PYTHON

What is Python?

- Python is simple & easy
- Free & Open Source
- High Level Language
- Developed by Guido van Rossum
- Portable

Variables

- A variable is a name given to a memory location in a program.
- In programming, **variables** are like containers that store information.

Types of Operators

- An operator is a symbol that performs a certain operation between operands.
- Arithmetic Operators (+ , - ,*, / , % ,**)
- Relational / Comparison Operators (== , != , > , < , >= , <=)
- Assignment Operators (= , +=, -= ,*= , /= , %= ,**=)
- Logical Operators (not , and , or)

Typecasting

Typecasting (or **type conversion**) in Python means changing the data type of a value into another type.

Common Typecasting Functions:

- **int()**: Converts data to an integer.
- **float()**: Converts data to a decimal (floating-point number).
- **str()**: Converts data to a string.
- **bool()**: Converts data to True or False.

Examples:

```
num_str = "123"       # String
num = int(num_str)    # Convert to integer
print(num + 10)       # Output: 133
```

Strings

String is data type that stores a sequence of characters.

1. Concatenation (Joining strings together):

```
first_name  = "John"
last_name   = "Doe"
full_name  = first_name + " " + last_name
print(full_name)  # Output: John Doe
```

2. Repetition:

```
word = "Hi "
repeated = word * 3
print(repeated)  # Output: Hi Hi Hi
```

3. Indexing (Accessing characters by position):

```
text = "Python"
print(text[0])  # Output: P (1st character)
print(text[-1]) # Output: n (last character)
```

4. Slicing (Extracting part of a string):

```
text = "Python"
print(text[0:3])   # Output: Pyt (characters from index 0 to 2)
print(text[2:])   # Output: thon (from index 2 to the end)
```

5. Length of a String:

```
text = "Hello, World!"
print(len(text))      # Output: 13 (number of characters)
```

String Methods:

Python provides many built-in methods to work with strings:

- **lower()**: Converts to lowercase.
- **upper()**: Converts to uppercase.
- **strip()**: Removes leading/trailing spaces.
- **replace()**: Replaces parts of a string.
- **split()**: Splits a string into a list.
- **join()**: Joins a list of strings into one string.

```
text = " Hello, World! "
print(text.lower())     # Output: " hello, world! "
print(text.upper())    # Output: " HELLO, WORLD! "
print(text.strip())      # Output: "Hello, World!"
print(text.replace("World", "Python"))  # Output: " Hello, Python! "
print(text.split(","))          # Output: [' Hello', ' World!']
words = ["Python", "is", "fun"]
print(" ".join(words))   # Output: "Python is fun"
```

Special Characters in Strings:

You can use **escape sequences** for special characters:

- **\n**: New line
- **\t**: Tab
- ****: Backslash
- **\'** or **\"**: Quotes inside a string

Indexing

```
# String indexing
        s = "Python"
        print(s[0])  # Output: 'P'

# List indexing
        lst = [10, 20, 30, 40]
        print(lst[2])  # Output: 30

# Tuple indexing
        tpl = (1, 2, 3)
        print(tpl[1])  # Output: 2
```

Negative Indexing

```
# String
        s = "Python"
        print(s[-1])  # Output: 'n' (last character)

# List
        lst = [10, 20, 30, 40]
        print(lst[-2])  # Output: 30
```

Slicing

```
# Syntax:  sequence[start:stop:step]

# String slicing
s = "Python"
print(s[1:4])  # Output: 'yth' (indices 1 to 3)

# List slicing
lst = [10, 20, 30, 40, 50]
print(lst[2:])    # Output: [30, 40, 50] (from index 2 to the end)
print(lst[:3])    # Output: [10, 20, 30] (from start to index 2)
print(lst[::2])     # Output: [10, 30, 50] (every second element)
print(lst[::-1])   # Output: [50, 40, 30, 20, 10] (reverse order)
```

Multidimensional Indexing

For nested lists or arrays, use multiple indices.

```
# Nested list
matrix = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
print(matrix[1][2])# Output: 6 (row 1, column 2)

# NumPy array (example for advanced indexing)
import numpy as np
arr = np.array([[1, 2, 3], [4, 5, 6]])
print(arr[0, 1])        # Output: 2 (row 0, column 1)
```

IndexError

Accessing an index out of range raises an IndexError

```
lst = [10, 20, 30]
print(lst[3])           # IndexError: list index out of range
```

String Functions

```
str = “I am a coder.”
str.endsWith(“er.“) #returns true if string ends with substr
str.capitalize( )#capitalizes 1st char
str.replace( old, new ) #replaces all occurrences of old with new
str.find( word ) #returns 1st index of 1st occurrence
str.count(“am“) #counts the occurrence of substr in string
```

Conditional statements

Conditional statements in Python are used to **execute** specific blocks of code based on whether a condition evaluates to **True** or **False**. They allow decision-making in a program.

```
if (condition):
   # code to execute if the condition is True
elif (another_condition):
   # code to execute if the `elif` condition is True
else:
   # code to execute if all the above conditions are False
```

Nested Conditional Statements

```
x = 15
if x > 10: # condition
   if x < 20: # Inside above condition for more complex decision making
      print("x is between 10 and 20")  # Output: x is between 10 and 20
   else:
      print("x is greater than or equal to 20")
else:
   print("x is less than or equal to 10")
```

Conditional Expressions (Ternary Operator)

Python supports a shorthand for simple if-else statements using a ternary operator.

```
        x = 5
        result = "Positive" if x > 0 else "Non-positive"
        print(result)  # Output: Positive
```

Logical Operators in Conditionals

Use logical operators (and, or, not) to combine multiple conditions.

```
x = 7
if x > 5 and x < 10:
   print("x is between 5 and 10")  # Output: x is between 5 and 10
if not x < 5:
   print("x is not less than 5")  # Output: x is not less than 5
```

Lists in Python

A built-in data type that stores set of values

It can store elements of different types (integer, float, string, etc.)

A list **is a collection of ordered, mutable, and heterogeneous elements.**

```
marks = [87, 64, 33, 95, 76]    #marks[0], marks[1]..
student = [”Karan”, 85,“Delhi”] #student[0], student[1]..
student[0] = “Arjun”            #allowed in python
len(student)                    #returns length
```

Lists are defined using square brackets [].

```
# Empty list
empty_list = []

# List of integers
numbers = [1, 2, 3, 4, 5]

# List of mixed data types
mixed = [1, "Python", 3.14, True]

# Nested lists
nested = [[1, 2], [3, 4], [5, 6]]
print(numbers)  # Output: [1, 2, 3, 4, 5]
```

Access elements using indexing or slicing.

```
# Indexing
fruits = ["apple", "banana", "cherry"]
print(fruits[0])  # Output: 'apple'

# Negative indexing
print(fruits[-1])  # Output: 'cherry'

# Slicing
print(fruits[1:3])  # Output: ['banana', 'cherry']
print(fruits[:2])   # Output: ['apple', 'banana']
```

List Slicing

Similar to String Slicing

```
list_name[ starting_idx : ending_idx ]  #ending idx is not included
marks = [87, 64, 33, 95, 76]
marks[ 1 : 4 ] is [64, 33, 95] # 4th index is not included because it’s a ending index
print(marks[:3])  # Output: [87, 64, 33] (from start to index 2)
print(marks[3:])  # Output: [95, 76] (from index 3 to the end)

print(my_list[::2])  # Output: [87, 33, 76] (every second element) pick index 0,2,4 skip 1 values in between
print(my_list[1:5:2])  # Output: [64, 95] here is 5th index is not included.

# Negative Indices
print(my_list[-3:])  # Output: [33, 95, 76] (last three elements)
print(my_list[:-2])  # Output: [87, 64, 33] (all except the last two)

# Reversing a list
print(my_list[::-1])  # Output: [5, 4, 3, 2, 1, 0]
```

List Methods

```
list = [2, 1, 3]
list.append(4) #adds one element at the end #output is [2, 1, 3, 4]
list.sort( ) #sorts in ascending order #output is [1, 2, 3]
list.sort( reverse=True ) #sorts in descending order
list.reverse( ) #reverses list [3,2,1]
list.insert( idx, el ) #insert element at index
```

Adding Elements

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)        # Output: [1, 2, 3, 4]
```

extend(iterable): Adds all elements from an iterable (like another list) to the end of the list.

```
my_list.extend([5, 6])
print(my_list)        # Output: [1, 2, 3, 4, 5, 6]
```

insert(index, x)

```
my_list.insert(2, 99)
print(my_list)        # Output: [1, 2, 99, 3, 4, 5, 6]
```

remove(x): Removes the first occurrence of the element x. Raises a ValueError if x is not found.

```
my_list.remove(99)
print(my_list)        # Output: [1, 2, 3, 4, 5, 6]
```

pop([index]): Removes and returns the element at the specified index. If no index is given, removes the last element.

```
print(my_list.pop())  # Output: 6
print(my_list)        # Output: [1, 2, 3, 4, 5]
```

clear(): Removes all elements from the list.

```
my_list.clear()
print(my_list)        # Output: []
```

count(x): Returns the number of occurrences of x in the list.

```
print(my_list.count(2))  # Output: 1
```

Sorting and Reversing

sort(key=None, reverse=False)**: Sorts the list in place.**

```
my_list.sort(reverse=True)
print(my_list)        # Output: [5, 4, 3, 2, 1]
```

reverse(): Reverses the list in place.

```
my_list.reverse()
print(my_list)        # Output: [1, 2, 3, 4, 5]
```

---

sum(): Returns the sum of the list elements (works only for numeric lists).

```
print(sum(my_list))  # Output: 15
```

max(): Returns the maximum element in the list.

```
print(max(my_list))  # Output: 5
```

min(): Returns the minimum element in the list.

```
print(min(my_list))  # Output: 1
```

Tuples in Python

In Python, a **tuple** is an immutable, ordered collection of elements. Like lists, tuples can store a sequence of values of any type, but once a tuple is created, its contents cannot be changed (i.e., no addition, removal, or modification of elements).

```
# Creating tuples
empty_tuple = ()
single_element_tuple = (1,)  # Note the trailing comma for a single-element tuple
multi_element_tuple = (1, 'hello', 3.14)
# Without parentheses (optional for grouping)
implicit_tuple = 1, 2, 3
```

Indexing and Slicing:

```
t = (10, 20, 30, 40)
print(t[1])    # Output: 20
print(t[-1])   # Output: 40
print(t[1:3])  # Output: (20, 30)
```

Concatenation and Repetition:

```
t1 = (1, 2)
t2 = (3, 4)
print(t1 + t2)  # Output: (1, 2, 3, 4)
print(t1 * 2)   # Output: (1, 2, 1, 2)
```

Length, Minimum, and Maximum:

```
t = (5, 10, 15)
print(len(t))  # Output: 3
print(min(t))  # Output: 5
print(max(t))  # Output: 15
```

Dictionary in Python

Dictionaries are used to store data values in key:value pairs. They are unordered, mutable(changeable) & don't allow duplicate keys. Dictionaries are widely used for storing and retrieving data by unique keys, instead of by index (as in lists or tuples).

```
# Using curly braces
my_dict = {"name": "Alice", "age": 25, "city": "New York"}

# Using the dict() constructor
another_dict = dict(id=101, dept="HR")

# Creating an empty dictionary
empty_dict = {}

# Access value by key
print(my_dict["name"])  # Output: Alice

# Using the `get()` method (avoids KeyError if the key doesn't exist)
print(my_dict.get("age"))    # Output: 25
print(my_dict.get("height"))  # Output: None (key does not exist)

# Default value with `get()`
print(my_dict.get("height", "Unknown"))    # Output: Unknown
```

Modifying Dictionaries:

```
# Adding or updating a key-value pair
my_dict["age"] = 26  # Update
my_dict["gender"] = "Female"  # Add
print(my_dict)

# Removing a key-value pair
del my_dict["city"]  # Remove key 'city'
print(my_dict)

# Using `pop()` method
removed_value = my_dict.pop("gender")  # Removes and returns value
print(removed_value)  # Output: Female

# Clearing the dictionary
my_dict.clear()  # Removes all elements
```

Iterating Through a Dictionary:

```
# Iterating through keys
for key in my_dict:
    print(key)

# Iterating through values
for value in my_dict.values():
    print(value)

# Iterating through key-value pairs
for key, value in my_dict.items():
    print(f"{key}: {value}")
```

Dictionary Methods:

Method	Description
`clear()`	Removes all items from the dictionary.
`copy()`	Returns a shallow copy of the dictionary.
`get(key, default)`	Returns the value for a key, or the default if the key does not exist.
`items()`	Returns a view object of key-value pairs.
`keys()`	Returns a view object of keys.
`values()`	Returns a view object of values.
`pop(key, default)`	Removes and returns the value for a key. If the key doesn't exist, returns the default.
`popitem()`	Removes and returns the last inserted key-value pair (Python 3.7+).
`update()`	Updates the dictionary with another dictionary or iterable of key-value pairs.

clear()

Removes all key-value pairs from the dictionary, leaving it empty.

```
my_dict = {"name": "Alice", "age": 25}
my_dict.clear()
print(my_dict)  # Output: {}
```

copy()

```
original = {"name": "Alice", "age": 25}
copy_dict = original.copy()
print(copy_dict)  # Output: {'name': 'Alice', 'age': 25}
```

get(key, default)

Returns the value associated with the key. If the key does not exist, returns the specified default value (or None if default is not provided).

```
my_dict = {"name": "Alice"}
print(my_dict.get("name"))       # Output: Alice
print(my_dict.get("age", "N/A"))  # Output: N/A
```

items()

Returns a view object containing key-value pairs as tuples.

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.items())  # Output: dict_items([('name', 'Alice'), ('age', 25)])

for key, value in my_dict.items():
    print(key, value)
```

keys()

Returns a view object containing the dictionary's keys.

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.keys())  # Output: dict_keys(['name', 'age'])

for key in my_dict.keys():
    print(key)
```

values()

Returns a view object containing the dictionary's values.

```
my_dict = {"name": "Alice", "age": 25}
print(my_dict.values())  # Output: dict_values(['Alice', 25])

for value in my_dict.values():
    print(value)
```

pop(key, default)

Removes the specified key and returns its value. If the key does not exist, returns the specified default (or raises a KeyError if default is not provided).

```
my_dict = {"name": "Alice", "age": 25}
age = my_dict.pop("age")
print(age)      # Output: 25
print(my_dict)  # Output: {'name': 'Alice'}
```

popitem()

Removes and returns the last inserted key-value pair as a tuple (in Python 3.7+). Raises a KeyError if the dictionary is empty.

```
my_dict = {"name": "Alice", "age": 25}
item = my_dict.popitem()
print(item)      # Output: ('age', 25)
print(my_dict)  # Output: {'name': 'Alice'}
```

setdefault(key, default)

Returns the value of the specified key. If the key does not exist, inserts the key with the specified default value.

```
my_dict = {"name": "Alice"}
value = my_dict.setdefault("age", 30)
print(value)    # Output: 30
print(my_dict)  # Output: {'name': 'Alice', 'age': 30}
```

update([other])

Updates the dictionary with key-value pairs from another dictionary or an iterable of key-value pairs. Existing keys are overwritten.

```
my_dict = {"name": "Alice", "age": 25}
my_dict.update({"age": 26, "city": "New York"})
print(my_dict)  # Output: {'name': 'Alice', 'age': 26, 'city': 'New York'}
```

Set in Python

Set is the collection of the unordered items. Each element in the set must be unique & immutable.

```
nums = { 1, 2, 3, 4 }
set2 = { 1, 2, 2, 2 } #repeated elements stored only once, so it resolved to {1, 2}
null_set = set( ) #empty set syntax
```

Set Methods

```
set.add( el ) #adds an element
set.remove( el ) # removes the element
set.clear( ) #empty the set
set.pop( )  # removes a random value
set.union( set2 ) #combines both set values & returns new
set.intersection( set2 ) #combines common values & returns new
```

OOPS (Object-Oriented Programming System)

It is programming approach based on classes and Objects. In simple language oops is a method to represent the real-world things(entity). It uses concepts like **classes** and **objects** to organize code and make it easier to understand, reuse, and maintain.

Class

A **class** is like a blueprint for an object like real world entity has some properties or behaviour.

For example, if a class is "Car," it defines what a car has (like wheels, engine) and what it can do (like drive, horn).

```
class Car: # Class name
    def __init__(self, brand, color):
        self.brand = brand      # Car's brand
        self.color = color      # Car's color

    def drive(self):
        print(f"The {self.color} {self.brand} is driving.")

# Creating a car object
my_car = Car("Toyota", "red")
# Using the drive method
my_car.drive()    #output - The red Toyota is driving.
```

Object :-

As we know class is a logical entity while an object is a physical entity or real entity that's works on classes data. An **object** is like the actual thing made from the blueprint. If the class is "Car," an object is a specific car, like "a red Toyota."

```
my_car = Car("Toyota", "red")  # Create an object
my_car.drive()  # Call the object's function
```

Inheritance

When we define a class that inherits all the properties of other class called as inheritance.

1. Single Inheritance in Python

Single inheritance is a type of inheritance where a **child class** inherits from only one **parent class**. This is the simplest form of inheritance.

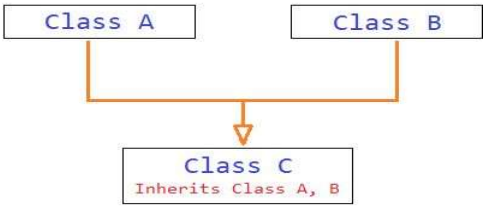
```
class ParentClass:
    # Parent class methods and attributes
    pass
class ChildClass(ParentClass):
    # Child class inherits from ParentClass
    Pass
```

Example:-

```
class Animal:
    def eat(self):
        print("This animal eats food.")
class Dog(Animal):  # Dog inherits from Animal
    def bark(self):
        print("The dog barks.")

# Creating objects
animal = Animal()
animal.eat()  # Output: This animal eats food.

dog = Dog()
dog.eat()  # Inherited method from Animal – by using child class call parent class
dog.bark()  # Unique method in Dog
```

2. Multiple Inheritance: A class can inherit from more than one parent class.

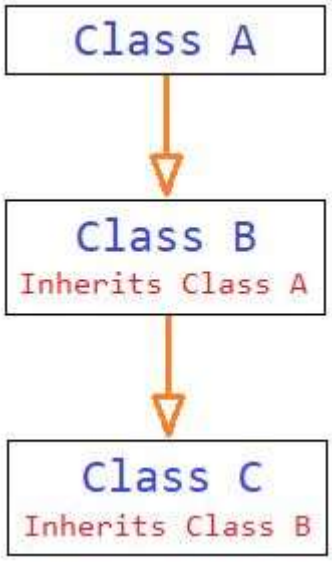
```
class A:
    pass
class B:
    pass
class C(A, B):  # Inherits from both A and B
    pass
```



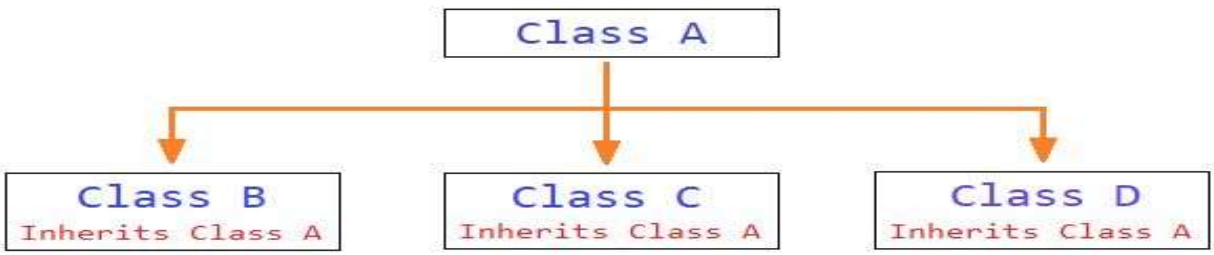
3. Multi-level inheritance :-

Class B Acquires the properties of class A and class C acquires the properties of class B is called as Multi-level Inheritance

```
class Grandparent:
    def grandparent_method(self):
        print("This is a grandparent class method")

class Parent(Grandparent):
    def parent_method(self):
        print("This is a parent class method")

class Child(Parent):
    def child_method(self):
        print("This is a child class method")

# Create an instance of the Child class
child_obj = Child()

# Access methods from all levels of inheritance
child_obj.grandparent_method()
child_obj.parent_method()
child_obj.child_method()
```



4. Hierarchical Inheritance :- When two or more classes inherit the same parent class, this type of inheritance is Hierarchical Inheritance.



Polymorphism

The term Polymorphism has been derived from the words poly means many and morphism means form. In programming, polymorphism means to create many forms from one.

There are three types of polymorphism available in python and they are:

1.Method Overloading (achieved differently in Python)
2.Method Overriding (in inheritance)

1. Polymorphism Through Method Overloading

Python does not natively support method overloading (i.e., multiple methods with the same name but different arguments). However, you can achieve it using **default arguments** or ***args/**kwargs**.

```
class Calculator:
    def add(self, a, b=0, c=0):  # Default values for parameters
        return a + b + c

calc = Calculator()
print(calc.add(5))       # Output: 5
print(calc.add(5, 10))   # Output: 15
print(calc.add(5, 10, 15))  # Output: 30
```

2. Polymorphism Through Method Overriding

In inheritance, a child class can redefine a method of its parent class. The method behaves differently based on the object calling it.

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

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

class Cat(Animal):
    def sound(self):
        print("Cats meow.")

# Polymorphism in action
animals = [Dog(), Cat(), Animal()]
for animal in animals:
    animal.sound()
#Output
Dogs bark.
Cats meow.
Animals make different sounds.
```

Functions in Python

A **function** in Python is a reusable block of code designed to perform a specific task. Functions help in organizing code, improving reusability, and reducing redundancy.

Types of Functions

1. **Built-in Functions**: Provided by Python, e.g., `print()`, `len()`, `type()`.
2. **User-defined Functions**: Created by the user to perform specific tasks.

Defining a Function

A function is defined using the `def` keyword.

1. A Simple Function

```
def greet():
    print("Hello, World!")
greet()
# Output: Hello, World!
```

2. Function with Parameters

```
def add_numbers(a, b):
    return a + b

result = add_numbers(5, 3)
print(result)
# Output: 8
```

3. Function with Default Arguments

```
def greet_person(name="Guest"):
    print(f"Hello, {name}!")

greet_person("Alice")
# Output: Hello, Alice!

greet_person()
# Output: Hello, Guest!
```

4. Function with Arbitrary Arguments

- ***args**: Accepts multiple positional arguments.
- ****kwargs**: Accepts multiple keyword arguments.

```
def display_items(*args, **kwargs):
    print("Positional arguments:", args)
    print("Keyword arguments:", kwargs)

display_items(1, 2, 3, name="Alice", age=30)
# Output:
# Positional arguments: (1, 2, 3)
# Keyword arguments: {'name': 'Alice', 'age': 30}
```

Purpose of Abstraction in Python

Abstraction is a fundamental concept in object-oriented programming (OOP) that focuses on **hiding the implementation details** of a feature and showing only its essential functionality to the user. In Python, abstraction is primarily achieved using **abstract classes** and **interfaces**

How Abstraction Works in Python

Abstraction is implemented using:

1. **Abstract Classes**
2. **Abstract Methods**

```
from abc import ABC, abstractmethod

class AbstractClassName(ABC):
    @abstractmethod
    def abstract_method(self):
        pass  # Must be implemented by subclasses
```

Example of Abstraction : Abstract Class and Abstract Method

```
from abc import ABC, abstractmethod

class Animal(ABC):
    @abstractmethod
    def sound(self):
        pass  # Abstract method

class Dog(Animal):
    def sound(self):
        return "Bark"

class Cat(Animal):
    def sound(self):
        return "Meow"

# Abstract class cannot be instantiated
# animal = Animal()  # This will raise an error

dog = Dog()
cat = Cat()
print(dog.sound())  # Output: Bark
print(cat.sound())  # Output: Meow
```

Vehicle Example Using Abstraction in Python

Here's how you can define a generic Vehicle class with abstract methods like start() and stop(). These methods will be implemented differently in subclasses such as Car and Bike.

```
from abc import ABC, abstractmethod

# Abstract base class
class Vehicle(ABC):
    @abstractmethod
    def start(self):
        pass  # Abstract method for starting the vehicle

    @abstractmethod
    def stop(self):
        pass  # Abstract method for stopping the vehicle

# Subclass for Car
class Car(Vehicle):
    def start(self):
        return "Car is starting with a key ignition."

    def stop(self):
        return "Car is stopping using hydraulic brakes."

# Subclass for Bike
class Bike(Vehicle):
    def start(self):
        return "Bike is starting with a kick or electric start."

    def stop(self):
        return "Bike is stopping using disc brakes."

# Using the classes
car = Car()
bike = Bike()

print(car.start())  # Output: Car is starting with a key ignition.
print(car.stop())   # Output: Car is stopping using hydraulic brakes.

print(bike.start())  # Output: Bike is starting with a kick or electric start.
print(bike.stop())   # Output: Bike is stopping using disc brakes.
```

Loops in python

In Python, loops are used to repeatedly execute a block of code. There are two main types of loops: **for loops** and **while loops**.

1. for loop

A for loop is used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code for each item in the sequence.

```
# Looping through a list
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
Output:
apple
banana
cherry
```

You can also use range() to iterate a specific number of times:

```
for i in range(5):  # Iterates over numbers from 0 to 4
    print(i)
Output:
0
1
2
3
4
```

2. while loop

A while loop repeatedly executes a block of code as long as a condition is True.

```
# Looping while a condition is true
count = 0
while count < 5:
    print(count)
    count += 1  # Increment the count
Output:
0
1
2
3
4
```

3. break, continue, and else in loops

- **break**: Used to exit the loop prematurely when a certain condition is met.
- **continue**: Skips the current iteration and moves to the next iteration of the loop.
- **else**: A loop can have an else block that executes if the loop completes normally (i.e., not by a break).

```
# Using break and continue
for i in range(10):
    if i == 5:
        break  # Stops the loop when i is 5
    if i % 2 == 0:
        continue  # Skips even numbers
    print(i)
Output:
1
3
```

In this case, the loop stops at i == 5, and only odd numbers less than 5 are printed.

4. Nested Loops

You can also nest loops inside each other. For example, a for loop inside another for loop.

```
for i in range(3):
    for j in range(3):
        print(f"i={i}, j={j}")
Output:
i=0, j=0
i=0, j=1
i=0, j=2
i=1, j=0
i=1, j=1
i=1, j=2
i=2, j=0
i=2, j=1
i=2, j=2
```

PANDAS IN PYTHON

Pandas is a powerful open-source library in Python used for data manipulation and analysis. It provides two primary data structures, **Series** and **DataFrame**, which are highly efficient for handling and analyzing structured data.

Key Features of Pandas

1. **Data Structures**:
   - **Series**: A one-dimensional labeled array capable of holding any data type.
   - **DataFrame**: A two-dimensional labeled data structure similar to a table in a relational database.
2. **File I/O**:
   - Read and write data from/to files in various formats such as CSV, Excel, JSON, SQL, etc.
3. **Data Manipulation**:
   - Handling missing data.
   - Filtering, slicing, and subsetting data.
   - Merging and joining datasets.
   - Grouping data and applying aggregate functions.
   - Pivoting and reshaping data.
4. **Data Cleaning**:
   - Identifying and correcting data errors.
   - Transforming and normalizing data.
5. **Time Series Support**:
   - Handling time-stamped data.
   - Resampling and frequency conversion.

Example Usage

Importing Pandas

```
import pandas as pd
```

Creating a DataFrame

```
data = {
    'Name': ['Alice', 'Bob', 'Charlie'],
    'Age': [25, 30, 35],
    'Salary': [50000, 60000, 70000]
}

df = pd.DataFrame(data)
print(df)
```

Accessing Data

```
# Access a column
print(df['Name'])

# Access a row by index
print(df.loc[0])

# Access a specific value
print(df.at[0, 'Name'])
```

Filtering Data

```
# Filter rows where Age > 28
filtered_df = df[df['Age'] > 28]
print(filtered_df)
```

Grouping and Aggregation

```
# Group by and calculate mean salary
grouped = df.groupby('Age')['Salary'].mean()
print(grouped)
```

Handling Missing Data

```
data_with_nan = {'Name': ['Alice', None, 'Charlie'], 'Age': [25, None, 35]}
df_with_nan = pd.DataFrame(data_with_nan)

# Fill missing values
df_with_nan.fillna({'Name': 'Unknown', 'Age': 0}, inplace=True)
print(df_with_nan)
```

Reading and Writing Files

```
# Read from a CSV file
df = pd.read_csv('data.csv')

# Write to a CSV file
df.to_csv('output.csv', index=False)
```

NUMPY IN PYTHON

NumPy is a powerful library in Python widely used for numerical and scientific computing. It provides support for arrays, mathematical operations, and tools for working with large datasets efficiently.

- **Faster:** NumPy arrays are faster and take up less space than regular Python lists.
- **Easy Math:** You can do calculations on whole arrays at once, like adding, multiplying, or finding averages.
- **Built-in Tools:** NumPy has ready-made functions to solve math problems, work with statistics, and much more.
- **Mathematical Functions:** Includes optimized implementations of mathematical operations (e.g., linear algebra, statistical functions).
- **Interoperability:** Works seamlessly with other Python libraries like pandas, matplotlib, and scikit-learn.

Basics of NumPy

Creating Arrays

```
import numpy as np

# 1D array
array_1d = np.array([1, 2, 3])

# 2D array
array_2d = np.array([[1, 2, 3], [4, 5, 6]])

# Create arrays with default values
zeros = np.zeros((2, 3))      # 2x3 array of zeros
ones = np.ones((3, 3))        # 3x3 array of ones
random = np.random.rand(2, 2)  # 2x2 array of random values
```

Array Operations

```
# Arithmetic operations
a = np.array([1, 2, 3])
b = np.array([4, 5, 6])
sum_array = a + b
product_array = a * b

# Broadcasting
scalar_mult = a * 2  # Multiplies each element by 2
```

Array Properties

```
array = np.array([[1, 2], [3, 4], [5, 6]])
print(array.shape)  # (3, 2) - dimensions of the array
print(array.size)   # 6 - total number of elements
print(array.ndim)   # 2 - number of dimensions
print(array.dtype)  # int64 (or the data type of the array)
```

Indexing and Slicing

```
array = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

# Access elements
element = array[1, 2]  # Access the element at row 1, column 2 (5)

# Slicing
row = array[1, :]     # Second row
column = array[:, 1]  # Second column
sub_array = array[0:2, 1:3]  # Sub-array
```

Mathematical Functions

```
array = np.array([1, 2, 3, 4, 5])
mean = np.mean(array)  # Average
std = np.std(array)    # Standard deviation
sum = np.sum(array)    # Sum of elements
```

Reshaping Arrays

```
array = np.array([1, 2, 3, 4, 5, 6])

reshaped = array.reshape(2, 3)  # Reshapes into 2x3 array
flattened = reshaped.flatten() # Converts back to 1D
```

DATA VISUALISATION

Data visualization involves the graphical representation of data to make it easier to understand and analyze trends, patterns, and relationships. In Python, several libraries help create powerful and insightful visualizations.
Key Libraries for Data Visualization in Python

1. Matplotlib

- A foundational library for creating static, animated, and interactive visualizations.
- Offers fine-grained control over plot elements.

```
import matplotlib.pyplot as plt
# Simple line plot
x = [1, 2, 3, 4, 5]
y = [2, 3, 5, 7, 11]
plt.plot(x, y, marker='o')
plt.title("Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```

2. Seaborn

- Built on Matplotlib, Seaborn provides a high-level interface for creating attractive and informative statistical graphics.
- Great for visualizing datasets with ease.

```
import seaborn as sns
import pandas as pd
# Sample dataset
data = {'Category': ['A', 'B', 'C'], 'Values': [10, 20, 15]}
df = pd.DataFrame(data)

# Bar plot
sns.barplot(x='Category', y='Values', data=df)
plt.show()
```