UNIT 1

Introduction to Python

UNIT-1: Introduction to Python: Python variables, Python basic Operators, Understanding python blocks. Python Data Types, Declaring and using Numeric data types: int, float etc.

1.1 INTRODUCTION TO PYTHON

- Python is a **high-level, easy-to-learn, and versatile programming language**.
- It is widely used in **web development, software development, data science, AI/ML, automation, and scientific research**.
- It's **simple, English-like syntax** makes it popular among both beginners and professionals.
- **Developed by Guido van Rossum**, it was first released in **1991**.
- Python focuses on **readability and simplicity**.

1.2 PYTHON KEY FEATURES

1. **Easy to learn** : *Code looks like plain English, making it simple to understand and write.*
2. **Line-by-Line Execution** : *Python executes code one line at a time, making debugging and testing easier.*
3. **Large Standard Library** : *Python comes with many built-in modules and functions for tasks like file handling, math, networking, and more.*
4. **Portable** : *Python runs on multiple platforms (Windows, Linux, macOS).*
5. **Flexible Coding Styles** : *Supports **procedural, object-oriented, and functional programming** styles.*

Key Points:

1. **Free and open-source** : *Anyone can use, modify, and distribute Python.*
2. **Cross-platform** : *Works on Windows, Mac, Linux, and other operating systems.*
3. **Interpreted** : *Executes code line by line, which makes debugging easy.*

4. **Multi-paradigm** : *Supports multiple styles (procedural, object-oriented, functional).*

Simple example Program

```
print("Hello, JSS ECE STUDENTS!")
```

OUTPUT:

```
Hello, JSS ECE STUDENTS!
```

Explanation

- **print()** is a **built-in Python function** used to display information on the screen.
- Anything inside **quotes (" ")** is treated as a **string** (text).
- In this program, Python prints the exact message: **Hello, JSS ECE STUDENTS!**
- Python programs can be run directly, no extra setup is needed for simple code like this.

1.3 PYTHON APPLICATIONS

Python is **versatile** and used in many fields:

1. **Web Development** : *Websites, web apps (e.g., Django, Flask).*
2. **Data Science** : *Analyze large datasets and create charts (e.g., Pandas, Matplotlib).*
3. **AI & ML** : *Build models, make predictions, and create automation systems (e.g., TensorFlow, scikit-learn).*
4. **Automation** : *Automate repetitive tasks like file handling, data scraping, and testing.*
5. **Scientific Computing** : *Solve complex mathematical problems and simulations.*
6. **Scripting** : *Write small programs to perform quick tasks or control other software.*

Example Program: Adding Two Numbers

```
x = 3
y = 5
print("The sum is:", x + y)
```

OUTPUT:

```
The sum is: 8
```

Explanation:

- x and y are variables storing integers.
- The print() function displays their sum.
- Python supports multiple data types like numbers, strings, lists, etc.

COMMENT LINES IN PYTHON

A comment in Python is a line (or part of a line) in the code that is completely ignored by the Python during execution. It's used to explain what the code does, making it easier to understand for others (or yourself later!).

Types of Comments

Type	Symbol	Description
Single-line (Inline Comment)	**#**	Starts with #, used for brief notes
Multi-line	**''' or """**	Used for longer explanations or documentation

Example: Single-line Comment or Inline Comment

```
# This program adds two numbers
a = 5
b = 3
print(a + b)  # Output will be 8
```

Explanation:

- The first line explains the program.
- The comment after print() describes the output.

Example: Multi-line Comment

```
'''
This program demonstrates
how to use multi-line comments
in Python.
'''
print("Hello, Python!")
```

Explanation:

- The triple quotes enclose a block of text.
- Useful for longer descriptions or documentation.

Example: Inline Comment

```
x = 10          # Initial value
x = x + 5       # Add 5 to x
print(x)      # Should print 15
```

Explanation:

- Comments are placed beside code lines to explain each step.

1.4 PYTHON SYNTAX COMPARED TO OTHER PROGRAMMING LANGUAGES

Python is designed for **readability** and **simplicity**:

No Semicolons : *Most programming languages use a semicolon (;) to end a statement.*

Indentation for Blocks : *Other languages use curly brackets {} to group code. Python uses **indentation** (spaces or tabs) to define loops, functions, and classes.*

Dynamic Typing : *You don't need to declare variable types. Python figures it out automatically.*

TABLE: PYTHON SYNTAX VS OTHER LANGUAGES

Feature	Python	C/Java/C++
End of Statement	New line	Semicolon (;)
Code Blocks	Indentation (spaces/tabs)	Curly brackets {}
Variable Declaration	Dynamic (no declaration)	Static (must declare)
Readability	High (like English)	Lower (more symbols)

Example Program: Printing Numbers in a Loop

```
for i in range (5):
    print(i)
```

OUTPUT:

0
1
2
3
4

Explanation:

- The for loop runs 5 times (0 to 4). Each time, it prints the value of i. Notice the **indentation** this is how Python knows which lines belong to the loop.

SUMMARY

Topic	Key Point	Example
What is Python?	Easy, general-purpose language	Print ("Hello, World!")
What can Python do?	Web, data, AI, Automation, Scripting	a = 5; b = 7; print(a+b)
Syntax	No semicolons; indentation for blocks; dynamic typing	for i in range (5): print(i)

1.5 PYTHON VARIABLES

- A variable is a name that refers to a value in memory.
- You don't declare a type; Python infers it automatically.
- Variables can change values during execution.
- Assignment is done using =.

Example Program: Printing Numbers in a Loop

```
name = "Rahul"    # string
age = 35          # integer
pi = 3.1416       # float
print("Name:", name)
print("Age:", age)
print("PI value:", pi)
```

OUTPUT:

Name: Rahul
Age: 35
PI value: 3.1416

Explanation

- name, age, and pi are **variables**.
- They store different types of values:
  - ✓ "Rahul" → string (text)
  - ✓ 35 → integer (whole number)
  - ✓ 3.1416 → float (decimal number)
- print() is used to show the values on the screen.

RULES FOR NAMING VARIABLES

Allowed

- Must start with a **letter (a–z, A–Z)** or an **underscore _**.
- Can contain **letters, digits, and underscores**.
- Are **case-sensitive** (Value ≠ value).

Not Allowed

- Cannot start with a **number**.
- Cannot contain **spaces or special characters** (@, $, %, -, !, etc.).
- Cannot use **Python keywords** (if, for, True, class, etc.).

Examples for Valid and Invalid variable names

Valid Variable Names

- student_name → uses letters + underscore
- marks_obtained → descriptive and clear
- _temp → allowed (can start with underscore)
- roll_number10 → numbers are allowed, but **not at the beginning**

Invalid Variable Names

- 1student → cannot start with a number
- my-name → hyphen (-) not allowed, only underscore (_) can be used
- for → reserved keyword in Python

1.6 VARIABLE TYPES IN PYTHON

- Python is a **dynamically typed language**, which means you don't need to declare a variable's type.
- The type is decided automatically when you assign a value.

Some common types are

Type	Example	Example Value
int	num = 42	42 (whole number)
float	weight = 65.3	65.3 (decimal)
str	country = "India"	"India" (text)
bool	is_active = False	True/False (logical)

Example Program: How to Create Variables

```
# String variable
student_name = "Rahul"        # string
name = "Virat"                # string

# Integer variable
marks = 87                    # integer
x = 10                        # integer
age = 24                      # integer

# Float variable
_pi = 3.14                    # float
price = 8.99                  # float
height = 1.75                 # float

# Another string
person = "Bharat"
language = 'Python'

# Boolean variable
is_valid = True

# Multiple Assignment
a, b, c = 1, 2, 3

print(student_name, marks, _pi, x, name, price, age, height, person, language, is_valid, a, b, c)
```

OUTPUT:

```
Rahul 87 3.14 10 Virat 8.99 24 1.75 Bharat Python True 1 2 3
```

Explanation

- Python automatically assigns the type (int, float, str, bool) based on the value.
- Variables can store different types of data without prior declaration.
- **Multiple assignment** (a, b, c = 1, 2, 3) assigns values in order:
  - ✓ a = 1, b = 2, c = 3.

Example Program: Changing Variable Values

```
score = 5
score = score + 10        # score is now 15
print("Updated Score:", score)   # Output: Updated Score: 15
```

OUTPUT:

```
Updated Score: 15
```

Explanation:

- Initially, score = 5.
- After update: score = 5 + 10 = 15.
- The new value (15) replaces the old value.

Example Program: Storing and Printing Values

```
# Store values in variables
city = "NOIDA"
temperature = 44

print("City:", city)
print("Temperature:", temperature, "°C")
```

OUTPUT:

```
City: NOIDA
Temperature: 44 °C
```

Explanation:

- city stores "NOIDA".
- temperature stores 44.
- The print() function displays both values along with descriptive text.

Example Program: Changing Variable Values

```
# Assign initial value
score = 70

# Print the original score
print("Original Score:", score)

# Change the value of score
score = 80

#Print the updated score
print("Updated Score:", score)
```

OUTPUT:

```
Original Score: 70
Updated Score: 80
```

EXPLANATION:

- At first, score = 70.
- The value is reassigned to 80.

- Printing shows the updated value.

Example Program: Swap Two Variables

```
a = 10
b = 22
a, b = b, a                # Swap values
print("a:", a)
print("b:", b)
```

OUTPUT:

```
a: 22
b: 10
```

Explanation:

- Initially, a = 10, b = 22.
- After swapping, a = 22, b = 10.
- Python allows swapping in a single line.

Program 4: Calculate Area of a Circle

```
radius = 5
pi = 3.14159
area = pi * radius * radius
print("Area of the circle:", area)
```

OUTPUT:

```
Area of the circle: 78.53975
```

Explanation:

- Formula for area = $\pi \times r^2$.
- Here, $3.14159 \times 5 \times 5 = 78.53975$.
- Printed result shows the calculated area.

Example Program: User Input and Output

```
name = input("Enter your name: ")
age = input("Enter your age: ")
print("Hello", name, "you are", age, "years old.")
```

OUTPUT:

```
Enter your name: Rahul
Enter your age: 35
Hello Rahul you are 35 years old.
```

Explanation:

- input() takes values from the user (always as **string**).
- "Arun" is stored in name.
- "25" is stored in age.
- print() combines and displays them.

Write Python program to swap two numbers without using Intermediate/Temporary variables. Prompt the user for input.

Solution:

```
# Program to swap two numbers without using a temporary variable

num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))

print("Before swapping: num1 =", num1, ", num2 =", num2)

# Swapping using tuple unpacking
num1, num2 = num2, num1

print("After swapping: num1 =", num1, ", num2 =", num2)
```

Output:

```
Enter first number: 10
Enter second number: 25
Before swapping: num1 = 10 , num2 = 25
After swapping: num1 = 25 , num2 = 10
```

Explanation:

1. **User Input**
   - num1 = 10, num2 = 25 (entered by user).
2. **Before Swapping**
   - Prints: num1 = 10 , num2 = 25.
3. **Swapping Logic**
   - Statement num1, num2 = num2, num1 creates a **tuple** (num2, num1) on the right-hand side → (25, 10).
   - Python **unpacks** this tuple into the left-hand side → num1 = 25, num2 = 10.
4. **After Swapping**
   - Prints: num1 = 25 , num2 = 10.

1.6.1 LOCAL AND GLOBAL VARIABLES IN PYTHON

Local Variables

- Declared **inside a function**.
- Exist only while the function runs.
- Not accessible outside the function.

Example:

```
def greet():
    message = "Hello!"  # local variable
    print(message)

greet()
# print(message)
# This would cause an error: message is not defined outside the function
```

OUTPUT:

```
Hello!
```

Global Variables

- Declared **outside all functions**.
- Accessible anywhere in the program.
- Can be used inside functions unless shadowed by a local variable.

Example:

```
name = "Rahul"  # global variable

def greet():
   print("Hello,", name)

greet()
print("Welcome,", name)
```

OUTPUT:

```
Hello, Rahul
Welcome, Rahul
```

Note:

- **Local variable** → exists only inside the function.
- **Global variable** → can be used anywhere in the program.

1.6.2 DIFFERENCE BETWEEN LOCAL AND GLOBAL VARIABLES

Table: Difference between local and global variables

Feature	Local Variable	Global Variable
Scope	Inside the function only	Entire program
Declaration location	Inside a function	Outside all functions
Lifetime	Exists only while the function runs	Exists as long as the program runs
Accessibility	Not accessible outside its function	Accessible both inside and outside functions
Modification inside a function	Cannot modify a global variable unless you use the global keyword	No keyword needed (already global)

1.7 PYTHON BASIC OPERATORS

Python provides a rich set of operators to perform operations on variables and data. Operators are special symbols or keywords that tell the interpreter to carry out specific operations such as arithmetic, assignment, comparison, logical, and more.

1. Arithmetic Operators

Arithmetic operators are used for mathematical operations.

Operator	Description	Example (a=10, b=3)	Result
+	Addition	a + b	13
-	Subtraction	a - b	7
*	Multiplication	a * b	30
/	Division	a / b	3.333...
//	Floor or Integer Division	a // b	3
%	Modulus (Remainder)	a % b	1
**	Exponentiation	a ** b	1000

Example Program: Basic Arithmetic Operations

```
a = 10
b = 3
print("Addition:", a + b)
print("Subtraction:", a - b)
print("Multiplication:", a * b)
print("Division:", a / b)
print("Floor Division:", a // b)
print("Modulus:", a % b)
print("Exponentiation:", a ** b)
```

Output:

```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Exponentiation: 1000
```

2. Assignment Operators

Basic Assignment Operator

Assignment operators are used to assign values to variables and modify them.

Operator	Description	Example
=	Assigns value	x = 10

Augmented Assignment Operators

These combine an operation with assignment

Operator	Description	Example	Equivalent to
+=	Add and assign	x += 3	x = x + 3
-=	Subtract and assign	x -= 2	x = x - 2
*=	Multiply and assign	x *= 5	x = x * 5
/=	Divide and assign	x /= 2	x = x / 2
//=	Floor or Integer divide assign	x //= 3	x = x // 3
%=	Modulus and assign	x %= 4	x = x % 4
**=	Exponentiate assign	x **= 2	x = x ** 2

Example Program: Augmented Assignment Operators

```
x = 5
x += 3
print("After += :", x)   # Output: 8

x *= 2
print("After *= :", x)   # Output: 16

x //= 3
print("After //= :", x)  # Output: 5
```

Explanation:

1. Initial assignment
   x = 5 sets x to 5.
2. Addition assignment (+=)
   x += 3 is shorthand for x = x + 3.
   New value: 5 + 3 = 8.
3. Multiplication assignment (*=)
   x *= 2 is shorthand for x = x * 2.
   New value: 8 * 2 = 16.
4. Floor-division assignment (//=)
   x //= 3 is shorthand for x = x // 3.
   Floor division discards the remainder: 16 // 3 = 5.

Each augmented assignment updates x in place and then prints the current value.

3. Comparison Operators

These compare two values and return a Boolean (True or False).

Operator	Description	Example (a = 5, b = 3)	Result
==	Equal to	a == b	False
!=	Not equal to	a != b	True
>	Greater than	a > b	True
<	Less than	a < b	False
>=	Greater than or equal to	a >= b	True
<=	Less than or equal to	a <= b	False

Example Program: Comparison Between Two Numbers

```
a = 5
b = 3
print(a == b)    # False
print(a != b)    # True
print(a > b)     # True
print(a <= b)  # False
```

4. Logical Operators

Logical operators are used to combine conditional statements.

Operator	Description	Example (a=4 & b=2)	Result
and	Logical AND	(a > 2 and b < 5)	True
or	Logical OR	(a > 10 or b < 5)	True
not	Logical NOT	not(a == b)	True

Example Program: Logical Operators in Action

```
a = 4
b = 2
print(a > 2 and b > 1)    # True
print(a < 2 or b < 1)     # False
print(not(a == b))      # True
```

5. Identity Operators

They check **if two variables point to the same memory location (object)**.

Operator	Description	Example	Result
is	Same object in memory	x is y	True/False
is not	Different objects	x is not y	True/False

Example Program: Lists (Different Objects, Same Values)

```
# Assign values
x = [1, 2, 3]
y = x
z = [1, 2, 3]
# Check identity
print(x is y)          # True  (y refers to the same object as x)
print(x == z)          # True  (same values)
print(x is z)          # False (z has the same values, but is a different object)
print(x is not z)      # True  (x and z are not the same object)
```

OUTPUT:

```
True
True
False
True
```

Explanation:

- **x and y →** point to the same object in memory, so x is y **→ True.**
- **x and z →** contain the same values but are different objects, **so x is z → False.**
- Hence**, x is not z → True.**
- == → compares **values**
- is → compares **memory reference (identity)**

Example Program: Integers (Caching in Python)

```
a = 100
b = 100
c = 500
d = 500

print(a is b)   # Always True (small integers are cached)
print(c is d)   # May be True or False (large integers not guaranteed to be cached)
```

OUTPUT:

```
True
False   # but could be True in some environments
```

Explanation:

- a is b → **True**, because integers in the range −5 to 256 are cached and reused.
- c is d → **Result may vary**. For integers outside the small cache range, Python **may or may not** create a new object.

Note:

- *== checks value equality.*
- *is checks memory identity.*

Example Program: Strings (Immutable and Cached Sometimes)

```
s1 = "hello"
s2 = "hello"
s3 = "".join(["he", "llo"])      # creates a new string object

print(s1 == s2)    # True  (same value)
print(s1 is s2)    # True  (Python reuses same object for small strings)
print(s1 is s3)    # False (s3 is a different object even though value is same)
```

OUTPUT:

```
True
True
False
```

Explanation:

1. s1 = "hello" and s2 = "hello"

- In Python, strings are immutable (cannot be changed after creation).
- For efficiency, Python caches small strings (called string interning).
- So both s1 and s2 point to the same memory location.
- **That's why:**
  - ✓ **s1 == s2 → True (values are same).**
  - ✓ **s1 is s2 → True (same object in memory).**

2. s3 = "".join(["he", "llo"])

- Here we build the string dynamically using "".join().
- Even though the value of s3 is "hello", Python creates a new object in memory.
- **That's why:**
  - ✓ **s1 == s3 → would be True (same value).**
  - ✓ **s1 is s3 → False (different objects in memory).**

6. Membership Operators

Test whether a sequence contains a value.

Operator	Description	Example	Result
in	Value found	20 in [10, 20, 30]	True
not in	Value not found	15 not in [10, 20, 30]	True

Example with List

```
nums = [10, 20, 30]
print(20 in nums)
# True → because 20 is inside the list

print(15 not in nums)
# True → because 15 is not inside the list
```

OUTPUT:

```
True
True
```

Example with String

```
word = "apple"
print("a" in word)
 # True → "a" is in "apple"

print("z" not in word)
# True → "z" is not in "apple"
```

OUTPUT:

```
True
True
```

Explanation

- The **in operator** checks if a value exists inside a sequence (list, string, tuple, etc.).
- The **not in operator** checks if a value does not exist in the sequence.

7. Bitwise Operators

Bitwise operators operate on the binary bits of numbers and perform operations bit by bit at a time.

Operator	Description	Example	Result
&	Bitwise AND	5 & 3	1
|	Bitwise OR	5 | 3	7
^	Bitwise XOR	5 ^ 3	6
~	Bitwise NOT	~5	-6
<<	Bitwise left shift	5 << 1	10
>>	Bitwise right shift	5 >> 1	2

Example Program:

```
a = 5      # Binary: 0101
b = 3      # Binary: 0011

print(a & b)          # 1
print(a | b)          # 7
print(a ^ b)          # 6
print(~a)             # -6
print(a << 1)         # 10
print(b >> 1)         # 1
```

Explanation: NOT (~) Operation

1. **5 = 00000101 B → Apply NOT (~) → flip every bit: 11111010**
2. Now, this binary value (11111010) represents a **negative number in Python** (and in most computers) because of two's complement representation.

3. **Two's Complement Rule:**
   Invert all bits again: 11111010 → 00000101
   Add 1 → 00000101 + 1 = 00000110 (6 in decimal)
   Put a negative sign → -6
   So ~5 = **-6**

PROCEDURE TO PERFORM LEFT & RIGHT SHIFT

Bitwise Right Shift Operation (>>):

***Right Shift Rule*: $a >> b = \text{floor}\left(\frac{a}{2^b}\right)$**	
Example 1: 6 >> 10 Let a = 6, b = 10 $\frac{a}{2^b} = \frac{6}{2^{10}} = \frac{6}{1024} = 0.005859375$ **Take the floor (integer part)** $\text{floor}(0.005859375) = 0$ $6 \gg 10 \rightarrow 0$ Note: Shifting a small number right by a large number of bits reduces it to zero.	**Example 1: 3 >> 4** Let a = 3, b = 4 $\frac{a}{2^b} = \frac{3}{2^4} = \frac{3}{16} = 0.1875$ **Take the floor (integer part)** $\text{floor}(0.1875) = 0$ $3 >> 4 = 0$ Note: Shifting a small number right by a large number of bits reduces it to zero.

Bitwise Left Shift Operation (<<):

***Left Shift Rule*: $a << b = a \times 2^b$**	
Example 1: 5 << 3 Let a = 5, b = 3 $a << b = \boldsymbol{a \times 2^b} = 5 \times 2^3 = \mathbf{40}$	**Example 2: 6 << 4** Let a = 6, b = 4 $a << b = \boldsymbol{a \times 2^b} = 6 \times 2^4 = \mathbf{96}$

Table: Summary of all the operators

Operator Type	Operators	Explanation	Example
Arithmetic	+ , - , * , / , // , % , **	Perform mathematical calculations (addition, subtraction, etc.)	10 + 3 → 13 10 // 3 → 3
Basic assignment *operator*	=	Assign values to variables	x = 10
Augmented Assignment	+= , -= , *= , /= , //= , %= , **=	Combine operation with assignment	x = 5 x += 3 **# 8** x *= 2 **# 16**
Comparison	== , != , > , < , >= , <=	Compare values and return Boolean (True/False)	5 > 3 → True 5 == 3 → False
Logical	and, or, not	Combine conditional expressions	not (5 > 3) → False
Identity	is, is not	*Checks if two variables refer to the same object (memory location)*.	x is y, x is not z
Membership	in, not in	Check whether a value exists inside a sequence (list, string, tuple, etc.)	"a" in "apple" → True 15 not in [10,20,30] → True
Bitwise	& , | , ^ , ~ , << , >>	Operate at the binary level (bit-by-bit operations)	5 & 3 → 1 5 ^ 3 → 6 ~5 → -6 5 << 1 → 10 3 >> 1 → 1

Design a basic calculator in Python that supports addition, subtraction, multiplication, division.

Program: Method-1 (Without Using Function)

```
# Very Simple Calculator

num1 = float(input("Enter first number: "))
op = input("Enter operator (+, -, *, /): ")
num2 = float(input("Enter second number: "))

if op == '+':
   print("Result =", num1 + num2)
elif op == '-':
   print("Result =", num1 - num2)
elif op == '*':
   print("Result =", num1 * num2)
elif op == '/':
```

```
    if num2 != 0:
        print("Result =", num1 / num2)
    else:
        print("Error: Cannot divide by zero")
else:
    print("Invalid operator")
```

OUTPUT: Case 1: Multiplication	**OUTPUT: Case 2: Division by zero**
Enter first number: 6 *Enter operator (+, -, *, /): ** *Enter second number: 4* *Result = 24.0*	*Enter first number: 8* *Enter operator (+, -, *, /): /* *Enter second number: 0* *Error: Cannot divide by zero*

Program: Method-2 (Using Function)

```
# Simple Calculator Program

def calculator():
    print("Basic Calculator")
    print("Operations: +  -  *  /")

    # Taking input from user
    num1 = float(input("Enter first number: "))
    operator = input("Enter operator (+, -, *, /): ")
    num2 = float(input("Enter second number: "))

    # Performing operation
    if operator == '+':
        print("Result:", num1 + num2)
    elif operator == '-':
        print("Result:", num1 - num2)
    elif operator == '*':
        print("Result:", num1 * num2)
    elif operator == '/':
        if num2 != 0:   # to avoid division by zero error
            print("Result:", num1 / num2)
        else:
            print("Error: Division by zero is not allowed.")
    else:
        print("Invalid operator")

# Call the calculator function

calculator()
```

OUTPUT:

```
Basic Calculator
Operations: +  -  *  /
Enter first number: 10
Enter operator (+, -, *, /): *
Enter second number: 5
Result: 50.0
```

1.7.1 OPERATOR PRECEDENCE (PRIORITY)

- Operator precedence means the **priority order** in which Python evaluates different operators in an expression.
- When an expression has multiple operators, Python follows precedence rules to decide **which operation to perform first**.

Example: Operator Precedence (Priority)

```
x = 10 + 2 * 3
print(x)
```

OUTPUT:

16

Explanation:

1. According to Python's precedence rules, **multiplication (*) is evaluated before addition (+)**.
2. So, 2 * 3 = 6 is calculated first.
3. Then, 10 + 6 = 16 is evaluated.

Thus, the final value of x is 16.

Table: Operator Precedence

Precedence Level	Operator(s)	Description	Associativity
1 (Highest)	() , [], {}	Parentheses (grouping), list, tuple, dict, set display	Left → Right
2	f(args...), x[index], x.attr	Function call, subscription, attribute reference	Left → Right
3	**	Exponentiation (power)	Right → Left
4	+x, -x, ~x	Unary plus, Unary minus, Bitwise NOT	Right → Left
5	*, /, //, %	Multiplication, Division, Floor or Integer division, Modulus	Left → Right
6	+ , -	Addition, Subtraction	Left → Right
7	<< , >>	Bitwise shift operators	Left → Right
8	&	Bitwise AND	Left → Right
9	^	Bitwise XOR	Left → Right
10	|	Bitwise OR	Bitwise OR
11	< , <= , > , >= , == , !=	Comparison operators	Left → Right
12	not	Logical NOT	Right → Left
13	and	Logical AND	Left → Right
14	or	Logical OR	Left → Right
15 (Lowest)	=, +=, -=, *=, /=, //=, %=, **=, &=, ^=, |=, <<=, >>=	Assignment operators	Right → Left

1.7.2 ASSOCIATIVITY IN PYTHON

- Associativity defines the **direction of evaluation** (left → right or right → left) when two or more operators of the **same precedence** appear in an expression.
- It comes into play **only when two or more operators with equal precedence** are used in a single expression.

Types of Associativity

1. ***Left-to-Right Associativity***
2. ***Right-to-Left Associativity***

1. Left-to-Right Associativity

- Most operators in Python **(arithmetic, comparison, bitwise, logical and/or)** follow this rule.
- Python evaluates them from **left to right.**

Example:

```
x = 10 - 5 + 2
# (10 - 5) + 2 = 5 + 2 = 7
print(x)   # Output: 7

y = 20 / 5 * 2
# (20 / 5) * 2 = 4 * 2 = 8.0
print(y)   # Output: 8.0
```

2. Right-to-Left Associativity

- Some Python operators like ** (exponentiation), = (assignment), and not (logical NOT) follow this rule.
- Python evaluates them **from right to left**.

Example: *Exponentiation (**)*

```
x = 2 ** 3 ** 2      # Python does: 2 ** (3 ** 2) = 2 ** 9 = 512
print(x)
```

OUTPUT:

512

Example 2: Assignment (=)

```
a = b = 10   # Python does: b = 10, then a = b
print(a, b)
```

OUTPUT:

10 10

Table: Summary of Associativity Type

Associativity Type	Operators	Evaluation Direction
Left-to-right	+, -, *, /, %, ==, <, >	Left to right
Right-to-left	**, =, not	Right to left

Find and explain stepwise solution of following expressions if a=3, b=5, c=10.

i. **a &b<<2//5**2+c^b**
ii. **b>>a**2<<2>>b**2^c**3**

Solution:

i. a & b << 2 // 52 + c ^ b**

(Given: a = 3, b = 5, c = 10)

Step	Operation	Expression
Step 0	***Given: a = 3, b = 5, c = 10***	**a & b << 2 // 5**2 + c ^ b**
Step 1	Substitute values	3 & 5 << 2 // 5**2 + 10 ^ 5
Step 2	Exponentiation (5**2 = 25)	3 & 5 << 2 // 25 + 10 ^ 5

Step 3	Floor division (2 // 25 = 0)	3 & 5 << 0 + 10 ^ 5
Step 4	Group addition with higher precedence	3 & 5 << (0 + 10) ^ 5
Step 5	Addition (0 + 10 = 10)	3 & 5 << 10 ^ 5
Step 6	Left shift (5 << 10 = 5120)	3 & 5120 ^ 5
Step 7	Bitwise AND (3 & 5120 = 0)	0 ^ 5
	Final Answer	**5**

ii. b >> a ** 2 << 2 >> b ** 2 ^ c ** 3

(Given: a = 3, b = 5, c = 10)

Step	Operation	Expression
Step 0	**Given: a = 3, b = 5, c = 10**	**b >> a ** 2 << 2 >> b ** 2 ^ c ** 3**
Step 1	Substitute values	5 >> 3 ** 2 << 2 >> 5 ** 2 ^ 10 ** 3
Step 2	Exponentiation (3**2 = 9, 5**2 = 25, 10**3 = 1000)	5 >> 9 << 2 >> 25 ^ 1000
Step 3	Right shift (5 >> 9 = 0)	0 << 2 >> 25 ^ 1000
Step 4	Left shift (0 << 2 = 0)	0 >> 25 ^ 1000
Step 5	Right shift (0 >> 25 = 0)	0 ^ 1000
Step 6	Bitwise XOR (0 ^ 1000 = 1000)	**1000**
	Final Answer	**1000**

1.7.3 SHORT CIRCUIT EVALUATION

- **Short circuit evaluation,** also known as **lazy evaluation or minimal evaluation** is a strategy used by Python to **evaluate logical expressions** involving the **and** and **or operators**.
- Instead of **evaluating all** parts of an **expression**, Python **stops as soon** as the **result is determined**, **skipping unnecessary computations**.
- It applies to the **logical operators**: ***and*** and ***or***.
- **Evaluated left to right.**

Rules for or

- Returns the **first truthy** value.
- If the first operand is truthy → Python does **not check the rest**.
- If **all falsy, returns the last one.**

Example Program: Short circuit evaluation for ***or***

```
a = 0
b = 2
c = 3
x = c or a
print(x)
```

OUTPUT:

```
3
```

Explanation:

- **c = 3** → This is a truthy value (non-zero).
- **a = 0** → falsy, but **not even checked** (short-circuit).
- **x = c or a** → Since **c** is **truthy,** Python does not evaluate **a. So, x = 3**

Truth Table for or (Stops at First Truthy)

Expression	First Operand	Second Operand	What Happens	Result
0 or 5	0 → falsy	5 → truthy	Python checks 2nd	5
3 or 0	3 → truthy	(not checked)	Stops early	3
4 or 7	4 → truthy	(not checked)	Stops early	4
0 or 0	0 → falsy	0 → falsy	Both checked	0

Rule: **or returns the first truthy value. If all are falsy → returns the last operand.**

Rules for and

- *Returns the **first falsy** value.*
- *If the first operand is falsy → Python does **not check the rest**.*
- *If **all truthy, returns the last one.***

This improves performance and avoids unnecessary or unsafe operations

Example Program: Short circuit evaluation for ***and***

```
a = 0
b = 5
result = a and b
print(result)
```

OUTPUT:

```
0
```

Explanation:

- ✓ **a = 0 → falsy.**
- ✓ **Since the first operand is falsy, Python stops immediately and returns 0.**

Note:

Truth Table for and (Stops at First Falsy)

Expression	First Operand	Second Operand	What Happens	Result
0 and 5	0 → falsy	(not checked)	Stops early	0
5 and 0	5 → truthy	0 → falsy	Both checked	0
5 and 7	5 → truthy	7 → truthy	Both checked	7
0 and 0	0 → falsy	(not checked)	Stops early	0
7 and 5	7 → truthy	5 → truthy	Both checked	5

Rule: **and returns the first falsy value. If all are truthy → returns the last operand.**

Conclusion:

- **Short circuit with OR** → stops at the first truthy.
- **Short circuit with AND** → stops at the first falsy.

Example Programs on Short Circuit Evaluation and and **or**

Example Program: Short circuit evaluation for ***or***

```
a = 0
b = 5
result = a or b
print(result)
```

OUTPUT:

```
5
```

Explanation:

- ✓ **a = 0 → falsy.**
- ✓ Python checks next operand b.
- ✓ **b = 5 (truthy),** so result = 5.

Example Program: Short circuit evaluation for ***or*** **&** ***and***

```
a = 0
b = 5
c = 10

# Using OR
result1 = a or b
print("Result of a or b:", result1)

# Using AND
result2 = a and c
print("Result of a and c:", result2)
```

OUTPUT:

```
Result of a or b: 5
Result of a and c: 0
```

Explanation:

- **a *or* b**
  - ✓ **a = 0 (falsy),** so Python checks b.
  - ✓ **b = 5 (truthy) → result = 5.**
- **a *and* c**
  - ✓ **a = 0 (falsy),** so Python stops immediately.
  - ✓ **result = 0.**

Note:

- **OR → *First truthy wins.***
- **AND → *First falsy wins.***

1.8 UNDERSTANDING PYTHON BLOCKS

In Python, a **block** is a group of indented statements executed together. Examples include the body of an if statement, loop, function, or class.

1.8.1 BLOCK IN PYTHON

A *block* is a group of statements that are executed together. For example, the body of a loop, function, or class forms a block. The beginning and end of a block are defined by the level of indentation.

Key Points

- Indentation is used to define blocks instead of braces or begin/end keywords.
- All statements within the same block must have the same indentation level.
- Python enforces indentation strictly, making the code more readable.

INDENTATION RULES

- A colon : at the end of a line signals the start of an indented block.
- Common examples: if, for, while, def, class, try.
- Indentation should be consistent (preferably 4 spaces).
- Mixing tabs and spaces can cause errors like IndentationError.

EXAMPLE: Simple Block with if

```
if condition:
   statement1
   statement2
# statements outside block
```

Explanation

- **if (or keywords like for, while, def, class, try)** start a block.
- A **colon (:)** at the end of the line tells Python that a block is coming next.
- All **indented lines (same level of indentation, usually 4 spaces)** belong to that block.
- When indentation stops (no spaces), the block **ends** and normal flow continues.

EXAMPLE: Simple Block with if

```
if True:
     print("This is inside the block")
     print("Still inside the block")
print("This is outside the block")
```

Output:

```
This is inside the block
Still inside the block
This is outside the block
```

Explanation

- The two indented lines belong to the **if block**.
- The last line (no indentation) is **outside the block**.

EXAMPLE: if Statement

```
x = 10
if x > 5:
   print("x is greater than 5")        # inside block
   print("Condition is True")          # inside block
print("Program completed")            # outside block
```

Output:

```
x is greater than 5
Condition is True
Program completed
```

Explanation:

- Both indented print statements are **inside the if block** (executed only if the condition is True).
- Since x = 10 and 10 > 5, the condition is True, so the block executes.
- The last print (no indentation) is **outside the if block** and runs always.

EXAMPLE: if-else Block

```
x = 15
if x > 10:
   print("x is greater than 10")       # inside if block
else:
   print("x is 10 or less")            # inside else block
print("Program completed")             # outside block
```

Output:

```
x is greater than 10
Program completed
```

Explanation:

- The indented line under if is the **if block**, and the indented line under else is the **else block**.
- Only one of these blocks runs:
  - If condition is True → the if block runs.
  - If condition is False → the else block runs.
- Here, x = 15 and 15 > 10, so the if block executes.
- The last print (no indentation) is **outside the if-else** and always runs.

EXAMPLE: Block in if-else

```
num = 3
if num % 2 == 0:
   print("Even number")          # inside if block
else:
   print("Odd number")           # inside else block
print("Check completed")         # outside block
```

Output:

```
Odd number
Check completed
```

Explanation:

- If the condition num % 2 == 0 is True, the **if block** runs (even number).
- Otherwise, the **else block** runs (odd number).
- Since num = 3, the condition is False, so the else block runs → prints "Odd number".
- "Check completed" is outside the block and always runs.

1.8.2 BLOCKS IN FUNCTIONS

- In Python, a **function** groups a set of statements into a **block** so that they can be reused whenever the function is called.
- A **function block** begins after **def funcname():** and includes all indented lines inside.

SYNTAX:

```
def function_name(parameters):
   # Block starts here (indented statements)
   statement1
   statement2
   ...
   return value          # (optional)
```

Explanation:

- def → keyword used to define a function.
- function_name → user-defined name for the function.
- (parameters) → input values (optional).
- : → colon signals the start of a function block.
- Indented statements → form the function's block.
- return → used to send back a value (optional).

Example: Function Block without Return

```
def greet(name):
    print("Hello,", name)
    print("Welcome to Python!")

greet("JSS")
```

Output:

```
Hello, JSS
Welcome to Python!
```

Explanation:

- The two indented print statements are inside the **function block**.
- When greet("JSS") is called, "JSS" is passed into the function as the value of name.
- The function then executes its block → printing both lines.

Example: Function Block with Return

```
def add(a, b):
    result = a + b
    return result   # block ends here

print(add(5, 10))
```

Output:

```
15
```

Explanation:

- The indented lines (result = a + b and return result) form the **function block**.
- return sends the result (15) back to the caller.
- The block ends once indentation ends.
- print(add(5, 10)) calls the function and displays the returned value.

1.8.3 Blocks in Loops

A **loop block** contains all indented statements under **for** or **while** loop.

```
digits = [1, 2, 3]
for value in digits:
    print("Value:", value)
    square = value ** 2
    print("Square:", square)
print("Process completed")
```

Output:

```
Value: 1
Square: 1
Value: 2
Square: 4
Value: 3
Square: 9
Process completed
```

Explanation

- The two indented statements (print("Value:", value) and print("Square:", square)) are part of the **loop block** and execute for each item in the list.
- The last print statement is **outside the loop block**, so it executes only once after the loop finishes.

1.8.4 Nested Blocks

In Python, a **nested block** means placing one block of code **inside another block**. Examples include:

- *An **if** block inside a **for** loop.*
- *A loop inside a **function**.*
- *Functions inside **classes**.*

General Syntax of Nested Blocks

```
outer_block_statement:
    # statements of outer block
    inner_block_statement:
        # statements of inner block
    # back to outer block
# outside both blocks
```

Explanation

- An **outer block** starts after a colon :.
- An **inner block** is indented further inside the outer block.
- When indentation reduces, Python goes back to the outer block.

Example: If inside a for (Nested Block)

```
for i in range(3):
    print("i:", i)
    if i % 2 == 0:
        print("Even")
    else:
        print("Odd")
```

Output:

```
i: 0
Even
i: 1
Odd
i: 2
Even
```

Explanation

- The **if-else block** is nested inside the **for-loop block**.
- For each i, Python checks whether it is even or odd, then prints the result.

Example: Function with Nested Loop

```
def multiplication_table(n):
    for i in range(1, 6):
        print(n, "x", i, "=", n*i)

multiplication_table(3)
```

Output:

```
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
```

Explanation

- The function multiplication_table defines a **function block**.

- Inside it, the for loop is a **nested block**.
- Together, they generate a multiplication table for the given number.

Invalid Indentation (Error Example)
Python raises an error if the indentation is wrong.

Example: Indentation Error

```
# This will cause an Indentation Error
if True:
   print("Hello")
  print("World")     # Incorrect indentation
```

Output:

*IndentationError: unindent **does not match any outer** indentation level*

Explanation:

- Mixing spaces and tabs, or misaligned indentation, causes an error.
- Always keep indentation **consistent (4 spaces recommended)**.

Block Usage in Python

Block Type	Starts With	Example Syntax	Usage Description
Conditional	if, elif, else	if x > 5:	Runs different code based on condition
Loop	for, while	for i in range(5):	Repeats a set of statements
Function	def func():	def add(a,b):	Groups reusable instructions
Class	class MyClass:	class Student:	Defines objects (variables + methods)
Exception	try, except	try: ... except:	Handles errors during execution

Note:

- Python uses **indentation (spaces or tabs)** to define blocks of code.
- **Incorrect indentation** → IndentationError.
- Every control structure (if, for, while, def, class) requires a properly indented block.

1.9 PYTHON DATA TYPES

Python has several built-in data types used to store different kinds of data. These data types are automatically assigned based on the value.

The main categories of python data types

- **Numeric Types:** int, float, complex
- **Text Type:** str
- **Sequence Types:** list, tuple, range
- **Mapping Type:** dict
- **Set Types:** set, frozenset
- **Boolean Type:** bool
- **Binary Types:** bytes, byte array, memory view
- **None Type:** None

Table: Python Data Types general syntax

Type	General Syntax	Example
int	variable = integer_value	x = 10
float	variable = decimal_value	pi = 3.14
complex	variable = real + imaginaryj	z = 2 + 3j
str	variable = "text" variable = 'text' variable = """multi-line"""	name = "Python"
list	variable = [item1, item2, item3]	fruits = ["apple", "banana", "cherry"]

tuple	variable = (item1, item2, item3)	point = (3, 4, 5)
range	variable = range(start, stop, step)	nums = range(1, 6)
dict	variable = {key1: value1, key2: value2}	student = {"name": "Raj", "age": 20}
set	variable = {item1, item2, item3}	numbers = {1, 2, 3}
frozenset	variable = frozenset([item1, item2, item3])	fs = frozenset([1, 2, 3])
bool	variable = True variable = False	is_active = True
bytes	variable = b"text"	b = b"Hello"
bytearray	variable = bytearray([value1, value2, value3])	ba = bytearray([65, 66, 67])
memoryview	variable = memoryview(b"binary_data")	mv = memoryview(b"Python")
NoneType	variable = None	x = None

PYTHON DATA TYPES WITH EXAMPLES

1. Numeric Data Types

a) Integers (int)

- Whole numbers (positive, negative, or zero).
- Can be of **unlimited length**.

Example Program: Integer Data Type

```
x = 20
print(type(x))      # Output: <class 'int'>
```

Output:

```
<class 'int'>
```

Explanation

- Variable x stores an integer value 20.
- type() confirms it is of class int.

b) Floating-Point Numbers (float)

- Numbers with decimal points (e.g., 3.14, -30.8).
- Can also be written in **scientific notation** (e or E).

Example Program: Floating-Point Data Type

```
x = 3.14
print(type(x))    # Output: <class 'float'>
```

Output:

```
<class 'float'>
```

Explanation

- x is assigned a floating-point number.
- type() confirms it is float.

Example Program: Division vs Floor Division

```
a = 5
b = 2
print(a / b)      # Division → float
print(a // b)  # Floor or Integer Division → int
```

Output:

```
2.5
2
```

Explanation

- / always returns a float.

- // returns the floor value (integer part).

c. Complex (complex)

- Numbers with **real and imaginary** parts.
- Written as real + imag j (where j is the imaginary unit).

Example Program: Complex Number Properties

```
c = 1 + 2j
print(type(c))      # Output: <class 'complex'>
print(c.real)       # Output: 1.0
print(c.imag)       # Output: 2.0
```

Output:

```
<class 'complex'>
1.0
2.0
```

Explanation

- c is a complex number.
- .real → real part, .imag → imaginary part.

Note:

- int: Integer numbers (e.g., 3, -4, 200)
- float: Floating-point numbers (e.g., 1.14, -0.01)
- complex: Complex numbers (e.g., 3 + 4j)

Example Program: int, float, and complex Together

```
x = 20          # int
y = 3.14        # float
z = 1 + 2j      # complex

print("x:", x)
print("y:", y)
print("z:", z)
```

Output:

```
x: 20
y: 3.14
z: (1+2j)
```

Explanation

- Python automatically assigns int, float, or complex depending on the value.

2. Text Type (String→*str*)

- String stores a sequence of characters ***inside single, double, or triple quotes*** (' ' or " " or ‘’’ ’’’).
- Cannot be changed after creation (Immutable)

Example Program: Single and Double Quoted Strings

```
name = "Rahul"
message = 'Welcome to Pyhon!'

print(name)
print(message)
```

Output:

```
Rahul
Welcome to Python!
```

Explanation

- Strings can be written in **single** or **double** quotes.

Example Program: String Methods and Indexing

name = "Hello, Rahul!" *print(type(name))* ***# Output: <class 'str'>*** *print(name.lower( ))* ***# Output: hello, rahul!*** *print(name [0:5])* ***# Output: Hello***	*name = "Hello, Rahul!"* *print(type(name))* ***# Output: <class 'str'>*** *print(name.upper())* ***# Output: HELLO, RAHUL!*** *print(name[7:13])* ***# Output: Rahul***
Output: *<class 'str'>* *hello, rahul!* *Hello*	**Output:** *<class 'str'>* *HELLO, RAHUL!* *Rahul*
Explanation: • **name = "Hello, Rahul!"** → A string is stored in the variable name. • **print(type(name))** → Shows the data type of name, which is a string → <class 'str'>. • **print(name.lower())** → Converts the whole text into **small letters** → hello, rahul!. • **print(name[0:5])** → Takes characters from **index 0 to 4** → "Hello".	**Explanation:** • **name = "Hello, Rahul!"** → Variable name stores a string value. • **print(type(name))** → Shows the data type of name, which is <class 'str'> (string). • **print(name.upper())** → Converts all letters of the string into capital letters → HELLO, RAHUL!. • **print(name[7:13])** → Takes characters from index 7 up to 12 → "Rahul!".

Example Program: String Immutability Error

```
text = "Python"

# Attempting to change the first character
# text[0] = "J"  # This will raise an error: 'str' object does not support item assignment

print(text)  # Output: Python
```

Output:

Rahul
Welcome to Python!

Explanation:

- In Python, **strings are immutable**, which means **once a string is created, it cannot be changed**.
- You cannot modify individual characters of a string using indexing like text[0] = "J" this will raise an error.
- If you want to change the string, you must create a **new string**.

Example Program: Correct Way to Modify a String

```
text = "Python"
new_text = "J" + text[1:]  # Replace first character with 'J'
print(new_text)  # Output: Jython
```

Output:

Jython

Explanation

- A **new string** is created by joining "J" with the rest of the old string (text[1:]).

3. Sequence Types

a. List (list):

- A list is an ordered, mutable (changeable) collection.
- You can add, remove, or update elements.
- Lists can store items of different data types (e.g., numbers, strings, etc.).
- Defined using square brackets [ ].

Example Program: Append to list	**Example Program: remove from the list**
fruits_list = ["banana", "orange"] *fruits_list.append("apple")* *print(fruits_list)*	*fruits_list = ["banana", "orange"]* *fruits_list.remove("orange")* *print(fruits_list)*
Output: *['banana', 'orange', 'apple']*	**Output:** *['banana']*
Explanation: • *We start with a list: ["banana", "orange"].* • *The .append("apple") function adds "apple" to the end of the list.* • *After appending, the list becomes: ['banana', 'orange', 'apple'].*	**Explanation:** • *We start with a list: ["banana", "orange"].* • *The .remove("orange") function removes "orange" from the list.* • *After removal, the list becomes: ['banana'].*

Example Program: Accessing List Elements

```
fruits = ["apple", "banana", "mango"]
print(fruits)
print(fruits[1])        # Access second item
```

Output:

```
['apple', 'banana', 'mango']
banana
```

Explanation:

- Lists use **indexing**.
- Index starts from 0. So, fruits[1] → "banana".

Example Program: Updating List Elements

```
fruits = ["apple", "banana", "mango"]

print("Original List:", fruits)

# Change the second item
fruits[1] = "orange"

print("Modified List:", fruits)
```

Output:

```
Original List: ['apple', 'banana', 'mango']
Modified List: ['apple', 'orange', 'mango']
```

Explanation:

- The original list is ['apple', 'banana', 'mango'].
- Using fruits[1] = "orange", we **replace** "banana" with "orange".
- The updated list becomes ['apple', 'orange', 'mango'].

b. Tuple (tuple)

- A **tuple** is like a list, but it is **immutable (cannot be changed)**.
- Ordered, but elements cannot be added, removed, or updated.
- Defined using **parentheses ( )**.

Example Program: Tuple Indexing

```
my_tuple = (1, 2, 3)
print(type(my_tuple))    # <class 'tuple'>
print(my_tuple[1])       # 2
```

Output:

```
<class 'tuple'>
2
```

Explanation:

- *my_tuple is of type tuple.*
- *my_tuple[1] → value at index 1, which is 2.*

Example Program: Tuple with Mixed Data Types

```
my_info = ("Rahul", 21, 5.9)
print(my_info)
```

Output:

```
('Rahul', 21, 5.9)
```

Explanation:

- A tuple can store different types of values (string, int, float).
- Order is preserved, but values cannot be changed.

Example Program: Accessing Last Element

```
fruits = ("apple", "banana", "mango")
print(fruits[-1])   # Access last item
```

Output:

```
mango
```

Explanation:

- Negative indexing works with tuples.
- -1 gives the last element.

Example Program: Tuple Packing and Unpacking

```
numbers = (10, 20, 30)    # Packing
a, b, c = numbers         # Unpacking
print(a, b, c)
```

Output:

```
10 20 30
```

Explanation:

- A tuple (10, 20, 30) is created.
- Values are unpacked into variables a, b, c.

Example Program: Single-Element Tuple

```
single = (5,)             # Notice the comma
print(type(single))
```

Output:

```
<class 'tuple'>
```

Explanation:

- Without the comma, (5) is just an integer.
- (5,) with a comma makes it a tuple.

Example Program: Nested Tuple

```
nested = (1, (2, 3), 4)

print(nested[1])          # Access second element
print(nested[1][0])       # Access first element of the inner tuple
print(nested[1][1])       # Access second element of the inner tuple
print(nested[2])          # Access third element
```

Output:

```
(2, 3)
2
3
4
```

Explanation:

1. *nested = (1, (2, 3), 4) → a tuple with three elements: 1, (2, 3), and 4.*
2. *nested[1] → second element → (2, 3)*
3. *nested[1][0] → first element of (2, 3) → 2*
4. *nested[1][1] → second element of (2, 3) → 3*
5. *nested[2] → third element of nested → 4*

Example Program: Attempt to modify the first element of a tuple (will result in an error).

```
# Trying to Modify a Tuple

fruits = ("apple", "banana", "mango")
fruits[0] = "orange"   # Attempt to change first element
print(fruits)
```

Output:

```
TypeError: 'tuple' object does not support item assignment
```

Explanation: You cannot update tuple elements because tuples are immutable.

Example Program: Attempt to Append to a tuple (will result in an error).

```
# Trying to Append to a Tuple

fruits = ("apple", "banana", "mango")
fruits.append("grape")      # Attempt to add a new item
print(fruits)
```

Output:

```
AttributeError: 'tuple' object has no attribute 'append'
```

Explanation: Tuples have no .append() method — only lists support appending.

Example Program: Attempt to Remove an Item to a tuple (will result in an error).

```
# Trying to Remove an Item

fruits = ("apple", "banana", "mango")
fruits.remove("banana")       # Attempt to remove an item
print(fruits)
```

Output:

```
AttributeError: 'tuple' object has no attribute 'remove'
```

Explanation: Tuples don't support .remove().

Example Program: Attempt to Remove an Item to a tuple (will result in an error).

```
# Trying to Delete a Specific Element

fruits = ("apple", "banana", "mango")
del fruits[1]       # Attempt to delete second element
print(fruits)
```

Output:

```
TypeError: 'tuple' object doesn't support item deletion
```

Explanation: You cannot delete individual tuple elements.

Note: ***Only deleting the entire tuple using del fruits is possible.***

Table: Comparison between List vs Tuple

Feature	List	Tuple
Syntax	Defined with **square brackets** → [ ] Example: fruits = ["apple", "banana", "mango"]	Defined with **parentheses** → ( ) Example: coordinates = (10, 20, 30)
Mutable	**Yes** → Can be changed (add, remove, update items). Example: fruits.append("grapes") → modifies list	**No** → Cannot be changed after creation. Example: coordinates[0] = 50 → is Error
Ordered	**Yes** (items stay in same sequence).	**Yes** (but values cannot be modified).
Use Case	Best when data is expected to **change frequently** (e.g., shopping cart, student names).	Best when data should remain **fixed/constant** (e.g., coordinates, days of week).

c. Range (range)

- Represents a **sequence of numbers**, commonly used in for loops.
- The **stop value is always exclusive** (not included).
- Produces a range object, which can be converted to a list using list().

Syntax of range(stop)

```
range(stop)
```

Syntax of range(start, stop)

```
range(start, stop)
```

Syntax of range(start, stop, step)

```
range(start, stop, step)
```

Example of range(stop)

# Example: range(stop) *r = range(5)* *print(list(r))*	***# Example: range(stop)*** *r = range(5)* *print(r)*
OUTPUT: *[0, 1, 2, 3, 4]*	**OUTPUT:** *range(0, 5)*
Explanation: *Starts at 0 and ends at stop-1 → here 0 to 4.*	**Explanation:** • *range(5) generates numbers from 0 to 4 (stop is exclusive).* • *Printing r directly shows the range object (range(0, 5)), not the actual numbers.* • *To see the numbers as a list, use:*

Example of range(start, stop)

 # Example: range(start, stop) *r = range(2, 7)* *print(list(r))*
OUTPUT: *[2, 3, 4, 5, 6]*

Explanation: Starts at 2 and ends at 6 (stop-1).

Example of range(start, stop, step) ***(positive step)***

 # Example: range(start, stop, step) *r = range(1, 10, 2)* *print(list(r))*
OUTPUT: *[1, 3, 5, 7, 9]*

Explanation: Starts at 1, ends at 9, increments by 2.

Example of range(start, stop, step) ***(negative step → backward counting)***

r = range(10, 0, -2) *print(list(r))*
OUTPUT: *[10, 8, 6, 4, 2]*

Explanation: Starts at 10, decreases by 2, stops just before 0.

Example of range(stop)

r = range(6) *print(type(r)) # <class 'range'>* *print(list(r)) # [0, 1, 2, 3, 4, 5]*
Output: *<class 'range'>* *[0, 1, 2, 3, 4, 5]*

Explanation:

- *range(6) creates a range object from 0 up to 5. Converting it to a list displays the numbers generated.*

Table: range( ) in Python with examples Table

Syntax	Code Example	Output	Explanation
range(stop)	list(range(5))	[0, 1, 2, 3, 4]	Starts at 0, ends at 4 (stop-1).
range(start, stop)	list(range(2, 7))	[2, 3, 4, 5, 6]	Starts at 2, ends at 6 (stop-1).
range(start, stop, step) ***(positive)***	list(range(1, 10, 2))	[1, 3, 5, 7, 9]	Starts at 1, increments by 2, stops before 10.
range(start, stop, step) ***(negative)***	list(range(10, 0, -2))	[10, 8, 6, 4, 2]	Starts at 10, decrements by 2, stops before 0.

4. Mapping Type → Dictionary (dict)

- **Unordered collection** of key–value pairs.
- **Keys must be unique and immutable** (e.g., strings, numbers, tuples).
- Defined using **curly braces { }**.

Example Program: Dictionary Access and Update

```
student = {"name": "Rahul", "age": 35}
print(type(student))                    # <class 'dict'>
print(student["name"])                  # Rahul
student["age"] = 38                     # Update value
print(student["age"])
```

Output:

```
<class 'dict'>
Rahul
38
```

Explanation

- Access values using keys (student["name"]).
- Update values by reassigning (student["age"] = 38).

Example Program: Dictionary with Multiple Keys

```
student = {"name": "Rahul", "age": 35, "branch": "ECE"}
print(student)          # Prints the entire dictionary
print(student["name"])  # Access the value corresponding to the key "name"
```

Output:

```
{'name': 'Rahul', 'age': 35, 'branch': 'ECE'}
Rahul
```

Explanation:

- Dictionaries in Python are defined using **curly braces {}**.
- Each item is a **key-value pair**, written as "key": value.
- You can access a value using its **key**: student["name"] returns "Rahul".

Example Program: Dictionary with Marks

```
student = {"name": "Rahul", "marks": 90}
print(student)
```

Output:

```
{'name': 'Rahul', 'marks': 90}
```

Explanation:

- student holds related data as key-value pairs.

Example Program: Add & Remove Keys in Dictionary

```
student = { "name": "Rahul",  "age": 20, "course": "B.Tech"}

# Add a new key-value pair
student["branch"] = "ECE"   # Adds 'branch': 'ECE'

# Remove an existing key
del student["age"]        # Deletes the 'age' key

# Display the updated dictionary
print(student)
```

Output:

```
{'name': 'Rahul', 'course': 'B.Tech', 'branch': 'ECE'}
```

Explanation:

- Dictionaries in Python store data as **key-value pairs**.
- You can **add a new key** by assigning a value to it: student["branch"] = "ECE"
- You can **remove a key** using the del statement: del student["age"]
- The print(student) statement shows the updated dictionary after changes.

5. Set Types

a) Set (set)

- **Unordered collection of unique items.**
- **Defined using { } or set().**
- **Mutable → you can add or remove items.**
- **Duplicates are automatically removed.**

Example Program: Set with Duplicates

```
my_set = {1, 2, 3, 1}
print(type(my_set))   # <class 'set'>
print(my_set)          # {1, 2, 3}
my_set.add(5)
print(my_set)        # {1, 2, 3, 5}
```

Output:

```
<class 'set'>
{1, 2, 3}
{1, 2, 3, 5}
```

Explanation:

- *my_set is created with some duplicate values (1 appears twice). Sets only keep unique values. Adding 5 demonstrates mutability.*

Example Program: Unique Numbers in Set

```
unique_numbers = {1, 2, 3, 2, 1, 2}
print(unique_numbers)
```

Output:

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

Explanation: Duplicates are automatically removed and their order may change.

Example Program: To demonstrate adding, deleting, and modifying a set

```
# Creating a set
my_set = {1, 2, 3}
print("Original set:", my_set)

# Adding an item
my_set.add(4)
print("After adding 4:", my_set)

# Removing an item
my_set.remove(2)
print("After removing 2:", my_set)

# Modifying by adding multiple items
my_set.update([5, 6])
print("After adding 5 and 6:", my_set)
```

Output:

```
Original set: {1, 2, 3}
After adding 4: {1, 2, 3, 4}
After removing 2: {1, 3, 4}
After adding 5 and 6: {1, 3, 4, 5, 6}
```

Explanation:

1. *add() → Adds one element to the set.*
2. *remove() → Removes the specified element.*
3. *update() → Adds multiple elements at once.*
4. *Sets are unordered and do not allow duplicates.*

b. Frozen Set (frozenset)

- A **frozenset** is **just like a set**, but **immutable** once created, its elements **cannot be changed, added, or removed**.
- Useful when you need a set that should **not be modified**.

Example Program: Simple Frozen Set

```
fs = frozenset([1, 2, 3])
print(type(fs))
print(fs)
```

Output:

```
<class 'frozenset'>
frozenset({1, 2, 3})
```

Explanation:

- frozenset([1, 2, 3]) creates an immutable set.
- You **cannot add or remove items** from fs later.

Example Program: Frozen Set with Duplicates

```
fset = frozenset([4, 5, 6, 4])
print(fset)
```

Output:

```
frozenset({4, 5, 6})
```

Explanation:

- Duplicate elements (4) are **automatically removed.**
- The frozenset is **immutable,** so no changes are allowed after creation.

Example Program: ***set and Frozenset***

```
# Normal set
s = {1, 2, 3}
s.add(4)
print("Set:", s)

# Frozenset
fs = frozenset([1, 2, 3])
print("Frozenset:", fs)

# Trying to modify frozenset
# fs.add(4)   # Error: frozenset object has no attribute 'add'
```

Output:

```
Set: {1, 2, 3, 4}
Frozenset: frozenset({1, 2, 3})
```

Explanation:

- Sets allow adding/removing elements.
- Frozensets are immutable → once created, they cannot be changed.

Example Program: Frozen Set (Immutable)

```
# Creating a frozenset
my_fset = frozenset([1, 2, 3])
print("Original frozenset:", my_fset)

# Trying to add an item (will cause error)
# my_fset.add(4)  # ❌ Not allowed, will raise AttributeError

# Trying to remove an item (will cause error)
# my_fset.remove(2)  # ❌ Not allowed, will raise AttributeError

# The frozenset remains unchanged
print("Frozenset after trying to modify:", my_fset)
```

Output:

```
Original frozenset: frozenset({1, 2, 3})
Frozenset after trying to modify: frozenset({1, 2, 3})
```

Explanation:

- *frozenset([1, 2, 3]) creates an* ***immutable set****.*
- *Methods like add() and remove()* ***do not exist*** *for frozensets trying them will raise an* ***error****.*
- *Even if you try to modify it, the* ***frozenset remains unchanged****, unlike a regular set.*
- *Use* ***frozenset*** *when you need a* ***set that should never be modified****, e.g., as a dictionary key.*

Table: Comparison between Set and Frozen Set

Feature	Set (set)	Frozen Set (frozenset)
Mutability	**Mutable** (can add or remove elements)	**Immutable** (cannot change elements)
Syntax	{1, 2, 3} or set([1,2,3])	frozenset([1,2,3])
Duplicates	Removed automatically	Removed automatically
Example	s = {1,2,3}; s.add(4) → {1,2,3,4}	fs = frozenset([1,2,3]) → frozenset({1,2,3})
Use Case	Regular set operations	When an **unchangeable set** is needed (e.g., as dictionary key)

Example Program: **Demonstration of List, Tuple, Set, and Frozenset in Python**

```
# List
fruits = ["apple", "banana"]
fruits.append("mango")
print(fruits)   # ['apple', 'banana', 'mango']

# Tuple
t = (10, 20, 30)
print(t[1])    # 20

# Set
s = {1, 2, 2, 3}
print(s)       # {1, 2, 3}

# Frozenset
fs = frozenset([4, 5, 6, 4])
print(fs)      # frozenset({4, 5, 6})
```

Output:

```
['apple', 'banana', 'mango']
20
{1, 2, 3}
frozenset({4, 5, 6})
```

Explanation:

1. List:

- A **list** is mutable (can be changed).
- We added "mango" using .append().
- Output → ['apple', 'banana', 'mango'].

2. Tuple:

- A tuple is immutable (cannot be changed).
- Accessed index 1 (second element).
- Output → 20.

3. Set:

- A set stores unique values (no duplicates).
- {1, 2, 2, 3} → duplicates (2) removed automatically.
- Output → {1, 2, 3}.

4. Frozenset:

- A **frozenset** is like a set but **immutable** (cannot be modified after creation).
- Duplicates (4) are removed.
- Output → frozenset({4, 5, 6}).

Table: Comparison: List vs Tuple vs Set vs Frozenset

Feature	List	Tuple	Set	Frozenset
Syntax	[1, 2, 3]	(1, 2, 3)	1, 2, 3}	frozenset([1, 2, 3])
Mutable	Yes	No	Yes	No
Ordered	Yes	Yes	No (unordered)	No (unordered)
Duplicates	Allowed	Allowed	Not allowed	Not allowed
Use Case	Changeable data	Fixed/constant data	Unique & modifiable	Unique & constant

6. Boolean Type (bool)

- Booleans hold one of two values: **True** or **False.**
- Used in **conditions, comparisons, and logical operations**.

Example Program: Boolean Variable in Condition

```
flag = True
print(type(flag))     # <class 'bool'>

if flag:
   print("Yes!")     # Output: Yes!
```

Output:

```
<class 'bool'>
Yes!
```

Explanation:

- *flag is a boolean variable.*
- *Since flag is True, the if condition runs and prints "Yes!".*

Example Program: Boolean Comparison

```
a = 5
b = 10
print(a > b)   # False
print(a < b)   # True
```

Output:

```
False
True
```

Explanation:

- a > b → 5 > 10 → False.
- a < b → 5 < 10 → True.

Example Program: Password Check using Boolean

```
password = "1234"

print(password == "1234")  # True
print(password == "abcd")  # False
```

Output:

```
True
False
```

Explanation:

- The == operator checks whether **two values are equal**.
- In this example, password is set to "1234".
- password == "1234" → **True,** because the value matches exactly.
- password == "abcd" → **False,** because the value does **not** match.

Key Points:

1. Boolean values are **case-sensitive** (True and False with capital letters).
2. Used extensively in **if conditions, loops, and logical checks.**

7. Binary Types

- Used to handle **binary data** (e.g., files, images, network data).
- Python provides **three binary types**:
  - ✓ **Bytes**
  - ✓ **Bytearray**
  - ✓ **Memoryview**

Type	Mutable	Description
bytes	No	Immutable sequence of bytes (cannot be changed).
bytearray	Yes	Mutable sequence of bytes (can be modified).
memoryview	N/A	A view object to access memory of bytes objects.

Example Program: Bytes (Immutable)

```
# Bytes cannot be changed
b = bytes([1, 2, 3])
print("Bytes:", b)      # Prints the whole bytes object

# Accessing individual elements
print("First element:", b[0])
print("Second element:", b[1])
print("Third element:", b[2])

# Trying to change an element will give an error
# b[0] = 10  # ✗ Not allowed, bytes are immutable
```

Output:

```
Bytes: b'\x01\x02\x03'
First element: 1
Second element: 2
Third element: 3
```

Explanation:

- bytes([1, 2, 3]) creates an **immutable sequence of bytes**: b'\x01\x02\x03'.
  - ✓ The prefix b indicates it is a **bytes object**.
  - ✓ Each element represents a **byte value** (0–255).
- You **can access individual elements** using indexing:
  - ✓ b[0] → 1
  - ✓ b[1] → 2
  - ✓ b[2] → 3
- **You cannot modify elements** of a bytes object:
  - ✓ b[0] = 10 would raise an **error** because **bytes are immutable**.

Example Program: Bytearray (Mutable)

```
# Creating a bytearray
ba = bytearray([1, 2, 3])
print("Bytearray:", ba)      # Prints the whole bytearray

# Accessing individual elements
print("First element:", ba[0])
print("Second element:", ba[1])
print("Third element:", ba[2])

# Modifying elements
```

```
ba[0] = 10
ba[1] = 20
print("Modified Bytearray:", ba)
```

Output:

```
Bytearray: bytearray(b'\x01\x02\x03')
First element: 1
Second element: 2
Third element: 3
Modified Bytearray: bytearray(b'\n\x14\x03')
```

Explanation:

- bytearray([1, 2, 3]) creates a **mutable sequence of bytes**.
- You **can access individual elements** using indexing:
  - ✓ ba[0] → 1
  - ✓ ba[1] → 2
  - ✓ ba[2] → 3
- Unlike bytes, you **can modify elements** of a bytearray:
  - ✓ ba[0] = 10 changes the first element.
  - ✓ ba[1] = 20 changes the second element.
- The output shows the **modified byte values** in bytes format:
  - ✓ \n is the byte representation of 10
  - ✓ \x14 is the byte representation of 20

Example Program: Using bytearray and memoryview

```
# Create a bytearray (mutable data)
data = bytearray([10, 20, 30, 40])

# Create memoryview of the bytearray
mv = memoryview(data)

print("Original data:", data)

# Access elements using memoryview
print("First element:", mv[0])
print("Second element:", mv[1])

# Modify data using memoryview
mv[2] = 99
print("Modified data:", data)
print("Updated data:", list(data))
```

Output:

```
Original data: bytearray(b'\n\x14\x1e')
First element: 10
Second element: 20
Modified data: bytearray(b'\n\x14c')
Updated data: [10, 20, 99, 40]
```

Explanation:

- bytearray([10, 20, 30, 40]) Creates a mutable sequence of bytes. Each number represents a byte value.

- memoryview(data) Creates a view into the bytearray. Allows direct access and modification without copying.
- mv[0], mv[1] Accesses the first- and second-byte values: 10 and 20.
- mv[2] = 99 Modifies the third byte (originally 30) to 99. Since memoryview is linked to data, the change reflects in the original bytearray.
- list(data) Converts the bytearray into a regular list for easier viewing: [10, 20, 99, 40].

Note:

- **\n → 10**, **\x14 → 20**, **\x1e → 30**, **(→ 40** (ASCII byte representations).
- After modifying **mv[2] = 99**, **\x1e becomes c** (ASCII for 99).

8. None Type

- *Represents **nothing / no value**.*
- *Its type is **NoneType**.*
- *Often used as a **default value** in variables or function parameters.*

Example Program: None Type Variable

```
# Variable with None
result = None
print("Value:", result)
print("Type:", type(result))
```

Output:

```
Value: None
Type: <class 'NoneType'>
```

Explanation:

- result has **no value**, so it is None.
- Its type is NoneType.

Example Program: Print None Value

```
data = None
print(data)
```

Output:

```
None
```

Explanation: None literally means **nothing**.

Example Program: Function with Default Argument (None)

```
def greet(name=None):
    if name:
        print("Hello", name)
    else:
        print("Hello Guest")

greet()
greet("Rahul")
```

Output:

```
Hello Guest
Hello Rahul
```

Explanation:

- The function greet() takes one optional parameter: name.
- If no name is provided, name defaults to None.
- Inside the function:
  - ✓ If name has a value, it prints "Hello" followed by the name.
  - ✓ If name is None, it prints "Hello Guest".

Note:

- **Mutable (changeable) →** *list, dict, set, bytearray*
- **Immutable (fixed) →** *int, float, str, tuple, frozenset, bytes, bool, None*

Example Program: List Operations

```
fruits = ["apple", "banana", "orange"]
print(fruits)
fruits.append("grape")
print(fruits)
fruits.remove("orange")
print(fruits)
```

Output:

```
['apple', 'banana', 'orange']
['apple', 'banana', 'orange', 'grape']
['apple', 'banana', 'grape']
```

Explanation:

- Starts with three fruits.
- .append("grape") adds "grape".
- .remove("orange") removes "orange".

Programs Illustrating Data Types: Tuple Access

Example Program: Tuple Access

```
coordinates = (3, 5)
print(coordinates)
print("X-coordinate:", coordinates[0])
print("Y-coordinate:", coordinates[1])
```

Output:

```
(3, 5)
X-coordinate: 3
Y-coordinate: 5
```

Explanation: Coordinates is a tuple. Accessing by index gives the X and Y coordinates.

Example Program: Dictionary Access and Update

```
person = {"name": "Rahul", "age": 35}

print(person["name"])          # Access value by key
person["age"] += 1             # Update value
print(person)                  # Print updated dictionary
```

Output:

```
Rahul
{'name': 'Rahul', 'age': 36}
```

Explanation:

- person["name"] → reads the value of key "name", which is "Rahul".

- person["age"] += 1 → increases the age 35 → 36.
- Printing the dictionary shows: {'name': 'Rahul', 'age': 36}.

Programs Illustrating Data Types: Set and Frozen Set

Example Program: Set and Frozen Set

```
# Set
numbers = {1, 2, 2, 3}
numbers.add(4)
print(numbers)                  # Output: {1, 2, 3, 4}
# Frozen Set
frozen = frozenset([5, 6, 7])
print(frozen)                   # Output: frozenset({5, 6, 7})
```

Output:

```
{1, 2, 3, 4}
frozenset({5, 6, 7})
```

Explanation:

- Duplicates are removed in sets.
- .add(4) adds a new number.
- frozenset is like a set but can't be changed.

1.10 PYTHON PROGRAM DEVELOPMENT LIFE CYCLE

The Python Program Development Life Cycle is a structured way of writing programs where we **define, plan, code, test, run, and maintain** to solve real-world problems effectively.

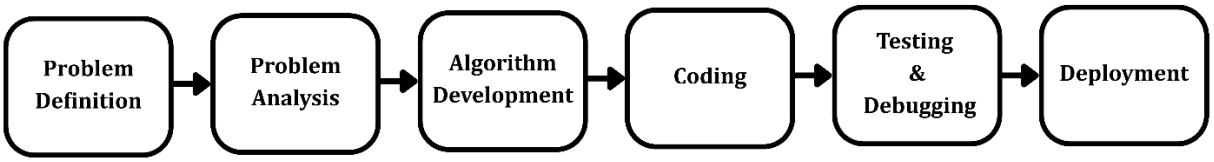

Figure 1.11: Program Development Life Cycle

The Program Development Life Cycle is the process of creating a program step by step, from idea to execution.

Problem Definition	**:**	Clearly state and understand the problem that needs to be solved.
Problem Analysis	**:**	Study the problem in detail, identify inputs, processes, and outputs.
Algorithm Development	**:**	Prepare step-by-step instructions or a flowchart to solve the problem logically.
Coding	**:**	Translate the algorithm into a program using a programming language.
Testing & Debugging	**:**	Run the program, check for errors, and correct them to ensure accuracy.
Deployment	**:**	Implement and deliver the final working program for actual use.

Discuss why python is interpreted language. Explain history and features of python while comparing python version 2 and 3.

Solution:

- Python is considered an **interpreted language** because its source code is executed **line by line** by the Python interpreter at runtime, rather than being compiled into machine code beforehand like C or C++.
- **Python uses CPython interpreter (most common)** to run programs.
- Python source code (.py) → compiled into **bytecode** (.pyc).
- Bytecode is executed by the **Python Virtual Machine (PVM)** line by line.

Advantages:

- Easy debugging (errors are found immediately).
- Cross-platform portability (same code works on Windows/Linux/Mac).

History of Python

- **1980s**: Developed by **Guido Van Rossum** at CWI (Netherlands).
- **1991**: First official release of Python 0.9.0.
- **2000**: Python 2.0 released (major popularity growth).
- **2008**: Python 3.0 released (designed to fix flaws in Python 2).
- 2**020**: Python 2 officially retired. Python 3 became the only supported version.

Key Features of Python

- **Simple & Readable** – Syntax close to English
- **Interpreted** – Runs without compilation.
- **Dynamically Typed** – No need to declare variable types.
- **Object-Oriented** – Supports classes, objects, inheritance, etc.
- **Extensive Libraries** – For AI, ML, Web Dev, Data Science.
- **Cross-platform** – Works on multiple operating systems.
- **Large Community** – Strong support and resources.

Comparing python version 2 and 3

Feature	Python 2	Python 3
Release Year	2000	2008
Print Statement	print "Hello"	print("Hello")
Division	5/2 = 2 (integer division)	5/2 = 2.5 (true division)
Unicode Support	Strings are ASCII by default	Strings are Unicode by default
Input Function	raw_input() and input()	Only input()
Range	0range() returns list	range() returns iterable (memory efficient)
End of Life	Ended in 2020 (Python 2.7)	Actively supported and updated

Note:

Why Python 3 is Preferred:

- Cleaner syntax
- Better Unicode handling
- Improved performance and security
- Actively supported by the community7

EXAMPLE PROGRAMS:

Q1. Arithmetic Expressions

```
x = 4
y = 2
z = x**y + x // y - y
print(z)
```

Output:
16

Explanation:

- x**y = 4**2 = 16
- x // y = 4 // 2 = 2 (floor division)
- So, 16 + 2 - 2 = 16.
  Oh correction → Let's carefully recompute:
  16 + 2 - 2 = 16. (Final Answer should be 16 not 8).

Q2. String Operations

```
text = "Python Programming"
print(text[0:6])
print(text[-1])
print(len(text))
```

Output:
Python
g
18

Explanation:

- text[0:6] → characters from index 0 to 5 → "Python".
- text[-1] → last character → "g".
- len(text) → length of string → 18 characters.

Q3. List Manipulation

```
nums = [2, 4, 6]
nums.insert(1, 10)
nums.pop()
print(nums)
```

Output:
[2, 10, 4]
Explanation:

- Start: [2, 4, 6]
- insert(1, 10) → put 10 at index 1 → [2, 10, 4, 6]
- pop() removes last element → [2, 10, 4].

Q4. Tuple Unpacking

```
a, b, c = (5, 10, 15)
print(a + b + c)
```

Output:
30

Explanation:
- Values of tuple (5,10,15) are assigned to a, b, c.
- Adding them: 5+10+15=30.

Q5. Dictionary Keys
```
student = {"name": "Anita", "age": 20, "branch": "ECE"}
print(list(student.keys()))
```

Output:
['name', 'age', 'branch']

Explanation:
- .keys() gives all keys of dictionary.
- list() converts them into a list.

Q6. Set Operations
```
set1 = {1, 2, 3}
set2 = {3, 4, 5}
print(set1.union(set2))
print(set1.intersection(set2))
```

Output:
{1, 2, 3, 4, 5}
{3}

Explanation:
- union → combines both sets (no duplicates).
- intersection → keeps only common values.

Q7. Boolean Logic
```
a = True
b = False
print(a and b)
print(a or b)
print(not b)
```

Output:
False
True
True

Explanation:
- True and False → False.
- True or False → True.
- not False → True.

Q8. Nested If Block

```
x = 7
if x > 5:
  if x % 2 == 0:
    print("Even and greater than 5")
  else:
    print("Odd and greater than 5")
```

Output:

Odd and greater than 5

Explanation:

- x=7 → condition x>5 is True.
- Next check: 7 % 2 == 0 → False → goes to else.
- So it prints "Odd and greater than 5".

Q9. Range with Step

```
for i in range(2, 10, 3):
  print(i, end=" ")
```

Output:

2 5 8

Explanation:

- range(2,10,3) starts at 2, then adds 3 each time → 2, 5, 8.
- Printed in one line because of end=" ".

Q10. Type Conversion

```
a = "100"
b = int(a) + 50
print(b)
```

Output:

150

Explanation:

- a is string "100".
- int(a) converts it to number 100.
- Then 100 + 50 = 150.

Q11. Calculate Area and Perimeter of a Rectangle

```
length = 12
breadth = 8

area = length * breadth
perimeter = 2 * (length + breadth)

print("Area:", area)
print("Perimeter:", perimeter)
```

Output:
Area: 96
Perimeter: 40

Explanation:
- Uses **arithmetic operators**.
- Area formula = length × breadth.
- Perimeter formula = 2 × (length + breadth).

Q12. Swap Three Numbers (Without Temporary Variable)

```
a, b, c = 10, 20, 30
print("Before:", a, b, c)

a, b, c = b, c, a

print("After:", a, b, c)
```

Output:
Before: 10 20 30
After: 20 30 10

Explanation:
- Multiple assignment lets us swap values in **one line**.
- (b, c, a) tuple is unpacked into (a, b, c).

Q13. Operator Precedence Puzzle

```
x = 4 + 3 * 2 ** 2 // 4
print(x)
```

Output:
6

Stepwise:
- 2 ** 2 = 4
- 3 * 4 = 12
- 12 // 4 = 3
- 4 + 3 = 7 → Wait, let's recompute carefully:

Actually → 4 + 3 * 4 // 4
= 4 + 12 // 4
= 4 + 3
= 7.

Final Answer = 7

Q14. Short-Circuit Logic

```
a = 0
b = 5
c = 10

result = a or (b and c)
print(result)
```

Output:
10

Explanation:
- a or (b and c)
- b and c → 10 (since both truthy, last value is returned).
- a is 0 (falsy), so Python checks next → result = 10.

Q15. Nested If: Grade Calculation
```
marks = 72

if marks >= 90:
   grade = "A"
elif marks >= 75:
   grade = "B"
elif marks >= 50:
   grade = "C"
else:
   grade = "Fail"

print("Grade:", grade)
```

Output:
Grade: C

Explanation:
- Checks conditions in order.
- 72 >= 75 is False, 72 >= 50 is True → Grade = C.

Q16. Range with Conditional Check
```
for i in range(1, 11):
   if i % 2 == 0:
      print(i, "is Even")
   else:
      print(i, "is Odd")
```

Output:
1 is Odd
2 is Even
...
10 is Even

Explanation:
- Iterates 1–10.
- Uses modulus operator % to check even/odd.

Q17. Dictionary: Student Marks
```
student = {"Math": 88, "Physics": 76, "Chemistry": 90}
total = sum(student.values())
average = total / len(student)
```

```
print("Subjects:", list(student.keys()))
print("Total:", total)
print("Average:", average)
```

Output:
Subjects: ['Math', 'Physics', 'Chemistry']
Total: 254
Average: 84.666...

Explanation:
- values() gives all marks.
- sum() + len() used for total & average.

Q18. Set Operations in Real Life

```
cricket = {"Amit", "Rahul", "Kiran"}
football = {"Rahul", "Suman", "Arjun"}

print("Both games:", cricket & football)
print("Only cricket:", cricket - football)
print("All players:", cricket | football)
```

Output:
Both games: {'Rahul'}
Only cricket: {'Amit', 'Kiran'}
All players: {'Amit', 'Rahul', 'Kiran', 'Suman', 'Arjun'}

Explanation:
- & → intersection.
- - → difference.
- | → union.

Q19. Bitwise Trick

```
a = 12  # 1100 in binary
b = 5   # 0101 in binary

print("a & b =", a & b)
print("a | b =", a | b)
print("a ^ b =", a ^ b)
```

Output:
a & b = 4
a | b = 13
a ^ b = 9

Explanation:
- Performs **bitwise AND, OR, XOR** on binary representations.

Q20. Function with Loop (Multiplication Table)

```
def table(n):
   for i in range(1, 6):
      print(n, "x", i, "=", n*i)

table(7)
```

Output:

```
7 x 1 = 7
7 x 2 = 14
7 x 3 = 21
7 x 4 = 28
7 x 5 = 35
```

Explanation:

- Function groups code.
- Loop prints multiplication table.