Day 4-100 of Data Science

Python Functions

- In Python, a function is a reusable block of code that performs a specific task.
- Functions help in organizing code, promoting reusability, and enhancing readability.
- Functions are defined using the def keyword, followed by the function name and a pair of parentheses.
- Parameters, if any, are listed inside the parentheses.

In [1]:
```
def greet(name):
    """This function greets the person passed in as a parameter."""
    print(f"Hello, {name}! How are you doing?")

# Calling the function
greet("Vamsi")
```

```
Hello, Vamsi! How are you doing?
```

You can also have functions with multiple parameters, and you can return values from a function using the return keyword. Here's an example:

In [2]:
```
def add_numbers(x, y):
    """This function adds two numbers and returns the result."""
    sum_result = x + y
    return sum_result

# Calling the function
result = add_numbers(5, 7)
print("The sum is:", result)
```

```
The sum is: 12
```

OOPs Concept

- Object-oriented programming (OOP) is a programming paradigm that uses objects, which are instances of classes, for structuring and organizing code.
- Python supports OOP principles and provides features like classes, objects, inheritance, encapsulation,polymorphism and .

Class and Object:

- Class: A class is a blueprint for creating objects. It defines a data structure and behavior that the objects of the class will have.
- Object: An object is an instance of a class. It represents a real-world entity and has characteristics (attributes) and behavior (methods).

In [1]:
```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says Woof!")

# Creating objects of the Dog class
dog1 = Dog("Buddy", 3)
dog2 = Dog("Max", 5)

# Accessing attributes and calling methods
print(f"{dog1.name} is {dog1.age} years old.")
dog2.bark()
```

```
Buddy is 3 years old.
Max says Woof!
```

Inheritance:

- Inheritance allows a class to inherit the properties and methods of another class. It promotes code reuse and helps in building a hierarchy of classes.

In [2]:
```
class Animal:
    def __init__(self, species):
        self.species = species

    def make_sound(self):
        print("Generic animal sound")

class Cat(Animal):
    def make_sound(self):
        print("Meow!")

class Dog(Animal):
    def make_sound(self):
        print("Woof!")

# Creating objects of derived classes
cat = Cat("Felis catus")
dog = Dog("Canis lupus familiaris")

# Calling overridden methods
cat.make_sound()
dog.make_sound()
```

```
Meow!
Woof!
```

Encapsulation:

- Encapsulation involves bundling data (attributes) and methods that operate on the data within a single unit (class). It helps in restricting access to certain components.

In [3]:
```python
class Car:
    def __init__(self, make, model):
        self.__make = make  # Private attribute
        self.__model = model  # Private attribute
        self.__mileage = 0  # Private attribute

    def drive(self, miles):
        self.__mileage += miles

    def get_mileage(self):
        return self.__mileage

    def display_info(self):
        print(f"{self.__make} {self.__model}, Mileage: {self.__mileage} miles")


# Creating an object of the Car class
my_car = Car("Toyota", "Camry")

# Accessing attributes through methods
my_car.drive(50)
my_car.display_info()

# Trying to access a private attribute directly (will result in an error)
# Uncommenting the line below will raise an AttributeError
# print(my_car.__mileage)
```

```
Toyota Camry, Mileage: 50 miles
```

Polymorphism:

- Polymorphism allows objects to be treated as instances of their parent class, even when they are actually instances of a derived class. It enables code to work with objects of multiple types.

In [4]:
```python
class Bird:
    def make_sound(self):
        pass

class Crow(Bird):
    def make_sound(self):
        print("Caw!")

class Parrot(Bird):
    def make_sound(self):
        print("Squawk!")
```

```
# Polymorphic function
def bird_sound(bird_obj):
    bird_obj.make_sound()

# Creating objects of different classes
crow = Crow()
parrot = Parrot()

# Calling the polymorphic function with different objects
bird_sound(crow)
bird_sound(parrot)
```

```
Caw!
Squawk!
```

Abstraction

- It allows you to hide the complex implementation details and expose only the necessary features.

In [5]:
```
from abc import ABC, abstractmethod

# Abstract class with abstract method
class Shape(ABC):
    @abstractmethod
    def calculate_area(self):
        pass

# Concrete classes implementing the abstract class
class Square(Shape):
    def __init__(self, side):
        self.side = side

    def calculate_area(self):
        return self.side ** 2

class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius

    def calculate_area(self):
        return 3.14 * self.radius ** 2

# Function that works with the abstract class without knowing the concrete implementat
def print_area(shape):
    print(f"Area: {shape.calculate_area()}")

# Creating objects of concrete classes
square = Square(4)
circle = Circle(3)

# Using the abstraction to calculate and print the area
print_area(square)
print_area(circle)
```

```
Area: 16
Area: 28.26
```

Python File I/OPython File Operation

A file is a container in computer storage devices used for storing data.

When we want to read from or write to a file, we need to open it first. When we are done, it needs to be closed so that the resources that are tied with the file are freed.

Hence, in Python, a file operation takes place in the following order:

1) Open a file 2) Read or write (perform operation) 3) Close the file

Opening Files in Python

In Python, we use the open() method to open files.

To demonstrate how we open files in Python, let's suppose we have a file named test.txt with the following content.

In [8]:
```
# open file in current directory
file1 = open("test.txt")
```

In [10]:
```
#By default, the files are open in read mode (cannot be modified). The code above is e

file1 = open("test.txt", "r")
file1
```

Out[10]:
```
<_io.TextIOWrapper name='test.txt' mode='r' encoding='cp1252'>
```

Reading Files in Python

After we open a file, we use the read() method to read its contents. For example,

In [11]:
```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)
```

```
Hello, World!
```

Closing Files in Python

When we are done with performing operations on the file, we need to properly close the file.

Closing a file will free up the resources that were tied with the file.

It is done using the close() method in Python. For example,

In [12]:
```
# open a file
file1 = open("test.txt", "r")

# read the file
read_content = file1.read()
print(read_content)

# close the file
file1.close()
```

```
Hello, World!
```

Exception Handling in Files

If an exception occurs when we are performing some operation with the file, the code exits without closing the file.

A safer way is to use a try...finally block.

The try...except block is used to handle exceptions in Python.

Here's the syntax of try...except block:

try:

```
    # code that may cause exception
```

except:

```
    # code to run when exception occurs
```

In [13]:
```
try:
    file1 = open("test.txt", "r")
    read_content = file1.read()
    print(read_content)

finally:
    # close the file
    file1.close()
```

```
Hello, World!
```

In [14]:
```
try:
    numerator = 10
    denominator = 10

    result = numerator/denominator

    print(result)
except:
    print("Error: Denominator cannot be 0.")

# Output: Error: Denominator cannot be 0.
```

```
1.0
```

In [ ]:

Follow us on Socail Media

Linkedin: https://www.linkedin.com/company/eternaltek/about/?viewAsMember=true

Medium: https://medium.com/@eternaltek.info

WhatsApp Channel: https://whatsapp.com/channel/0029Va5onCbDjiOTi6D1vU36

In [ ]: