1. Variables and Variable Types

Concept: Variables store data values. Python does not require declaring the data type explicitly; it is inferred based on the value assigned.

Variable Types:

- `int`: Integer values
- `float`: Decimal numbers
- `str`: Sequence of characters (text)
- `bool`: Boolean values (`True` or `False`)

```
# Assigning a string to the variable 'name'
name = "Alice"

# Assigning an integer to the variable 'age'
age = 25

# Assigning a float to the variable 'height'
height = 5.4

# Assigning a boolean value to 'is_student'
is_student = True

# Display all variables
print(name)         # Output: Alice
print(age)          # Output: 25
print(height)       # Output: 5.4
print(is_student)   # Output: True

# Check variable types
type(age)
type(is_student)
```

```
Alice
25
5.4
True
bool
```

2. Getting User Input

Concept: Python uses the `input()` function to get input from the user. Input is always read as a string, so type conversion is necessary.

```
# Ask the user to enter their name
name = input("Enter your name: ")  # Takes input as a string

# Ask the user to enter their age
age = int(input("Enter your age: "))  # Converts the input string to integer

# Display a personalized message
print("Hello", name, "you are", age, "years old.")
```

```
Enter your name: Gaurav
Enter your age: 24
Hello Gaurav you are 24 years old.
```

3. Arithmetical Operations

Concept: Python supports all basic arithmetic operators: `+`, `-`, `*`, `/`, `//`, `%`, `**`

```
a = 11   # Assigning integer value 10 to variable a
b = 3    # Assigning integer value 3 to variable b

print(a + b)   # Addition
print(a - b)   # Subtraction
print(a * b)   # Multiplication
print(a / b)   # Division
print(a // b)  # Floor Division (Rounds the result down to the nearest whole number)
print(a % b)   # Modulus (remainder)
print(a ** b)  # Exponentiation:(11^3)
```

```
14
8
33
3.6666666666666665
3
2
1331
```

4. Compound Assignment Operators

Concept: These are shortcuts to perform arithmetic and assignment together.

```
x = 5           # Initial value of x is 5
x += 3          # x = x + 3 => x becomes 8
print(x)

x *= 2          # x = x * 2 => x becomes 16
print(x)

x -= 4          # x = x - 4 => x becomes 12
print(x)
```

```
8
16
12
```

5. Built-in Functions (abs, round, max)

Concept: Python provides useful built-in functions:

- `abs()`: Returns absolute value
- `round()`: Rounds a number
- `max()`: Returns the largest of arguments

```
num = -7.6
print(abs(num))     # Absolute value: 7.6

print(round(num))   # Rounded value: -8

print(max(5, 8, 2))  # Maximum value: 8
```

```
7.6
-8
8
```

6. Relational Operators and Boolean Type

Concept: Used to compare values. Return boolean results: `True` or `False`.

```
a = 15
b = 10

print(a == b)  # False: a is not equal to b
print(a != b)  # True: a is not equal to b
print(a > b)   # True: a is greater than b
print(a < b)   # False: a is not less than b
print(a >= b)  # True: a is greater than or equal to b
print(a <= b)  # False: a is not less than or equal to b
```

```
False
True
True
False
True
False
```

7. Modules in Python

Concept: Modules are pre-written Python code files. Use `import` to include them.

```
import math  # Importing the math module

print(math.sqrt(16))  # Prints square root of 16 => 4.0
print(math.pi)        # Prints value of Pi => 3.14159...
```

```
4.0
3.141592653589793
```

8. f-strings

Concept: Formatted string literals (f-strings) allow embedding expressions inside string literals using `{}`.

```
name = "Bob"
age = 30

# Use f-string to include variables in a sentence
print(f"My name is {name} and I am {age} years old.")
```

```
My name is Bob and I am 30 years old.
```

10. Data Types

A **data type** tells Python what kind of value a variable holds, and what operations can be performed on that value.

1. List

"List is a collection which is ordered and changeable. Allows duplicate members."

Explanation:

- **Ordered**: Items are stored in a specific order. You can access them using an index.
- **Changeable (Mutable)**: You can modify the list by adding, removing, or changing elements.
- **Allows Duplicates**: Same value can appear more than once.

```
# Creating a list of fruits
fruits = ["apple", "banana", "cherry"]

# Accessing elements
print(fruits[0])       # Prints 'apple'

# Appending an element
fruits.append("orange")
print(fruits)          # ['apple', 'banana', 'cherry', 'orange']

# Inserting at specific index
fruits.insert(1, "grape")
print(fruits)          # ['apple', 'grape', 'banana', 'cherry', 'orange']

# Removing an element
fruits.remove("banana")
print(fruits)          # ['apple', 'grape', 'cherry', 'orange']

# Popping the last element
last_item = fruits.pop()
print(last_item)       # 'orange'
print(fruits)          # ['apple', 'grape', 'cherry']

# Slicing the list
print(fruits[1:2])     # ['grape']

# Sorting the list
fruits.sort()
print(fruits)          # ['apple', 'cherry', 'grape']

# Reversing the list
fruits.reverse()
print(fruits)          # ['grape', 'cherry', 'apple']
```

2. Tuple

“Tuple is a collection which is ordered and unchangeable. Allows duplicate members.”

Explanation:

- **Ordered**: Items are in a fixed sequence and accessed by index.
- **Unchangeable (Immutable)**: You cannot change or update items once the tuple is created.
- **Allows Duplicates**: Same values can exist more than once.

```
# Creating a tuple of fruits
fruits = ("apple", "banana", "cherry")

# Accessing elements by index
print(fruits[0])        # Output: 'apple'

# Tuple is immutable - we cannot append, insert or remove directly
# But we can convert to a list, modify, and convert back if needed

# Converting tuple to list to add an element
fruits_list = list(fruits)
fruits_list.append("orange")
fruits = tuple(fruits_list)
print(fruits)           # ('apple', 'banana', 'cherry', 'orange')

# Getting the last item (no pop, but we can access by index)
last_item = fruits[-1]
print(last_item)        # 'orange'

# Length of the tuple
print(len(fruits))      # 4

# Checking existence
print("apple" in fruits)   # True

# Slicing the tuple
print(fruits[1:3])      # ('banana', 'cherry')

# Looping through the tuple
for fruit in fruits:
    print(fruit)        # Prints each fruit one by one
```

3. Set

“Set is a collection which is unordered, unchangeable*, and unindexed. No duplicate members.”

Explanation:

- **Unordered**: Items don’t have a fixed position.
- **Unchangeable***: You can't modify specific items, but you can add/remove whole items.
- **Unindexed**: You can’t access items by position.
- **No Duplicates**: All values must be unique.

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

print(my_set)       # Output: {1, 2, 3} (duplicate 2 is removed)

# Add a new item
my_set.add(4)
print(my_set)       # Output: {1, 2, 3, 4}

# Remove an item
my_set.remove(1)
print(my_set)       # Output: {2, 3, 4}

# Trying to access by index
#print(my_set[0])  ❌ Will cause an error
```

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

4. Dictionary

"Dictionary is a collection which is ordered and changeable. No duplicate members."**

Explanation:

- **Ordered (as of Python 3.7+)**: Items maintain the order in which they were added.
- **Changeable**: You can add, update, or remove key-value pairs.
- **No Duplicates**: Keys must be unique. If you try to use the same key again, the old value is overwritten.

```
# Creating a dictionary of fruits with quantity as value
fruits = {
    "apple": 10,
    "banana": 5,
    "cherry": 12
}
# Accessing elements (value by key)
print(fruits["apple"])  # Output: 10

# Adding a new key-value pair
fruits["orange"] = 7
print("After adding orange:", fruits)
# Output: {'apple': 10, 'banana': 5, 'cherry': 12, 'orange': 7}

# Updating the value of an existing key
fruits["banana"] = 8
print("After updating banana:", fruits)
# Output: {'apple': 10, 'banana': 8, 'cherry': 12, 'orange': 7}

# Removing a key-value pair using del
del fruits["banana"]
print("After deleting banana:", fruits)
# Output: {'apple': 10, 'cherry': 12, 'orange': 7}

# Popping a key-value pair and returning its value
removed_value = fruits.pop("orange")
print("Removed value:", removed_value)  # Output: 7
print("After popping orange:", fruits)
# Output: {'apple': 10, 'cherry': 12}

# Length of the dictionary (number of key-value pairs)
print("Length:", len(fruits))  # Output: 2
```

```python
# Checking for key existence
print("apple" in fruits)   # Output: True
print("banana" in fruits)  # Output: False

# Looping through dictionary keys
print("Looping through keys:")
for fruit in fruits:
    print(fruit)
# Output:
# apple
# cherry

# Looping through key-value pairs
print("Looping through key-value pairs:")
for fruit, quantity in fruits.items():
    print(f"{fruit}: {quantity}")
# Output:
# apple: 10
# cherry: 12

# Getting all keys
print("Keys:", fruits.keys())
# Output: dict_keys(['apple', 'cherry'])

# Getting all values
print("Values:", fruits.values())
# Output: dict_values([10, 12])

# Getting all items (key-value pairs)
print("Items:", fruits.items())
# Output: dict_items([('apple', 10), ('cherry', 12)])
```

```
True
```

`if-else` Statement in Python

Key Concepts:

- **`if`** checks a condition (expression that returns `True` or `False`)
- If `True`, the code under `if` block runs.
- If `False`, the code under `else` runs (if provided).

Note: **`if-elif-else`** is used for multiple conditions

Nested `if-else` in Python

Key Concepts:

- You can place an `if` or `else` inside another `if` or `else` block.
- Useful when a decision depends on another condition.

```
username = input ("Enter your Username:")
password = input ("Enter your Password:")

if username == "admin":
    if password == "1234":
        print("Login successful!")
    else:
        print("Incorrect password.")
else:
    print("Unknown user.")
```

```
Enter your Username:admin
Enter your Password:1234
Login successful!
```

`while` Loop in Python

Used to **repeat a block of code** as long as a given condition is `True`.

```
i = 1
while i <= 5:
    print(i)
    i += 1
```

```
1
2
3
4
5
```

`break` Statement

Used to **exit a loop** (both `while` and `for`) immediately when a certain condition is met.

```
i = 1
while i <= 5:
    if i == 3:
        break  # loop exits here
    print(i)
    i += 1
```

```
1
2
```

`for` Loop in Python

Used to **iterate over a sequence** (like a list, string, tuple, or range).

```
for i in range(1, 6):
    print(i)
```

```
1
2
3
4
5
```

`continue` Statement

Used to **skip the current iteration** and continue with the next one.

```
for i in range(1, 6):
    if i == 3:
        continue  # skips rest of the code for i==3
    print(i)
```

```
1
2
4
5
```

Combined Example: `while` loop with `break` and `continue`

```
attempts = 0
while attempts < 3:
    username = input("Enter username: ")

    if username == "":
        print("Empty input, try again.")
        continue  # skip to next attempt

    if username == "admin":
        print("Welcome admin!")
        break  # exit loop on success
    else:
        print("Incorrect username.")

    attempts += 1
```

```
Enter username:
Empty input, try again.
Enter username:
Empty input, try again.
Enter username:
Empty input, try again.
Enter username: admin
Welcome admin!
```

Nested Loops in Python

A **nested loop** is a loop **inside another loop**. This is useful when dealing with multi-dimensional data like **matrices**, **tables**, or patterns.

```
# Define the number of rows we want in our pattern
rows = input("Enter the number of rows: ")

# Convert the input string to an integer
rows = int(rows)

# Outer loop: Iterates over each row (from 1 to rows inclusive)
for i in range(1, rows + 1):

    # Inner loop: Iterates from 1 to i (inclusive), so the number of elements increases each row
    for j in range(1, i + 1):

        # Print the current value of j with a space (staying on the same line using end=" ")
        print(j, end=" ")

    # After inner loop ends, move to the next line (start a new row)
    print()
```

```
Enter the number of rows: 7
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
1 2 3 4 5 6
1 2 3 4 5 6 7
```

Program form the slide

```
# Initialize an empty list to store prime numbers
primes = []

# Loop through numbers from 2 to 99 (inclusive of 2, exclusive of 100)
for num in range(2, 100):

    # Assume the current number is prime (we'll check this below)
    possible_prime = True

    # Check for factors from 2 to num-1
    for j in range(2, num):

        # If num is divisible by any j, it's not prime
        if num % j == 0:
            possible_prime = False  # Mark as not prime
            break  # No need to check further, break the inner loop

    # If no factor was found, then num is a prime number
    if possible_prime == True:
        primes.append(num)  # Add it to the list of primes

# After all numbers are checked, print the list of prime numbers
print(primes)
```

```
[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47, 53, 59, 61, 67, 71, 73, 79, 83, 89, 97]
```

Functions in Python

A function is a **block of reusable code** that performs a specific task. It helps in **modularizing** code and **avoiding repetition**.

```python
# Define a function to add two numbers
def add_numbers(a, b):
    result = a + b
    return result

# Call the function and print the result
sum_value = add_numbers(5, 3)
print("The sum is:", sum_value)
```

```
The sum is: 8
```

- `def` → keyword to define a function
- `add_numbers` → name of the function (you choose)
- `parameters` → inputs the function accepts (a & b in this case)
- `return` → (optional) sends back a result

Functions and Variable Scope

Scope defines **where a variable is accessible**.

Types:

- **Local Scope** → Inside a function
- **Global Scope** → Outside all functions

```
# Global variable
message = "Global Message"

def print_message():
    # Local variable (only accessible inside this function)
    message = "Local Message"
    print("Inside function:", message)

print_message()
print("Outside function:", message)
```

```
Inside function: Local Message
Outside function: Global Message
```

Function to Find the Maximum in a List

```
def find_max(numbers):
    # Assume the first number is the max initially
    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num
    return max_num

nums = [23, 45, 67, 89, 12]
a = find_max(nums)
print("Maximum number is:", a)
```

```
Maximum number is: 89
```