Advanced Python

Functions

- A function is a group of related statements that performs a specific task.
- Functions break long programs up into smaller components
- It is defined using **def** keyword

Advantages of python-

- Avoid rewriting the same logic or code again and again in a program
- We can track a large Python program easily when it is divided into multiple functions.
- Make the code reusable
- We can call Python functions anywhere and also call multiple times in a single program

Syntax- def function_name(parametrs):

    statements

    return expression

function_name() # Calling a function

Example-

```
def my_func():
        x = 10
        print("Value inside function:",x)
 x = 20 my_func()
print("Value outside function:",x)
```

Passing arguments in a function

```
def function1(name):
  print("hello ” , name,  “Welcome!”)

function1("preeti")
```

Return statement

- Used to exit a function

Types of Functions

1. **Built-in functions** - Functions that are built into Python.
2. **User-defined functions** - Functions defined by the users themselves.

Scope of variables

Global Variables

- A variable declared outside of the function or in global scope
- global variable can be accessed inside or outside of the function

Example-

```
x = "global"
def foo():
        y = ”local”
        print(x)
foo()
```

Local Variables

- A variable declared inside the function's body or in the local scope

Example-

```
def foo():
        y = "local"
foo()
print(y)
```

 # Python program for simple calculator

```
# Function to add two numbers
def add(num1, num2):
    return num1 + num2
# Function to subtract two numbers
def subtract(num1, num2):
    return num1 - num2
# Function to multiply two numbers
def multiply(num1, num2):
    return num1 * num2
# Function to divide two numbers
def divide(num1, num2):
    return num1 / num2

print("Please select operation -\n" \
      "1. Add\n" \
      "2. Subtract\n" \
      "3. Multiply\n" \
        "4. Divide\n")

# Take input from the user select = int(input("Select operations form 1, 2, 3, 4 :"))

number_1 = int(input("Enter first number: "))
number_2 = int(input("Enter second number: "))

if select == 1:
    print(number_1, "+", number_2, "=", add(number_1, number_2))
elif select == 2:
    print(number_1, "-", number_2, "=",  subtract(number_1, number_2))
elif select == 3:
    print(number_1, "*", number_2, "=", multiply(number_1, number_2))
elif select == 4:
    print(number_1, "/", number_2, "=",divide(number_1, number_2))
else:
    print("Invalid input")
```

Types of Arguments

Default arguments

A default argument is a parameter that assumes a default value if a value is not provided in the function call for that argument

```
def myFun(x, y=50):
   print("x: ", x)
   print("y: ", y)


myFun(10)
```

Keyword arguments

- To send the arguments in key=value syntax

```
def student(firstname, lastname):
        print(firstname, lastname)

# Keyword arguments
student(firstname='Kiran', lastname='Kumari')
```

Variable-length arguments

- *args (Non-Keyword Arguments)
- **kwargs (Keyword Arguments)

```
# *args

def myFun(*argv):
        for arg in argv:
                print(arg)


myFun('Delhi', 'Noida', 'MP', 'UP')
```

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

myFun(Name='Kiran',age=20,course='python')
```

Factorial of a number-

```
def factorial(number):
        if number == 0:
                return 1
        else:
                return number * factorial(number-1)
```

Modules in Python

- A file containing a set of functions.
- A module can define functions, classes, and variables.

Import Module in Python

- To import functions and class defined in a module to another module

Syntax:

import module_name

Example-

```
import calculator
print(calculator.add(10, 2))
```

from module_name import function/class

Example-

```
from math import sqrt, factorial
print(sqrt(16)) print(factorial(6))
```

Import all Names

```
from module_name import *
```

Python built-in modules

```
import math

# using square root(sqrt) function contained in math module
print(math.sqrt(25))

print(math.pi)

# 2 radians = 114.59 degrees print(math.degrees(2))

print(math.sin(2))

print(math.cos(0.5))

# 1 * 2 * 3 * 4 = 24
print(math.factorial(4))
```

import random

```
print(random.randint(0, 5))

# print random floating point number between 0 and 1
print(random.random())

# random number between 0 and 100
print(random.random() * 100)

List = [1, 4, True, 800, "python", 27, "hello"]

# choosing a random number from a list print(random.choice(List))
```

import datetime

```
from datetime import date import
time

print(time.time())

# Converts a number of seconds to a date object
print(date.fromtimestamp(454554))
```

Directories List for Modules

```
# importing sys module
import sys

# importing sys.path
print(sys.path)
```

Renaming the Python module

Syntax: Import **Module_name** as **Alias_name**

```
import math as mt

print(mt.sqrt(16))
print(mt.factorial(6))
```

Python File Handling

Read a file

```
f = open("sample_data.txt ", "r")
print(f.read())
```

Open a file on a different location:

```
f = open("D:\\Newfolder\sample_data.txt", "r")
print(f.read())
```

To append the content of file

```
f = open("sample_data.txt ", "a")
f.write("Welcome to the programmings!")
f.close()
```

To overwrite the context:

```
f = open("sample_data.txt ", "w")
f.write("This is the new content!")
f.close()
```

To create a new file:

```
f = open("New_file.txt", "x")
```

Delete a file:

```
import os
os.remove("sample_data.txt")
```

Check if file exists, *then* delete it:

```
import os

if os.path.exists("sample_data.txt"):

        os.remove("sample_data.txt") else:   print("The file does not exist")
```

Remove the folder:

```
import os
os.rmdir("folder1")
```

Object-Oriented Programming (OOP)

1. Class

A **class** is a blueprint for creating objects. It defines attributes (variables) and methods (functions) that describe the behaviour and properties of the objects.

Example:

```
class Car:
    def __init__(self, brand, color):
        self.brand = brand
        self.color = color

    def display_info(self):
        print(f"This car is a {self.color} {self.brand}.")
```

---

2. Object

An **object** is an instance of a class. It represents a specific example of the class and has its own unique data.

Example:

```
# Creating objects of the Car class
car1 = Car("Toyota", "Red")
car2 = Car("Ford", "Blue")

# Accessing methods and attributes
car1.display_info()
car2.display_info()
```

Output:

```
This car is a Red Toyota.
This car is a Blue Ford.
```

3. Method

A **method** is a function defined inside a class that operates on the attributes of the class.

Example:

```
class Calculator:
    def add(self, a, b):
        return a + b

    def subtract(self, a, b):
        return a - b

calc = Calculator()
print(calc.add(10, 5))        # Output: 15
print(calc.subtract(10, 5))   # Output: 5
```

__init__() Function

The __init__() function is called automatically every time the class is initiated.

Example-

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

p1 = Person("Allen", 20)

print(p1.name)
print(p1.age)
```

The self-Parameter

The self-parameter is a reference to the current instance of the class, and it is used to access variables (attributes) and other methods that belongs to the class.

Key Points About self:

1. **Represents the Instance**: self represents the instance of the class through which the method is being invoked.
2. **Mandatory in Instance Methods**: It must be the first parameter of any instance method in a class, although you can name it something else (not recommended for readability).
3. **Access Attributes and Methods**: It allows you to access and modify the attributes and methods associated with the particular instance.

Example-

```
class Person:
    def __init__(self, name, age):
        self.name = name  # Assign instance variable 'name' to the provided value
        self.age = age    # Assign instance variable 'age' to the provided value

    def greet(self):
        print(f"Hello, my name is {self.name} and I am {self.age} years old.")

# Creating an instance of the class
person1 = Person("Alice", 25)
person1.greet()
person2 = Person("Bob", 30)
person2.greet()

# Output: Hello, my name is Alice and I am 25 years old.
# Output: Hello, my name is Bob and I am 30 years old.
```

Explanation of above example:

1. When person1 is created, the __init__ method is called, and self refers to person1. The attributes name and age are set for person1.
2. Similarly, when person2 is created, self refers to person2.
3. When greet is called, self.name and self.age access the respective attributes of the instance (person1 or person2).

4. Inheritance

Inheritance allows a class (child) to inherit the attributes and methods of another class (parent). This promotes code reusability.

Parent class is the class being inherited from, also called base class.

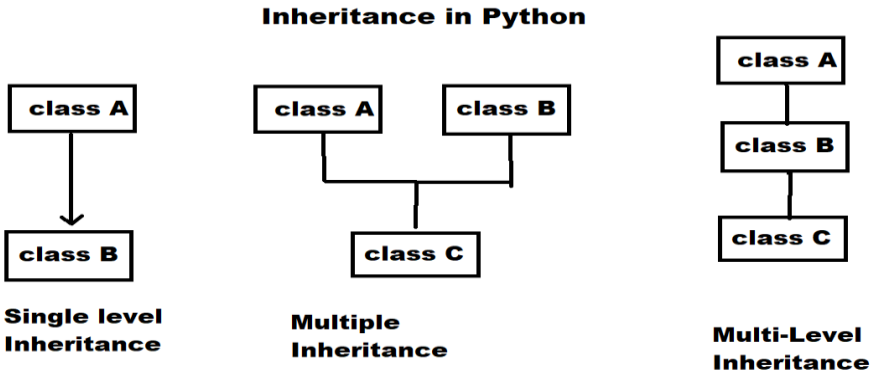
Child class is the class that inherits from another class, also called derived class.

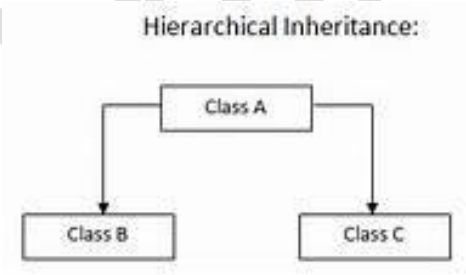
Example:

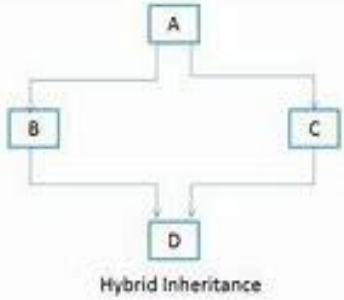
```
class Animal:
    def sound(self):
        print("Animals make sounds.")

class Dog(Animal):  # Dog inherits from Animal
    def sound(self):
        print("Dogs bark.")

# Creating objects
animal = Animal()
dog = Dog()

animal.sound()  # Output: Animals make sounds.
dog.sound()     # Output: Dogs bark.
```

Types of Inheritance

Type	Description	Example
Single Inheritance	A class inherits from one parent class.	class Dog(Animal)
Multiple Inheritance	A class inherits from more than one parent class.	class Dog(Animal, Mammal)
Multilevel Inheritance	A class inherits from a derived class, forming a chain.	class Dog(Mammal)
Hierarchical Inheritance	Multiple classes inherit from the same parent class.	class Dog(Animal), class Cat(Animal)
Hybrid Inheritance	A combination of different types of inheritance.	class Bat(Mammal, Bird)







1. Single Inheritance

- Single inheritance occurs when a child class inherits from a single parent class.
- The child class inherits all methods and attributes of the parent class.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Dog(Animal):
    def bark(self):
        print("Dog barks.")

dog = Dog()
dog.speak()  # Inherited from Animal
dog.bark()   # Defined in Dog
```

Output:

```
Animal makes a sound.
Dog barks.
```

2. Multiple Inheritance

- Multiple inheritance occurs when a child class inherits from more than one parent class.
- The child class inherits methods and attributes from all of its parent classes.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")
```

```
class Mammal:
    def feed(self):
        print("Mammal feeds milk to its young.")

class Dog(Animal, Mammal):
    def bark(self):
        print("Dog barks.")

dog = Dog()
dog.speak()  # Inherited from Animal
dog.feed()   # Inherited from Mammal
dog.bark()   # Defined in Dog
```

Output:

```
Animal makes a sound.
Mammal feeds milk to its young.
Dog barks.
```

3. Multilevel Inheritance

Multilevel inheritance occurs when a class is derived from another derived class, forming a chain of inheritance.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Mammal(Animal):
    def feed(self):
        print("Mammal feeds milk to its young.")
```

```
class Dog(Mammal):
    def bark(self):
        print("Dog barks.")

dog = Dog()
dog.speak()  # Inherited from Animal
dog.feed()   # Inherited from Mammal
dog.bark()   # Defined in Dog
```

Output:

```
Animal makes a sound.
Mammal feeds milk to its young.
Dog barks.
```

4. Hierarchical Inheritance

- Hierarchical inheritance occurs when multiple child classes inherit from a single parent class.
- All child classes share the attributes and methods of the parent class.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Dog(Animal):
    def bark(self):
        print("Dog barks.")

class Cat(Animal):
    def meow(self):
        print("Cat meows.")
```

```
dog = Dog()
dog.speak()  # Inherited from Animal
dog.bark()   # Defined in Dog

cat = Cat()
cat.speak()  # Inherited from Animal
cat.meow()   # Defined in Cat
```

Output:

```
Animal makes a sound.
Dog barks.
Animal makes a sound.
Cat meows.
```

5. Hybrid Inheritance

- Hybrid inheritance is a combination of any of the above types of inheritance.
- It often involves mixing multiple inheritance with other inheritance types.

Example:

```
class Animal:
    def speak(self):
        print("Animal makes a sound.")

class Mammal(Animal):
    def feed(self):
        print("Mammal feeds milk to its young.")

class Bird(Animal):
    def fly(self):
        print("Bird flies.")
```

```
class Bat(Mammal, Bird):
    def hang(self):
        print("Bat hangs upside down.")

bat = Bat()
bat.speak()  # Inherited from Animal
bat.feed()   # Inherited from Mammal
bat.fly()    # Inherited from Bird
bat.hang()   # Defined in Bat
```

Output:

```
Animal makes a sound.
Mammal feeds milk to its young.
Bird flies.
Bat hangs upside down.
```

Super() Function

- The super() function in Python is used to call a method from a parent (or superclass) within a child (or subclass).
- This is particularly useful in inheritance scenarios.

Example-

```
class Animal:
    def __init__(self, species):
        self.species = species

    def sound(self):
        print("Animals make sounds.")

class Dog(Animal):
    def __init__(self, species, breed):
        super().__init__(species)  # Call the parent class constructor
        self.breed = breed

    def sound(self):
        super().sound()  # Call the parent class sound method
        print("Dogs bark.")

# Create an instance of Dog
dog = Dog("Mammal", "Labrador")
print(dog.species)
dog.sound()
```

Output

```
Animals make sounds.
Dogs bark.
```

5. Polymorphism

Polymorphism allows methods in different classes to have the same name but behave differently depending on the class.

Example:

```
class Bird:
    def move(self):
        print("Birds can fly.")

class Fish:
    def move(self):
        print("Fish swim.")

# Using polymorphism
for creature in [Bird(), Fish()]:
    creature.move()
```

Output:

```
Birds can fly.
Fish swim.
```

---

6. Data Abstraction

Data Abstraction hides implementation details and shows only the necessary features of an object.

Example- Creating class and function

7. Encapsulation

Encapsulation restricts direct access to some of an object's components, which is done using private attributes and methods (prefixed with _ or __).

- **Public Member**: Accessible anywhere from outside of class.
- **Private Member**: Accessible within the class (use double underscore "__" to make private member.
- **Protected Member**: Accessible within the class and its sub-classes. Use single underscore "_" to make private member.

Example:

```
class BankAccount:
    def __init__(self, balance):
        self.__balance = balance  # Private attribute

    def deposit(self, amount):
        self.__balance += amount

    def get_balance(self):
        return self.__balance

account = BankAccount(1000)
account.deposit(500)
print(account.get_balance())  # Output: 1500
```

List Comprehension

Syntax: newlist = [*expression* for *item* in *iterable*

if *condition* == True]

The *expression* is the current item in the iteration and also the outcome.

Example

```
fruits = ["apple", "banana", "cherry", "kiwi",
"mango"] newlist = []

for x in
fruits:   if
"a" in x:
   list1.append(x)

print(list1)
```

Using List Comprehension-

```
fruits = ["apple", "banana", "cherry", "kiwi",
"mango"]

list1 = [x for x in fruits if "a" in x]

print(list1)

#newlist = [x.upper() for x in fruits]
```

Lambda Function

A lambda function is a small anonymous function.

Syntax

lambda *arguments* : *expression*

- Argument(s) - any value passed to the lambda function
- expression - expression is executed and returned

Ex1-

```
x = lambda a : a + 10
print(x(5))
```

Ex2-

```
def cube(y):
return y*y*y
lambda_cube = lambda y: y*y*y


## using def keyword print("Using function defined with `def`
keyword, cube:", cube(5))

# using the lambda function
print("Using lambda function, cube:", lambda_cube(5))
```

map() function

Syntax :

map(fun, iter)

Example-

```
def addition(n):
return n + n


# We double all numbers using map()
numbers = (1, 2, 3, 4)
result = map(addition, numbers)
print(list(result))
```

Generator in python

- A **generator** in Python is a type of iterable, like lists or tuples.
- Generators produce items one at a time and only when needed. Unlike lists that store all values in memory.
- This makes them more memory-efficient, especially for working with large datasets or streams of data.

Characteristics of a Generator:

1. **Defined using Functions**:
    - Generators are created using a function with the yield keyword.
    - When a generator function is called, it doesn't execute the function body immediately. Instead, it returns a generator object.
2. **Yield Statement**:
    - The yield statement is used to pause the function, saving its state. When the generator is resumed (e.g., using next()), it continues execution from where it left off.
3. **Lazy Evaluation**:
    - Generators don't compute all their values at once. Instead, they yield values on-the-fly, which makes them efficient in terms of memory.

Examples of a Generator Function:

```
def my_generator():
   yield 1
   yield 2
   yield 3

gen = my_generator()

print(next(gen))  # Outputs: 1
print(next(gen))  # Outputs: 2
print(next(gen))  # Outputs: 3

# Calling next(gen) again will raise StopIteration
```

Example 1: A Simple Generator

Code**:**

```
def countdown(start):
   while start > 0:
      yield start
      start -= 1

gen = countdown(5)  # Creates a generator object
for num in gen:
   print(num)
```

Explanation:

The function countdown generates numbers starting from start and counts down to 1.

How it works:
The yield keyword pauses the function and returns the current value of start.
When the generator is resumed, it continues from where it left off.

Output:

```
5
4
3
2
1
```

Use: If the range was very large, the generator would not create a list in memory, saving resources.

Example 2: Infinite Sequence Generator

```
def fibonacci():
    a, b = 0, 1
    while True:
        yield a
        a, b = b, a + b

gen = fibonacci()
for i in range(10):  # Print the first 10 Fibonacci numbers
    print(next(gen))
```

Explanation:

- Generates an infinite sequence of Fibonacci numbers.
- The generator starts with a = 0 and b = 1.
- On each iteration, it yields the current value of a and updates a and b.
- This continues indefinitely unless explicitly stopped.

Output:

```
0
1
1
2
3
5
8
13
21
34
```

Use: Useful when you need to generate an infinite sequence or very large sequences without precomputing and storing them in memory.

Advantages of Generators:

- **Memory Efficiency**: Suitable for processing large datasets.
- **Convenience**: Easy to implement lazy iterables.
- **Composability**: Can be used in data pipelines.

Common Use Cases:

- Reading large files line-by-line.
- Generating infinite or large sequences (e.g., primes, Fibonacci).
- Data streaming or processing pipelines.

Iterator

- Iterator in Python is simply an object that can be iterated upon. An object which will return data, one element at a time.
- **iterator object** implements two special methods,

  __iter__() and __next__()

Exception Handling

The try block lets you test a block of code for errors.

The except block lets you handle the error.

The else block lets you execute code when there is no error.

Finally: This block is always executed after the try and except blocks.

Example

The try block will generate an exception, because x is not defined:

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

Example

Print one message if the try block raises a NameError and another for other errors:

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

Regular Expression

- A **Reg**ular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern.
- It can detect the presence or absence of a text by matching it with a particular pattern
- It can also split a pattern into one or more sub-patterns.

Example-

```
import re
pattern = '^e...r$'
test_string = 'error'
result = re.match(pattern, test_string)
if result:
  print("Search successful.")
else:
  print("Search unsuccessful.")

# re.match() only returns true if the line of string starts with the given pattern
```

Metacharacters are characters that are interpreted in a special way by a RegEx engine.
To specify regular expressions, metacharacters are used.

Here's a list of metacharacters: [ ] . ^ $ * + ? {} () \ |

MetaCharacters	Description
\	Used to drop the special meaning of character following it
[ ]	Represent a character class
^	Matches the beginning
$	Matches the end
.	Matches any character except newline
|	Means OR (Matches with any of the characters separated by it.
?	Matches zero or one occurrence
*	Any number of occurrences (including 0 occurrences)
+	One or more occurrences
{}	Indicate the number of occurrences of a preceding RegEx to match.
()	Enclose a group of RegEx

re.findall()

The re.findall() method returns a list of strings containing all matches.

```
# Program to extract numbers from a string
import re
string = 'Hello 123, my name is 25’
pattern = '\d+'
result = re.findall(pattern, string)
print(result)
# Output: ['123’, ‘25]
```

Here **\d** matches any decimal digit, this is equivalent to the set class [0-9]

re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

```
import re
string = 'Hello 123, my name is 25’
pattern = '\d+'
result = re.split(pattern, string)
print(result)
# Output: ['Hello ', ', my name is ', '']
```

re.search()

- The re.search() The method looks for the first location where the RegEx pattern produces a match with the string.
- It takes two arguments: a pattern and a string.

Example

```
import re
string = "Python is very easy language"
# check if 'Python' is at the beginning
match = re.search('\APython', string)
if match:
  print("pattern found")
else:
  print("pattern not found")
# Output: pattern found
```

List of special sequences

Special Sequence	Description
\A	Matches if the string begins with the given character
\b	Matches if the word begins or ends with the given character. \b(string) will check for the beginning of the word and (string)\b will check for the ending of the word.
\B	It is the opposite of the \b i.e. the string should not start or end with the given regex.
\d	Matches any decimal digit, this is equivalent to the set class [0-9]
\D	Matches any non-digit character, this is equivalent to the set class [^0-9]
\s	Matches any whitespace character.
\S	Matches any non-whitespace character
\w	Matches any alphanumeric character, this is equivalent to the class [a-zA-Z0-9_].
\W	Matches any non-alphanumeric character.
\Z	Matches if the string ends with the given regex

JSON

JSON is a syntax for storing and exchanging data.

JSON is text, written with JavaScript object notation.

Import the json module:

```
import json
```

Convert from JSON to Python

If you have a JSON string, you can parse it by using the json.loads() method.

```
import json

# some JSON: x =  '{ "name":"John", "age":30,
"city":"New York"}'

# Converting JSON to Python
object y = json.loads(x)

# the result is a Python dictionary:
print(y["age"])
```

Convert from Python to JSON

If you have a Python object, you can convert it into a JSON string by using the json.dumps() method.

```
z=json.dumps(y)
```