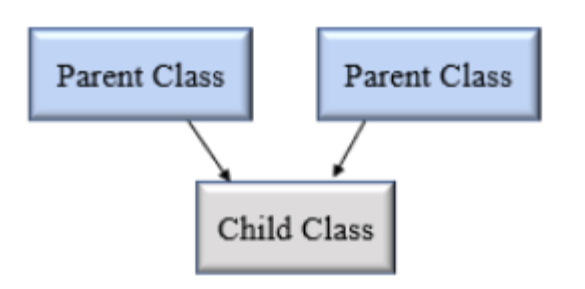



```
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Create object of Car
car = Car()

# access Vehicle's info using car object
car.Vehicle_info()
car.car_info()
```

```
Inside Vehicle class
Inside Car class
```

Multiple Inheritance

- In multiple inheritance, one child class can inherit from multiple parent classes.
  - parent classes - two or more (multiple).
  - child class - one



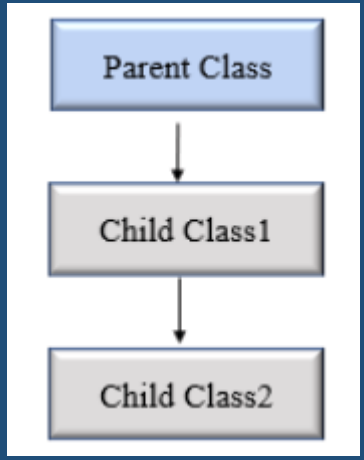

```python
# Parent class 1
class Person:
    def person_info(self, name, age):
        print('Inside Person class')
        print('Name:', name, 'Age:', age)

# Parent class 2
class Company:
    def company_info(self, company_name, location):
        print('Inside Company class')
        print('Name:', company_name, 'location:', location)

# Child class
class Employee(Person, Company):
    def Employee_info(self, salary, skill):
        print('Inside Employee class')
        print('Salary:', salary, 'Skill:', skill)
```

```python
# Create object of Employee
emp = Employee()

# access data
emp.person_info('Lekhya', 20)
emp.company_info('Google', 'Bangalore')
emp.Employee_info(90000, 'Machine Learning')
```

```
Inside Person class
Name: Lekhya Age: 20
Inside Company class
Name: Google location: Bangalore
Inside Employee class
Salary: 90000 Skill: Machine Learning
```

Multilevel Inheritance

- In multilevel inheritance, a class inherits from a child class.
  - consider three classes A, B, C.
    - A is the superclass,
    - B is the child class of A,
    - C is the child class of B.
- A chain of classes is called multilevel inheritance.

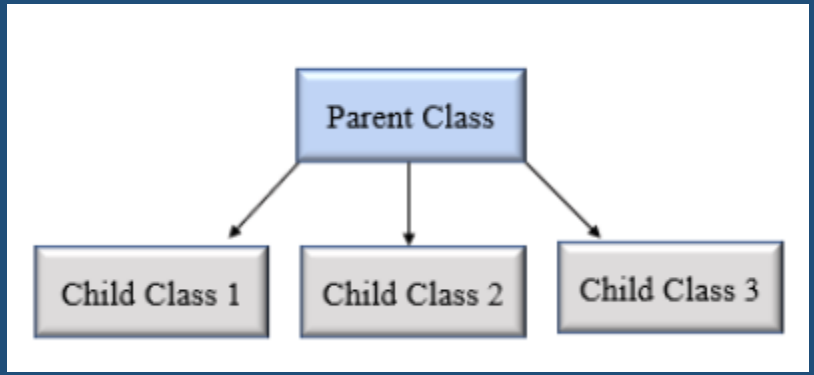



```python
# Base class
class Vehicle:
    def Vehicle_info(self):
        print('Inside Vehicle class')

# Child class
class Car(Vehicle):
    def car_info(self):
        print('Inside Car class')

# Child class
class SportsCar(Car):
    def sports_car_info(self):
        print('Inside SportsCar class')

# Create object of SportsCar
s_car = SportsCar()

# access Vehicle's and Car info using SportsCar object
s_car.Vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

Hierarchical Inheritance

- In Hierarchical inheritance, more than one child class can inherit from single parent class.
  - parent - one
  - child - two or more (multiple)

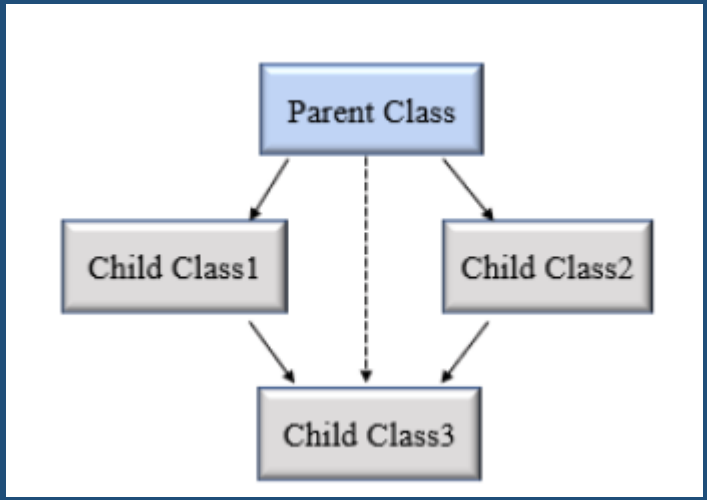



```python
class Vehicle:
    def info(self):
        print("This is Vehicle")

class Car(Vehicle):
    def car_info(self, name):
        print("Car name is:", name)

class Truck(Vehicle):
    def truck_info(self, name):
        print("Truck name is:", name)

obj1 = Car()
obj1.info()
obj1.car_info('BMW')

obj2 = Truck()
obj2.info()
obj2.truck_info('Ford')
```

```
This is Vehicle
Car name is: BMW
This is Vehicle
Truck name is: Ford
```

Hybrid Inheritance

- When inheritance is consists of multiple types or a combination of different inheritance is called hybrid inheritance.




```python
class Vehicle:
    def vehicle_info(self):
        print("Inside Vehicle class")

class Car(Vehicle):
    def car_info(self):
        print("Inside Car class")

class Truck(Vehicle):
    def truck_info(self):
        print("Inside Truck class")

# Sports Car can inherits properties of Vehicle and Car
class SportsCar(Car, Vehicle):
    def sports_car_info(self):
        print("Inside SportsCar class")

# create object
s_car = SportsCar()

s_car.vehicle_info()
s_car.car_info()
s_car.sports_car_info()
```

```
Inside Vehicle class
Inside Car class
Inside SportsCar class
```

super() function

- When a class inherits all properties and behavior from the parent class is called inheritance.
- In such a case, the inherited class is a subclass and the latter class is the parent class.
- super() function
  - Used to refer to parent class.
  - Returns a temporary object of the parent class that allows us to call a parent class method inside a child class method.
- Benefits of using the super() function.
  - It is not required to remember or specify the parent class name to access its methods.
  - The super() function support code reusability as there is no need to write the entire function

```python
class Company:
    def company_name(self):
        return 'Google'

class Employee(Company):
    def info(self):
        # Calling the superclass method using super()function
        c_name = super().company_name()
        print("Srithan works at", c_name)

# Creating object of child class
emp = Employee()
emp.info()
```

```
Srithan works at Google
```

isinstance(obj, class)

*Checks if **obj** is an instance of **class**.*
*Second argument can also be **tuple of classes** to check if obj is instance of **either one**.*

```
>>> isinstance(42, int)
True
>>> isinstance(42, (float, list, int))
True
>>> isinstance('42', (float, list, int))
False
```

Python isinstance() function

```
If isinstance(object , type):
    # Your code to be executed
```

issubclass()

- Used to verify whether a particular class is subclass of another class
- This function returns True if the given class is the subclass of the specified class. Otherwise, it returns False.
- Syntax

  issubclass(class, classinfo)

```
class Company:
    def fun1(self):
        print("Inside parent class")

class Employee(Company):
    def fun2(self):
        print("Inside child class.")

class Player:
    def fun3(self):
        print("Inside Player class.")
```

```
# Result True
print(issubclass(Employee, Company))

# Result False
print(issubclass(Employee, list))

# Result False
print(issubclass(Player, Company))

# Result True
print(issubclass(Employee, (list, Company)))

# Result True
print(issubclass(Company, (list, Company)))
```

Keywords of Python

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

What are the keywords we have not used till now?

MYSQL

- A relational database management system
- You can learn about it .. By doing a simple course
- Install it
  - https://www.mysql.com/downloads/
  - Click on MySQL Community (GPL) Downloads »
  - Select the required installer - MySQL Installer for Windows
  - And also download Connector/Python
  - Install mysql – set username, password.
- Check whether you have mysql-connector-python package or not

PYTHON MYSQL

- We can connect to the MySQL server using the connect() method.
- Creating a database
- Creating tables
- Fetching the data from tables
- Refer the code section…. For the above

ITERATORS

- Iterator in Python is simply an object which will return data, one element at a time.
- a Python iterator object must implement two special methods, __iter__() and __next__(),
- An object is called iterable if we can get an iterator from it. Most built-in containers in Python like: list, tuple, string etc. are iterables.
  - The iter() function (which in turn calls the __iter__() method) returns an iterator from them.
  - next() function is used to manually iterate through all the items of an iterator.

- Building Custom Iterators
  - implement __iter__ and __next__ methods.
  - The __iter__() method returns the iterator object itself. If required, some initialization can be performed.
  - The __next__() method must return the next item in the sequence.

PICKLE MODULE

- **Serialization** - It is the process of converting a data structure into a linear form that can be stored or transmitted over a network.
  - In Python, serialization allows you to take a complex object structure and transform it into a stream of bytes that can be saved to a disk.
- **Deserialization** : It is the process of converting a stream of bytes back into a data structure.
- Pickle is the standard Python library used to serialize and deserialize objects.
- We can use the terms **pickling** and **unpickling** to refer to serializing and deserializing with Python pickle module.
- majorly used methods of pickle module:
- **pickle.dump()** - creates a file containing the serialization result.
  - Syntax : *pickle.dump(<Structure>, File Object)*
  - Structure can be any data structure of python.
  - File object is the file handle of the file.
- **pickle.load()** - reads a file to start unpickling process.
  - Syntax : *structure = pickle.load(File Object)*

```python
import pickle

def write():
    f = open("binarywrite.bin", "wb")
    list = ['CTP', 'Chemistry', 'Maths', 'English']
    pickle.dump(list, f)
    f.close()

def read():
    f = open("binarywrite.bin", "rb")
    L = pickle.load(f)
    print(L)
    f.close()

write()
print("Data Written successfully")

read()
print("Data read and displayed successfully")
```

```
Data Written successfully
['CTP', 'Chemistry', 'Maths', 'English']
Data read and displayed successfully
```

```python
import pickle

def write():
    f = open("srithan.bin", "wb")
    d = {'name':"Srithan", 'hobbies':['Animation', 'Drawing', 'Watching TV'], 'age':9}
    pickle.dump(d, f)
    f.close()

def read():
    f = open("srithan.bin", "rb")
    D = pickle.load(f)
    print(D)
    f.close()

write()
print("Data Written successfully")

read()
print("Data read and displayed successfully")
```

```
Data Written successfully
{'name': 'Srithan', 'hobbies': ['Animation', 'Drawing', 'Watching TV'], 'age': 9}
Data read and displayed successfully
```