Name – Gathram Kesava Murthy

Registered mail id- kesavagathram.murthy@outlook.com

Course Name – Master Generative AI: Data Science Course

Assignment Name - Python Baiscs Assignment

Submission Date – 25th October 2024

1.Explain the key features of Python that make it a popular choice for programming

Python is one of the most popular programming languages for good reason. It’s easy to learn, fast to develop on and has some fantastic built-in functionality. But what really takes Python above and beyond other languages out there, is it’s incredible versatility. Here are just a few reasons why Python is so versatile and as a result, so widely used:

1. Easy to Learn and Use

- Readable and Intuitive Syntax
- Minimalistic Code

2. **Interpreted Language**

- No Compilation Required
- Cross-platform Execution

3. Dynamic Typing

- No Need to Declare Variable Types
- Flexibility

4. Extensive Standard Library

- Built-in Modules and Functions

5. Versatility

- Multipurpose Language
- Embeddable

6. Object-Oriented and Functional Programming

- Support for Multiple Paradigms
- Inheritance, Encapsulation, and Polymorphism

7. Scalability and Performance

- Manageable for Large Projects

8. Automatic Memory Management

- Garbage Collection: Inbuilt garbage collector in python does all the work of allocating and de-allocating memory by itself.

9. Integration Capabilities

- Easy Integration with Other Languages
- Web Services

10. Wide Range of Applications

- Data Science and Machine Learning
- Web development
- Automation and scripting

2. Describe the role of predefined keywords in Python and provide examples of how they are used in a program

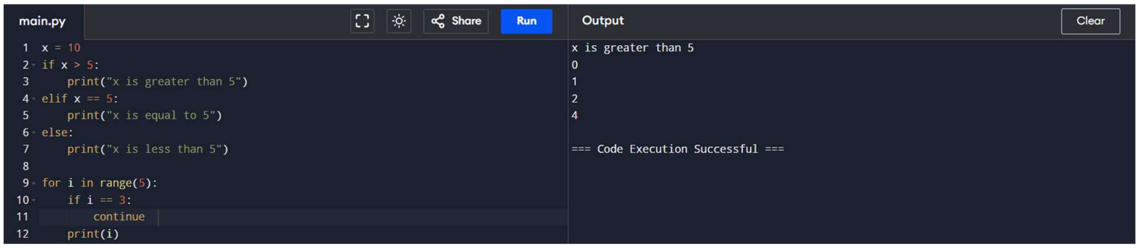
In Python, predefined keywords, or reserved words, are special words that hold specific, non-overridable meanings within the language. Python uses these keywords to define the core structure and flow of a program. Each keyword has a unique purpose and cannot be redefined or used as an identifier (like a variable or function name), as this would interfere with the language's syntax and expected functionality. Let us explore each category of keywords in Python in detail, with examples of how they are used in programs.

1. Control Flow Keywords

Control flow keywords are used to control the execution path of the code based on conditions or repetition.

- **if, elif, else:** These keywords handle conditional statements, allowing you to execute different blocks of code based on specific conditions.
- **for, while:** These keywords handle loops, enabling you to repeat a block of code multiple times.
- **break, continue:** These keywords are used within loops to control their flow. break exits the loop early, while continue skips the current iteration and moves to the next one.

Example:



```python
x = 10
if x > 5:
    print("x is greater than 5")
elif x == 5:
    print("x is equal to 5")
else:
    print("x is less than 5")

for i in range(5):
    if i == 3:
        continue
    print(i)
```

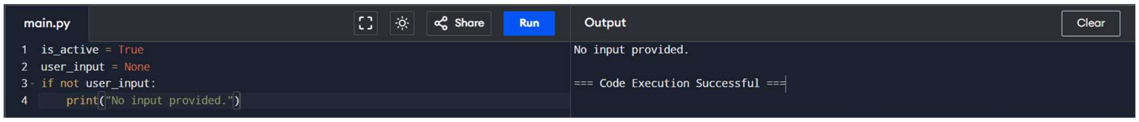
```
x is greater than 5
0
1
2
4

=== Code Execution Successful ===
```

2. Data Type Keywords

Data type keywords are used to represent certain data types in Python.

- **True, False:** Represent Boolean values, often used in conditions and comparisons.
- **None:** Represents a null or empty value, often used to indicate the absence of a value or to initialize a variable that will later hold another type of data.

Example:



```python
is_active = True
user_input = None
if not user_input:
    print("No input provided.")
```

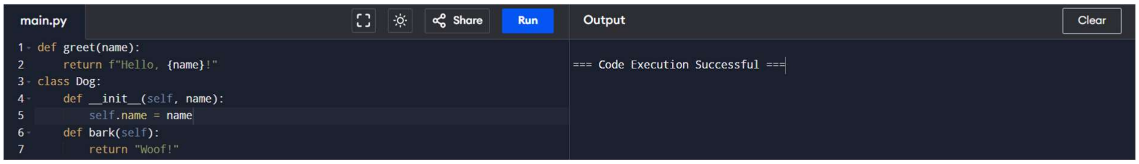
```
No input provided.

=== Code Execution Successful ===
```

3. Function and Class Definition Keywords

These keywords are used to define reusable blocks of code (functions) or object-oriented structures (classes).

- **def**: Used to define a function, which is a named block of reusable code.
- **class**: Used to define a class, which serves as a blueprint for creating objects.
- **return**: Used within a function to send a result back to the caller.

Example:



```python
def greet(name):
    return f"Hello, {name}!"
class Dog:
    def __init__(self, name):
        self.name = name
    def bark(self):
        return "Woof!"
```

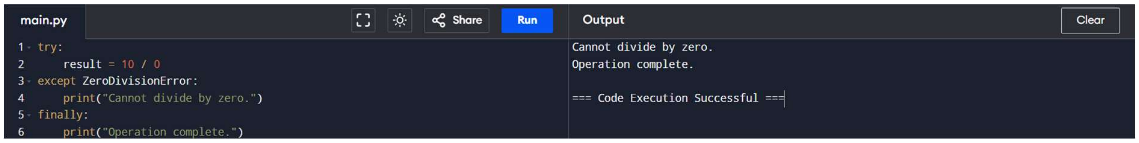
```
=== Code Execution Successful ===
```

4. Exception Handling Keywords

Exception handling keywords manage errors and unexpected behaviour, allowing the program to continue running smoothly even if issues arise.

- **try, except, finally**: try wraps a block of code that may produce an error. If an error occurs, except handles it, while finally executes code regardless of whether an error occurred.
- **raise**: Explicitly raises an exception, often used to enforce constraints or signal errors.

Example:



```python
try:
    result = 10 / 0
except ZeroDivisionError:
    print("Cannot divide by zero.")
finally:
    print("Operation complete.")
```

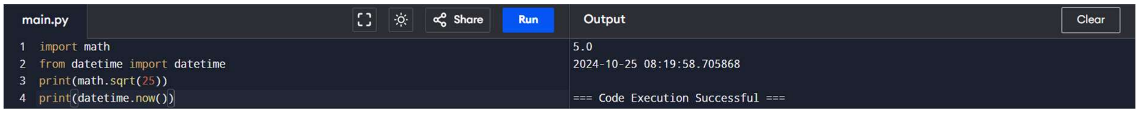
```
Cannot divide by zero.
Operation complete.

=== Code Execution Successful ===
```

5. Import Keywords

Import keywords allow you to bring in external modules and libraries, enabling you to use pre-built functions and classes.

- **import**: Imports an entire module or a specific function from it.
- **from**: Specifies the module to import functions from.

Example:



```python
import math
from datetime import datetime
print(math.sqrt(25))
print(datetime.now())
```

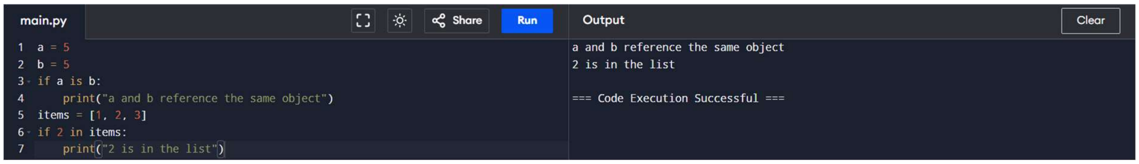
```
5.0
2024-10-25 08:19:58.705868

=== Code Execution Successful ===
```

6. Logical Operators and Comparison Keywords

These keywords are used for performing logical and comparison operations, commonly used within conditions.

- **and, or, not**: Logical operators that combine Boolean values or expressions.
- **is**: Checks if two variables point to the same object.
- **in, not in**: Used to check membership in data structures like lists, strings, and dictionaries.

Example:



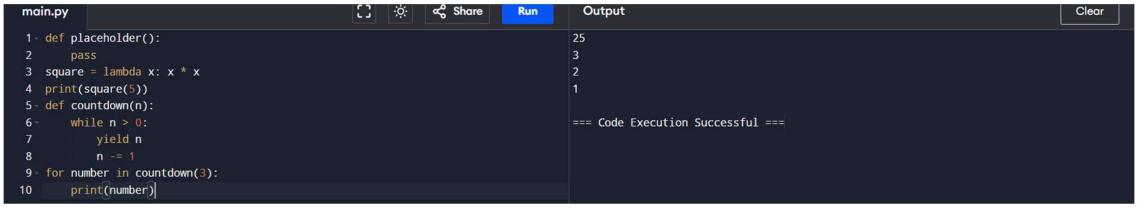
```python
a = 5
b = 5
if a is b:
    print("a and b reference the same object")
items = [1, 2, 3]
if 2 in items:
    print("2 is in the list")
```

```
a and b reference the same object
2 is in the list

=== Code Execution Successful ===
```

7. Other Keywords

- **pass**: Used as a placeholder, allowing you to define empty loops, functions, or classes. pass is useful when stubbing out code that will be implemented later.
- **lambda**: Creates anonymous functions, often used for simple, one-time-use functions.
- **yield**: Used in functions to return a generator object, which can be iterated over but retains its state between iterations.

Example:



```python
def placeholder():
    pass
square = lambda x: x * x
print(square(5))
def countdown(n):
    while n > 0:
        yield n
        n -= 1
for number in countdown(3):
    print(number)
```

```
25
3
2
1

=== Code Execution Successful ===
```

3. Compare and contrast mutable and immutable objects in Python with examples

In Python, objects are classified as *mutable* or *immutable* based on whether their state (content) can be modified after they are created. This distinction has important implications for memory management, data integrity, and how objects behave when used in various data structures. Here’s an in-depth look at mutable and immutable objects, with examples to illustrate the differences.

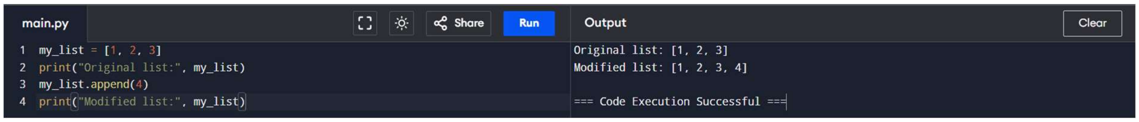
Mutable Objects

Mutable objects are those whose contents or state can be changed after they are created. When you modify a mutable object, you are directly altering its contents without creating a new object. This flexibility is beneficial but requires caution when these objects are used in multiple places, as changes to one reference affect all references to that object.

Examples of Mutable Objects:

- Lists
- Dictionaries
- Sets

Example:



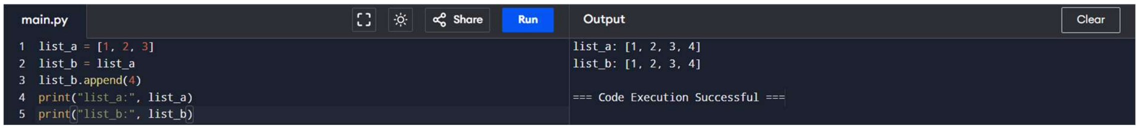
```python
my_list = [1, 2, 3]
print("Original list:", my_list)
my_list.append(4)
print("Modified list:", my_list)
```

```
Original list: [1, 2, 3]
Modified list: [1, 2, 3, 4]

=== Code Execution Successful ===
```

Key Characteristics of Mutable Objects:

- **Modifiable**: They can be modified in place (e.g., items can be added, removed, or changed).
- **Shared references**: If a mutable object is assigned to another variable, both variables reference the same object. Thus, changes through one reference are reflected in all references.

Example of Shared Reference in Mutable Objects:



```python
list_a = [1, 2, 3]
list_b = list_a
list_b.append(4)
print("list_a:", list_a)
print("list_b:", list_b)
```

```
list_a: [1, 2, 3, 4]
list_b: [1, 2, 3, 4]

=== Code Execution Successful ===
```

Immutable Objects

Immutable objects cannot be changed once created. Any operation that tries to modify an immutable object creates a new object instead of altering the original. This immutability is beneficial when you want to ensure data integrity, as it protects against accidental changes.

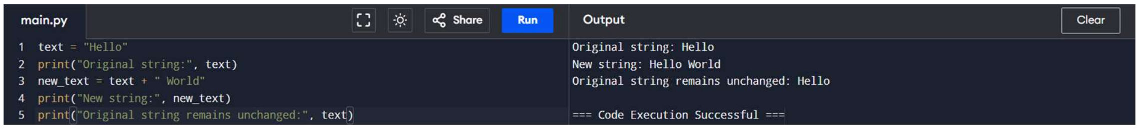
Examples of Immutable Objects:

- Integers
- Floats
- Strings
- Tuples
- Booleans

Key Characteristics of Immutable Objects:

- **Non-modifiable**: Once created, they cannot be changed. Any modification attempts produce a new object.
- **Distinct copies**: When assigned to another variable or used in operations, immutable objects result in separate objects rather than shared references.

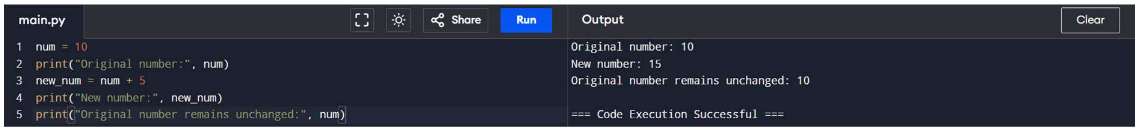
Example with Strings:



```python
text = "Hello"
print("Original string:", text)
new_text = text + " World"
print("New string:", new_text)
print("Original string remains unchanged:", text)
```

```
Original string: Hello
New string: Hello World
Original string remains unchanged: Hello

=== Code Execution Successful ===
```

Example of Immutability with Integers:



```python
num = 10
print("Original number:", num)
new_num = num + 5
print("New number:", new_num)
print("Original number remains unchanged:", num)
```

```
Original number: 10
New number: 15
Original number remains unchanged: 10

=== Code Execution Successful ===
```

Comparing Mutable and Immutable Objects

Feature	Mutable Objects	Immutable Objects
Modifiability	Can be changed in place	Cannot be changed once created
Examples	List, dictionary, set	String, tuple, integer, float
Reference Behavior	Shared references reflect changes	Each change creates a new object
Use Cases	Ideal for data that changes frequently (e.g., lists of records)	Ideal for constants, keys in dictionaries, and hashable objects

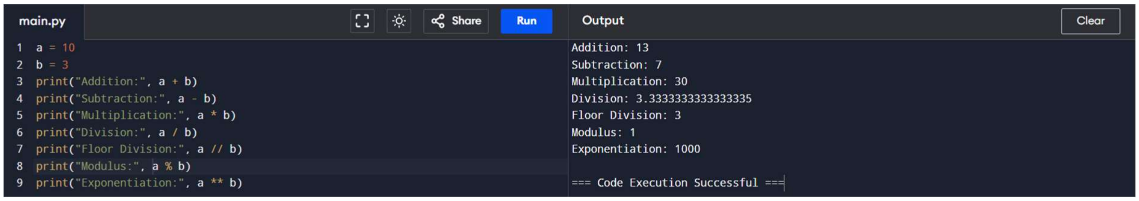
4. Discuss the different types of operators in Python and provide examples of how they are used

1. Arithmetic Operators

Arithmetic operators perform mathematical operations such as addition, subtraction, multiplication, etc.

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
//	Floor Division
%	Modulus (remainder)
**	Exponentiation

Example:



```python
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)
```

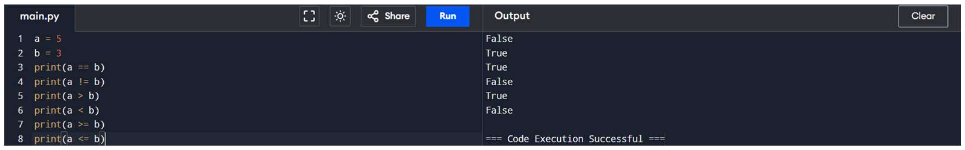
```
Addition: 13
Subtraction: 7
Multiplication: 30
Division: 3.3333333333333335
Floor Division: 3
Modulus: 1
Exponentiation: 1000

=== Code Execution Successful ===
```

2. Comparison Operators

Comparison operators compare values and return a Boolean (True or False). They are often used in conditional statements.

Operator	Description	Example
==	Equal to	a == b
!=	Not equal to	a != b
>	Greater than	a > b
<	Less than	a < b
>=	Greater than or equal	a >= b
<=	Less than or equal	a <= b

Example:



```python
a = 5
b = 3
print(a == b)
print(a != b)
print(a > b)
print(a < b)
print(a >= b)
print(a <= b)
```

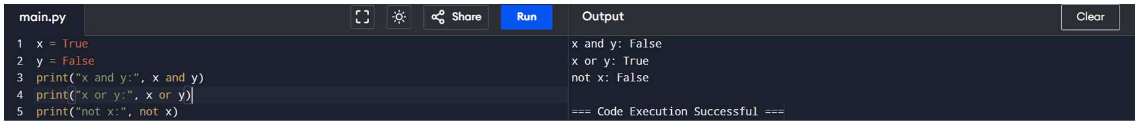
```
False
True
True
False
True
False

=== Code Execution Successful ===
```

3. Logical Operators

Logical operators are used to combine multiple conditions. They return a Boolean value (True or False).

Operator	Description	Example
and	True if both are True	a and b
or	True if at least one is True	a or b
not	True if the operand is False	not a

Example:



```python
x = True
y = False
print("x and y:", x and y)
print("x or y:", x or y)
print("not x:", not x)
```

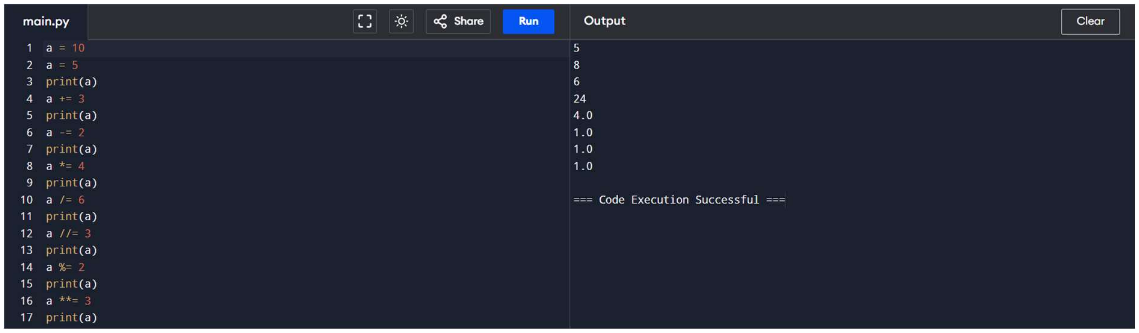
```
x and y: False
x or y: True
not x: False

=== Code Execution Successful ===
```

4. Assignment Operators

Assignment operators are used to assign values to variables. Some also perform a mathematical operation along with assignment.

Operator	Description
=	Simple assignment
+=	Addition assignment
-=	Subtraction assignment
*=	Multiplication assignment
/=	Division assignment
//=	Floor Division assignment
%=	Modulus assignment
**=	Exponentiation assignment

Example:



```python
a = 10
a = 5
print(a)
a += 3
print(a)
a -= 2
print(a)
a *= 4
print(a)
a /= 6
print(a)
a //= 3
print(a)
a %= 2
print(a)
a **= 3
print(a)
```

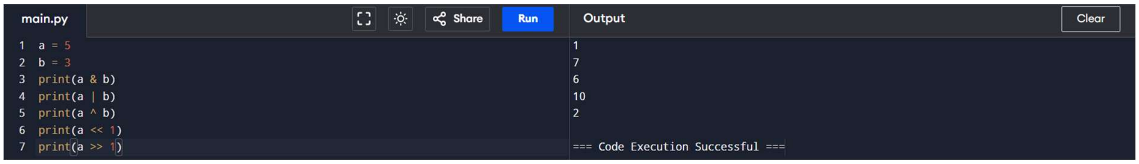
```
5
8
6
24
4.0
1.0
1.0
1.0

=== Code Execution Successful ===
```

5. Bitwise Operators

Bitwise operators perform operations on binary representations of integers.

Operator	**Description**
&	**Bitwise AND**
`	`
^	**Bitwise XOR**
~	**Bitwise NOT (unary)**
<<	**Left Shift**
>>	**Right Shift**

Example:



```python
a = 5
b = 3
print(a & b)
print(a | b)
print(a ^ b)
print(a << 1)
print(a >> 1)
```

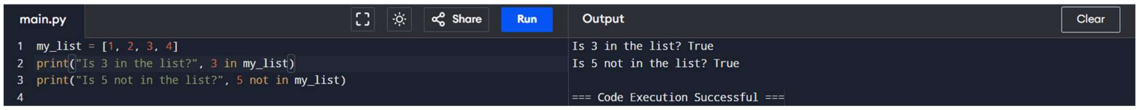
```
1
7
6
10
2

=== Code Execution Successful ===
```

6. Membership Operators

Membership operators check for membership within data structures like strings, lists, or tuples.

Operator	**Description**	**Example**
in	**True if element is present**	**"a" in "abc"**
not in	**True if element is not present**	**"x" not in "abc"**

Example:



```python
my_list = [1, 2, 3, 4]
print("Is 3 in the list?", 3 in my_list)
print("Is 5 not in the list?", 5 not in my_list)

```

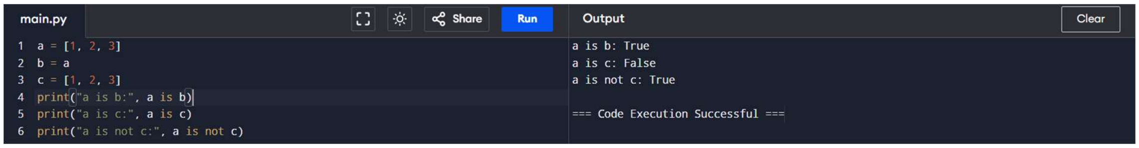
```
Is 3 in the list? True
Is 5 not in the list? True

=== Code Execution Successful ===
```

7. Identity Operators

Identity operators check if two variables point to the same object in memory.

Operator	**Description**	**Example**
is	**True if same object**	**a is b**
is not	**True if not the same object**	**a is not b**

Example:



```python
a = [1, 2, 3]
b = a
c = [1, 2, 3]
print("a is b:", a is b)
print("a is c:", a is c)
print("a is not c:", a is not c)
```

```
a is b: True
a is c: False
a is not c: True

=== Code Execution Successful ===
```

5. Explain the concept of type casting in Python with examples

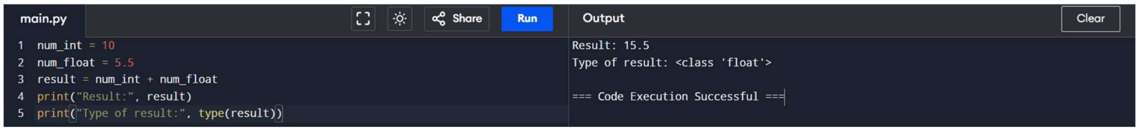
***Type casting* (or *type conversion*)** is the process of converting one data type into another. This is often necessary when working with different types of data, such as converting a string representation of a number to an integer to perform mathematical operations. Python provides two types of type casting:

1. **Implicit Type Casting**: Done automatically by Python, without any manual intervention.
2. **Explicit Type Casting**: Requires the programmer to specify the desired data type.

1. Implicit Type Casting

In implicit type casting, Python automatically converts a smaller data type (e.g., int) to a larger data type (e.g., float) when performing operations that involve both. This is also known as *type promotion* and ensures that data is not lost in operations.

Example:



```
num_int = 10
num_float = 5.5
result = num_int + num_float
print("Result:", result)
print("Type of result:", type(result))
```

```
Result: 15.5
Type of result: <class 'float'>

=== Code Execution Successful ===
```

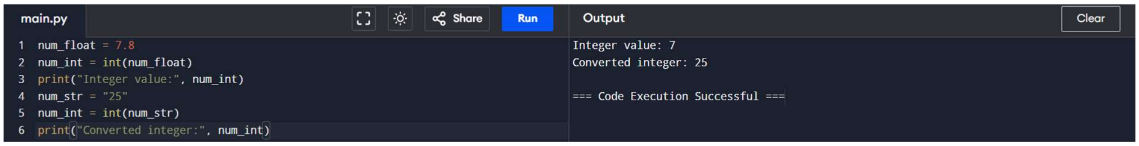
2. Explicit Type Casting

In explicit type casting, also known as *type conversion*, the programmer manually converts an object from one type to another. This is done using Python's built-in functions: int(), float(), str(), list(), tuple(), set(), etc.

a. Integer Casting

To convert a value to an integer, use int (). It truncates decimal points when applied to floats and raises an error if applied to a non-numeric string.

Example:



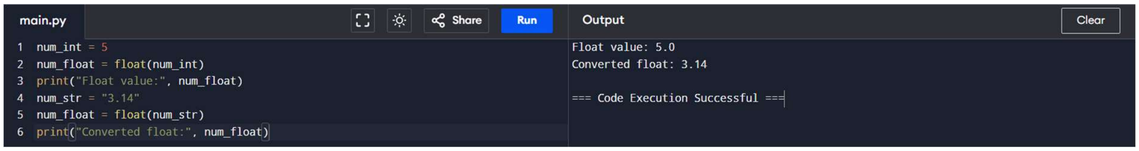
```
num_float = 7.8
num_int = int(num_float)
print("Integer value:", num_int)
num_str = "25"
num_int = int(num_str)
print("Converted integer:", num_int)
```

```
Integer value: 7
Converted integer: 25

=== Code Execution Successful ===
```

b. Float Casting

To convert a value to a float, use float (). It can handle integers, strings that represent numeric values, and Boolean.

Example:



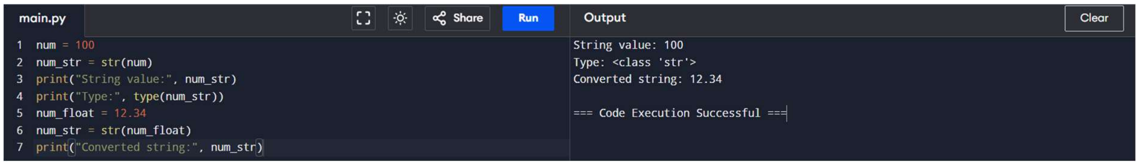
```
num_int = 5
num_float = float(num_int)
print("Float value:", num_float)
num_str = "3.14"
num_float = float(num_str)
print("Converted float:", num_float)
```

```
Float value: 5.0
Converted float: 3.14

=== Code Execution Successful ===
```

c. String Casting

To convert a value to a string, use str (). This is useful for converting numbers to strings for concatenation or formatting**.**

Example:



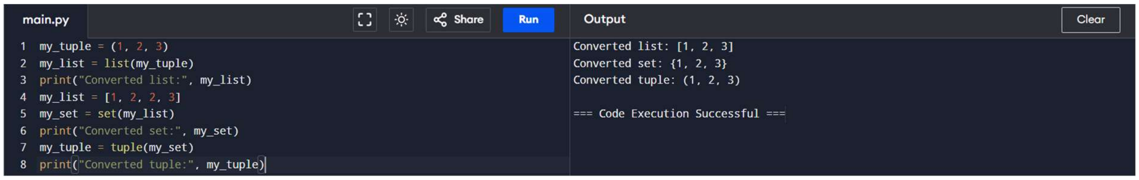
```python
num = 100
num_str = str(num)
print("String value:", num_str)
print("Type:", type(num_str))
num_float = 12.34
num_str = str(num_float)
print("Converted string:", num_str)
```

```
String value: 100
Type: <class 'str'>
Converted string: 12.34

=== Code Execution Successful ===
```

d. List, Tuple, and Set Casting

Lists, tuples, and sets can also be converted to each other, which can be helpful when working with data structures and performing specific operations (e.g., converting a list to a set to remove duplicates).

Example:



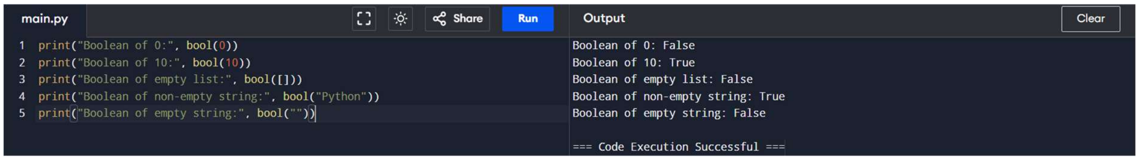
```python
my_tuple = (1, 2, 3)
my_list = list(my_tuple)
print("Converted list:", my_list)
my_list = [1, 2, 2, 3]
my_set = set(my_list)
print("Converted set:", my_set)
my_tuple = tuple(my_set)
print("Converted tuple:", my_tuple)
```

```
Converted list: [1, 2, 3]
Converted set: {1, 2, 3}
Converted tuple: (1, 2, 3)

=== Code Execution Successful ===
```

e. Boolean Casting

Values can also be converted to booleans using bool(). Python treats several values as False by default, including 0, None, empty collections (like [], (), {}, ""), and the special constant False itself. All other values are treated as True.

Example:



```python
print("Boolean of 0:", bool(0))
print("Boolean of 10:", bool(10))
print("Boolean of empty list:", bool([]))
print("Boolean of non-empty string:", bool("Python"))
print("Boolean of empty string:", bool(""))
```

```
Boolean of 0: False
Boolean of 10: True
Boolean of empty list: False
Boolean of non-empty string: True
Boolean of empty string: False

=== Code Execution Successful ===
```

6. How do conditional statements work in Python? Illustrate with examples

conditional statements are used to execute code blocks based on specific conditions. They allow a program to perform different actions based on whether a condition evaluates to True or False. Python's primary conditional statements are:

1. if statement
2. if-else statement
3. if-elif-else statement

Each of these statements uses an expression that evaluates to a Boolean value (True or False)

1. if Statement

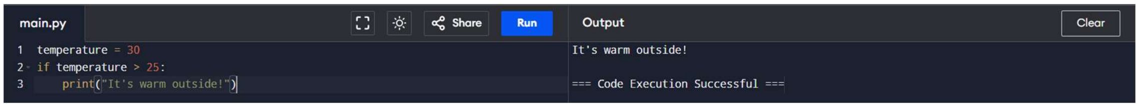
The if statement is the simplest conditional statement. It checks whether a given condition is True, and if so, executes the associated code block.

Syntax:

if condition:

  # code to execute if condition is True

Example:



```
temperature = 30
if temperature > 25:
    print("It's warm outside!")
```

```
It's warm outside!

=== Code Execution Successful ===
```

2. if-else Statement

The if-else statement allows for two possible paths: one if the condition is True and another if it's False.

Syntax:

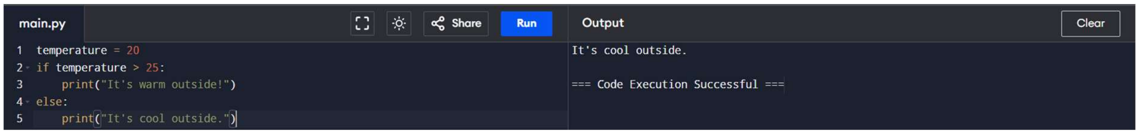
if condition:

  # code to execute if condition is True

else:

  # code to execute if condition is False

Example:



```
temperature = 20
if temperature > 25:
    print("It's warm outside!")
else:
    print("It's cool outside.")
```

```
It's cool outside.

=== Code Execution Successful ===
```

3. if-elif-else Statement

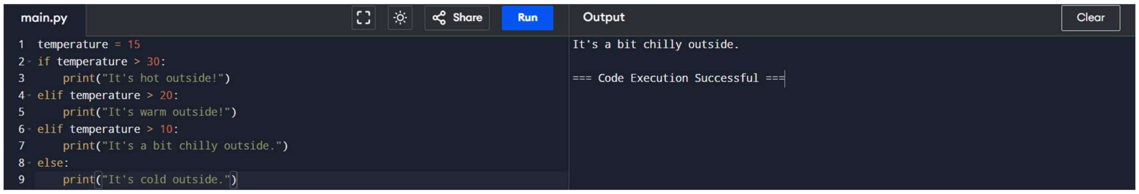
The if-elif-else statement is useful when multiple conditions need to be checked in sequence. Once a condition evaluates to True, the associated code block executes, and the rest of the conditions are ignored.

Syntax:

if condition1:

```
    # code to execute if condition1 is True
elif condition2:
    # code to execute if condition2 is True
elif condition3:
    # code to execute if condition3 is True
else:
    # code to execute if all above conditions are False
```

Example:



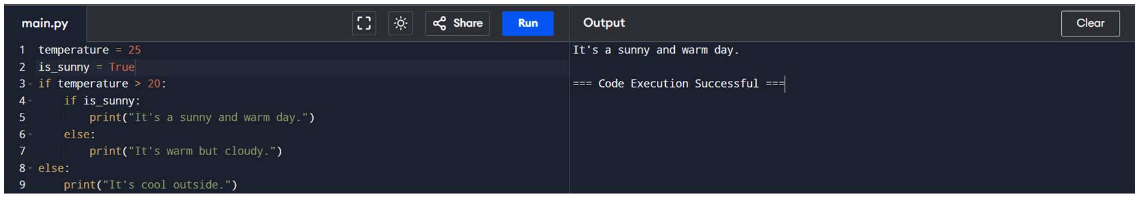
```python
temperature = 15
if temperature > 30:
    print("It's hot outside!")
elif temperature > 20:
    print("It's warm outside!")
elif temperature > 10:
    print("It's a bit chilly outside.")
else:
    print("It's cold outside.")
```

```
It's a bit chilly outside.

=== Code Execution Successful ===
```

4. Nested if Statements

Nested if statements occur when an if statement is placed inside another if, elif, or else block. This structure is used for more complex conditions.

Example:



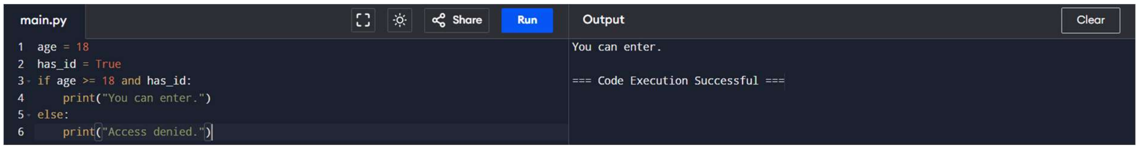
```python
temperature = 25
is_sunny = True
if temperature > 20:
    if is_sunny:
        print("It's a sunny and warm day.")
    else:
        print("It's warm but cloudy.")
else:
    print("It's cool outside.")
```

```
It's a sunny and warm day.

=== Code Execution Successful ===
```

5. Using Logical Operators with Conditional Statements

Python supports the logical operators and, or, and not to combine multiple conditions in a single statement.

Example with and and or:



```python
age = 18
has_id = True
if age >= 18 and has_id:
    print("You can enter.")
else:
    print("Access denied.")
```

```
You can enter.

=== Code Execution Successful ===
```

7. Describe the different types of loops in Python and their use cases with examples.

Loops are used to execute a block of code repeatedly based on certain conditions. The primary types of loops in Python are:

1. for loop
2. while loop
3. Nested loops

1. for Loop

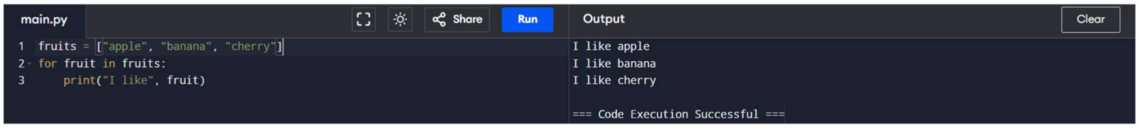
The for loop is used to iterate over a sequence (like a list, tuple, dictionary, set, or string) or any iterable object. It allows you to execute a block of code a specific number of times.

Syntax:

for variable in iterable:

    # code to execute

Example:



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

```
I like apple
I like banana
I like cherry

=== Code Execution Successful ===
```

2. while Loop

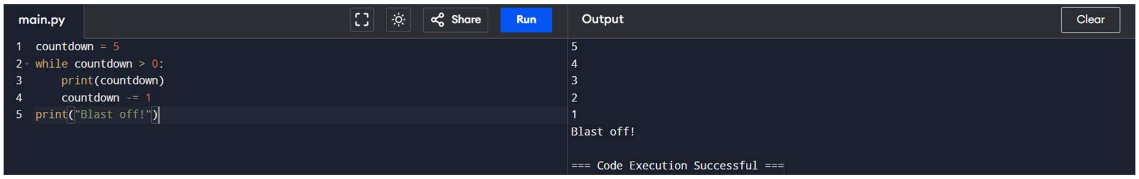
The while loop continues to execute as long as a specified condition is True. It is useful when the number of iterations is not known beforehand and depends on dynamic conditions.

Syntax:

while condition:

    # code to execute

Example:



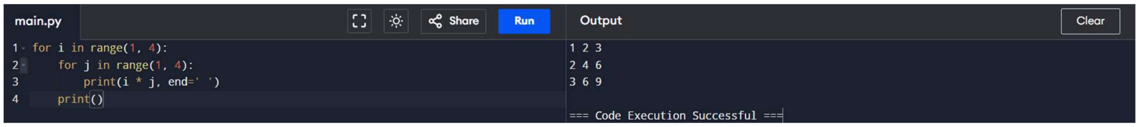
```
countdown = 5
while countdown > 0:
    print(countdown)
    countdown -= 1
print("Blast off!")
```

```
5
4
3
2
1
Blast off!

=== Code Execution Successful ===
```

3. Nested Loops

Nested loops are loops placed inside another loop. This is useful when you need to perform a repetitive action within another repetitive action.

Example:



```python
for i in range(1, 4):
    for j in range(1, 4):
        print(i * j, end=' ')
    print()
```

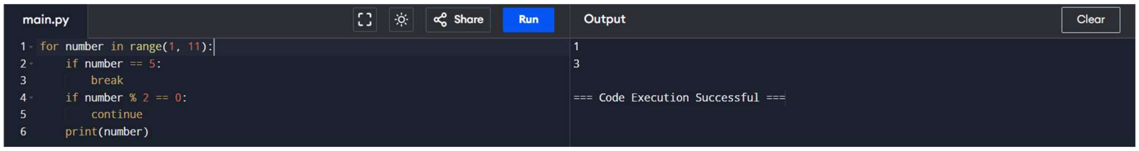
```
1 2 3
2 4 6
3 6 9

=== Code Execution Successful ===
```

4. Loop Control Statements

Python also provides control statements to modify the behavior of loops:

- break: Exits the loop immediately, regardless of the loop condition.
- continue: Skips the current iteration and moves to the next iteration of the loop.
- pass: A null statement that is syntactically required but does nothing. It's often used as a placeholder.

Example of break and continue:



```python
for number in range(1, 11):
    if number == 5:
        break
    if number % 2 == 0:
        continue
    print(number)
```

```
1
3

=== Code Execution Successful ===
```