Python Programming (MR25)
UNIT-1
1. Introduction

Contents:

Introduction to Python: Features, Applications, Installation, IDEs, Python Syntax, Indentation, Comments, Variables, Data Types, Type Casting, Operators: Arithmetic, Relational, Logical, Assignment, Membership, Identity, Bitwise, Input/Output functions (input(), print()), Control Structures: if, if-else, if-elif-else, Nested Conditions, Looping: for, while, Nested Loops, break, continue, pass.

# **Introduction to Python**

### **1. Features of Python**

a) Simple & Easy to Learn – Syntax is close to English.
b) Interpreted Language – No need for compilation, executed line by line.
c) Cross-platform – Works on Windows, Mac, Linux.
d) Dynamic Typing – No need to declare variable types.
e) Extensive Libraries – Rich standard library (math, os, datetime, etc.) and third-party packages (NumPy, Pandas, TensorFlow).
f) Portable & Extensible – Can integrate with C, C++, Java.
g) High Productivity – Less code, faster development.
h) Free and Open Source
i) Object-Oriented language
j) Databases Connectivity

### **2. Applications of Python**

* **Web Development** – Django, Flask, FastAPI.
* **Data Science & Machine Learning** – Pandas, NumPy, Scikit-learn, TensorFlow, PyTorch.
* **Artificial Intelligence** – Natural Language Processing, Computer Vision.
* **Automation & Scripting** – Writing small scripts to automate tasks.
* **Game Development** – Pygame.
* **Desktop Applications** – Tkinter, PyQt.
* **Networking & Cybersecurity** – Socket programming, penetration testing tools.
* **IoT & Embedded Systems** – MicroPython, Raspberry Pi.

### **3. Installation of Python**

* **Windows**:

1. Download from [python.org](https://www.python.org/downloads/).
2. Run installer → check “Add Python to PATH” → Install.

* **Linux / Mac**:

  * Python is often pre-installed.
  * Update via package manager:

    ```bash
    sudo apt install python3   # Ubuntu/Debian
    brew install python        # macOS (with Homebrew)
    ```

### **4. Python IDEs (Integrated Development Environments)**

* **IDLE** (comes with Python by default).
* **PyCharm** (popular for large projects).
* **VS Code** (lightweight, widely used).
* **Jupyter Notebook** (best for Data Science, ML).
* **Spyder** (scientific computing).
* **Thonny** (beginner-friendly).

### **5. Python Syntax Basics**

* **Hello World Example**

  ```python
  print("Hello, World!")
  ```
* **Indentation** – Python uses spaces instead of `{}` or `;`

  ```python
  if 5 > 2:
      print("Five is greater than two")
  ```
* **Variables** – No type declaration

  ```python
  x = 10
  y = "Python"
  ```
* **Comments**

  ```python
  # This is a single-line comment
  """
  This is a
  multi-line comment
  """
  ```
* **Data Types** – int, float, str, list, tuple, dict, set.

6. Comments:

Comments in Python are used to explain the code and make it more readable for humans. They are ignored by the Python interpreter and do not affect the execution of the program. Python supports two types of comments:

1. **Single-line comments**: Single-line comments start with a hash `#` character and continue until the end of the line. They are typically used for short comments on a single line.

   ```python
   # This is a single-line comment
   print("Hello, world!")  # This is another single-line comment
   ```

2. **Multi-line comments**: Python does not have a built-in syntax for multi-line comments, but you can use triple quotes (`'''` or `"""`) to create multi-line strings, which are often used as multi-line comments.

   ```python
   """
   This is a multi-line comment.
   It spans multiple lines and is enclosed in triple quotes.
   """
   print("Hello, world!")
   ```

   Alternatively, you can use multiple single-line comments to achieve a similar effect:

   ```python
   # This is a multi-line comment
   # It spans multiple lines
   print("Hello, world!")
   ```

It is recommended to use comments to document your code, explain complex parts, or provide context for future readers. However, excessive comments can make the code harder to read, so it's important to strike a balance and use comments judiciously.

7. Variables:

In Python, variables are used to store data values. Variables are created when they are first assigned a value. You can assign a value to a variable using the assignment operator (`=`). Variable names can contain letters, digits, and underscores, but they cannot start with a digit. Here are some examples of variable assignments:

Rules for Naming Variables in Python:

1. **Must start with a letter or underscore (_)**
   - Valid: `name`, `_value`
   - Invalid: `1name`, `@value`
2. **Can only contain letters, digits, and underscores**
   - Valid: `student_name`, `marks123`
   - Invalid: `student-name`, `marks#1`
3. **Cannot use Python keywords** (like `if`, `for`, `while`, `class`, `def`, etc.)
   - Valid: `total_marks`
   - Invalid: `for = 10`
4. **Case-sensitive**
   - `Age`, `age`, and `AGE` are three different variables.
5. **No special characters allowed** (`@`, `#`, `$`, `%`, etc. are not allowed).
6. **Variable names should be meaningful** (recommended for readability).
   - Good: `student_age`
   - Bad: `x`, `abc` (unless used temporarily)

```python
x = 10  # x is a variable with the value 10
name = "Alice"  # name is a variable with the value "Alice"
is_valid = True  # is_valid is a variable with the value True
```

Python is dynamically typed, which means **you do not need to explicitly declare the type of a variable.** The type of a variable is determined at runtime based on the value assigned to it. You can use the `type()` function to check the type of a variable:

```python
x = 10
print(type(x))  # Output: <class 'int'>

name = "Alice"
print(type(name))  # Output: <class 'str'>

is_valid = True
print(type(is_valid))  # Output: <class 'bool'>
```

Variables can be reassigned to new values, even if they had a different type before:

```python
x = 10
print(x)  # Output: 10

x = "Hello"
print(x)  # Output: Hello
```

However, reassigning a variable does not change its type; it simply changes the value that the variable refers to.

8. Numeric Data Types:

Numeric data types in Python are used to represent numeric values. Python supports several numeric data types, including integers, floating-point numbers, and complex numbers.

1. Integers (`int`): Integers are whole numbers without any decimal point. They can be positive, negative, or zero. In Python, integers have unlimited precision.

2. **Floating-Point Numbers (`float`)**: Floating-point numbers are numbers with a decimal point or in exponential form (e.g., 2.5, -3.14, 1e-5). Floating-point numbers in Python are implemented using the `float` data type, which is based on the IEEE 754 standard.

3. Complex Numbers (`complex`): Complex numbers are numbers with a real and imaginary part. They are represented by `a + bj`, where `a` is the real part, `b` is the imaginary part, and `j` is the imaginary unit (sqrt(-1)).

Python also provides functions and operators for working with numeric data types. For example, you can use arithmetic operators (`+`, `-`, `*`, `/`, `//`, `%`, `**`) for mathematical operations, and functions like `abs()`, `round()`, `int()`, `float()`, and `complex()` for conversions and other operations.

9. Type Casting in Python:

Type casting (or type conversion) means changing one data type into another.
In Python, there are two main types:

- **Implicit Type Casting** (Type Conversion) → Done automatically by Python.
- **Explicit Type Casting** (Type Casting) → Done manually by the programmer using built-in functions.

a) Implicit Type Casting

- Python **automatically converts** smaller data types into larger compatible types to avoid data loss.

Example:

```
x = 10     # int
y = 3.5    # float
z = x + y  # int + float → float
print(z)   # 13.5
print(type(z))  # <class 'float'>
```

b) Explicit Type Casting

- Programmer converts the type manually using built-in functions:

1. **int()**: Converts a value to an integer.

   ```python
   x = int(3.14)
   print(x)  # Output: 3
   ```

2. **float()**: Converts a value to a floating-point number.

   ```python
   x = float("3.14")
   print(x)  # Output: 3.14
   ```

3. **str()**: Converts a value to a string.

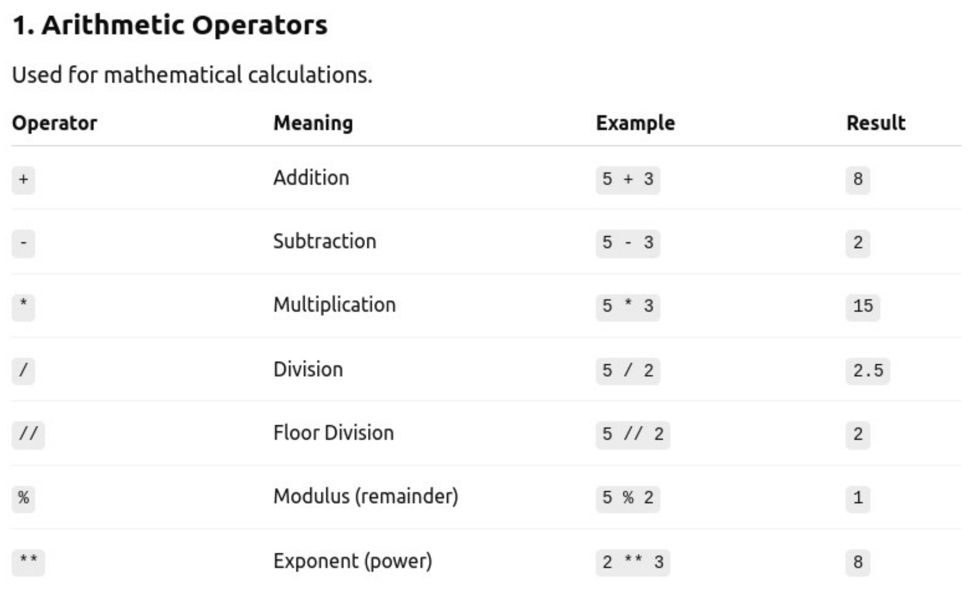
   ```python
   x = str(123)
   print(x)  # Output: "123"
   ```

4. **bool()**: Converts a value to a boolean.

   ```python
   x = bool(0)
   print(x)  # Output: False
   ```

10. Operators:

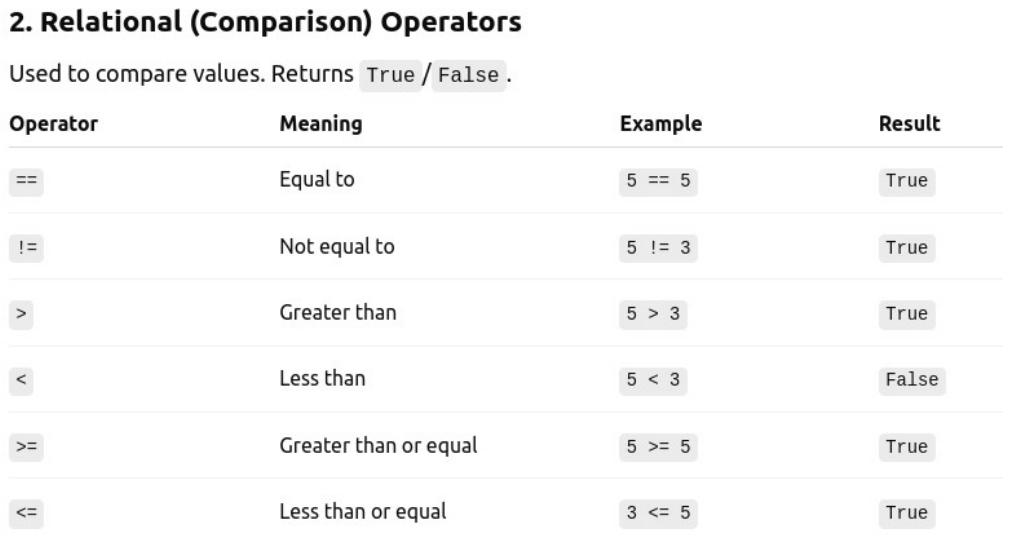
Python Operators are divided into **7 categories**:
Arithmetic, Relational, Logical, Assignment, Membership, Identity, Bitwise.



1. Arithmetic Operators

Used for mathematical calculations.

Operator	Meaning	Example	Result
`+`	Addition	`5 + 3`	`8`
`-`	Subtraction	`5 - 3`	`2`
`*`	Multiplication	`5 * 3`	`15`
`/`	Division	`5 / 2`	`2.5`
`//`	Floor Division	`5 // 2`	`2`
`%`	Modulus (remainder)	`5 % 2`	`1`
`**`	Exponent (power)	`2 ** 3`	`8`



2. Relational (Comparison) Operators

Used to compare values. Returns `True` / `False`.

Operator	Meaning	Example	Result
`==`	Equal to	`5 == 5`	`True`
`!=`	Not equal to	`5 != 3`	`True`
`>`	Greater than	`5 > 3`	`True`
`<`	Less than	`5 < 3`	`False`
`>=`	Greater than or equal	`5 >= 5`	`True`
`<=`	Less than or equal	`3 <= 5`	`True`

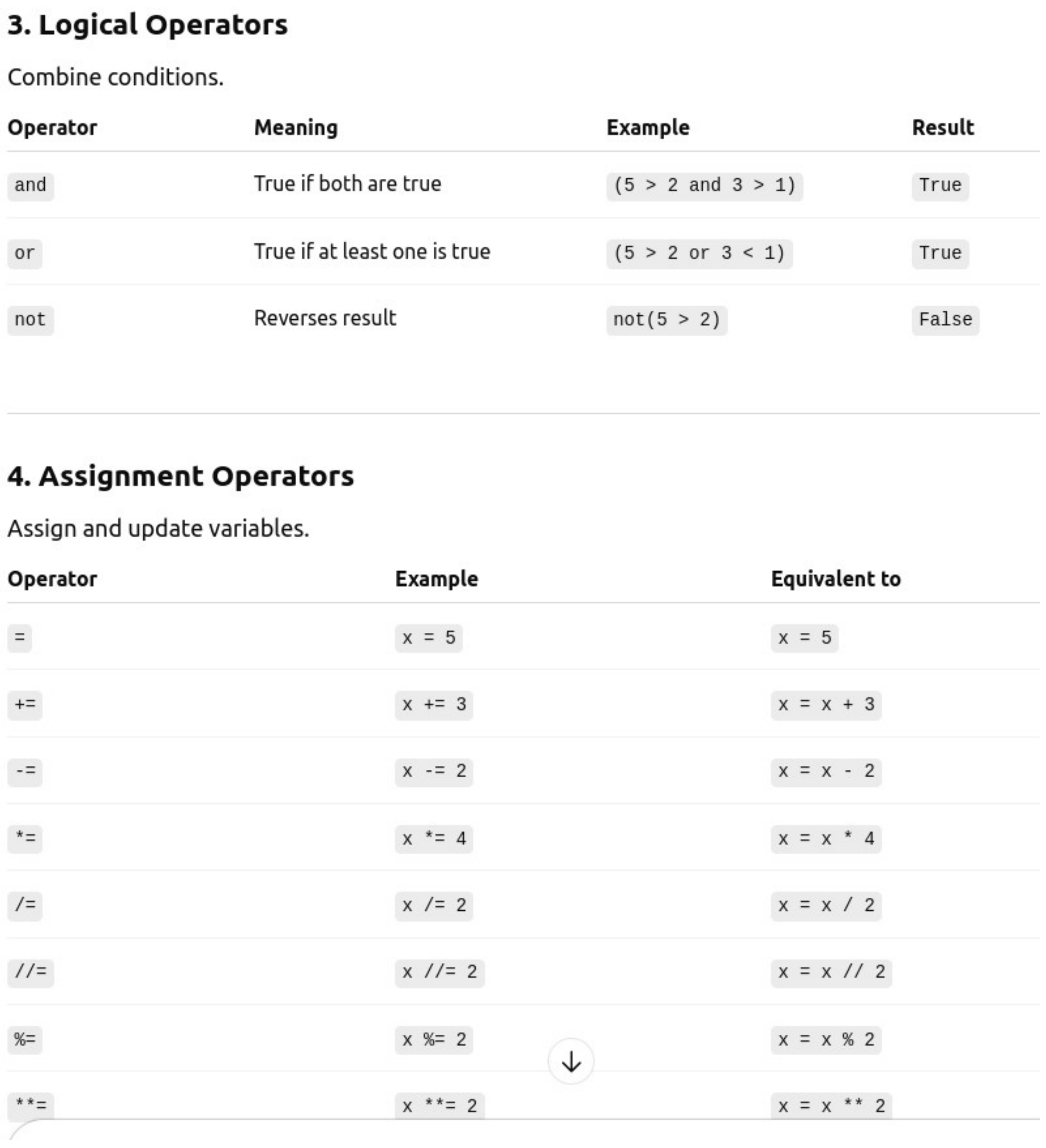
Example of Equality (`==`): Compares if two strings are equal.

```python
string1 = "hello"
string2 = "world"
if string1 == string2:
    print("Strings are equal")
else:
    print("Strings are not equal")
```

Example of Inequality (`!=`): Compares if two strings are not equal.

```python
```

```
if string1 != string2:
    print("Strings are not equal")
else:
     print("Strings are equal")
```



3. Logical Operators

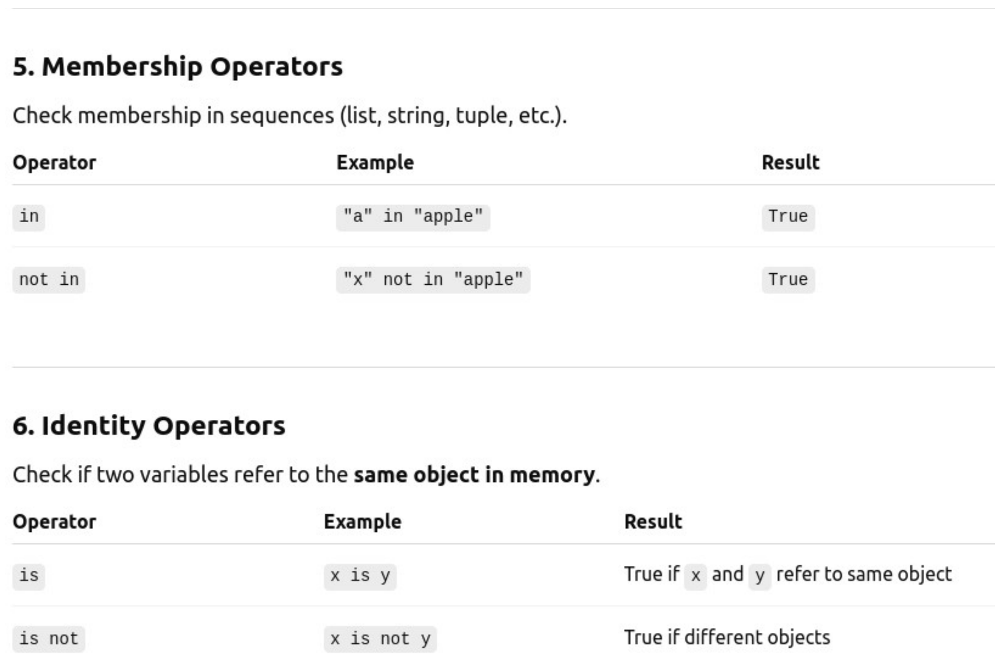
Combine conditions.

Operator	Meaning	Example	Result
`and`	True if both are true	`(5 > 2 and 3 > 1)`	`True`
`or`	True if at least one is true	`(5 > 2 or 3 < 1)`	`True`
`not`	Reverses result	`not(5 > 2)`	`False`

4. Assignment Operators

Assign and update variables.

Operator	Example	Equivalent to
`=`	`x = 5`	`x = 5`
`+=`	`x += 3`	`x = x + 3`
`-=`	`x -= 2`	`x = x - 2`
`*=`	`x *= 4`	`x = x * 4`
`/=`	`x /= 2`	`x = x / 2`
`//=`	`x //= 2`	`x = x // 2`
`%=`	`x %= 2`	`x = x % 2`
`**=`	`x **= 2`	`x = x ** 2`



5. Membership Operators

Check membership in sequences (list, string, tuple, etc.).

Operator	Example	Result
`in`	`"a" in "apple"`	`True`
`not in`	`"x" not in "apple"`	`True`

6. Identity Operators

Check if two variables refer to the **same object in memory**.

Operator	Example	Result
`is`	`x is y`	True if `x` and `y` refer to same object
`is not`	`x is not y`	True if different objects

Example of Membership Operators: Membership operators are used to test if a value is present in a sequence (e.g., lists, tuples, strings).

```python
x = [1, 2, 3, 4, 5]
print(3 in x)  # Output: True
print(6 not in x)  # Output: True
```

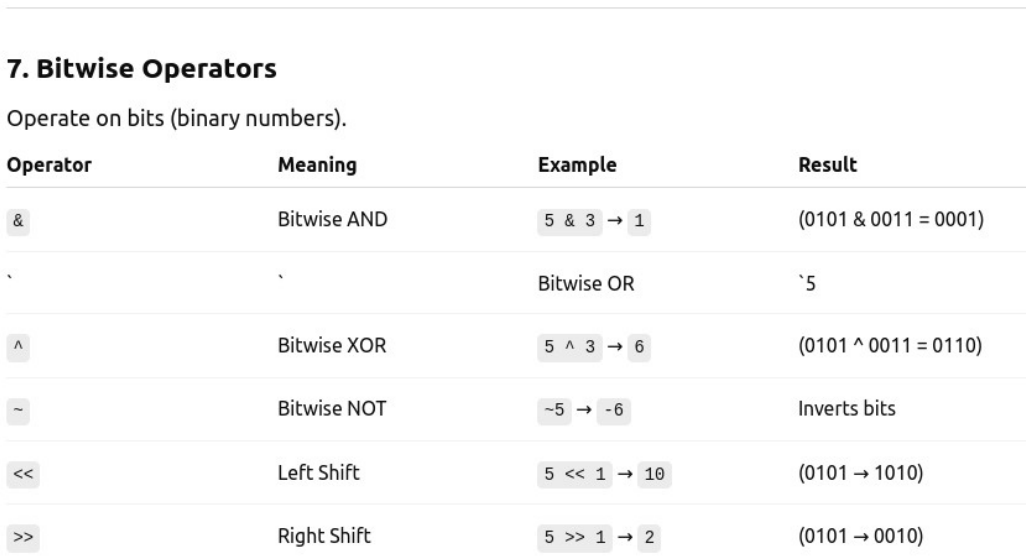
Examples of Identity Operators: Identity operators are used to compare the memory locations of two objects.

```
 # Example 1: Using 'is' and '=='
x = [1, 2, 3]
y = [1, 2, 3]

print(x == y)   # True → contents are the same
print(x is y)   # False → stored at different memory locations

# Example 2: Assign same reference
z = x
print(x is z)   # True → z points to the same object as x

# Example 3: Using 'is not'
a = 10
b = 10
print(a is b)     # True (small integers are cached in Python)
print(a is not b) # False
```



7. Bitwise Operators

Operate on bits (binary numbers).

Operator	Meaning	Example	Result
`&`	Bitwise AND	`5 & 3` → `1`	(0101 & 0011 = 0001)
\`	\`	Bitwise OR	\`5
`^`	Bitwise XOR	`5 ^ 3` → `6`	(0101 ^ 0011 = 0110)
`~`	Bitwise NOT	`~5` → `-6`	Inverts bits
`<<`	Left Shift	`5 << 1` → `10`	(0101 → 1010)
`>>`	Right Shift	`5 >> 1` → `2`	(0101 → 0010)

Example of Bit-wise operators:

```
a = 12  # 1100
b = 6   # 0110

print("a & b =", a & b)   # 0100 → 4
print("a | b =", a | b)   # 1110 → 14
print("a ^ b =", a ^ b)   # 1010 → 10
print("~a =", ~a)         # -(a+1) = -13
print("a << 2 =", a << 2) # 110000 → 48
print("a >> 2 =", a >> 2) # 0011 → 3
```

11. Input/Output functions (input(), print())

a) Reading input from the keyboard:

In Python, you can read input from the keyboard using the `input()` function. The `input()` function takes a single argument, which is a prompt message to display to the user. It then waits for the user to enter a value and press Enter. The value entered by the user is returned as a string. Here's an example:

```python
name = input("Enter your name: ")
print("Hello, " + name + "!")
```

b) Displaying output with the Print function

```
# Printing text
print("Hello, world!")
```

```
# Printing variables
name = "Alice"
age = 30
print("Name:", name)
print("Age:", age)

# Printing expressions
result = 10 + 20
print("Result:", result)

# Printing multiple values
x = 10
y = 20
print("x =", x, "y =", y)

# Printing with formatting
value = 3.14159
print("Value:", format(value, ".2f"))  # Output: "Value: 3.14"
print(f"{value} is a floating point no.")

# Printing multiple values
x = 10
y = 20
print("The value of x is", x, "and the value of y is", y)  # Output: The value of x is 10 and the value of y
is 20

# Using the end argument to change the end character
print("Hello, ", end="")
print("world!")  # Output: Hello, world!
```

Control Structures – Notes from Jupyter notebook

Decision Structures: if, if-else, if-elif-else, Nested Conditions.

Looping: for, while, Nested Loops, break, continue, pass.