

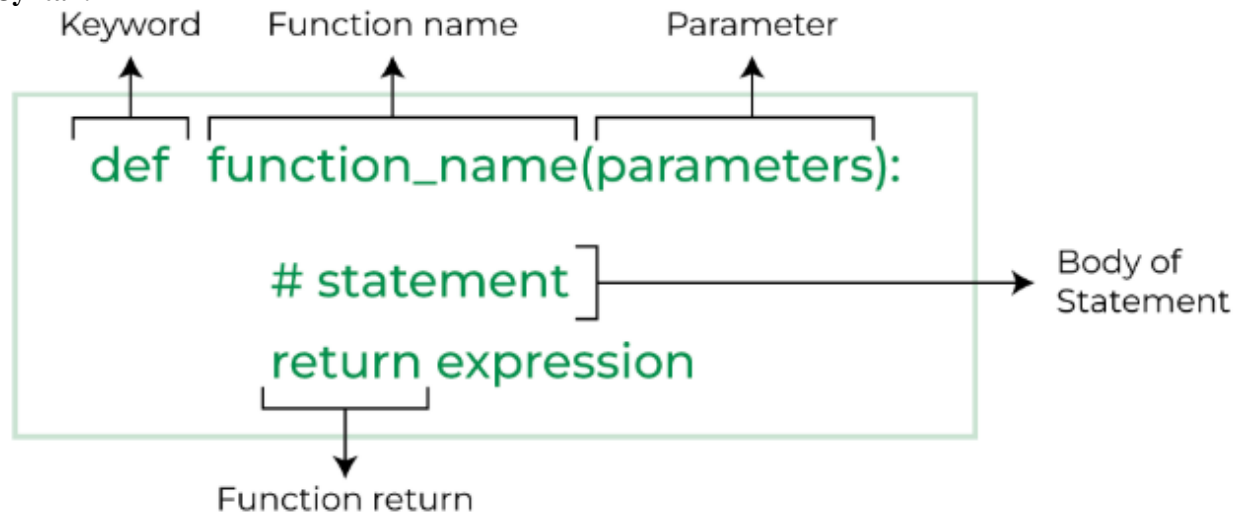
Chapter-2

Part-1: Functions:

Topic-1: Defining and using functions, including the use of arguments and return values:

Functions in Python are blocks of statements that return a specific task. The idea is to combine a number of frequently performed tasks into a function so that we can reuse the code within the function repeatedly rather than writing the same code for different inputs.

Syntax:



Python's `def` keyword allows us to define a function. It can have any kind of attributes and functions that we need. We may learn how to write a function in Python by looking at the following example. Using the `def` keyword, we may create a Python function definition in this manner.

Once a function has been created in Python, it can be called by using the function's name, Python, followed by parenthesis that contains the function's parameters.

Example:

```
def fun():
```

```
    print("Welcome to Parul")
```

```
fun()
```

Function with arguments:

The values that are supplied inside the function's parenthesis are called arguments. A comma can be used to separate any number of arguments in a function.

Example:

```
def checkEven(x):
```

```
    if (x % 2 == 0):
```

```
        print("even")
```

```
    else:
```

```
        print("odd")
```

```
checkEven(2)
checkEven(3)
```

Types of function arguments:

Default Argument: When a value is not supplied in the function call for a parameter, the parameter is said to be a default argument. The example that follows shows how to build functions in Python using default arguments.

Example:

```
def myFun(x, y=5):

    print("x: ", x)

    print("y: ", y)

myFun(1)
```

Keyword Arguments: Allowing the caller to supply the argument name together with values eliminates the requirement for the caller to remember the parameters' order.

Example:

```
def student(name, surname):

    print(name, surname)

student(name='Parul', surname='University')

student(lastname='ParulInstituteof', firstname='Technology')
```

Positional Arguments: In order to assign the first parameter (or value) to name and the second argument (or value) to age, we utilised the Position argument during the function call. The values may be used incorrectly if the positions are switched or if you lose track of their order.

Example:

```
def fullName(name, surname):
    print("Hi, I am", name)
    print("My surname is", surname)

fullName ("Ram", "Joshi")
fullName ("Joshi", "Ram")
```

Arbitrary arguments: *args and **kwargs are two Python arbitrary keyword arguments that use special symbols to pass a variable number of arguments to a function. Two unique symbols are present: Python **kwargs (Keyword Arguments) and *args (Non-Keyword Arguments)

Example-1:

```
def demoFun(*argv):

    for arg in argv:

        print(arg)

demoFun('Hi', 'Welcome', 'to', 'ParulUniversity')
```

Output:

```
Hi
Welcome
to
ParulUniversity
```

Example-2:

```
def myFun(**kwargs):
    for key, value in kwargs.items():
        print("%s == %s" % (key, value))

myFun(first='Welcome', mid='to', last='ParulUniversity ')
```

Output:

```
first == Welcome
mid == to
last == ParulUniversity
```

Docstrings:

The Document string, or Docstring for short, is the first string that appears after the function. This is employed to explain how the function works. Although it is not required, using docstring in functions is thought to be a good idea.

The docstring of a function can be printed using the syntax below:

Example:

```
def checkEven(x):
    """This function is to check whether the number is even or odd"""

    if (x % 2 == 0):
        print("even")
    else:
        print("odd")

print(checkEven.__doc__)
```

Nested functions:

The term "nested function" refers to a function that is defined inside another function. The variables of the enclosing scope are accessible to nested functions. The purpose of inner functions is to insulate them from anything that occurs outside of them.

Recursive functions:

In Python, recursion is the process by which a function calls itself. In order to tackle mathematical and recursive problems, you frequently need to construct a recursive function.

Example:

```
def factorial(n):  
    if n == 0:  
        return 1  
    else:  
        return n * factorial(n - 1)  
  
print(factorial(4))
```

Output:

24

Return statements:

To terminate a function, return the provided value or data item to the function caller, and return to the function caller, use the function return statement.

Syntax:

```
return [expression_list]
```

A variable, an expression, or a constant that is returned at the conclusion of the function's execution can make up the return statement. A None object is returned if none of the aforementioned conditions are met by the return statement.

Example:

```
def square(val):  
    return val**2
```

```
print(square(4))  
print(square(-3))
```

Pass by Reference:

Pass by reference indicates that a variable already exists in memory since you must provide the function (reference) to it.

The function and the caller share the same variable and object when pass-by-reference is used.

Example:

```
def demoList(x):  
    x.append("Parul")  
    print("Inside function:", x)  
  
list = ["University"]  
demoList (list)  
print("Outside function:", demoList)
```

Output:

```
Inside function: ['Parul', 'University']  
Outside function: ['Parul', 'University']
```

Pass by Value:

This method involves passing a copy of the function's real variables as a parameter. As a result, changes made to the function's parameters won't affect the variable itself.

In the context of the function caller, the copies of the variables and objects are totally separate.

Example:

```
def demoInteger(x):  
    x = x + 5  
    print("Inside function:", x)  
  
x = 15  
print("Before function call:", x)  
demoInteger(x)  
print("After function call:", x)
```

Output:

Before function call: 15
Inside function: 20
After function call: 15

Part-2: OOPS Concepts :

Topic-1: Object and Class:

An instance of a class is called an object. An instance is a replica of the class that has real values, whereas a class is similar to a blueprint. Python is an object-oriented programming language, meaning that it focusses mostly on functions. In essence, Python objects are a single entity that has data variables and methods that work with that data.

Important points on class:

Keyword class is used to construct classes.

The variables that are part of a class are called attributes.

Using the dot (.) operator, attributes can be retrieved and are always public. For instance,

Democlass.Demoattribute

Example:

```
class Shape:  
    type = "circle"
```

An object is made up of:

State: An object's properties serve as a representation of it. It also shows an object's characteristics.

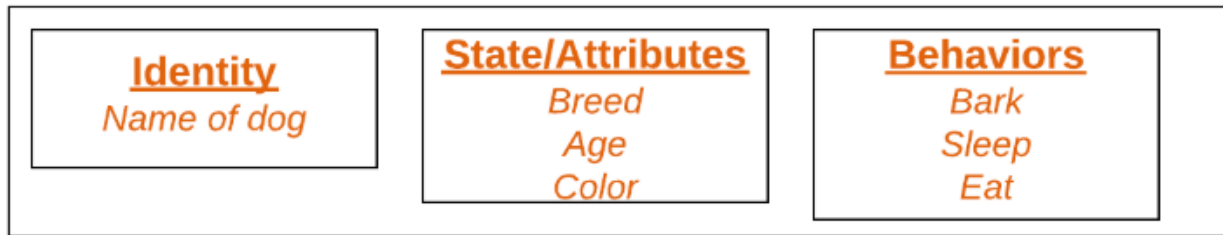
Behaviour: An object's methods serve as its representation. It also shows how one object reacts to other objects.

Identity: It allows a thing to communicate with other objects and provides it a distinctive name.

Syntax for object:

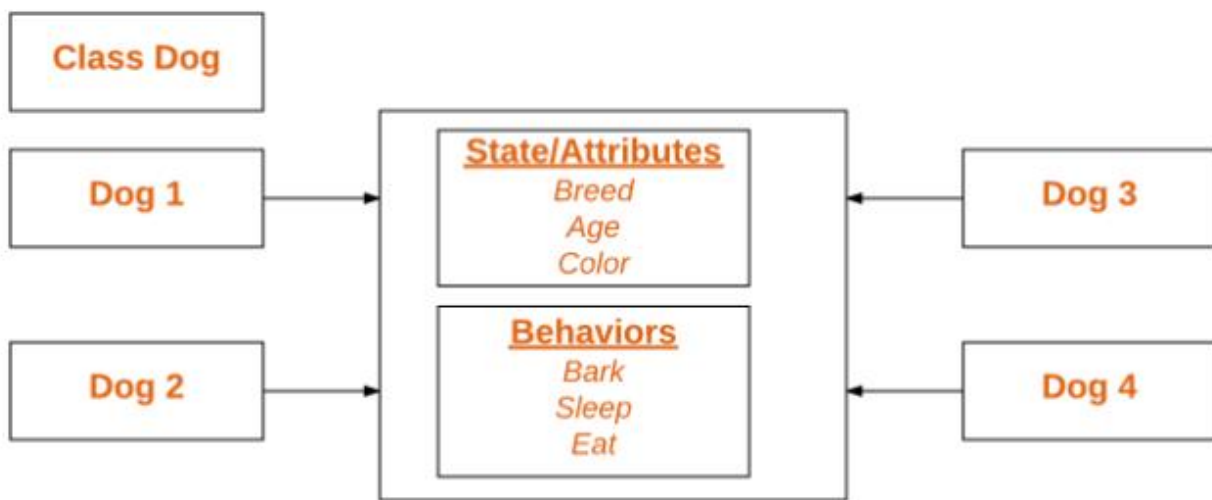
```
obj = ClassName()  
print(obj.attr)
```

Example:



Declaring or Instantiating a class:

A class is said to be instantiated when one of its objects is formed. The class's characteristics and behaviour are shared by all instances. However, each object has a unique set of values for those characteristics, or the state. There can be any number of instances of a single class.



Example:

class Dog:

 attr1 = "mammal"

 attr2 = "dog"

 def fun(self):

 print("I'm a", self.attr1)

 print("I'm a", self.attr2)

Rodger = Dog()

print(Rodger.attr1)

Rodger.fun()

Output:

mammal

I'm a mammal

I'm a dog

__init__() method

The `__init__` method is comparable to Java and C++ constructors. The object's state is initialised using constructors. Similar to methods, a constructor comprises a set of statements, or instructions, that are carried out during the formation of an object. It starts as soon as a class object is created. You can use the method to initialise your object in any way you choose.

Example:

class Person:

```
# init method or constructor
def __init__(self, name):
    self.name = name

# Sample Method
def say_hi(self):
    print('Hello, my name is', self.name)
```

```
p = Person('Parul')
p.say_hi()
```

Output:

Hello, my name is Parul

`__str__()` method:

`__str__()` is a specific function in Python. It is used to specify the string representation of a class object. It is frequently used to provide an object with a textual representation that is readable by humans, which is useful for debugging, logging, or displaying object information to users. The `__str__()` function is automatically invoked when a class object is used to construct a string using the built-in `print()` and `str()` operations. Define the `__str__()` method to change how a class's objects are represented in strings.

class PU:

```
def __init__(self, name, college):
    self.name = name
    self.company = college

def __str__(self):
    return f"My name is {self.name} and I study in {self.college}."
```

```
obj = PU("Jack", "ParulUniversity")
print(obj)
```

Example:

My name is Jack and I study in ParulUniversity.

Topic-2: Access Modifiers:

A Class in Python has three types of access modifiers:

- **Public Access Modifier:** Theoretically, public methods and fields can be accessed directly by any class.

Example:

```
class person:
```

```

def __init__(self, name, age):
    self.personName = name
    self.personAge = age

def displayAge(self):
    print("Age: ", self.personAge)

obj = person("R2J", 20)
print("Name:", obj.personName)
obj.displayAge()

```

Protected Access Modifier: Theoretically, protected methods and fields can be accessed within the same class it is declared and its subclass. It is common practice to describe a protected data member or class method with a single underscore "_." Because programmers would be attempting to access it using the plain name rather than calling it with the appropriate prefix, the Python interpreter does not treat it as protected data like other languages do.

Example:

```

class Student:
    _name = None
    _roll = None
    _branch = None

    def __init__(self, name, roll, branch):
        self._name = name
        self._roll = roll
        self._branch = branch

    def _displayRollAndBranch(self):
        print("Roll:", self._roll)
        print("Branch:", self._branch)

class person(Student):
    def __init__(self, name, roll, branch):
        Student.__init__(self, name, roll, branch)

    def displayDetails(self):
        print("Name:", self._name)
        self._displayRollAndBranch()

stu = Student("ABC", 1234567, "Computer Science")

print(stu._name)
stu._displayRollAndBranch()

# Throws error
# print(stu.name)
# stu.displayRollAndBranch()

```



```
obj = Person("RST", 1706256, "Information Technology")
print("")
obj.displayDetails()
```

Private Access Modifier: Theoretically, private methods and fields can be only accessed within the same class it is declared. A double underscore "__" is added before a data member of a class to indicate that the member is private.

Example:

```
class Student:
    __name = None
    __roll = None
    __branch = None

    def __init__(self, name, roll, branch):
        self.__name = name
        self.__roll = roll
        self.__branch = branch

    def __displayDetails(self):
        print("Name:", self.__name)
        print("Roll:", self.__roll)
        print("Branch:", self.__branch)

    def accessPrivateFunction(self):
        self.__displayDetails()

obj = Student("RST", 1706256, "Information Technology")
print("")

# Throws error
# obj.__name
# obj.__roll
# obj.__branch
# obj.__displayDetails()

print(obj._Student__name)
print(obj._Student__roll)
print(obj._Student__branch)
obj._Student__displayDetails()

print("")

obj.accessPrivateFunction()
```

Topic-3: Abstraction:

Data abstraction in Python is a programming concept that hides complex implementation details while exposing only essential information and functionalities to users. In Python, we can achieve data

abstraction by using abstract classes and abstract classes can be created using abc (abstract base class) module and abstractmethod of abc module.

Abstract class is a class in which one or more abstract methods are defined. When a method is declared inside the class without its implementation is known as abstract method.

In Python, abstract method feature is not a default feature. To create abstract method and abstract classes we have to import the "**ABC**" and "**abstractmethod**" classes from abc (Abstract Base Class) library. Abstract method of base class force its child class to write the implementation of the all abstract methods defined in base class. If we do not implement the abstract methods of base class in the child class then our code will give error.

Example:

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

```
# Create Abstract base class
```

```
class Car(ABC):
```

```
    def __init__(self, brand, model, year):
        self.brand = brand
        self.model = model
        self.year = year
```

```
    @abstractmethod
```

```
    def printDetails(self):
        pass
```

```
# Concrete method
```

```
    def accelerate(self):
        print("Speed up ...")
```

```
    def break_applied(self):
        print("Car stopped")
```

```
# Child class
```

```
class Hatchback(Car):
```

```
    def printDetails(self):
        print("Brand:", self.brand)
        print("Model:", self.model)
        print("Year:", self.year)
```

```
    def sunroof(self):
        print("Not having this feature")
```

```
# Child class
```

```
class Suv(Car):
```

```
    def printDetails(self):
        print("Brand:", self.brand)
        print("Model:", self.model)
        print("Year:", self.year)
```

```
    def sunroof(self):
        print("Available")
```

```
# Instance of the Hatchback class
car1 = Hatchback("Maruti", "Alto", "2022")

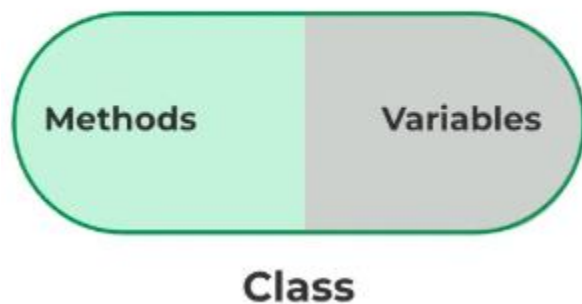
# Call methods
car1.printDetails()
car1.accelerate()
car1.sunroof()
```

Topic-4: Encapsulation:

One of the core ideas of object-oriented programming (OOP) is encapsulation. It explains the concept of data wrapping as well as ways that manipulate data in a single unit. This limits direct access to variables and methods and can stop data from being accidentally changed. Only an object's method has the ability to modify an object's variable in order to prevent unintentional changes. These variables are referred to as private variables.

Since it contains all of the data, including member functions, variables, etc., a class is an example of encapsulation. By limiting access to properties that are concealed from the public, information hiding aims to guarantee that an object's state is always valid.

Encapsulation in Python



Access specifiers are used in Python encapsulation to manage access to class members:

Public Members: By default, methods and attributes are accessible from outside the class and are public.

Protected Members: To designate that an attribute or method is meant for internal use within the class and its subclasses, use a single underscore (`_`) prefix.

Private Members: To make an attribute or method private, use the double underscores (`__`) prefix. It becomes more difficult to access from outside the class as a result of name mangling.

Topic-5: Polymorphism:

Polymorphism is defined as having several forms. Polymorphism in programming refers to the usage of the same function name for various types (but with distinct signatures). The data types and amount of arguments utilised in the function are the main differences.

Example:

In-built functions:

```
print(len("person"))
```

```
print(len([10, 20, 30]))
```

Output:

6

3

User-defined functions:

```
def add(x, y, z = 0):  
    return x + y+z
```

```
print(add(2, 3))  
print(add(2, 3, 4))
```

Output:

5

9

Class methods:

```
class India():  
    def capital(self):  
        print("New Delhi is the capital of India.")  
  
    def language(self):  
        print("Hindi is the most widely spoken language of India.")  
  
    def type(self):  
        print("India is a developing country.")
```

```
class USA():  
    def capital(self):  
        print("Washington, D.C. is the capital of USA.")  
  
    def language(self):  
        print("English is the primary language of USA.")  
  
    def type(self):  
        print("USA is a developed country.")
```

```
obj_ind = India()  
obj_usa = USA()  
for country in (obj_ind, obj_usa):  
    country.capital()  
    country.language()  
    country.type()
```

Output:

New Delhi is the capital of India.

Hindi is the most widely spoken language of India.
India is a developing country.
Washington, D.C. is the capital of USA.
English is the primary language of USA.
USA is a developed country.

In Inheritance:

```
class Bird:
    def intro(self):
        print("There are many types of birds.")

    def flight(self):
        print("Most of the birds can fly but some cannot.")

class sparrow(Bird):
    def flight(self):
        print("Sparrows can fly.")

class ostrich(Bird):
    def flight(self):
        print("Ostriches cannot fly.")

obj_bird = Bird()
obj_spr = sparrow()
obj_ost = ostrich()

obj_bird.intro()
obj_bird.flight()

obj_spr.intro()
obj_spr.flight()

obj_ost.intro()
obj_ost.flight()
```

Output:

There are many types of birds.
Most of the birds can fly but some cannot.
There are many types of birds.
Sparrows can fly.
There are many types of birds.
Ostriches cannot fly.

Topic-6: Inheritance:

By deriving a class from another class, you can use this process to establish a hierarchy of classes that share a set of attributes and methods. The capacity of one class to inherit or derive properties from another class is known as inheritance.

A class's properties, or base class, can be passed down to a derived class through inheritance. The following are some advantages of Python inheritance:

It accurately depicts relationships in the real world.

It gives a code the potential to be reused. We are not required to repeatedly write the same code.

Additionally, it enables us to expand a class's functionality without changing it.

Because of its transitive nature, all of class B's subclasses would inevitably inherit from class A if class B were to inherit from another class A.

An easy-to-understand model structure is provided by inheritance.

An inheritance lowers the cost of development and upkeep.

Syntax:

```
Class BaseClass:
```

```
    {Body}
```

```
Class DerivedClass(BaseClass):
```

```
    {Body}
```

Example:

```
class Person:
```

```
    def __init__(self, name, sname):
        self.name = name
        self.sname = sname
```

```
    def getFullName(self):
        return self.name, self.sname
```

```
    def isStudent(self):
        return False
```

```
class Student(Person):
```

```
    def __init__(self, name, sname, branch, enrollment_number):
        self.branch = branch
        self.enrollment_number = enrollment_number
        Person.__init__(self, name, sname)
        # super().__init__(name, sname)
```

```
    def isStudent(self):
        return True
```

```
    def getDetails(self):
        return self.branch, self.enrollment_number
```

```
p = Person("Ram", "Sharma")
print(p.getFullName())
print(p.isStudent())
```

```
p = Student("Krishna", "Yadav", "CSE", 1234567)
print(p.getFullName())
print(p.getDetails())
print(p.isStudent())
```

Output:

```
('Ram', 'Sharma')
False
('Krishna', 'Yadav')
('CSE', 1234567)
True
```

Types of inheritance:

There are 5 different types of inheritance in Python. They are as follows:

- **Single inheritance:** When a child class inherits from only one parent class, it is called single inheritance.

Syntax:

```
class Parent:
    pass
```

```
class Child(Parent):
    pass
```

Example:

```
class Animal:
    def sound(self):
        return "Animals make sounds"
```

```
class Dog(Animal):
    def sound(self):
        return "Dog barks"
```

```
dog = Dog()
print(dog.sound())
```

Output:

Dog barks

- **Multiple inheritances:** When a child class inherits from multiple parent classes, it is called multiple inheritances.

Syntax:

```
class Parent1:
    pass
```

```
class Parent2:
    pass
```

```
class Child(Parent1, Parent2):
    pass
```

Example:

```
class Father:
    def profession(self):
        return "Engineer"
```

```
class Mother:
    def hobby(self):
        return "Painting"

class Child(Father, Mother):
    pass

child = Child()
print(child.profession())
print(child.hobby())
```

Output:

Engineer
Painting

- **Multilevel inheritance:** When we have a child and grandchild relationship. This means that a child class will inherit from its parent class, which in turn is inheriting from its parent class.

Syntax:

```
class Grandparent:
    pass

class Parent(Grandparent):
    pass

class Child(Parent):
    pass
```

Example:

```
class Grandparent:
    def legacy(self):
        return "Family legacy"

class Parent(Grandparent):
    def advice(self):
        return "Work hard"

class Child(Parent):
    def dream(self):
        return "Become a scientist"

child = Child()
print(child.legacy())
print(child.advice())
print(child.dream())
```

Output:

Family legacy
Work hard
Become a scientist

- **Hierarchical inheritance** More than one derived class can be created from a single base.

Syntax:

```
class Parent:  
    pass
```

```
class Child1(Parent):  
    pass
```

```
class Child2(Parent):  
    pass
```

Example:

```
class Vehicle:  
    def fuel(self):  
        return "Runs on fuel"
```

```
class Car(Vehicle):  
    def wheels(self):  
        return "Has 4 wheels"
```

```
class Bike(Vehicle):  
    def wheels(self):  
        return "Has 2 wheels"
```

```
car = Car()  
bike = Bike()  
print(car.fuel())  
print(car.wheels())  
print(bike.fuel())  
print(bike.wheels())
```

Output:

```
Runs on fuel  
Has 4 wheels  
Runs on fuel  
Has 2 wheels
```

- **Hybrid inheritance:** This form combines more than one form of inheritance. Basically, it is a blend of more than one type of inheritance.

Syntax:

```
class Parent1:  
    pass
```

```
class Parent2:  
    pass
```

```
class Child1(Parent1):  
    pass
```

```
class Child2(Parent1, Parent2):  
    pass
```

Example:

```
class Human:
    def species(self):
        return "Homo sapiens"

class Father(Human):
    def profession(self):
        return "Engineer"

class Mother(Human):
    def hobby(self):
        return "Gardening"

class Child(Father, Mother):
    def ambition(self):
        return "Doctor"

child = Child()
print(child.species())
print(child.profession())
print(child.hobby())
print(child.ambition())
```

Output:

```
Homo sapiens
Engineer
Gardening
Doctor
```

Part-3: Exceptions and File handling:**Topic-1: Handling exceptions:**

Python errors can be divided into two categories: syntax errors and exceptions. Errors are issues with a program that cause the program to terminate. However, exceptions are triggered when internal events occur that alter the program's regular flow.

Various kinds of Python exceptions:

When an error arises while a program is running, one of the many built-in Python exceptions can be triggered. The following are a few of the most typical Python exception types:

SyntaxError: When the interpreter comes across a syntax issue in the code, like a misspelt keyword, a missing colon, or an unbalanced parenthesis, this exception is raised.

TypeError: When a function or operation is done to an object of the incorrect type—for example, adding a string to an integer—a TypeError exception is produced.

NameError: When a variable or function name is not present in the current scope, this exception is raised.

IndexError: When an index for a list, tuple, or other sequence type is out of range, an IndexError exception is raised.

KeyError: When a key cannot be located in a dictionary, an exception called KeyError is triggered.

ValueError: When a function or method is called with an invalid argument or input—for example,

attempting to convert a string to an integer when the string does not represent a valid integer—ValueError is produced.

AttributeError: When an attribute or method cannot be located on an object, such as when attempting to access a class instance's nonexistent attribute, an AttributeError exception is produced.

IOError: When an input/output error causes an I/O operation to fail, like reading or writing a file, this exception is triggered.

Try and Except Statement:

In Python, exceptions are caught and handled using try and except statements. The try block encloses statements that have the ability to raise exceptions, while the except block contains statements that manage the exception.

Example:

```
try:
    print(x)
except:
    print("An exception occurred")
```

Output:

An exception occurred

try: You can check a block of code for mistakes with the try block.

except: You can deal with the mistake using the except block.

Example:

```
try:
    print(x)
except NameError:
    print("Variable x is not defined")
except:
    print("Something else went wrong")
```

Output:

Variable x is not defined

else: When there isn't an error, you can run code using the else block.

Example:

```
try:
    print("Hello")
except:
    print("Something went wrong")
else:
    print("Nothing went wrong")
```

Output:

Hello
Nothing went wrong

finally: Regardless of the outcome of the try- and except-blocks, code can be executed using the finally block.

Example:

```
try:
    print(x)
except:
    print("Something went wrong")
finally:
    print("The 'try except' is finished")
```

Output:

Something went wrong
The 'try except' is finished

Topic-2: Working with files:

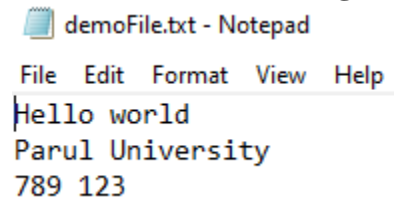
The process of creating, opening, reading, writing, and closing a file using a programming interface is referred to as file handling. It entails controlling the data flow between the application and the storage device's file system, guaranteeing that data is managed effectively and safely.

Python File Modes:

The mode that we want to use when opening a file dictates what we want to do with it. This table lists the various modes that are available:

Mode	Description
r	Read mode. Opens a file for reading. The file must exist.
w	Write mode. Opens a file for writing. Creates a new file if it doesn't exist or truncates the file if it exists.
a	Append mode. Opens a file for appending. Creates a new file if it doesn't exist.
b	Binary mode. Opens a file in binary mode.
t	Text mode. Opens a file in text mode. Default mode.
x	Exclusive creation mode. Creates a new file and fails if it already exists.
r+	Read and write mode. Opens a file for both reading and writing. The file must exist.
w+	Write and read mode. Opens a file for both writing and reading. Creates a new file if it doesn't exist or truncates the file if it exists.
a+	Append and read mode. Opens a file for appending and reading. Creates a new file if it doesn't exist.

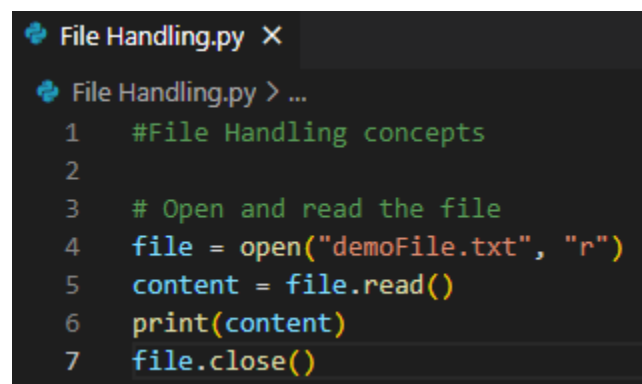
We will use the following demoFile for file handling:



```
demoFile.txt - Notepad
File Edit Format View Help
Hello world
Parul University
789 123
```

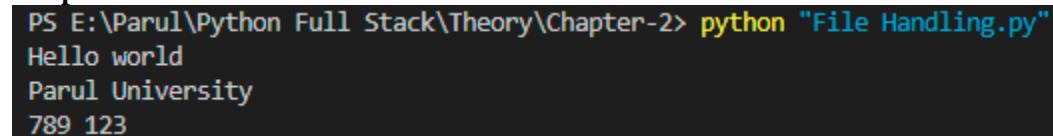
Reading a File :

Each file can be read file.read(), which reads the file's whole contents. We can use close the file after reading it using file.close(), which is required to free up system resources, reads the file and then closes it.



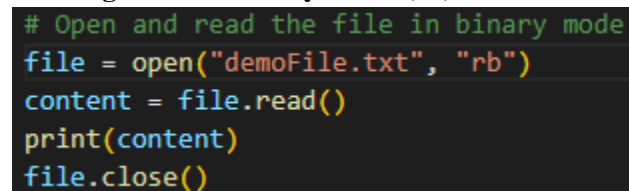
```
File Handling.py X
File Handling.py > ...
1 #File Handling concepts
2
3 # Open and read the file
4 file = open("demoFile.txt", "r")
5 content = file.read()
6 print(content)
7 file.close()
```

Output:



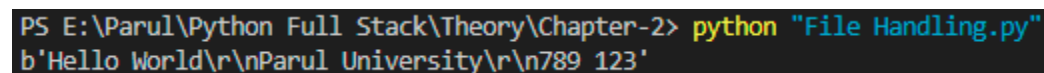
```
PS E:\Parul\Python Full Stack\Theory\Chapter-2> python "File Handling.py"
Hello world
Parul University
789 123
```

Reading a File in Binary Mode (rb)



```
# Open and read the file in binary mode
file = open("demoFile.txt", "rb")
content = file.read()
print(content)
file.close()
```

Output:



```
PS E:\Parul\Python Full Stack\Theory\Chapter-2> python "File Handling.py"
b'Hello World\r\nParul University\r\n789 123'
```

Writing Data to a File:

file.write() is used to write to a file. The write() function adds the given string to the file. The contents of the file are deleted if it exists. A new file is made if it doesn't already exist.

```
# write the file
file = open("demoFile.txt", "w")
file.write("Parul is in Vadodara, Gujarat.")
file = open("demoFile.txt", "r")
content = file.read()
print(content)
file.close()
```

Output:

```
PS E:\Parul\Python Full Stack\Theory\Chapter-2> python "File Handling.py"
Parul is in Vadodara, Gujarat.
```

Using Append Mode to Write to a File (a):

A file.write() is used for this. The write() function appends the given string to the end of the file without deleting its contents.

```
# write the file by appending content
file = open("demoFile.txt", "a")
file.write("This line is appended.")
file = open("demoFile.txt", "r")
content = file.read()
print(content)
file.close()
```

Output:

```
PS E:\Parul\Python Full Stack\Theory\Chapter-2> python "File Handling.py"
Parul is in Vadodara, Gujarat.This line is appended.
```

Closing a File:

To guarantee that all resources utilised by the file are appropriately freed, it is imperative to close the file. The file.close() method guarantees that any modifications made to the file are preserved while also closing the file.

with statement:

Making use of resource management is accomplished through the use of 'with' statements. Even if an exception is raised, it guarantees that the file is correctly closed once its suite is finished. When the block of code is exited, the file is immediately closed using the open() method, even in the event of an error. This lowers the possibility of resource leaks and file corruption.

```
# making use of "with" statement
with open("demoFile.txt", "r") as file:
    content = file.read()
    print(content)
```

Output:

```
PS E:\Parul\Python Full Stack\Theory\Chapter-2> python "File Handling.py"
Parul is in Vadodara, Gujarat.This line is appended.
```

Managing Exceptions in File Closing:

Even in cases where an error occurs during file operations, handling exceptions is crucial to ensuring that files are closed correctly.

```
# using exception handling in file handling
try:
    file = open("demoFile.txt", "r")
    content = file.read()
    print(content)
finally:
    file.close()
```

Output:

```
PS E:\Parul\Python Full Stack\Theory\Chapter-2> python "File Handling.py"
Parul is in Vadodara, Gujarat.This line is appended.
```