PYTHON PROGRAMMING NOTES BY ZEN CODE

Introduction to Python and Programming.

Programming is the process of giving instructions to a computer to perform specific tasks. This process involves writing code in a programming language that the computer can understand and execute.

Why Python?

- Python is a high-level programming language
- It is widely used in:
  - Web Development (Django, Flask)
  - Data Science and Machine Learning
  - Automation
  - Scripting
  - Pygame.

Setting Up Python and IDEs

Installing Python

→ Downloading Python:

1. Visit the official website: python.org.
2. Download the latest version for your OS.

→ Install Python:

1. Run the installer and ensure you check the box "Add Python to PATH".

→ Verify Installation

1 Open a terminal or command prompt and type:

```
Python --version
```

This should display the installed python version (e.g., Python 3.13.1)

Choosing an IDE

→ What is an IDE?

An Integrated Development (IDE) is a software application that provides an interface and tools for writing, testing, and debugging code.

→ Popular Python IDEs:

- VS Code (we will use this one as our IDE).
- Pycharm
- Jupyter Notebooks
- IDLE

Writing Your First Python Program

→ The "Hello, World!" Program

- Open a folder in your VS code and type the code written below in a new file named 'hello.py'.

```
print ("Hello, World!")
```

- Make sure to save the file with a .py extension (e.g., hello_world.py).
- Run the program:

Use the run button at the top right corner of your IDE (VS code) or type this in your VS Code integrated terminal:

```
python hello.py
```

Output:

```
Hello, World!
```

Important pointers:

print() is a built-in function used to display output

Python code is executed line by line.

Understanding Python Syntax and Basics.

→ **Indentation:**

- Python uses identation (spaces) to define blocks of code.
- Example:

```
if 5 > 2:
    print ("Five greater than two")
```

↑ The spaces before the print statement are called indentation.

→ **Whitespace:**

Python is sensitive to whitespace. Ensure consistent indentation to avoid errors.

→ **Statements:**

Each line of code is a statement. You can write multiple statements on one line using a semicolon (;).

→ **Comments:**

Use # for single-line comments
Use ''' or """ for multi-line comments
Example:

```
# This is how a single-line comment looks like
'''
This is a
multi-line comment
'''
```

Python Fundamentals

→ Variables and Data Types in Python:

- Variables are used to store data that can be used and manipulated in a program.
- A variable is created when you assign a value to it using the = operator.
- Example:

```
name = "Alice"
age = 25
height = 5.6
```

→ Variable Naming Rules:

- Variable names can contain letters, numbers and underscores.
- Variable names must start with a letter or underscore.
- Variable names are case-sensitive.
- Avoid using Python keywords as variable names (e.g., print, if, break).

→ Data Types in Python:

Python supports several built-in data types:

- Integers (int): Whole numbers (10, -7)
- Floats (float): Decimal numbers (0.002, 3.14)
- Strings (str): Text data ("World", "Zen").
- Booleans (bool): True or False Values.
- Lists: Ordered, mutable collections
- Tuples: Ordered, immutable collections.
- Sets: Unordered collections of unique elements.
- Dictionaries: key-value pairs.

→ Checking Data Types:

- Use the type() function to check the data type of a variable.

```
print(type(10))      # o/p: <class 'int'>
print(type("zen"))   # o/p: <class 'str'>
```

Typecasting in Python:

→ What is Typecasting?

- Typecasting is the process of converting one data type to another.
- Python provides built-in functions for typecasting:
  - ↳ int() : converts to integer
  - ↳ float() : converts to float
  - ↳ str() : converts to string
  - ↳ bool() : converts to boolean.

→ Examples:

```
# converting a string into an integer
num = "10"
num_int = int (num)
print (num_int)          # O/P: 10

# converting a float to integer
pic = 7.14
pic_int = int (pic)
print (pic_int)          # O/P: 7
```

Taking User Input in Python

→ input() function:

- The input() functions allows us to take user input from the end-user.
- By default, input() returns a string. You can convert it to other data types according to your needs.

→ Example:

```
name = input ("Enter name:")
reg_no = int (input ("Enter your age:"))
print (f"Your name is {name}, and
    your reg_no is {reg_no}.")
```

Comments, Escape Sequences & Print Statement

→ **Comments:**

- Comments are used to explain the written code and are ignored by the python interpreter.
- Single-line comments start with #.
- Multi-line comments are enclosed in ''' or """.

→ **Escape Sequences:**

- Escape sequences are used to include special characters in strings.
- Common escape sequences:
  - \n : Newline
  - \t : Tab
  - \\ : Backslash
  - \" : Double quote
  - \' : Single quote

→ **Print Statement:**

- The print() function is used to display output on the screen.
- You can use "sep" and "end" parameters to customize the output.
- Example:

```
print("Hello", "World", sep="*", end="\n")
```

O/P: Hello*World

Operators in Python.

→ Types of operators:

1 ↳ Arithmetic Operators:

+	- Addition	%	- Modulus
-	- Subtraction	**	- Exponentiation
*	- Multiplication	//	- Floor Division
/	- Division		

↳ Example:

```
print (7+3)          # O/P: 10
print (10 ** 2)      # O/P: 100
```

2 ↳ Comparison Operators:

==	- Equal	<	- Less Than
!=	- Not Equal	>=	- Greater than or equal
>	- Greater Than	<=	- Less than or equal

↳ Example:

```
print (31 > 7)       # O/P: True
print (31 == 7)      # O/P: False.
```

3 ↳ Logical Operators:

- and
- or
- not

↳ Example:

```
print (True and False)   # O/P: False
print (True or False)    # O/P: True
print (not True)         # O/P: False
```

4 ↳ Assignment Operators:

=, +=, -=, *=, /=, %=, **=

↳ Example:

```
x = 10
x += 5      # same as x = x + 5
print (x)   # O/P: 15.
```

5 ↳ Membership Operators:

- in
- not in

↳ Example:

```
bikes = ["TVS", "Yamaha", "Royal Enfield"]
print ("Yamaha" in bikes)   # O/P: True
```

6 ↳ Identity operators:

- is
- is not

Example:

```
x = 100
y = 100
print (x is y)   # O/P: True
```

Contral Flow and Loops

If - Else Conditional Statements :

What are conditional statements ?

- Conditional statements are statements that allow you to execute code based on certain conditions.

Syntax :

```
if condition 1 :
    # code to execute if condition 1 is True.
elif condition2 :
    # Code to execute if condition 2 is True.
else :
    # Code to execute when all conditions are False
```

Example :

```
age = 18
if age < 18 :
    print ("You cannot drive")
elif age == 18 :
    print ("You just became eligible to drive")
else :
    print ("You can drive").
```

Match Case Statements

What is Match - Case ?

- Match - case is a new feature introduced in Python 3.10 for pattern matching.
- It simplifies complex conditional logic.

→ Syntax:

```
match value:
    case p1:
      # code to execute if value matches p1
    case p2:
      # code to execute if value matches p2
    case _:
      # Default case (if no patterns match)
```

→ Example:

```
port = 404

match port:
    case 200:
        print("Success!")
    case 404:
        print("Not Found")
    case _:
        print("Unknown Status").
```

For Loops in Python

→ What are for loops?

- For loops are used to iterate over a sequence (e.g., list, string, range).
- They execute a block of code repeatedly for each item in the sequence.

→ Syntax:

```
for item in sequence:
    # Code to execute for each item.
```

→ Example:

```
bikes = ["TVS", "Honda", "Yamaha"].

for bike in bikes:
    print(bike).
```

→ Using range() function:

- The range() function generates a sequence of numbers.
- Example:

```
for i in range(5):
    print(i)    # o/p: 0, 1, 2, 3, 4
```

While Loops in Python.

→ What are While loops?

- While loops execute a block of code as long as a condition is True.
- They are useful when the number of iterations is not known in advance.

→ Syntax:

```
while condition:
    # code to execute while condition is True
```

→ Example:

```
count = 0
while count < 7:
    print(count)
    count += 1
```

→ Infinite Loops:

- Be cautious to avoid infinite loops by ensuring the condition eventually becomes False.
- Example:

```
while True:
    print("This will print forever!").
```

Break, Continue and Pass Statements

→ Break:

- The break statement is used to exit a loop prematurely.

→ Continue:

The continue statement skips the rest of the code in the current iteration and moves to the next iteration.

→ Pass:

- The pass statement is a placeholder that does nothing. It is used when syntax requires a statement but no action is needed.

Strings in Python

→ Introduction:

Strings are one of the most fundamental data types in Python. A string is a sequence of characters enclosed within either single quotes (' '), double quotes (" "), or triple quotes (''' ''') or (""" """).

→ Creating Strings:

```
# Single-quoted string
a = 'Hello, Zen!'

# Double-quoted string
b = "Hello, World!"

# Triple-quoted string
c = ''' This is
how a multi-line
string looks like
in python. '''
```

→ String Indexing :

In python, each character in a string has an index :

```
text = "zencode"
print (text[0])      #o/p : z
print (text[1])      #o/p : e
print (text[-1])     #o/p : e (last character)
```

→ String Slicing :

You can extract parts of a string using slicing.

```
text = "Welcome, zen!"
print (text[0:5])    #o/p : Welco
print (text[:5])     #o/p : welco
print (text[9:])     #o/p : zen
```

Let us take a more detailed look at how these processes work in python.

String Slicing and Indexing:

→ String Indexing:

- Each character in a string has a unique index, starting from 0 for the first character and -1 for the last character.

```
text = "zencode"
print (text[0])    # O/P: z
print (text[1])    # O/P: e
print (text[-1])   # O/P: e (last character)
print (text[-2])   # O/P: d
```

→ String Slicing:

- Slicing allows you to extract a portion of a string using the syntax:
  string [start : stop : step]

```
text = "Hello, Python!"
print (text[0:5])      # O/P: Hello
print (text[:5])       # O/P: Hello
print (text[7:])       # O/P: Python!
print (text[::2])      # O/P: Hlo Pto!
print (text[-6:-1])    # O/P: ython (negative indexing)
```

→ Step Parameter:

- The step parameter defines the interval of slicing.

```
text = "Python Programming"
print (text [::2])  # O/P: Pto rgamn
print (text [::-1]) # O/P: gnimmargorp nohtyP
```

String Methods and Functions

→ **Introduction:**
Python provides a variety of built-in string methods and functions to manipulate and process strings efficiently.

→ **Common String Methods:**

↳ **Changing Case**

```
text = "zen code"
print (text.upper())   # O/P: "ZEN CODE"
print (text.lower())   # O/P: "zen code"
print (text.title())   # O/P: "Zen Code"
print (text.capitalize()) # O/P: "Zen code".
```

↳ **Removing Whitespace**

```
text = "  hello world  "
print (text.strip())   # O/P: "hello world"
print (text.lstrip())  # O/P: "hello world  "
print (text.rstrip())  # O/P: "  hello world"
```

↳ Finding and Replacing

```
text = "Python is fun"
print (text.find ("is"))          # O/P: 7
print (text.replace ("fun", "awesome")
```

↳ # O/P: "Python is awesome"

↳ Splitting and Joining

```
text = "TVS, Honda, Yamaha"
bikes = text.split (",")
print (bikes)     # O/P: ['TVS', 'Honda', 'orange']
```

Useful Built-in String Functions

- len () - Get length of a string.

```
text = "zencode"
print (len (text))          # O/P: 7.
```

- ord() and chr () - Character Encoding

```
print (ord ('A'))     # O/P: 65
print (chr (65))      # O/P: 'A'
```

String Formatting and f-Strings

→ **Introduction:**
String formatting is a powerful feature in Python that allows you to insert variables and expressions into strings in a structured way. Python provides multiple ways to format strings, including the older `.format()` method and the modern f-strings.

→ **Using .format() Method:**
The `.format()` method allows inserting values into placeholders `{}`.

```
name = "Alex"
age = 31
print("My name is {} and I am {} years old".
  format(name, age)).
```

→ **f-Strings:**
Introduced in Python 3.6, f-strings are the most concise and readable way to format strings:

```
name = "Alex"
age = 31
print(f"My name is {name} and I
 am {age} years old.")
```

↳ Using expressions in f-strings:

You can perform calculations directly inside f-strings:

```
x = 7
y = 3
print(f"The sum of {x} and {y} is {x+y}").
```

↳ Formatting Numbers:

```
pi = 3.14159265
print(f"Pi rounded to 2 decimal places: {pi:.2f}")
```

↳ Padding and Alignment:

```
text = "Python"
print(f"{text:>10}")  # Right Align
print(f"{text:<10}")  # Left Align
print(f"{text:^10}")  # Center Align.
```

Function and Modules:

→ Defining functions in python:
Functions help in reusability and modularity in python.

↳ Syntax:

```
def greet(name):
    return f"Hi, {name}!"
print(greet("zen"))   #o/p: Hi, zen!
```

→ Function Arguments & Return Values:
Functions can take parameters and return values.

↳ Types of Arguments:

1. Positional Arguments

```
def sub(a, b):
    return a - b
print(sub(5, 3))   #o/p: 2
```

2. Default Arguments

```
def greet(name = "User"):
    return f"Hello, {name}!"
print(greet())      #o/p: Hello, User!
```

3. Keyword Arguments.

```
def student(name, age):
    print(f"Name: {name}, Age: {age}")
student(age=24, name="anshika").
```

→ Lambda Functions in Python.

Lambda functions are anonymous, inline functions.

↳ Syntax / Example:

```
square = lambda x: x * x.
print(square(4))        # O/P: 16.
```

→ Recursion in Python.

A function calling itself to solve a problem.

↳ Example: Factorial problem using Recursion.

```
def factorial(n):
    if n == 1:
        return 1
    return n * factorial(n-1)

print(factorial(5))     # O/P: 120
```

• Always have a base case to avoid infinite recursion.

→ Modules and Pip

Python provides built-in and third-party modules.

↳ Example: Using external module like math

```
import math
print (math.sqrt(16))   # O/P: 4.0.
```

↳ Installing external libraries with pip.

```
Pip install requests.
```

→ Function Scope and Lifetime

In python, variables have Scope (where they can be accessed) and lifetime (how long they exist). Variables are created when a function is called and destroyed when it returns. Understanding scope helps avoid unintended errors and improves code organization.

↳ Types of Scope in python

1. Local Scope - Variables declared inside a function are accessible only within that function.
2. Global Scope - Variables declared outside any function can be used throughout the program.

↳ Example:

```
x = 10    # Global Variable

def func1():
    x = 5  # local variable
    print(x)  # O/P: 5.
func1()
print(x)  # O/P: 10 (global x remains unchanged)
```

↳ Using the global Keyword.

To modify a global variable inside a function, use the global Keyword:

```
x = 10  # Global Variable.

def modify_global():
    global x
    x = 5      # Modifies the global x.

modify_global()
print(x)    # O/P: 5.
```

This allows functions to change global variables

↳ Docstrings - Writing Function Documentation

Docstrings are used to document functions, classes and modules. In python, they are written in triple quotes. They are accessible using the __doc__ attribute.

```
def add(a,b):
    """Returns the sum of two numbers."""
    return a+b.

print(add.__doc__)

#O/P: Returns the sum of two numbers.
```

Data Structures in Python

Python provides powerful built-in data structures to store and manipulate collections of data efficiently.

→ Lists and List Methods

Lists are ordered, mutable collection of items.

```
numbers = [1,2,3,4,5]
mixed = [7.14, "zen", 31]
```

↳ Common List Methods:

```
my_list = [1,2,3]

my_list.append(4)      # [1,2,3,4]
my_list.insert(1,99) # [1,99,2,3,4]
```

```
my_list.remove(2)   # [1, 99, 3, 4].
my_list.reverse()   # [3, 99, 1].
my_list.sort()      # [1, 3, 99].
```

↳ List Comprehensions (Efficient List Creation).

```
squared = [x**2 for x in range(5)]
print(squared)      # o/p: [0, 1, 4, 9, 16]
```

→ Tuples and Operations on Tuples

Tuples are ordered but immutable collections (they cannot be changed after creation).

↳ Creating a Tuple:

```
my_tup = (1, 2, 3)
single_element = (5,)  # Tuple with one element.
```

↳ Tuple Unpacking:

```
a, b, c = my_tup
print(a, b, c)   # o/p: 1, 2, 3.
```

→ Sets and Set Methods

Sets are unordered, unique collections. It allows no duplicates.

↳ Creating a Set:

```
bikes = {"Yamaha", "TVS", "Honda"}
```

↳ Key Set Methods:

```
my_set = {1, 2, 3, 4}

my_set.add(7)      # {1, 2, 3, 4, 7}
my_set.remove(2)   # {1, 3, 4, 7}
my_set.discard(10) # no error if element not found
my_set.pop()       # Removes random element
```

↳ Set Operations:

```
a = {1, 2, 3}
b = {3, 4, 5}

print(a.union(b))        # {1, 2, 3, 4, 5}
print(a.intersection(b)) # {3}
print(a.difference(b))   # {1, 2}
```

→ Dictionaries and Dictionary Methods.

Dictionaries store Key-value

↳ Creating a Dictionary:

```
student = {"name": "Anshika", "age": 23, "grade": "O"
```

↳ Accessing & Modifying Values:

```
print (student ["name"])  # o/p : Anshika
student ["age"] = 24      # Updating value.
student ["city"] = "Chennai" # adding new key-value pair.
```

↳ Dictionary Methods:

```
print (student.keys ())    # Prints all keys
print (student.values ())  # Prints all values
print (studeat.items ())   # Prints all key-value pairs

student.pop ("age")  # Removes "age" key
student.clear ()  # Empties dictionary.
```

↳ Dictionary Comprehensions:

```
squares = {x: x**2 for x in range (5)}
print (squares) # {0:0, 1:1, 2:4, 3:9, 4:16}.
```

Object-Oriented Programming in Python

→ What is OOP?

- Imagine you are building with LEGOs. Instead of just having a pile of individual bricks, OOP lets you create pre-assembled units - like a car, house or a robot.
- OOP is basically a way of programming that focuses on creating "objects".
- An object is like a self-contained unit that bundles together like an entity.
- Data (Attributes): Information about the object. For a car, this might be its color, model, and speed.
- Actions (Methods): Things the object can do. A car can accelerate, brake and turn.

→ Advantages of OOP:

- Organization
- Reusability
- Easier Debugging
- Real-World Modelling

→ Four Pillars of OOP

1. Abstraction : Imagine you are driving a car. You use the steering, pedals, and gearshift, but you don't need to know the complex engineering under the hood. Abstraction basically means hiding complex details and showing only the essential information to the user.

2. Encapsulation : This is like putting all the car's engine parts inside a protective casing. Encapsulation bundles data (attributes) and the methods that operate on that data within a class. This protects the data.

3. Inheritance : Imagine creating a "SportsBike" class. Instead of starting from scratch, you can build it upon an existing "Bike" class. The "~~Bike~~" SportsBike" inherits all the features of a "Bike" (like spec and engine) and adds its own special features. This promotes code reusability.

4. Polymorphism : "Poly" means many, and "morph" means forms. This means objects of different classes can respond to the same "message" (method call) in their own specific way. For example, both a "Dog" and "Cat" might have a make_sound() method. The dog will bark, the cat will meow (same method, different action)

→ Classes & Objects

- **Class**: Think of class as a blueprint or a template. It defines what an object will be like - what data will it hold and what actions it can perform. It doesn't create the object itself, just the instructions for creating it. It's like an architectural plan for a house.

- **Object**: An object is a specific instance created from the class blueprint. If "Bike" is the class, then your red yamaha MT is an object (an instance) of the "Bike" class. Each object has its own unique set of data.

- **Example**:

```
class Dog:  # Defining a class called "Dog".
    species = 'Golden Retriever" # class attribute

def __init__(self, name, breed): # constructor.
    self.name = name
    self. breed = breed.

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

Creating dog objects:

```
my_dog = Dog("Bruno", "Pug")
another_dog = Dog("Lucy", "Labrador").
```

accessing attributes from created objects:

```
print(my_dog.name)      # O/P: Bruno
print(another_dog.breed) # O/P: Labrador.
```

Making them perform actions:

```
my_dog.bark()      # O/P: Bruno says Woof!
print(Dog.species) # O/P: Golden Retriever
```

- **Self:**
  Inside a class, self is like saying "this particular object". It is a way for the object to refer to itself. It's always the first parameter in a method definition, but python handles it automatically when you call the method. Do not type self when calling the method; Python inserts it for you.

→ **The Constructor: (__init__)**

The __init__ method is special. It's called the constructor. It's automatically run whenever you create a new object from a class.

The constructor's job is to initialize the object's attributes - to give them their starting values. It sets up the initial state of the object.

→ Inheritance

Inheritance is like a family tree. A child class (or subclass) inherits traits (attributes and methods) from its parent class (or superclass). This allows you to create new classes that are specialized versions of existing classes, without rewriting all the code.

Example:

```
class Animal:  # Parent Class (superclass)
    def __init__(self, name):
        self.name = name

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

class Dog(Animal):  # Dog inherits from Animal
    def speak(self):
        print("Woof!")
```

```
class Cat(Animal): # Cat also inherits from Anima
    def speak(self):
        print("Meow!").

# Creating objects:

my_dog = Dog("Rover")
my_cat = Cat("Fluffy").

# They both have a 'name' attribute:
print(my_dog.name)  # O/P: Rover
print(my_cat.name)  # O/P: Fluffy.
```

→ Polymorphism

Polymorphism, as we saw with the speak() method in the inheritance example, means that objects of different classes can respond to the same method call in their own specific way.

→ Method Overriding

Method overriding is how polymorphism is achieved in inheritance. When a child class defines a method with the same name as a method in its parent class, the child's version overrides the parent's version for objects of the child class. This allows specialized behavior in subclasses. The parent's

class method is still available, but when you call the method on a child's class object, the child's version is executed.

→ Getters & Setters

Getters and setters are methods that you create to control how attributes of your class are accessed and modified. They are a key part of the principle of encapsulation. Instead of directly accessing an attribute, you use methods to get and set its value.

↳ Advantages of using getters & setters

- Validation
- Read-Only attributes
- Side Effects
- Maintainability and Flexibility.

Example:

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

```
    def set_age(self, new_age):
        if new_age >= 0 and new_age <= 150:
            self._age = new_age.
        else:
            print("Invalid age!").

person = Person("Alice", 30).
print(person.get_age())   # O/P: 30.

person.set_age(35).
print(person.get_age())   # O/P: 35

person.set_age(-5)        # O/P: Invalid age!
print(person.get_age)     # O/P: 35.
```

@property decorator

Python offers a more elegant and concise way to define getters and setters using the @property decorator. This is the preferred way to implement them in modern python.

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

    @property  # This makes 'age' a property
    def age(self):
        return self._age
```

```
@age.setter
def age(self, new_age):
    if new_age >= 0 and new_age <= 150:
        self._age = new_age
    else:
        print("Invalid age!")

person = Person("Bob", 40)
print(person.age)   # O/P: 40
person.age = 45
print(person.age)
person.age = -22    # O/P: Invalid age!
```

With @property, accessing and setting the age attribute looks like you're working directly with a regular attribute, but you're actually using the getter and setter method behind the scenes. This combines the convenience of direct access with the control and protection of encapsulation.

Python : Advanced Concepts

⇒ Decorators in Python

Introduction :

Decorators in Python are a powerful and expressive feature that allows you to modify or enhance functions and methods in a clean and readable way.

They provide a way to wrap additional functionality around an existing function without permanently modifying it. This is often referred to as metaprogramming, where one part of the program tries to modify another part of the program at compile time. Decorators use Python's higher-order function capability, meaning functions can accept other functions as arguments and return new functions.

↳ Understanding decorators :

A decorator is simply a callable (usually a function) that takes another function as an argument and returns a replacement function. The replacement function typically extends or alters the behaviour of the original function.

↳ Example of a decorator:

```
def my_decorator(func):
    def wrapper():
        print("Something is happening before call")
        func()
        print("Something is happening after call")
    return wrapper.

@my_decorator
def say_hello():
    print("Hello!")

say_hello()
```

O/P:

```
Something is happening before call
Hello!
Something is happening after call.
```

Here, @my_decorator is syntatic for say_hello = my_decorator(say_hello)

It modifies the behavior of say_hello() by wrapping it inside wrapper().

The wrapper function adds behavior before and after the original function call.

Using Decorators with Arguments

Decorators themselves can also accept arguments. This requires another level of nesting: an outer function that takes the decorator's arguments and returns the actual decorator function.

```
def repeat(n):
    def decorator(func):
        def wrapper(a):
            for _ in range(n):
                func(a)
        return wrapper
    return decorator

@repeat(3)
def greet(name):
    print(f"Hello, {name}!")

greet("world")
```

O/P:

```
Hello, world!
Hello, world!
Hello, world!
```

→ Static and Class Methods in Python

· Introduction.

In Python, methods within a class can be of three main types:

→ **Instance Methods**: These are the most common type of method. They operate on instances of the class (objects) and have access to the instance's data through the self parameter.

→ **Class Methods**: These methods are bound to the class itself, not to any particular instance. They have access to class-level attributes and can be used to modify the class state. They recieve the class itself as the first argument.

→ **Static Methods**: These methods are associated with the class, but they don't have access to either the instance (self) or the class (cls). They are essentially regular functions that are logically grouped within a class for organizational purposes.

→ Instance Methods (Default Behavior)

Instance methods are the default type of method in python classes. They require an instance of the class to be called, and they automatically receive the instance as the first argument (self).

Example:

```
class Dog:
    def __init__(self, name):
        self.name = name # Instance attribute

    def speak(self):
        return f"{self.name} says Woof!"

dog = Dog("Bruno")
print(dog.speak()) # Bruno says Woof!
```

→ Class Methods (@classmethod)

A class method is marked with the @classmethod decorator. It takes the class itself (cls) as its first parameter, rather than the instance (self). Class methods are often used for:

Modifying class attributes
Factory Methods

Example :

```
class Animal:
    species = "Mammal"  # class attribute

    @classmethod
    def set_species(cls, new_species):
        cls.species = new_species  # modifies class attribute

    @classmethod
    def get_species(cls):
        return cls.species

print(Animal.get_species())  # Mammal
Animal.set_species("Reptile")
print(Animal.get_species())  # Reptile
```

→ Static Methods (@staticmethod)

Static methods are marked with the @staticmethod decorator. They are similar to regular functions, except they are defined within the scope of a class.

Example :

```
class MathUtils:
    @staticmethod
    def add(a, b):
        return a + b
```

```
print (MathUtils.add (3,5))  # 8.

# can also be called on an instance.
m = MathUtils ()
print (m.add(4,5))  # 9.
```

→ Magic (Dunder) Methods in Python

· Introduction.

Magic methods also called dunder (double underscore) methods, are special methods in Python that have double underscores at the beginning and end of their names.
eg: __init__, __str__, __add__.

These methods allow you to define how your objects interact with built-in Python operators, functions, and language constructs. They provide a way to implement operator overloading and customize the behavior of your classes in a pythonic way.

→ Common magic methods

↳ __init__ - Object initialization.

The __init__ method is the constructor. It's called automatically when a new instance of a class is created. It's used to initialize the object's attributes.

Example :

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

p = Person ("Alice", 30)
print (p.name, p.age)  # Alice 30.
```

↳ __str__ and __repr__ - String Representation

__str__ : This method should return a human-readable, informal string representation of the object. It's used by the str() function and by print().

__repr__ : This method should return an unambiguous, official string representation of the object. It's used by the repr() function and in the interactive interpreter when you just type the object's name and press enter.

Example :

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

```
def __str__(self):
    return f"Person({self.name}, {self.age})"

def __repr__(self):
    return f"Person(name='{self.name}',
      age={self.age})"

p = Person("Alice", 30)
print(str(p))     # Person(Alice, 30)
print(repr(p))    # Person(name='Alice', age=30)
print(p)          # Person(Alice, 30)
```

If `__str__` is not defined, Python will use `__repr__` as a fallback for `str()` and `print()`. It's good practice to define at least `__repr__` for every class you create.

4) **`__len__` - Define behavior for `len()`**

This method allows objects of your class to work with the built-in `len()` function. It should return the "length" of the object.

Example:

```
class Book:
    def __init__(self, title, pages):
        self.title = title
        self.pages = pages
```

```
    def __len__(self):
        return self.pages

b = Book("Python Handbook", 231)
print(len(b))   # 231
```

→ Exception Handling and Custom Errors

↳ Introduction

Exceptions are events that occur during the execution of a program that disrupt the normal flow of instructions. Python provides a robust mechanism for handling exceptions using try-except blocks. This allows your program to gracefully recover from errors or unexpected situations, preventing crashes and providing informative error messages.

↳ Basic Exception Handling

The try-except block is the fundamental construct for handling exceptions:

The try block contains the code that might raise an exception.

The except block contains the code that will be executed if a specific exception occurs within the try block.

↳ Example

```
try:
    x = 10 / 0  # This will raise a ZeroDivisionError
except ZeroDivisionError:
    print("Cannot divide by zero!")
```

o/p:

```
Cannot divide by zero!
```

↳ Handling Multiple Exceptions

You can handle multiple types of exceptions using multiple except blocks or by specifying a tuple of exception types in a single except block.

↳ Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("You can't divide by zero!")
except ValueError:
```

```
    print ("Invalid input ! Please enter a number")
```

↳ Using else and finally

else: The else block is optional and is executed only if no exception occurs within the try block.

finally: The finally block is also optional and is always executed, regardless of whether an exception occured or not.

↳ Example :

```
try:
    file = open ("test.txt", "r")
    content = file.read()
except FileNotFoundError:
    print ("File not found!")
else:
    print ("File read successfully.")
    print (f"File contents: \n{content}")
finally:
    file.close()
```

↳ Raising Exceptions (raise)

You can manually raise exceptions using the raise keyword. This is the useful for signalling error conditions in your own code.

```
def check_age(age):
    if age < 18:
        raise ValueError("Age must be 18 or older!")
    return "Access granted."

try:
    print(check_age(20)) # Access granted
    print(check_age(16)) # Raises ValueError
except ValueError as e:
    print(f"Error: {e}")
```

→ Map, Filter and Reduce

↳ Introduction

Map, filter, and reduce are higher-order functions in Python that operate on iterables (lists, tuples, etc.). They provide a concise and functional way to perform common operations on sequences of data without using explicit loops.

↳ Map.

The map() function applies a given function to each item of an iterable and returns an iterator.

Syntax: map(function, iterable, ...)

function : The function to apply to each item.

iterable : The iterable (e.g., list, tuple) whose items will be processed.

Example:

```
numbers = [1, 2, 3, 4, 5]

# Squaring the numbers using map.
squared_nos = map(lambda x: x**2, numbers)
print(list(squared_nos))

# O/P: [1, 4, 9, 16, 25].
```

↳ Filter

The filter() function constructs an iterator for which a function returns True. In other words, it filters the iterable based on a condition.

Syntax : filter(function, iterable).

function : A function that returns True or False for each item. If None is passed, it defaults to checking if the element is True (truthy value).

iterable : The iterable to be filtered.

Example :

```
numbers = [1, 2, 3, 4, 5, 6]

even_nos = filter(lambda x: x%2 == 0, numbers)
print(list(even_nos))

# o/p : [2, 4, 6]
```

↳ Reduce

The reduce() function applies a function of two arguments cumulatively to the items of an iterable, from left to right, so as to reduce the iterable to a single value. Reduce is not a built-in function, it must be imported from the functools module.

Syntax : reduce(function, iterable[, initializer])

function : A function that takes two arguments.

iterable : The iterable to be reduced.

initializer (optional) : If provided, it's placed before the items of the iterable in the calculation and serves as a default when the iterable is empty.

```
from functools import reduce
numbers = [1, 2, 3, 4, 5]
# Sum of all numbers using reduce
sum_of_numbers = reduce(lambda x, y: x+y, num
print(sum_of_numbers)  # O/P: 15.

# equivalent using a loop (for sum):
total = 0
for x in numbers:
    total += x
print(total) # 15.
```

When to use map, filter, reduce vs list generator expressions:

When there is need for
- Readability
- Performance
- Functional Programming Style.

→ Walrus Operator (:=)

→ Introduction

The walrus operator (:=), introduced in Python 3.8, is an assignment expression operator. It allows you to assign a value to a variable within an expression.

→ Use Cases

↳ Conditional Expressions : The most common use case is within if statements, while loops, and list comprehensions, where you need to both test a condition and use the value that was tested.

```
# Without walrus operator
data = input("Enter a value (or 'quit' to exit): ")
while data != "quit":
    print(f"You entered: {data}")
    data = input("Enter a value (or 'quit' to exit): ")

# With walrus operator.
while (data := input("Enter a value (or 'quit' to exit): ")) != "quit":
    print(f"You entered: {data}").
```

↳ List Comprehensions : You can avoid repeated calculations or function calls within a list comprehension.

```
nos = [1, 2, 3, 4, 5]

# Without walrus operator : Calculate x*2 twice
results = [x * 2 for x in numbers if x * 2 > 5]

# With walrus operator : Calculate x*2 only once
results = [y for x in numbers if (y := x * 2) > 5]
```

↳ Reading Files : You can read lines from a file and process them within a loop.

```
# Without Walrus
with open("my-file.txt", "r") as f:
    line = f.readline()
    while line:
        print(line.strip())
        line = f.readline()

# With walrus
with open("my-file", "r") as f:
    while (line := f.readline()):
        print(line.strip()).
```

→ Args and Kwargs

→ Introduction

*args and **Kwargs are special syntaxes in Python functions that allow you to pass a variable number of arguments to a function. They are used when you don't know in advance how many arguments a function might need to accept.

- *args : Allows you to pass a variable number of positional arguments.
- **Kwargs : Allows you to pass a variable number of keyword arguments.

→ *args (Positional Arguments)

*args collects any extra positional arguments passed to a function into a tuple. The name args is just a convention; you could use any valid variable name preceded by a single asterik

```
def my_function (*args):
    print (type (args)) #(class 'tuple')
    for arg in args:
        print (arg).

my_function (1, 2, 3, "hello") #o/p: 1 2 3 hello
```

→ **Kwargs (Keyword Arguments).

**Kwargs collects any extra keyword arguments passed into a dictionary. Again, Kwargs is the conventional name, but you could use any valid variable name preceded by two asterisks.

```
def my_function(**kwargs):
    print(type(kwargs)) # <class 'dict'>
    for key, value in kwargs.items():
        print(f"{key}: {value}").

my_function(name = "Alice", age = 30, city = "Drass")

# O/P:
# name : Alice
# age : 30
# city : Drass

my_function() # No output (empty dictionary)
my_function(a=1, b=2)

# O/P:
# a: 1
# b: 2
```

→ File Handling and OS operations

This section introduces you to file handling in Python, which allows your programs to interact with files on your computer.

File I/O refers to reading data from and writing data to files. Python provides built-in functions to make this process straightforward.

→ Read, Write, and Append Files

Python provides several modes for opening files:

- 'r' (Read mode): Opens the file for reading. This is the default mode. If the file doesn't exist, you'll get an error.
- 'w' (Write mode): Opens the file for writing. If the file exists, its contents will be overwritten. If the file doesn't exist, a new file will be created.
- 'a' (Append mode): Opens the file for appending. Data will be added to the end of the file. If the file doesn't exist, a new file will be created.

Reading from a file:

```
try:
    file = open("my_file.txt", "r")
    content = file.read()
    print(content)
    file.close()
except FileNotFoundError:
    print("File not found.")
```

Writing to a file:

```
file = open("new_file.txt", "w")
file.write("Hello, world!\n")
file.write("This is a new line.\n")
file.close().
```

Appending to a file:

```
file = open("my_file.txt", "a")
file.write("This is appended text.\n")
file.close()
```

Using with statement:

The with statement provides a cleaner way to work with files. It automatically closes the file, even if errors occurs.

```
try:
    with open("my-file.txt", "r") as file:
        content = file.read()
        print(content)
except FileNotFoundError:
        print("File Not Found.")
with open("output.txt", "w") as file:
        file.write("Data written using 'with'.\n")
```

→ OS and Shutil Modules in Python.

Python's os module provides functions for interacting with the operating system, such as working with directories and files. The shutil module offers higher-level file operations.

os module examples:

```
import os

# Get the current working directory.

current_dir = os.getcwd()
print("Current directory:", current_dir)

# Create a new directory.
# os.mkdir("new_directory") # creates 1 lvl of directory
# os.makedirs("path/to/new_directory") # creates nested directories

# List files and directories in a directory
files = os.listdir(".") # "." represents current directory
```

```
print ("Files in current directory:", files)

# Remove a file or directory
os.remove ("my_file.txt")

# Rename a file or directory
os.rename ("old_name.txt", "new_name.txt")
```

shutil module examples :

```
import shutil

# Copy a file
shutil.copy ("my_file.txt", "my_file_copy.txt")

# Move a file or directory
shutil.move ("my_file.txt", "new_directory/")
```

→ Creating Command Line Utilities

You can use Python to create simple command-line utilities. The argparse module makes it easier to handle command-line arguments.

```
import argparse

parser = argparse.ArgumentParser = "A cmd-line utility"
parser.add_argument ("filename", help = "file to process")
parser.add_argument ("-n", "--number", type = int,
    default = 1)
```

```
args = parser.parse_args()

try:
    with open(args.filename, "r") as file:
        content = file.read()
        for _ in range(args.number):
            print(content)
except FileNotFoundError:
    print("File not found.")
```

To run this script from the command line:

```
python my-script.py my_file.txt -n 3
```

Working with External Libraries

This section introduces you to the world of external libraries in Python. These libraries extend Python's capabilities and allow you to perform complex tasks more easily.

→ Virtual Environment & Package Management

As you start working on more Python projects, you'll likely use different versions of libraries. Virtual environments help isolate project dependencies, preventing conflicts between different projects.

Virtual Environments:

A virtual environment is a self-contained directory that contains its own Python interpreter and libraries. This means that libraries installed in one virtual environment won't interfere with libraries in another.

Creating a virtual environment:

```
python3 -m venv my-env
```

Activating the virtual environment:

- Windows: my-env / Scripts / activate
- macOS / Linux: source my-env / bin / activate.

Package Management (using pip):

pip is Python's package installer. It's used to install, upgrade, and manage external libraries.

Installing a package:

```
pip install requests # installs the "requests" library
pip install numpy == 1.20.0 # installs a specific vers
```

Listing installed packages:

```
pip list
```

↳ Upgrading a package:

```
pip install --upgrade requests
```

↳ Uninstalling a package:

```
pip uninstall requests.
```

↳ Deactiviting the virtual environment:

```
deactivate.
```

→ Requests Module - Working with APIs

The requests library simplifies making HTTP requests. This is essential for interacting with web APIs.

```
import requests

url = "https://api.github.com/users/octocat"
response = requests.get(url)

if response.status_code == 200:
    data = response.json() # Parse the JSON response
    print(data["name"]) # Access data from JSON
else:
    print(f"Error: {response.status_code}").

# Making a POST request (for sending data to an API):
data = {"key": "value"}
response = requests.post(url, json=data)
```

→ Regular Expressions in Python

Regular expressions (regex) are powerful tools for pattern matching in strings. Python's re module provides support for regex.

```
import re

text = "The quick brown fox jumps over the lazy dog."

# search for a pattern
match = re.search("brown", text)
if match:
    print("Match found!")
    print("Start:", match.start())
    print("End:", match.end())

# Find all occurrences of a pattern
matches = re.findall("the", text, re.IGNORECASE)
print("Matches:", matches)

# Replace all occurences of a pattern
new_text = re.sub("fox", "cat", text)
print("new text:", new_text).
```

→ Multithreading

These techniques allow your programs to perform multiple tasks concurrently, improving performance.

↳ Multithreading (using threading module):

Multithreading is suitable for I/O bound tasks (e.g., waiting for network requests).

```
import threading
import time

def worker(num):
    print(f"Thread {num}: Starting")
    time.sleep(2) # simulate some work.
    print(f"Thread {num}: Finishing").

threads = []
for i in range(3):
    thread = threading.Thread(target=worker, args=(i,))
    threads.append(thread)
    thread.start()

for thread in threads:
    thread.join # wait for all threads to finish

print("All threads completed.")
```