🌸 Introduction to Python and Programming 🌸

🌿 What is Programming? 🌿

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

🌼 Why Python? 🌼

Python is a high-level, interpreted programming language known for its simplicity and readability.

It is widely used in:

- 🌷 Web Development (Django, Flask)
- 🍀 Data Science and Machine Learning (Pandas, NumPy, TensorFlow)
- 🌸 Automation and Scripting
- 🌻 Game Development (Pygame)

Python has a large community and extensive libraries, making it beginner-friendly.

🌿 Setting Up Python and IDEs 🌿

🌼 Installing Python 🌼

1. Download Python:
   - Visit the official Python website: python.org.
   - Download the latest version for your operating system (Windows, macOS, or Linux).
2. Install Python:
   - Run the installer and ensure you check the box to Add Python to PATH (important for running Python from the command line).
3. Verify Installation:
   - Open a terminal or command prompt and type:

     ```
     python
     ```

     ```
     python --version
     ```

   - This should display the installed Python version (e.g., Python 3.13.5).

🌷 Choosing an IDE 🌷

What is an IDE?

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

Popular Python IDEs:

1. 🌸 VS Code: Lightweight, customizable, and supports extensions for Python. (We will use this one as our primary IDE)
2. 🍀 PyCharm: Powerful IDE with advanced features for professional developers.
3. 🌼 Jupyter Notebook: Great for data science and interactive coding.
4. 🌿 IDLE: Comes pre-installed with Python; good for beginners.

🌺 Writing Your First Python Program 🌺

The "Hello, World!" Program 🌻

Open a folder in your VS code and type the following code in a new file named hello.py:

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

Make sure to save the file with a .py extension (e.g., hello.py).

Run the program:

- Use the run button at the top of your IDE or alternatively type this in your VS Code integrated terminal:

```
python hello.py
```

Output:

```
Hello, World!
```

Key Takeaways:

- print() is a built-in function used to display output.
- Python code is executed line by line.

🌸 Understanding Python Syntax and Basics 🌸

🌿 Python Syntax Rules 🌿

🌼 Indentation: 🌼

Python uses indentation (spaces or tabs) to define blocks of code.
Example:

```python
if 5 > 2:
    print("Five is greater than two!")
    # Spaces before print are called indentation
```

🌷 Whitespace: 🌷

Python is sensitive to whitespace. Ensure consistent indentation to avoid errors. Ideally, use 4 spaces for indentation.

🍀 Statements: 🍀

Each line of code is a statement. You can write multiple statements on one line using a semicolon (;), but this is not recommended.

🌸 Comments: 🌸

- Use # for single-line comments.
- Use ''' or """ for multi-line comments.

Example:

```python
# This is a single-line comment
'''
This is a
multi-line comment
'''
```

🌺 Notes from Instructor 🌺

- Python is a versatile and beginner-friendly programming language.
- Setting up Python and choosing the right IDE is the first step to writing code.
- Python syntax is simple but requires attention to indentation and whitespace.

- Start with small programs like "Hello, World!" to get comfortable with the basics.

🌸 Python Fundamentals 🌸

🌿 Variables and Data Types in Python 🌿

🌼 What are Variables? 🌼

- 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:

```python
name = "Vansh"
age = 19
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, else).

🍀 Best Practices 🍀

- Use descriptive names that reflect the purpose of the variable.
- Use lowercase letters for variable names.
- Separate words using underscores for readability (e.g., first_name, total_amount).

🌸 Data Types in Python 🌸

Python supports several built-in data types:

- 🌼 Integers (int): Whole numbers (e.g., 10, -5).
- 🌷 Floats (float): Decimal numbers (e.g., 3.14, -0.001).
- 🍀 Strings (str): Text data enclosed in quotes (e.g., "Hello", 'Python').
- 🌸 Booleans (bool): Represents True or False.
- 🌻 Lists: Ordered, mutable collections (e.g., [1, 2, 3]).

- 🌿 Tuples: Ordered, immutable collections (e.g., (1, 2, 3)).
- 🌺 Sets: Unordered collections of unique elements (e.g., {1, 2, 3}).
- 💐 Dictionaries: Key-value pairs (e.g., {"name": "Vansh", "age": 19}).

🍃 Checking Data Types 🍃

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

```python
print(type(10))  # Output: <class 'int'>
print(type("Hello"))  # Output: <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:

python

```python
# Convert string to integer
num_str = "10"
num_int = int(num_str)
print(num_int)  # Output: 10

# Convert integer to string
num = 19
num_str = str(num)
print(num_str)  # Output: "19"

# Convert float to integer
pi = 3.14
pi_int = int(pi)
print(pi_int)  # Output: 3
```

🌸 Taking User Input in Python 🌸

🌼 Using the input() Function 🌼

- The input() function allows you to take user input from the keyboard.
- By default, input() returns a string. You can convert it to other data types as needed.

Example:

python

```python
name = input("Enter your name: ")
age = int(input("Enter your age: "))
print(f"Hello {name}, you are {age} years old.")
```

🌿 Comments, Escape Sequences & Print Statement 🌿

🌷 Comments 🌷

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

```python
# This is a single-line comment
'''
This is a
multi-line comment
'''
```

🍀 Escape Sequences 🍀

Escape sequences are used to include special characters in strings.

Common escape sequences:

- \n: Newline
- \t: Tab
- \\: Backslash
- \": Double quote
- \': Single quote

Example:

```python
print("Hello\nWorld!")
print("This is a tab\tcharacter.")
```

🌸 Print Statement 🌸

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

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

💐 Operators in Python 💐

Types of Operators

1. 🌷 Arithmetic Operators:
    - - (Addition), - (Subtraction), * (Multiplication), / (Division), % (Modulus), ** (Exponentiation), //

(Floor Division).

Example:

python

```
print(10 + 5)  # Output: 15
print(10 ** 2)  # Output: 100
```

2. 🍀 Comparison Operators:
   - == (Equal), != (Not Equal), > (Greater Than), < (Less Than), >= (Greater Than or Equal), <= (Less Than or Equal).

   Example:

   python

   ```
   print(10 > 5)  # Output: True
   print(10 == 5)  # Output: False
   ```

3. 🌸 Logical Operators:
   - and, or, not.

   Example:

   python

   ```
   print(True and False)  # Output: False
   print(True or False)  # Output: True
   print(not True)  # Output: False
   ```

4. 🌻 Assignment Operators:
   - =, +=, -=, *=, /=, %=, **=, //=.

   Example:

   python

   ```
   x = 10
   x += 5  # Equivalent to x = x + 5
   print(x)  # Output: 15
   ```

5. 🌿 Membership Operators:
   - in, not in.

   Example:

python

```python
fruits = ["apple", "banana", "cherry"]
print("banana" in fruits)  # Output: True
```

6. 🌺 Identity Operators:
   - is, is not.

   Example:

python

```python
x = 10
y = 10
print(x is y)  # Output: True
```

🌼 Summary 🌼

- Variables store data, and Python supports multiple data types.
- Typecasting allows you to convert between data types.
- Use input() to take user input and print() to display output.
- Comments and escape sequences help make your code more readable.
- Python provides a variety of operators for performing operations on data.

🌸 Control Flow and Loops 🌸

🌿 If-Else Conditional Statements 🌿

🌼 What are Conditional Statements? 🌼

- Conditional statements allow you to execute code based on certain conditions.
- Python uses if, elif, and else for decision-making.

Syntax:

python

```python
if condition1:
    # Code to execute if condition1 is True
elif condition2:
    # Code to execute if condition2 is True
else:
    # Code to execute if all conditions are False
```

Example:

```python
age = 19
if age < 18:
    print("You are a minor.")
elif age == 18:
    print("You just became an adult!")
else:
    print("You are an adult.")
```

🌷 Match Case Statements in Python (Python 3.10+) 🌷

🍀 What is Match-Case? 🍀

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

Syntax:

```python
match value:
    case pattern1:
        # Code to execute if value matches pattern1
    case pattern2:
        # Code to execute if value matches pattern2
    case _:
        # Default case (if no patterns match)
```

Example:

```python
status = 404
match status:
    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:

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

Example:

```python
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

Using range():

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

Example:

```python
for i in range(5):
    print(i)  # Output: 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:

```python
while condition:
    # Code to execute while condition is True
```

Example:

```python
count = 0
while count < 5:
    print(count)
    count += 1
```

🌼 Infinite Loops: 🌼

- Be careful to avoid infinite loops by ensuring the condition eventually becomes False.

Example of an infinite loop:

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

🌷 Break, Continue, and Pass Statements 🌷

🍀 Break 🍀

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

Example:

```python
for i in range(10):
    if i == 5:
        break
    print(i)  # Output: 0, 1, 2, 3, 4
```

🌸 Continue 🌸

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

Example:

```python
for i in range(5):
    if i == 2:
        continue
    print(i)  # Output: 0, 1, 3, 4
```

🌻 Pass 🌻

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

Example:

```python
for i in range(5):
    if i == 3:
        pass  # Do nothing
    print(i)  # Output: 0, 1, 2, 3, 4
```

🌿 Summary 🌿

- Use if, elif, and else for decision-making.
- Use match-case for pattern matching (Python 3.10+).
- Use for loops to iterate over sequences and while loops for repeated execution based on a condition.
- Control loop execution with break, continue, and pass.

🌸 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 🌷

You can create strings in Python using different types of quotes:

python

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

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

# Triple-quoted string (useful for multi-line strings)
c = '''This is
a multi-line
string.'''
```

🍀 String Indexing 🍀

Each character in a string has an index:

python

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

🌸 String Slicing 🌸

You can extract parts of a string using slicing:

python

```python
text = "Hello, Python!"
print(text[0:5])  # Output: Hello
print(text[:5])  # Output: Hello
print(text[7:])  # Output: Python!
print(text[::2])  # Output: Hlo Pto!
```

🌻 String Methods 🌻

Python provides several built-in methods to manipulate strings:

python

```python
text = " hello world "
print(text.upper())  # Output: " HELLO WORLD "
print(text.lower())  # Output: " hello world "
print(text.strip())  # Output: "hello world"
print(text.replace("world", "Python"))  # Output: " hello Python "
print(text.split())  # Output: ['hello', 'world']
```

🌿 String Formatting 🌿

Python offers multiple ways to format strings:

python

```python
name = "EII"
age = 19

# Using format()
print("My name is {} and I am {} years old.".format(name, age))

# Using f-strings (Python 3.6+)
print(f"My name is {name} and I am {age} years old.")
```

🌺 Multiline Strings 🌺

Triple quotes allow you to create multi-line strings:

python

```python
message = '''
Hello,
This is a multi-line string example.
Goodbye!
'''
print(message)
```

🌼 Summary 🌼

- Strings are sequences of characters.
- Use single, double, or triple quotes to define strings.
- Indexing and slicing allow accessing parts of a string.
- String methods help modify and manipulate strings.

- f-strings provide an efficient way to format strings.

🌸 String Slicing and Indexing 🌸

🌿 Introduction 🌿

In Python, strings are sequences of characters, and each character has an index. You can access individual characters using indexing and extract substrings using slicing.

🌼 String Indexing 🌼

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

python

```python
text = "Python"
print(text[0])  # Output: P
print(text[1])  # Output: y
print(text[-1])  # Output: n (last character)
print(text[-2])  # Output: o
```

🌷 String Slicing 🌷

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

].

python

```python
text = "Hello, Python!"
print(text[0:5])  # Output: Hello
print(text[:5])  # Output: Hello (same as text[0:5])
print(text[7:])  # Output: Python! (from index 7 to end)
print(text[::2])  # Output: Hlo Pto!
print(text[-6:-1])  # Output: ython (negative indexing)
```

🍀 Step Parameter 🍀

The step parameter defines the interval of slicing.

```python
text = "Python Programming"
print(text[::2])  # Output: Pto rgamn
print(text[::-1])  # Output: gnimmargorP nohtyP (reverses string)
```

🌸 Practical Uses of Slicing 🌸

String slicing is useful in many scenarios:

- Extracting substrings
- Reversing strings
- Removing characters
- Manipulating text efficiently

```python
text = "Welcome to Python!"
print(text[:7])  # Output: Welcome
print(text[-7:])  # Output: Python!
print(text[3:-3])  # Output: come to Pyt
```

🌻 Summary 🌻

- Indexing allows accessing individual characters.
- Positive indexing starts from 0, negative indexing starts from -1.
- Slicing helps extract portions of a string.
- The step parameter defines the interval for selection.
- Using [::-1] reverses a string.

🌸 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 🌷

python

```python
text = "hello world"
print(text.upper())  # Output: "HELLO WORLD"
print(text.lower())  # Output: "hello world"
print(text.title())  # Output: "Hello World"
print(text.capitalize())  # Output: "Hello world"
```

🍀 Removing Whitespace 🍀

python

```python
text = " hello world "
print(text.strip())  # Output: "hello world"
print(text.lstrip())  # Output: "hello world "
print(text.rstrip())  # Output: " hello world"
```

🌸 Finding and Replacing 🌸

python

```python
text = "Python is fun"
print(text.find("is"))  # Output: 7
print(text.replace("fun", "awesome"))  # Output: "Python is awesome"
```

🌻 Splitting and Joining 🌻

python

```python
text = "apple,banana,orange"
fruits = text.split(",")
print(fruits)  # Output: ['apple', 'banana', 'orange']

new_text = " - ".join(fruits)
print(new_text)  # Output: "apple - banana - orange"
```

🌿 Checking String Properties 🌿

python

```python
text = "Python123"
print(text.isalpha())  # Output: False
print(text.isdigit())  # Output: False
print(text.isalnum())  # Output: True
print(text.isspace())  # Output: False
```

🌺 Useful Built-in String Functions 🌺

🌼 len() - Get Length of a String 🌼

python

```python
text = "Hello, Python!"
print(len(text))  # Output: 14
```

🌷 ord() and chr() - Character Encoding 🌷

python

```python
print(ord('A'))  # Output: 65
print(chr(65))  # Output: 'A'
```

🍀 format() and f-strings 🍀

python

```python
name = "Vansh"
age = 19
print("My name is {} and I am {} years old.".format(name, age))
print(f"My name is {name} and I am {age} years old.")
```

🌸 Summary 🌸

- Python provides various string methods for modification and analysis.
- Case conversion, trimming, finding, replacing, splitting, and joining are commonly used.
- Functions like len(), ord(), and chr() are useful for working with string properties.

🌸 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 {}:

python

```python
name = "Vansh"
age = 19
print("My name is {} and I am {} years old.".format(name, age))
```

You can also specify positional and keyword arguments:

python

```python
print("{1} is learning {0}".format("Python", "Vansh"))  # Output: Vansh is Learning Python
print("{name} is {age} years old".format(name="EII", age=19))
```

🌷 f-Strings (Formatted String Literals) 🌷

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

python

```python
name = "Vansh"
age = 19
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:

python

```python
x = 10
y = 5
print(f"The sum of {x} and {y} is {x + y}")
```

🌸 Formatting Numbers 🌸

python

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

🌻 Padding and Alignment 🌻

python

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

🌿 Important Notes 🌿

- **Escape Sequences**: Use \n, \t, ', ", and \ to handle special characters in strings.
- **Raw Strings**: Use r"string" to prevent escape sequence interpretation.
- **String Encoding & Decoding**: Use .encode() and .decode() to work with different text encodings.
- **String Immutability**: Strings in Python are immutable, meaning they cannot be changed after creation.
- **Performance Considerations**: Using ''.join(list_of_strings) is more efficient than concatenation in loops.

🌺 Summary 🌺

- .format() allows inserting values into placeholders.
- f-strings provide an intuitive and readable way to format strings.
- f-strings support expressions, calculations, and formatting options.

🌸 Functions and Modules 🌸

🌿 1. Defining Functions in Python 🌿

Functions help in reusability and modularity in Python.

Syntax:

python

```python
def greet(name):
    return f"Hello, {name}!"

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

Key Points:

- Defined using def keyword.
- Function name should be meaningful.
- Use return to send a value back.

🌼 2. Function Arguments & Return Values 🌼

Functions can take parameters and return values.

Types of Arguments:

🌷 Positional Arguments 🌷

python

```python
def add(a, b):
    return a + b

print(add(5, 3))  # Output: 8
```

🍀 Default Arguments 🍀

python

```python
def greet(name="Guest"):
    return f"Hello, {name}!"

print(greet())  # Output: Hello, Guest!
```

🌸 Keyword Arguments 🌸

python

```python
def student(name, age):
    print(f"Name: {name}, Age: {age}")

student(age=19, name="Vansh")
```

🌻 3. Lambda Functions in Python 🌻

Lambda functions are anonymous, inline functions.

Syntax:

python

```python
square = lambda x: x * x
print(square(4))  # Output: 16

numbers = [1, 2, 3, 4]
squared = list(map(lambda x: x**2, numbers))
print(squared)  # Output: [1, 4, 9, 16]
```

🌿 4. Recursion in Python 🌿

A function calling itself to solve a problem.

Example: Factorial using Recursion

python

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

print(factorial(5))  # Output: 120
```

Important Notes:

- Must have a base case to avoid infinite recursion.
- Used in algorithms like Fibonacci, Tree Traversals.

🌺 5. Modules and Pip - Using External Libraries 🌺

🌼 Importing Modules 🌼

Python provides built-in and third-party modules.

Example: Using the math module

```python
import math
print(math.sqrt(16))  # Output: 4.0
```

🌷 Creating Your Own Module 🌷

Save this as mymodule.py:

```python
def greet(name):
    return f"Hello, {name}!"
```

Import in another file:

```python
import mymodule
print(mymodule.greet("EII"))  # Output: Hello, EII!
```

🍀 Installing External Libraries with pip 🍀

```
pip install requests
```

Example usage:

```python
import requests
response = requests.get("https://api.github.com")
print(response.status_code)
```

🌸 6. 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 (inside a function) – Variables declared inside a function are accessible only within that function.
2. 🌿 Global Scope (accessible everywhere) – Variables declared outside any function can be used throughout the program.

Example:

```python
x = 10  # Global variable
def my_func():
    x = 5  # Local variable
    print(x)  # Output: 5
my_func()
print(x)  # Output: 10 (global x remains unchanged)
```

🌼 Using the global Keyword 🌼

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

```python
x = 10  # Global variable
def modify_global():
    global x
    x = 5  # Modifies the global x
modify_global()
print(x)  # Output: 5
```

This allows functions to change global variables, but excessive use of global is discouraged as it can make debugging harder.

🌷 7. 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. Here's an example:

python

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

print(add.__doc__)  # Output: Returns the sum of two numbers.
```

Here is even proper way to write docstrings:

python

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

    Parameters:
    a (int): The first number.
    b (int): The second number.

    Returns:
    int: The sum of the two numbers.
    """
    return a + b
```

🍀 Summary 🍀

- Functions help in reusability and modularity.
- Functions can take arguments and return values.
- Lambda functions are short, inline functions.
- Recursion is a technique where a function calls itself.
- Modules help in organizing code and using external libraries.
- Scope and lifetime of variables decide their accessibility.
- Docstrings are used to document functions, classes, and modules.

🌺 Data Structures in Python 🌺

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

🌷 1. Lists and List Methods 🌷

Lists are ordered, mutable (changeable) collections of items. 🌸

Creating a List:

python

```python
numbers = [1, 2, 3, 4, 5]
mixed = [10, "hello", 3.14]
```

Common List Methods: 💐

python

```python
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.pop()          # Removes last element -> [1, 99, 3]
my_list.reverse()      # [3, 99, 1]
my_list.sort()         # [1, 3, 99]
```

List Comprehensions (Efficient List Creation) 🌿

python

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

🍀 2. Tuples and Operations on Tuples 🍀

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

Creating a Tuple: 🌸

python

```python
my_tuple = (10, 20, 30)
single_element = (5,)  # Tuple with one element (comma required)
```

Accessing Tuple Elements:

python

```python
print(my_tuple[1])  # Output: 20
```

Tuple Unpacking: 🌼

python

```python
a, b, c = my_tuple
print(a, b, c)  # Output: 10 20 30
```

Common Tuple Methods: 🌿

Method	Description	Example	Output
count(x)	Returns the number of times x appears in the tuple	(1, 2, 2, 3).count(2)	2
index(x)	Returns the index of the first occurrence of x	(10, 20, 30).index(20)	1

python

```python
my_tuple = (1, 2, 2, 3, 4)
print(my_tuple.count(2))  # Output: 2
print(my_tuple.index(3))  # Output: 3
```

Why Use Tuples? 🌺

- Faster than lists (since they are immutable)
- Used as dictionary keys (since they are hashable)
- Safe from unintended modifications

🌸 3. Sets and Set Methods 🌸

Sets are unordered, unique collections (no duplicates). 🍀

Creating a Set: 🌼

python

```python
fruits = {"apple", "banana", "cherry"}
my_set = {1, 2, 3, 4}
```

Key Set Methods: 🌿

python

```python
my_set.add(5)       # {1, 2, 3, 4, 5}
my_set.remove(2)    # {1, 3, 4, 5}
my_set.discard(10)  # No error if element not found
my_set.pop()        # Removes random element
```

Set Operations: 🌺

python

```python
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}
```

Use Case: Sets are great for eliminating duplicate values. 🌷

🍀 4. Dictionaries and Dictionary Methods 🍀

Dictionaries store key-value pairs and allow fast lookups. 🌸

Creating a Dictionary: 🌼

python

```python
student = {"name": "Vansh", "age": 19, "grade": "A"}
```

Accessing & Modifying Values: 🌿

python

```python
print(student["name"])     # Output: Vansh
student["age"] = 19        # Updating value
student["city"] = "New York"  # Adding new key-value pair
```

Common Dictionary Methods: 🌺

python

```python
print(student.keys())    # dict_keys(['name', 'age', 'grade', 'city'])
print(student.values())  # dict_values(['Vansh', 19, 'A', 'New York'])
print(student.items())   # dict_items([('name', 'Vansh'), ('age', 19), ...])
student.pop("age")       # Removes "age" key
student.clear()          # Empties dictionary
```

Dictionary Comprehensions: 🌷

python

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

🌿 5. When to Use Each Data Structure? 🌿

Data Structure	Features	Best For
List	Ordered, Mutable	Storing sequences, dynamic data
Tuple	Ordered, Immutable	Fixed collections, dictionary keys
Set	Unordered, Unique	Removing duplicates, set operations
Dictionary	Key-Value Pairs	Fast lookups, structured data

🌷 Object-Oriented Programming (OOP) in Python 🌷

We'll now explore how to organize and structure your Python code using objects, making it more manageable, reusable, and easier to understand.

🌸 1. What is OOP Anyway? 🌸

Imagine you're building with LEGOs. Instead of just having a pile of individual bricks (like in procedural programming), OOP lets you create pre-assembled units – like a car, a house, or a robot. These units have specific parts (data) and things they can do (actions). 🌿

That's what OOP is all about. It's a way of programming that focuses on creating "objects." An object is like a self-contained unit that bundles together:

- **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. 🌺

Why Bother with OOP? 🍀

OOP offers several advantages:

- **Organization**: Your code becomes more structured and easier to navigate. Large projects become much more manageable. 🌷
- **Reusability**: You can use the same object "blueprints" (classes) multiple times, saving you from writing the same code over and over. 🌸
- **Easier Debugging**: When something goes wrong, it's often easier to pinpoint the problem within a specific, self-contained object. 🌼
- **Real-World Modeling**: OOP allows you to represent real-world things and their relationships in a natural way. 🌿

The Four Pillars of OOP 🌺

OOP is built on four fundamental principles:

1. **Abstraction**: Think of driving a car. You use the steering wheel, pedals, and gearshift, but you don't need to know the complex engineering under the hood. Abstraction 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 from being accidentally changed or misused from outside the object. It controls access. 🌸
3. **Inheritance**: Imagine creating a "SportsCar" class. Instead of starting from scratch, you can build it upon an existing "Car" class. The "SportsCar" inherits all the features of a "Car" (like wheels and an engine) and adds its own special features (like a spoiler). This promotes code reuse and reduces redundancy. 🌼
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 a "Cat" might have a make_sound() method. The dog will bark, and the cat will meow – same method name, different behavior. 🌿

🍀 2. Classes and Objects: The Blueprint and the Building 🍀

- **Class**: Think of a class as a blueprint or a template. It defines what an object will be like – what data it will 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 (Instance)**: An object is a specific instance created from the class blueprint. If "Car" is the class, then your red Honda Civic is an object (an instance) of the "Car" class. Each object has its own unique set of data. It's like the actual house built from the architectural plan. 🌼

Let's see this in Python: 🌿

python

```python
class Dog:  # We define a class called "Dog"
    species = "Canis familiaris"  # A class attribute (shared by all Dogs)

    def __init__(self, name, breed):  # The constructor (explained later)
        self.name = name  # An instance attribute to store the dog's name
        self.breed = breed  # An instance attribute to store the dog's breed

    def bark(self):  # A method (an action the dog can do)
        print(f"{self.name} says Woof!")

# Now, let's create some Dog objects:
my_dog = Dog("Vansh", "Golden Retriever")  # Creating an object called my_dog
another_dog = Dog("EII", "Labrador")  # Creating another object

# We can access their attributes:
print(my_dog.name)  # Output: Vansh
print(another_dog.breed)  # Output: Labrador

# And make them perform actions:
my_dog.bark()  # Output: Vansh says Woof!
print(Dog.species)  # Output: Canis familiaris
```

- **self Explained**: Inside a class, self is like saying "this particular object." It's 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. You don't type self when calling the method; Python inserts it for you. 🌺
- **Class vs. Instance Attributes**: 🌷
  - **Class Attributes**: These are shared by all objects of the class. Like species in our Dog class. All dogs belong to the same species. They are defined outside of any method, directly within the class. 🌸
  - **Instance Attributes**: These are specific to each individual object. name and breed are instance attributes. Each dog has its own name and breed. They are usually defined within the **init** method. 🌼

🌿 3. The Constructor: Setting Things Up (init) 🌿

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

What's it for? 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. 🌷

python

```python
class Dog:
    def __init__(self, name, breed):  # The constructor
        self.name = name      # Setting the name attribute
        self.breed = breed     # Setting the breed attribute

# When we do this:
my_dog = Dog("Vansh", "Poodle")  # The __init__ method is automatically called

# It's like we're saying:
# 1. Create a new Dog object.
# 2. Run the __init__ method on this new object:
#    - Set my_dog.name to "Vansh"
#    - Set my_dog.breed to "Poodle"
```

You can also set default values for parameters in the constructor, making them optional when creating an object: 🌸

python

```python
class Dog:
    def __init__(self, name="Unknown", breed="Mixed"):
        self.name = name
        self.breed = breed

dog1 = Dog()                  # name will be "Unknown", breed will be "Mixed"
dog2 = Dog("Vansh")           # name will be "Vansh", breed will be "Mixed"
dog3 = Dog("EII", "Labrador")  # name will be "EII", breed will be "Labrador"
```

🍀 4. Inheritance: Building Upon Existing Classes 🍀

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. 🌸

- **super()**: Inside a child class, super() lets you call methods from the parent class. This is useful when you want to extend the parent's behavior instead of completely replacing it. It's especially important when initializing the parent class's part of a child object. 🌼

python

```python
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 (Dog is a subclass of Animal)
    def speak(self):  # We *override* the speak method (more on this later)
        print("Woof!")

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

# Create objects:
my_dog = Dog("Vansh")
my_cat = Cat("EII")

# They both have a 'name' attribute (inherited from Animal):
print(my_dog.name)  # Output: Vansh
print(my_cat.name)  # Output: EII

# They both have a 'speak' method, but it behaves differently:
my_dog.speak()  # Output: Woof!
my_cat.speak()  # Output: Meow!
```

python

```python
# Calling Parent Constructor with super()
class Bird(Animal):
    def __init__(self, name, wingspan):
        super().__init__(name)  # Call Animal's __init__ to set the name
        self.wingspan = wingspan  # Add a Bird-specific attribute

my_bird = Bird("Tweety", 10)
print(my_bird.name)      # Output: Tweety (set by Animal's constructor)
print(my_bird.wingspan)  # Output: 10 (set by Bird's constructor)
```

🌿 5. Polymorphism: One Name, Many Forms 🌿

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. This allows you to write code that can work with objects of different types without needing to know their exact class. 🌺

🌷 6. Method Overriding: Customizing Inherited Behavior 🌷

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

🌼 7. Operator Overloading: Making Operators Work with Your Objects 🌼

Python lets you define how standard operators (like +, -, ==) behave when used with objects of your own classes. This is done using special methods called "magic methods" (or "dunder methods" because they have double underscores before and after the name). 🌿

python

```python
class Point:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):  # Overloading the + operator
        # 'other' refers to the object on the *right* side of the +
        return Point(self.x + other.x, self.y + other.y)

    def __str__(self):  # String representation (for print() and str())
        return f"({self.x}, {self.y})"

    def __eq__(self, other):  # Overloading == operator
        return self.x == other.x and self.y == other.y

p1 = Point(1, 2)
p2 = Point(3, 4)
p3 = p1 + p2  # This now works! It calls p1.__add__(p2)
print(p3)     # Output: (4, 6) (This uses the __str__ method)
print(p1 == p2)  # Output: False (This uses the __eq__ method)
```

Other useful magic methods: (You don't need to memorize them all, but be aware they exist!) 🌺
sub (-), **mul** (*), **truediv** (/), **eq** (==), **ne** (!=), **lt** (<), **gt** (>), **len** (len()), **getitem**, **setitem**, **delitem** (for list/dictionary-like behavior – allowing you to use [] with your objects).

🍀 8. Getters and Setters: Controlling Access to Attributes 🍀

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 (like my_object.attribute), you use methods to get and set its value. This might seem like extra work, but it provides significant advantages. 🌸

Why use them? 🌼

- **Validation**: You can add checks within the setter to make sure the attribute is set to a valid value. For example, you could prevent an age from being negative. 🌿
- **Read-Only Attributes**: You can create a getter without a setter, making the attribute effectively read-only from outside the class. This protects the attribute from being changed accidentally. 🌺
- **Side Effects**: You can perform other actions when an attribute is accessed or modified. For instance, you could update a display or log a change whenever a value is set. 🌷
- **Maintainability and Flexibility**: If you decide to change how an attribute is stored internally (maybe you switch from storing degrees Celsius to Fahrenheit), you only need to update the getter and setter methods. You don't need to change every other part of your code that uses the attribute. This makes your code much easier to maintain and modify in the future. 🌸

python

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age  # Convention: _age indicates it's intended to be "private"

    def get_age(self):  # Getter for age
        return self._age

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

person = Person("Vansh", 19)
print(person.get_age())  # Output: 19
person.set_age(19)
print(person.get_age())  # Output: 19
person.set_age(-5)       # Output: Invalid age!
print(person.get_age())  # Output: 19 (age wasn't changed)
```

The Pythonic Way: @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. 🌿

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 methods behind the scenes. This combines the convenience of direct access with the control and protection of encapsulation. 🌺

python

```python
class Person:
    def __init__(self, name, age):
        self.name = name
        self._age = age  # Convention: _age for "private" attributes

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

    @age.setter  # This defines the setter for the 'age' property
    def age(self, new_age):
        if new_age >= 0 and new_age <= 150:
            self._age = new_age
        else:
            print("Invalid age!")

person = Person("EII", 19)
print(person.age)  # Output: 19 (Looks like direct attribute access, but calls the getter)
person.age = 19    # (Calls the setter - looks like attribute assignment)
print(person.age)  # Output: 19
person.age = -22   # Output: Invalid age!
```

Private Variables (and the _ convention): 🌷

It's important to understand that Python does not have truly private attributes in the same way that languages like Java or C++ do. There's no keyword that completely prevents access to an attribute from outside the class. 🌸

Instead, Python uses a convention: An attribute name starting with a single underscore (_) signals to other programmers that this attribute is intended for internal use within the class. It's a strong suggestion: "Don't access this directly from outside the class; use the provided getters and setters instead." It's like a "Please Do Not Touch" sign. 🌼

While you can still access obj._internal_value directly, doing so is considered bad practice and can lead to problems if the internal implementation of the class changes. Always respect the underscore convention! It's about good coding style and collaboration. 🌿

python

```python
class MyClass:
    def __init__(self):
        self._internal_value = 0  # Convention: _ means "private"

    def get_value(self):
        return self._internal_value

obj = MyClass()
# print(obj._internal_value)  # This *works*, but it's against convention
print(obj.get_value())          # This is the preferred way
```

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. We'll also explore basic operating system (OS) interactions using Python's built-in modules. 🌺

🌷 File I/O in Python 🌷

File Input/Output (I/O) refers to reading data from and writing data to files. Python provides built-in functions to make this process straightforward. Working with files generally involves these steps: 🌸

1. **Opening a file**: You need to open a file before you can read from it or write to it. This creates a connection between your program and the file. 🌼
2. **Performing operations**: You can then read data from the file or write data to it. 🌿
3. **Closing the file**: It's crucial to close the file when you're finished with it. This releases the connection and ensures that any changes you've made are saved. 🌺

🍀 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. 🌿

Here are some examples: 🌺

Reading from a file: 🌷

python

```python
try:
    file = open("my_file.txt", "r")  # Open in read mode
    content = file.read()  # Read the entire file content
    print(content)
    file.close()  # Close the file
except FileNotFoundError:
    print("File not found.")

# Reading line by line
try:
    file = open("my_file.txt", "r")
    for line in file:  # Efficient for large files
        print(line.strip())  # Remove newline characters
    file.close()
except FileNotFoundError:
    print("File not found.")
```

Writing to a file: 🌸

python

```python
file = open("new_file.txt", "w")  # Open in write mode (creates or overwrites)
file.write("Hello, Vansh!\n")  # Write some text
file.write("This is a new line.\n")
file.close()
```

Appending to a file: 🌼

python

```python
file = open("my_file.txt", "a")  # Open in append mode
file.write("This is appended text by EII, age 19.\n")
file.close()
```

Using with statement (recommended): 🌿

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

python

```python
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 by Vansh, age 19.\n")
```