# UNIT 1: Python Programming Basics

## A Brief History

- **Creator: Guido van Rossum,** a Dutch programmer**.**
- **Year:** Python development started in 1989 and was officially released in **1991**.
- He was working at a research institute, Centrum Wiskunde & Informatica (CWI), in the Netherlands.
- He was working on **ABC language** (used for teaching.)
- Why the name Python?
  → In the 1970s, there was a popular British comedy show called ***"Monty Python's Flying Circus."***
  → Guido van Rossum was a big fan of the show and chose the name ***Python*** for the language to make it sound unique and a bit mysterious.

## Introduction to Python

**Python is a:**

- **High-level language**:
  It is closer to human language and abstracts away most of the hardware details, making it easier to write and understand.
- **General-purpose language**:
  It can be used to build all kinds of software applications — from web development to data analysis, machine learning, automation, and more.
- **Interpreted language**:
  Python code is executed line-by-line by an interpreter, not compiled all at once like in compiled languages (e.g., C/C++). This makes debugging easier and faster during development.

## Key Features of Python

1. **Simple and Easy to Learn**
   → Clear and readable syntax, similar to English, which makes learning and writing code easy.
2. **Interpreted Language**
   → No need for compilation; code is executed line by line, making debugging easier.
3. **Dynamically Typed**
   → No need to declare variable data types; Python determines the type at runtime.
4. **Free and Open Source**
   → Python is freely available to download, we can use, and modify.
5. **Extensive Libraries**
   → Comes with a rich set of standard libraries and modules for various tasks like math, web, data processing, etc.
6. **Portability**
   → Python programs can run on different operating systems (Windows, macOS, Linux) without changes.
7. **Platform-Independent**
   → "Write once, run anywhere" — Python code doesn't depend on a specific platform.
8. **Versatile Language**
   → Widely used in various fields such as web development, data science, machine learning, artificial intelligence, automation, etc.
9. **Language Integration**
   → Python supports integration with other programming languages for flexibility and performance. Examples include:
   - **Cython**: Allows integration with C/C++ to improve performance.
   - **Jython**: Python implementation that runs on the Java platform, allowing Python code to interact with Java classes.
   - **IronPython**: Python implementation for the .NET framework, allowing integration with C# and other .NET languages.

**Debugging**: process of **checking and correcting** errors in code

**Applications of Python**

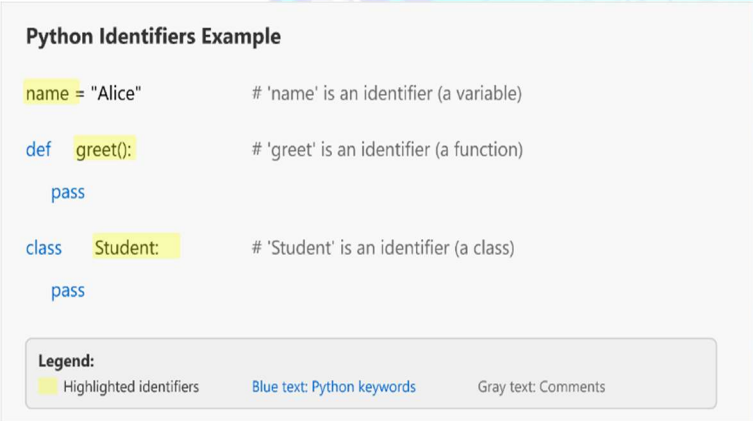
Python is used in a wide range of real-world applications:

1. **Web Development**
   → Backend frameworks like **Django** and **Flask**.
2. **Data Science and Scientific Computing**
   → Used for data analysis, visualization, and scientific research.
3. **Artificial Intelligence (AI) and Machine Learning (ML)**
   → Libraries like TensorFlow, Keras, Scikit-learn, etc.
4. **Game Development**
   → Simple games using libraries like **Pygame**.
5. **Desktop Applications**
   → GUI-based apps using Tkinter, PyQt, etc.
6. **Mobile Applications**
   → Using tools like **Kivy**.
7. **Cybersecurity and Ethical Hacking**
   → Everywhere used for writing scripts and tools in cybersecurity.

**Parts of Python Programming Language**

Python consists of the following key parts:

1. **Identifiers**
   → The Identifiers are the name given to identify variables, functions, classes and Objects, etc.




**Rules for Creating Identifiers in Python**

1. **Can contain:**
   - Letters (A–Z, a–z)
   - Digits (0–9) at end
   - Underscore ( _ )
     → Example: student_1, age23
2. **Cannot start with a digit**
   → Example: 1name (Invalid)
   name1 (Valid)
3. **Cannot use special characters**
   → Example: @, #, $, %, -, +
   → Identifiers like name@1 or user-name are **invalid**
4. **Cannot use Python keywords as identifiers**
   → Example: if, else, class, def…etc→ if = 5 (Invalid)
5. **Case-Sensitive**
   → Identifiers are case-sensitive
   → Name, name, and NAME are **all different**

**Python Case-Sensitive Identifiers**

```
name = "Ram"
Name = "Raj"
NAME = "Sham"
# Print all variables
print ("name =", name)
print ("Name =", Name)
print ("NAME =", NAME)
```

**Output:**

```
name = Ram
Name = Raj
NAME = Sham
```

**Valid and Invalid Identifiers**

| Identifier | Valid/Invalid | Reason | Rule No. |
|---|---|---|---|
| age | Valid | Follows all rules | — |
| Age22 | Valid | Starts with a letter | Rule 2 |
| 1age | Invalid | Starts with a digit | Rule 2 |
| age_22 | Valid | Contains letters, digit, _ | Rule 1 |
| @age | Invalid | Contains special character | Rule 3 |
| def | Invalid | Python keyword | Rule 4 |
| AGE | Valid | All caps are valid, different from age | Rule 5 |

**Keywords in Python**

**Keywords** are the **reserved words** in Python that have **special meanings**.
They are part of the **syntax** and **cannot be used as identifiers** (i.e., you cannot name variables or functions with them).

**Python Keywords**

```
a = 2
b = 1
if a > b:          # 'if' is keyword
   print ("a is large")
else :             # 'else' is keyword
   print ("b is large")
```

**Output:**

```
a is large
```

**List of Common Python Keywords**

| Category | Keywords |
|---|---|
| Control Flow | if elif else while for break continue pass |
| Functions | def return lambda yield |
| Classes & Objects | class self |
| Exception Handling | try except finally raise assert |
| Boolean & Null | True False None |
| Operators | and or not in is |
| Imports | import from as |
| Variable Scope | global nonlocal del |
| Others | with async await |

**Statements and Expressions**

**Statement**

- A **statement** is an instruction that the Python interpreter can execute.
- A program is made up of a series of statements.
- It could be an assignment, a loop, a function call, etc.
- Example:

```
x = 10  # This is an assignment statement
print(x)  # This is a function call statement
```

**Expression**

- An **expression** is a **combination of variables, values and operators** that produces a new value.
- Python evaluates the expression and returns the result.
- Example:

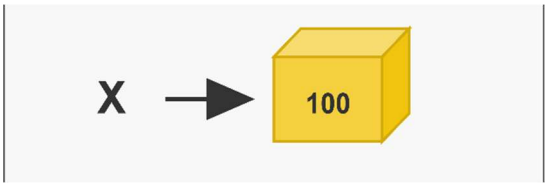
```
5 + 3      # This is an expression that evaluates to 8
2 * 2      # This evaluates to 4
x + y      # If x = 2, y = 3, it evaluates to 5
```

**Python Code:**

```
name = "Sam"
age = 22
x = 10
y = 5
total = x + y

if age > 18:
   print(name, "is an adult")
else:
   print(name, "is a minor")
```

**Variables:** name, age, x, y, total
**Keyword:** if, else
**Functions:** print( )

**Variables in Python**




- A variable is a name or container used for storing a value.
- Values may be numbers, text, or other data types.
- Syntax: variable_name = value
- Example:

```
x = 10              # Integer
name = "John"       # String
price = 99.99       # Float
is_active = True    # Boolean
```

**Rules for Naming Variables**

1. **Can contain:**
   - Letters (A–Z, a–z)
   - Digits (0–9) at end
   - Underscore (_)
     → Example: student_1, age23
2. **Cannot start with a digit**
   → Example: 1name (Invalid)

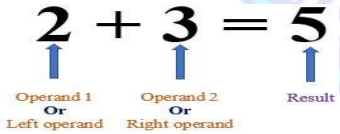
   name1 (Valid)
3. **Cannot use special characters**
   → Example: @, #, $, %, -, +
   → Identifiers like name@1 or user-name are **invalid**
4. **Cannot use Python keywords as identifiers**
   → Example: if, else, class, def
   → if = 5 (Invalid)
5. **Case-Sensitive**
   → Identifiers are case-sensitive
   → Name, name, and NAME are **all different** variables

***Note:***

- *Use descriptive variable names to improve **code readability**.*
- *Choose names that are **meaningful**, **clear**, and **easy to understand**.*

**Operators in Python**

- Operators are **symbols** that perform operations on **variables and values**.
- Example: x = 2 + 3




An **operator manipulates values**, and these values are called **operands**.

**Types of Operators in Python**

1. **Arithmetic Operators**
2. **Assignment Operators**
3. **Comparison Operators**
4. **Logical Operators**
5. **Bitwise Operators**
6. **Membership Operators**
7. **Identity Operators**

Let's discuss one by one

**1. Arithmetic Operators**

Arithmetic operators are operators used to perform mathematical operations such as **addition**, **subtraction**, **multiplication**, **division**, **modulus**, **exponent**, and **floor division**.

**List of arithmetic operators**

| Operator | Operator Name | Syntax (with operands) | Description | Example |
|---|---|---|---|---|
| + | Addition Operator | left_operand + right_operand | Adds the two operands and returns their sum | 2 + 3 = 5 |
| - | Subtraction Operator | left_operand - right_operand | Subtracts right operand from left operand | 2 - 3 = -1 |
| * | Multiplication | left_operand * right_operand | Multiplies the operands | 2 * 3 = 6 |
| / | Division | left_operand / right_operand | Divides left operand by right operand (returns quotient) | 3 / 2 = 1.5 |
| % | Modulus | left_operand % right_operand | Divides and returns the remainder | 3 % 2 = 1 |
| ** | Exponent | left_operand ** right_operand | Left operand raised to the power of right operand | 2 ** 3 = 8 |
| // | Floor Division | left_operand // right_operand | Returns only the integer part of the quotient | 9 // 2 = 4 |

**Note:** Here left_operand and right_operand can be **variables** or **values**.

Let's write a simple Python program that performs **all arithmetic operations** (+, -, *, /, %, **, //) on two numbers.

```
a = 2
b = 3

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

Output:

```
a = 2  b = 3
a + b = 5
a - b = -1
a * b = 6
a / b = 0.6666666666666666
a % b = 2
a ** b = 8
a // b = 0
```

**2. Assignment Operators in Python**

Assignment operators are operators used to **assign values** to variables. They can also combine arithmetic operations with assignment.

**List of assignment operators**

| Operator | Operator Name | Syntax (with operands) | Description | Equivalent Expression | Example (a=2, b=3) |
|---|---|---|---|---|---|
| = | Assignment | left_operand = right_operand | Assigns the value of right operand to left operand | – | a = 3 → a = 3 |
| += | Add and Assign | left_operand += right_operand | Adds right operand to left operand and assigns result | a = a + b | a = 2; a += 3 → a = 5 |
| -= | Subtract and Assign | left_operand -= right_operand | Subtracts right operand from left operand and assigns result | a = a - b | a = 2; a -= 3 → a = -1 |
| *= | Multiply and Assign | left_operand *= right_operand | Multiplies operands and assigns result | a = a * b | a = 2; a *= 3 → a = 6 |
| /= | Divide and Assign | left_operand /= right_operand | Divides left operand by right operand and assigns quotient | a = a / b | a = 2; a /= 3 → a = 0.666... |
| %= | Modulus and Assign | left_operand %= right_operand | Performs modulus and assigns remainder | a = a % b | a = 2; a %= 3 → a = 2 |
| **= | Exponent and Assign | left_operand **= right_operand | Raises left operand to the power of right operand and assigns result | a = a ** b | a = 2; a **= 3 → a = 8 |
| //= | Floor Division and Assign | left_operand //= right_operand | Performs floor division and assigns integer quotient | a = a // b | a = 2; a //= 3 → a = 0 |

Let's write a simple Python program that performs all assignment operations on two numbers.

**CODE**

```
# Assignment Operators in Python
a = 2
b = 3
print("Initial values: a =", a, ", b =", b)

# = (Assignment)
a = b
print("a = b → a =", a)

a = 2
# += (Add and Assign)
a += b
print("a += b → a =", a)

a = 2
# -= (Subtract and Assign)
a -= b
print("a -= b → a =", a)

a = 2
# *= (Multiply and Assign)
a *= b
print("a *= b → a =", a)

a = 2
# /= (Divide and Assign)
a /= b
print("a /= b → a =", a)

a = 2
# %= (Modulus and Assign)
a %= b
print("a %= b → a =", a)

a = 2
# **= (Exponent and Assign)
a **= b
print("a **= b → a =", a)

a = 2
# //= (Floor Division and Assign)
a //= b
print("a //= b → a =", a)
```

**OUTPUT**

Initial values: a = 2 , b = 3

**= (Assignment)**
a = b → a = 3

**+= (Add and Assign)**
a += b → a = 5

**-= (Subtract and Assign)**
a -= b → a = -1

***= (Multiply and Assign)**
a *= b → a = 6

**/= (Divide and Assign)**
a /= b → a = 0.6666666666666666

**%= (Modulus and Assign)**
a %= b → a = 2

****= (Exponent and Assign)**
a **= b → a = 8

**//= (Floor Division and Assign)**
a //= b → a = 0

# Program to swap two variables using Python assignment Operators

```
a = 5
b = 10

print("Before swapping: a =", a, ", b =",
b)

# Swapping
a, b = b, a

print("After swapping: a =", a, ", b =", b)
```

**Output:**
***Before swapping: a = 5 , b = 10***

### 3. Comparison Operators

Comparison operators are operators used to **compare the values of two operands** (left operand and right operand). They evaluate the relationship between the operands and return a Boolean value:

- **True** if the condition is satisfied
- **False** if the condition is not satisfied

**List of comparison operators**

| Operator | Operator Name | Syntax (with operands) | Description | Equivalent Expression | Example (a=2, b=3) |
|---|---|---|---|---|---|
| = = | Equal to | left_operand = = right_operand | Returns True if both operands are equal | a = = b | 2 = = 3 → False |
| != | Not equal to | left_operand != right_operand | Returns True if operands are not equal | a != b | 2 != 3 → True |
| > | Greater than | left_operand > right_operand | Returns True if left operand is greater than right operand | a > b | 2 > 3 → False |
| < | Less than | left_operand < right_operand | Returns True if left operand is less than right operand | a < b | 2 < 3 → True |
| >= | Greater than or equal to | left_operand >= right_operand | Returns True if left operand is greater than or equal to right operand | a >= b | 2 >= 3 → False |
| <= | Less than or equal to | left_operand <= right_operand | Returns True if left operand is less than or equal to right operand | a <= b | 2 <= 3 → True |

Let's write a simple program

```
a = 2
b = 3

print("a =", a, " b =", b)

print("a == b :", a == b)    # Equal to
print("a != b :", a != b)    # Not equal to
print("a > b  :", a > b)     # Greater than
print("a < b  :", a < b)     # Less than
print("a >= b :", a >= b)    # Greater than or equal to
print("a <= b :", a <= b)    # Less than or equal to
```

Output

```
a = 2  b = 3
a == b : False
a < b  : True
a >= b : False
a <= b : True
```

### 4. Logical Operators in Python

- **Logical operators** are used to **combine conditional statements** (expressions that return True or False).
- They return a **Boolean value** (True or False) based on the logic applied.

**Before jumping to logical operators, we need to know some basics**

Types of Logics (Basics)

**1. Logical AND (and)**

- Definition: Returns True only if both operands are True.
- Example: a=2, b=4
  (a < b) **and** (b > 0) → True

| A | B | A and B |
|---|---|---|
| True | True | True |
| True | False | False |
| False | True | False |
| False | False | False |

**2. Logical OR (or)**

- Definition: Returns True if at least one operand is True.
- Example: a=2, b=4
  (a > b) **or** (b > 0) → True

| A | B | A or B |
|---|---|---|
| True | True | True |
| True | False | True |
| False | True | True |
| False | False | False |

**3. Logical NOT (not)**

- Definition: Returns the opposite (negation) of the given operand. If the condition is True, it becomes False, and if it is False, it becomes True.
- Example: a=2
  **not**(a>1) →False

| A | not A |
|---|---|
| True | False |
| False | True |

**List of logical operators**

| Operator | Operator Name | Syntax (with operands) | Description | Example (a=2, b=3) | Result |
|---|---|---|---|---|---|
| **and** | Logical AND | left_operand **and** right_operand | Returns True if **both operands** are True | (a < b) **and** (b > 0) | True |
| **or** | Logical OR | left_operand **or** right_operand | Returns True if **at least one operand** is True | (a > b) **or** (b > 0) | True |
| **not** | Logical NOT | **not** operand | Returns True if operand is False, else returns False | **not**(a > b) | True |

Let's write a simple Python program

```
a = 2
b = 3

print("a =", a, " b =", b)

print("(a < b) and (b > 0) :", (a < b) and (b > 0))   # Logical AND
print("(a > b) or (b > 0) :", (a > b) or (b > 0))     # Logical OR
print("not(a > b) :", not(a > b))                     # Logical NOT
```

**Output**

```
a = 2  b = 3
(a < b) and (b > 0) : True
(a > b) or (b > 0) : True
not(a > b) : True
```

**5. Bitwise Operators**

- Bitwise operators work on binary (bit-level) representations of integers.
- They perform operations bit by bit**.**

**a = 5 → (binary: 0101)**

**b = 3 → (binary: 0011)**

**1. Bitwise AND (&)**

- Compares each bit and returns 1 if both bits are 1 or otherwise 0.

Example: a = 5 → (binary: 0101)

b = 3 → (binary: 0011)

a **&** b

5 **&** 3 = 1

**(0101 & 0011 = 0001)**

```
  0101
& 0011
  0001
```

```
# Program to demonstrate Bitwise AND

a = 5   # binary: 0101
b = 3   # binary: 0011

print("a =", a, " b =", b)
print("a & b =", a & b)   # Bitwise AND
```

Output:

```
a = 5  b = 3
a & b = 1
```

**2. Bitwise OR ( | )**

- Compares each bit and returns 1 if at least one bit is 1.

Example: a = 5 → (binary: 0101)

b = 3 → (binary: 0011)

a | b

5 | 3 = 7

**(0101 | 0011 = 0111)**

```
  0101
| 0011
  0111
```

```
# Program to demonstrate Bitwise OR

a = 5   # binary: 0101
b = 3   # binary: 0011

print("a =", a, " b =", b)
print("a | b =", a | b)   # Bitwise OR
```

Output:

```
a = 5  b = 3
a | b = 7
```

**3. Bitwise XOR ( ^ )**

- Returns 1 if the bits are **different**.

Example: a = 5 → (binary: 0101)
b = 3 → (binary: 0011)
a ^ b
5 ^ 3 = 6
**(0101 ^ 0011 = 0110)**

```
  0101
^ 0011
  0110
```

```
# Program to demonstrate Bitwise XOR

a = 5   # binary: 0101
b = 3   # binary: 0011

print("a =", a, " b =", b)
print("a ^ b =", a ^ b)   # Bitwise XOR
```

Output:

```
a = 5  b = 3
a ^ b = 6
```

**4. Bitwise NOT ( ~ )**

- Flips all the bits (1 becomes 0, 0 becomes 1).
- In Python, we have formula ***~n = -(n+1).***

Example: a = 5 → (binary: 0101)

~a
~5 = -6
**(~0101 = 1011)**

2's compliment=(1's compliment + 1)
0101
1's - 1010
2's - +1
**1011**

```
# Program to demonstrate Bitwise XOR

a = 5   # binary: 0101
b = 3   # binary: 0011

print("a =", a, " b =", b)
print("a ^ b =", a ^ b)   # Bitwise XOR
```

Output:

```
a = 5  b = 3
a ^ b = 6
```

**Note:**

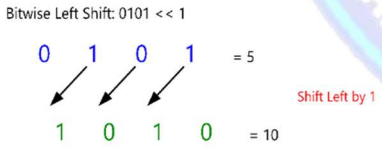
- Computers only understand **binary (0s and 1s)**.
- To represent **negative numbers**, Python (like most programming languages) uses **2's complement system**. Here -6 is negative number.

**5. Bitwise Left Shift (<<)**

- Shifts bits to the **left** by given positions (adds zeros at right).
- In Python, we have formula $a<<n=a\times 2^n$.

Example: a = 5 → (binary: 0101)

a << 1
5 << 1
**(0101 << 1= 1010)**




```
# Program to demonstrate Bitwise Left Shift

a = 5   # binary: 00000101

print("a =", a)

# Left shift by 1
print("a << 1 =", a << 1)   # equivalent to a * 2
```

Output:

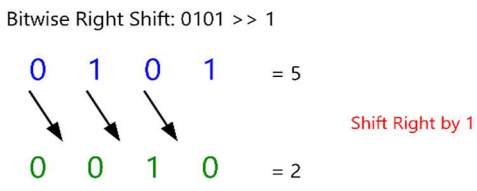
```
a = 5
a << 1 = 10
```

**6. Bitwise Right Shift (>>)**

- Shifts bits to the **right** by given positions.
- In Python, we have formula $a>>n=a/2^n$.

Example: a = 5 → (binary: 0101)

a >> 1
5 >> 1
**(0101 >> 1= 0010)**




```
# Program to demonstrate Bitwise Right Shift

a = 5   # binary: 00000101

print("a =", a)

# Right shift by 1
print("a >> 1 =", a >> 1)   # equivalent to a // 2
```

Output:

```
a = 5
a >> 1 = 2
```

**List of bitwise operators**

| Operator | Operator Name | Syntax (with operands) | Description | Equivalent Expression | Example (a=5, b=3) |
|---|---|---|---|---|---|
| **&** | Bitwise AND | left_operand & right_operand | Performs AND on each bit. Result bit is 1 if both bits are 1. | a & b | 5 & 3 = 1 |
| \| | Bitwise OR | left_operand \| right_operand | Performs OR on each bit. Result bit is 1 if at least one bit is 1. | a \| b | 5 \| 3 = 7 |
| ^ | Bitwise XOR | left_operand ^ right_operand | Performs XOR on each bit. Result bit is 1 if bits are different. | a ^ b | 5 ^ 3 = 6 |
| ~ | Bitwise NOT | ~left_operand | Inverts each bit. Returns the 2's complement (negative) of the number. | ~a | ~5 = -6 |
| << | Left Shift | left_operand << right_operand | Shifts bits of left operand left by given positions (adds zeros at right). | a << b | 5 << 1 = 10 |
| >> | Right Shift | left_operand >> right_operand | Shifts bits of left operand right by given positions (removes bits at right). | a >> b | 5 >> 3 = 2 |

**5. Identity Operators**

- Identity operators in Python are used to compare the **memory location** of two variables, not their values.
- They check whether two variables point to the same object in memory.
- They return a Boolean value **(True / False).**

**List of operators**

| Operator | Operator Name | Syntax | Description | Example (a=3, b=3) | Result |
|---|---|---|---|---|---|
| **is** | Identity | left_operand **is right**_operand | Returns True if both operands refer to the **same object** (same memory location). | a **is** b | True |
| **is not** | Not Identity | left_operand **is not** right_operand | Returns True if both operands **do not refer to the same object**. | a **is not** b | False |

Program to demonstrate **is** operator

```
# Assign integer value 3 to variable 'a'
a = 3

# Assign integer value 3 to variable 'b'
b = 3

# 'is' checks whether 'a' and 'b' refer to the same object in memory
# Since small integers are cached by Python, both 'a' and 'b' point to the
print(a is b)  # Output: True
```

Program to demonstrate **is not** operator

```
# Assign integer value 3 to variable 'a'
a = 3

# Assign integer value 3 to variable 'b'
b = 3

# 'is not' checks whether 'a' and 'b' do NOT refer to the same object in me
# Since small integers are cached by Python, both 'a' and 'b' point to the
print(a is not b)  # Output: False
```

### 6. Membership Operators:

Operators used to **test whether a value is present in a sequence** (like a list, tuple, string, or dictionary keys).

- **in** → Returns True if the value exists in the sequence.
- **not in** → Returns True if the value does **not** exist in the sequence.

**List of membership operators**

| Operator | Operator Name | Syntax | Description | Example | Result |
|---|---|---|---|---|---|
| **in** | Membership | element **in** sequence | Returns True if the element exists in the given sequence. | 'a' **in** 'apple' | True |
| **not in** | Not Membership | element **not in** sequence | Returns True if the element does **not** exist in the sequence. | 'b' **not** in 'apple' | True |

**Program**

```
# Define a string
word = "apple"

# Check if a character exists in the string using 'in'
print('a' in word)      # True, because 'a' is in "apple"

# Check if a character does not exist in the string using 'not in'
print('b' not in word)  # True, because 'b' is not in "apple"
```

**Output**

```
True
True
```

### Precedence and Associativity

**Precedence**

- Precedence determines which operator is **evaluated first** when an expression has **multiple operators**.
- Operators with **higher precedence** are evaluated before operators with **lower precedence**.

**Example:**

```
x = 10 + 2 * 3
# Multiplication (*) has higher precedence than addition (+)
x = 10 + 6
x = 16
```

**Associativity**

- Associativity determines the **order of evaluation** when operators of the **same precedence** appear in an expression.
- Can be **left-to-right** or **right-to-left** depending on the operator.
- **Example:**

```
x = 10 - 2 - 3
# Subtraction (-) has left-to-right associativity
x = (10 - 2) - 3
x = 8 - 3
x = 5
```

| Precedence | Operator(s) | Meaning | Associativity |
|---|---|---|---|
| 1 | () | Parentheses / Grouping | — |
| 2 | ** | Exponentiation | Right-to-left |
| 3 | +x, -x, ~x | Unary plus, minus, bitwise NOT | Right-to-left |
| 4 | *, /, //, % | Multiplication, Division, Floor Div, Modulus | Left-to-right |
| 5 | +, - | Addition and Subtraction | Left-to-right |
| 6 | <<, >> | Bitwise shift operators | Left-to-right |
| 7 | & | Bitwise AND | Left-to-right |
| 8 | ^ | Bitwise XOR | Left-to-right |
| 9 | \| | Bitwise OR | Left-to-right |
| 10 | ==, !=, >, <, >=, <=, is, is not, in, not in | Comparison, Identity, Membership | Left-to-right |
| 11 | not | Logical NOT | Right-to-left |
| 12 | and | Logical AND | Left-to-right |
| 13 | or | Logical OR | Left-to-right |
| 14 | =, +=, -=, *=, /=, etc. | Assignment Operators | Right-to-left |

**Python Operator Precedence Table (From Highest to Lowest)**
**Examples:**

(1) x = 5 + 3 * 2
x = 5 + 6
x= 11

(2) x = (5 + 3) * 2
x = (8) * 2
x = 16

(3) x = 10 + 4 * 2 // 5
x = 10 + 8 // 5
x = 11

(4) x = 2 ** 3 ** 2
x = 2 ** 9
x = 512

(5) x = 10 - 3 + 2
x = 7 + 2
x = 9

(6) x = 4 + 5 + (7 - 3) // 2
x = 4 + 5 + 4 // 2
x = 4 + 10
x = 14

(7) x = -3 ** 2
x = -9

(8) x = not True or False and True
x= False or False and True
x= False or False
x= False

**Evaluate the followings**

1) x = 3 + 4 * 2 ** 3 // 5 - 1
2) x = (6 + 2 * 3) ** 2 // 5 + 7
3) x = 2 ** 3 ** 2 % 7 + 4
4) x = not (False or True) and True or False
5) x = 10 // 3 + 5 * 2 - 7 % 3
6) x = -2 ** 3 + 4 * (3 - 1)
7) x = True and not False or False and True
8) x = (5 + 3) * 2 ** (4 - 2) // 3 + 1

**Data Types**
Data types in Python specify the type of value that a variable can hold and determine the operations that can be performed on that value.

**Basic data types in python are**

1. Numeric types
2. Boolean types
3. Sequence types
4. Set type
5. Dictionary type
6. None type

Let we discuss below

**1. Numeric Types**
Numeric data types are used to store numbers. Python supports three numeric types: **int, float, and complex**.

- **int (Integer):** Represents whole numbers without a decimal point.
  **Syntax:**
  ```
  variable_name = integer_value
  ```
  **Example:**
  ```
  a = 10
  print(type(a)) # <class 'int'>
  ```
- **float (Floating-point):** Represents real numbers with decimal points.
  **Syntax:**
  ```
  variable_name = float_value
  ```
  **Example:**
  ```
  b = 3.14
  print(type(b)) # <class 'float'>
  ```
- **complex (Complex):** Represents numbers in the form ***a + bj*** where ***a*** is real and ***b*** is imaginary.
  **Syntax:**
  ```
  variable_name = real + imaginary*j
  ```
  **Example:**
  ```
  c = 2 + 3j
  print(type(c)) # <class 'complex'>
  ```

**2. String Type (str):** Represents textual data enclosed in single (' ') or double (" ") quotes.
**Syntax:**
```
variable_name = "text"
variable_name = 'text'
```
**Example:**
```
name = "Python"
print(type(name)) # <class 'str'>
```

**3. Boolean Type (bool):** Represents logical values True or False.
**Syntax:**
```
variable_name = True
variable_name = False
```

**Example:**

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

**4. Sequence Types**

Sequence types store collections of items in a specific order. The main sequence types in Python are **list, tuple, and range**.

- **List:** Ordered, mutable collection of elements.

  **Syntax:**

  ```
  list_name = [item_1, item_2, item_3, ……, item_n]
  ```

  **Example:**

  ```
  fruits = ["apple", "banana"]
  print(type(fruits)) # <class 'list'>
  ```

- **Tuple:** Ordered, immutable collection of elements.

  **Syntax:**

  ```
  tuple_name = (item1, item2, item3, ……, item_n)
  ```

  **Example:**

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

- **Range:** Represents a sequence of numbers, often used in loops.

  **Syntax:**

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

  **Example:**

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

**5. Set Type (set):** A set is an unordered collection of unique and mutable elements. duplicate values are automatically removed.

**Syntax:**

```
set_name = {item_1, item_2, item_3, ……, item_n}
```

**Example:**

```
s = {1, 2, 3, 2}
print(s)            # {1, 2, 3}
print(type(s))      # <class 'set'>
```

**6. Dictionary Type (dict):** A dictionary is a collection of **key-value pairs**, where keys must be unique and immutable, and values can be of any type.

**Syntax:**

```
dict_name = {key_1: value_1, key_2: value_2, ……, key_n: value_n}
```

**Example:**

```
student = {"name": "Ravi", "age": 20}
print(student["name"])   # Ravi
print(type(student))     # <class 'dict'>
```

**7. None Type:** The **None type** represents the absence of a value or a null value. It is often used to denote "no result."

**Syntax:**

```
variable_name = None
```

**Example:**

```
x = None
print(type(x))   # <class 'NoneType'>
```

**Indentation in Python:**

Indentation refers to spaces at the beginning of a line of code. It is mandatory in Python and is used to define blocks of code such as loops, conditionals, and functions.

**Syntax:**

```
if condition:
    statement
```

**Example:**

```
if True:
    print("Hello")  # 4 spaces
```

Indentation symbol

**Purpose**:

- It tells Python which statements belong together (inside loops, functions, if-else blocks, etc.).
- Without proper indentation, Python cannot understand the structure of the code.

**Errors due to indentation**:

- **Indentation Error**: Raised if code is not properly indented.
- **Unexpected Indent Error**: Happens if you start with extra indentation where it is not needed.
- **Inconsistent Indent Error**: Mixing tabs and spaces or misaligned blocks.

**Comments in Python**

Comments are non-executable statements that explain the code and make it more readable.
Python ignores comments during execution.
It is used only for human to understand the code

**Syntax:**

```
# Single-line comment
```

**Example:**

```
# This is a comment
  print("Hello World")
```

Types of comments

**1. Single-line Comments**

A single-line comment starts with the # symbol.

**Syntax:**

```
# This is a single-line comment
```

**Example:**

```
x = 5   # Assigning 5 to x variable
print(x)
```

**2. Multiple Single-line Comments**

Writing multiple # lines one after another to explain code in more detail.

**Syntax:**

```
# This is a first-line comment
# This is a second-line comment
```

**Example:**

```
x = 5       # Assigning 5 to x variable
x = x + 1   # Addition assignment operation
print(x)
```

**3. Multi-line String Comments (Docstring Style)**

Multi-line comments can also be written using triple quotes (''' or """).
**Note:** Technically, Python treats them as **multi-line strings**, but if not assigned to any variable, they act as comments.

**Syntax:**

```
"""
 This is a multi-line comment.
It can span multiple lines.
"""
```

**Example:**

```
x = 5
x += 1
"""
First assign value 5 to x
Then perform addition assignment operation
"""
 print(x)
```

**Built-in Functions**

Built-in functions are **predefined functions** in Python that can be used directly without importing any library.
Commonly used console input/output functions are:

- input → means taking input from the user
- output → means displaying output to the user

**1. Input Function → input( )**

The input( ) function is used to read (take) data from the user via the keyboard.
By default, it takes the input as a **string**.

**Syntax:**

```
variable_name = input("message")
```

**Example:**

```
person = input("What is your name? ")
print(person)
```

**Output (if user enters "Ravi"):**

```
What is your name? Ravi
Ravi
```

**2. Output Function → print( )**

The print( ) function is used to display output (text, numbers, variables, etc.) on the console.

**Syntax:**

```
print(object, ..., sep=' ', end='\n')
```

- *object → one or more values to print*
- *sep → separator (default is space " ")*
- *end → what to print at the end (default is newline \n)*

**Example:**

```
print("Hello BCA AI")
```

**Output:**
*Hello BCA AI*

**Different ways of formatting in print statements**

**a. General**

**Syntax:**

```
print("a =",a," and b = ",b)
```

**Example:**

```
a = 2
b = 1
print("a =",a," and b = ",b)
```

**Output:**
*a = 2 and b = 1*

**b. str.format( )**

**Syntax:**

```
print("a = {0} and b = {1}".format(a, b))
```

**Example:**

```
a = 2
b = 1
print("a = {0} and b = {1}".format(a, b))
```

**Output:**
*a = 2 and b = 1*

**Output:**
*a = 2 and b = 1*

**b. f-strings**

**Syntax:**

```
print(f"a = {a} and b = {b}")
```

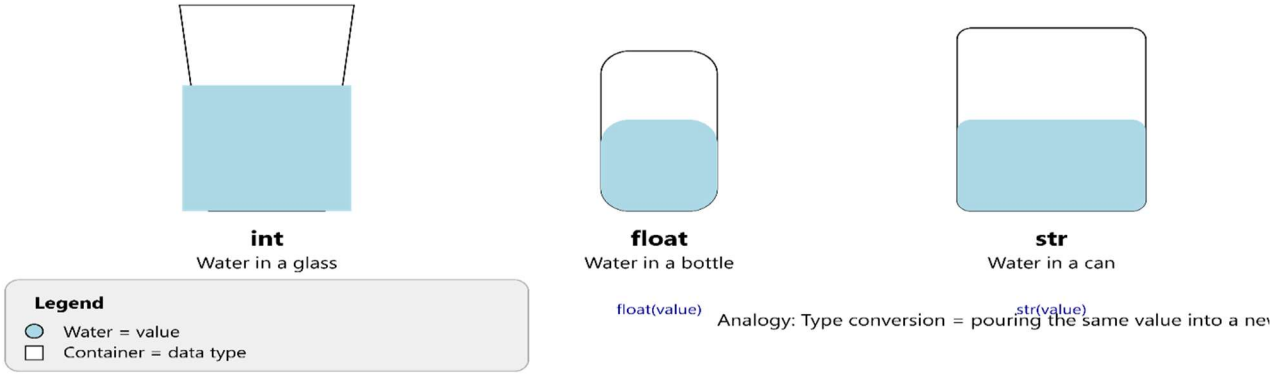
**Example:**

```
a = 2
b = 1
print(f"a = {a} and b = {b}")
```

**Output:**
*a = 2 and b = 1*

**Type Conversion in Python**

Type conversion is the process of converting data type of value to another type.
It is also called type casting
In Python, it can be done **automatically** (implicit) or **manually** (explicit) depending on some situations.



Types of type conversion

1. Implicit type conversion
2. Explicit type conversion

**Let's discuss……**

**1. Implicit Type Conversion (Type Casting)**

- Python automatically done this conversion during some operations .
- Converts smaller data type into larger data type to prevent data loss.
- Also called **Type Casting by Python (Type Promotion)**.

**Example:**

```
x = 10          # int
y = 5.5         # float
z = x + y       # int + float → float
print(z)        # Output: 15.5
```

**Implicit Conversion Table**

| From | To | Example | Result |
|---|---|---|---|
| int | float | 5 + 2.5 | float |
| int | complex | 4 + (3+2j) | complex |
| bool | int | True + 10 | int |
| bool | float | False + 2.5 | float |
| bool | complex | True + (3j) | complex |
| float | complex | 2.5 + (3+1j) | complex |
| int | bool | 10 + True | int |

**Easy Trick:** Smaller → Bigger
bool → int → float → complex

***Note:***

***Fail of Implicit Conversion***

```
x = "10"
y = 5
print(x + y)   #   TypeError
```

*Python does not auto-convert "10" into 10.*
*If it did, the result could mean either:*

*"10" + 5 → "105" (string concatenation), or*
*"10" + 5 → 15 (numeric addition).*
*Since it's ambiguous, Python raises an error instead of guessing*

*Due to this reason, we use explicit type conversion*

**2. Explicit type conversion**

**Explicit type conversion** in Python is the process of **manually converting one data type into another** using built-in functions.

let discuss……

**a. int( )**

The **int( ) function** in Python is a **built-in function** that converts a given value into an **integer number**.

- If the input is a number (like float, boolean), it converts it to the nearest integer (by truncating decimal part).
- If the input is a string containing digits, it converts that string into an integer.
- If the input cannot be interpreted as an integer, Python raises a **ValueError**.

**Syntax:**

```
int(value)
```

**Example:**

```
x = int(3.9)    # float → integer
print(x)        # 3
```

**Note:** Works with float, boolean, string (digit), complex (real part only).

**b. float( )**

The **float( )** function in Python is **a built-in function** that converts a given value into a **floating-point number** (decimal).

- If the input is an integer or boolean, it converts it into a float.
- If the input is a string containing digits (with or without decimal point), it converts that string into a float.
- If the input cannot be interpreted as a float, Python raises a **ValueError**.

**Syntax:**

```
float(value)
```

**Example:**

```
x = float(5)          # integer → float
print(x)              # 5.0

y = float("3.14")     # string → float
print(y)              # 3.14
```

**Note:** Works with integer, boolean, string (digit), complex (real part only).

**c. str( )**

The **str( ) function** in Python is a built-in function that converts a given value into a **string (text format)**.

- If the input is a **number (integer, float, complex)**, it converts it into its string representation.
- If the input is a **boolean**, it converts it into "True" or "False".
- If the input is already a **string**, it returns the same string.
- If the input cannot be interpreted as a string, Python raises a **ValueError**.

**Syntax:**

```
str(value)
```

**Example:**

```
x = str(25)           # integer → string
print(x)              # "25"

y = str(3.14)         # float → string
print(y)              # "3.14"

z = str(True)         # boolean → string
print(z)              # "True"

w = str(2+3j)         # complex → string
print(w)              # "(2+3j)"
```

**Note:** Works with all data types.

**d. bool( )**

The **bool( ) function** in Python is a built-in function that converts a given value into a **boolean value** (True or False).

- If the input is **zero** (0, 0.0, 0j), **empty** ("", [], {}, (), set()), or None, it converts to **False**.
- if the input is any **non-zero number**, **non-empty string**, or **non-empty collection**, it converts to **True**.
- If the input is already a boolean, it returns the same value.

**Syntax:**

```
bool(value)
```

**Example:**

```
x = bool(0)           # integer 0 → False
print(x)              # False

y = bool(25)          # non-zero integer → True
print(y)              # True

z = bool("")          # empty string → False
print(z)              # False
```

**Note:** Works with all data types.

**e. complex( )**

The **complex( ) function** in Python is a built-in function that converts a given value into a **complex number** in the form a + bj, where a is the real part and b is the imaginary part.

- If the input is a **number (integer, float, boolean, string with digits)**, it creates a complex number with that value as the **real part** and imaginary part as 0.
- If two arguments are given (real, imag), it returns a complex number in the form *real + imag*j*.
- If the input is a string in valid complex format (like "3+4j"), it converts it into a complex number**.**

**Syntax:**

```
complex(value)
complex(real, imag)
```

**Example:**

```
x = complex(5)          # integer → complex
print(x)                # (5+0j)

y = complex(3.5)        # float → complex
print(y)                # (3.5+0j)

z = complex(2, 4)       # real + imag
print(z)                # (2+4j)

w = complex("7+3j")     # string → complex
print(w)                # (7+3j)
```

**Note:** Works with integer, float, string, boolean.

**f. list( )**

The **list( ) function** in Python is a built-in function that converts a given iterable (sequence or collection) into a **list**.

- If the input is a **string**, it converts each character into a list element.
- If the input is a **tuple, set, or dictionary**, it converts it into a list. for a dictionary, only the **keys** are taken.
- If the input is already a list, it returns the same list.
- If the input is not sequence, Python raises a **TypeError**.

**Syntax:**

```
list(sequence_values)
```

**Example:**

```
x = list("hello")          # string → list of
                             characters
print(x)                   # ['h', 'e', 'l', 'l', 'o']

y = list((1, 2, 3))        # tuple → list
print(y)                   # [1, 2, 3]

z = list({10, 20, 30})     # set → list
print(z)                   # [10, 20, 30]  (order may vary)

w = list({"a": 1, "b": 2})   # dictionary → list of keys

print(w)                     # ['a', 'b']
```

**Note:** Works with **string, tuple, set, dictionary (keys),**.

**g. tuple( )**

The **tuple( ) function** in Python is a built-in function that converts a given iterable (sequence or collection) into a **tuple** (an ordered, immutable sequence).

- if the input is a **string**, it converts each character into a tuple element.
- If the input is a **list, set, or dictionary**, it converts it into a tuple. for a dictionary, only the **keys** are taken.
- If the input is already a tuple, it returns the same tuple.
- If the input is not sequence, Python raises a **TypeError**.

**Syntax:**

```
tuple(sequence_values)
```

**Example:**

```
x = tuple("hello")          # string → tuple of characters
print(x)                    # ('h', 'e', 'l', 'l', 'o')

y = tuple([1, 2, 3])        # list → tuple
print(y)                    # [1, 2, 3]

z = tuple({10, 20, 30})     # set → tuple

print(z)                    # (10, 20, 30)  (order may vary)

w = tuple({"a": 1, "b": 2})   # dictionary → tuple of keys
print(w)                      # ('a', 'b')
```

**Note:** Works with **string, list, set, dictionary (keys)**.

**h. set( )**

The **set( ) function** in Python is a built-in function that converts a given iterable (like string, list, tuple, etc.) into a **set**, which is an **unordered collection of unique elements**.

- if the input is a **string**, it converts each character into a set element.
- If the input is a **list, tuple, or dictionary**, it converts it into a set. for a dictionary, only the **keys** are taken.
- If the input contains **duplicates**, they are automatically removed.
- If the input is not sequence, Python raises a **TypeError**.

**Syntax:**

```
set(sequence_values)
```

**Example:**

```
x = set("hello")          # string → set of characters (unique)
print(x)                  # {'h', 'e', 'l', 'o'}

y = set([1, 2, 2, 3])     # list → set (removes duplicates)
print(y)                  # {1, 2, 3}

z = set((10, 20, 30))     # tuple → set

print(z)                  # {10, 20, 30}  (order may vary)

w = set({"a": 1, "b": 2})   # dictionary → set of keys
print(w)                    # {'a', 'b'}
```

**Note:** Works with **string, list, tuple, dictionary (keys)**.

**i. dict( )**

The **dict() function** in Python is a built-in function that creates a **dictionary object** (collection of key–value pairs).

**Syntax:**

```
dict()
dict(key=value, ...)
```

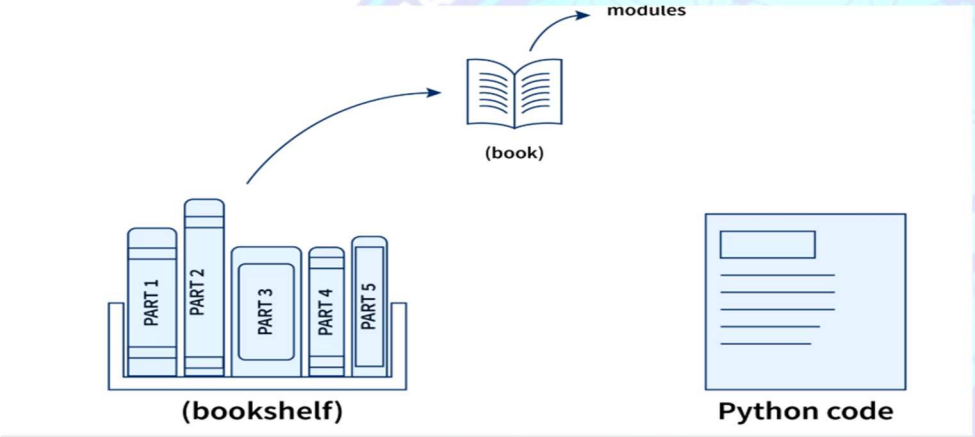
**Example:**

```
# Empty dictionary
x = dict()
print(x)    # {}

# List of tuples → dictionary
y = dict([("a", 1), ("b", 2)])
print(y)    # {'a': 1, 'b': 2}

# Tuple of tuples → dictionary
z = dict((("x", 10), ("y", 20)))
print(z)    # {'x': 10, 'y': 20}

# Keyword arguments → dictionary
w = dict(name="Alice", age=25)
print(w)    # {'name': 'Alice', 'age': 25}

# Copying dictionary
d = dict({"m": 100, "n": 200})
print(d)    # {'m': 100, 'n': 200}
```

**Note:** Works with **list of pairs, tuple of pairs**.

**Library in Python**

A **library** is a collection of pre-written code (functions, classes, variables) that we can reuse instead of writing from scratch.




**LIBRARY** = **L**et's **I**mport **B**unch of **R**eady-made **A**ctions **R**ather than **Y**awning while coding from scratch.

**Why Use Libraries?**

- **Saves time** (Don't reinvent the wheel)
- **Fewer bugs** (Well-tested code)
- **Special powers** (Like AI, data visualization, web scraping, etc.)

**Types of Python Libraries**

1. **Built-in libraries** – Already included with Python (e.g., math, os, random).
2. **External libraries** – Need to install via **pip** (e.g., numpy, pandas, matplotlib).

**pip- python install packages**

**Importing a Library — Syntax**

| Style | Syntax | Meaning |
|---|---|---|
| Whole library | import library_name | Brings the full toolbox. |
| Alias (nickname) | import library_name as shortname | Full toolbox but shorter name. |
| Selective import | from library_name import tool_name | Only brings the tool you need. |

**General Example (Built-in Library)**

```
import math
radius = 5
area = math.pi * (radius ** 2)
print("Area of circle:", area)
```

**Why easier?** No need to type 3.14159 manually, math.pi is accurate and ready.

One Script Showing All Three Styles

```
# 1. Import the whole library
import math
radius = 5
area1 = math.pi * (radius ** 2)
print("Area (whole library):", area1)

# 2. Import with alias
import math as m
area2 = m.pi * (radius ** 2)
print("Area (alias):", area2)

# 3. Import only what you need
from math import pi
area3 = pi * (radius ** 2)
print("Area (specific import):", area3)
```

**Control flow statements in python**

Control flow statements in Python are used to determine the order in which instructions are executed in a program

The Control flow statements in python are

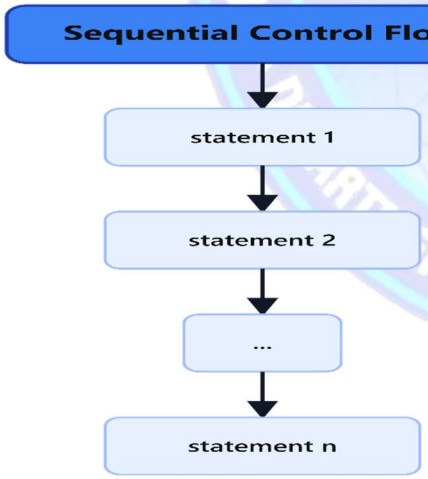
**1. Sequential control flow statements**

Sequential control flow statements in Python are executed one after another in the exact order in which they appear in the program, without any branching or repetition.

**Characteristics:**

1. Instructions are executed line by line from top to bottom.
2. There is no decision-making or looping involved.
3. It is the simplest form of control flow.

**Flow chart:**




**Syntax:**

```
statement 1
statement 2
statement 3
...
statement n
```

**Example:**

```
# Sequential execution of statements
print("Start of program")    # statement 1
x = 5                        # statement 2
y = 10                       # statement 3
z = x + y                    # statement 4
print("Sum:", z)             # statement 5
print("End of program")      # statement 6
```

**Output:**

```
Start of program
Sum: 15
End of program
```

**2. Decision control flow statements**

Decision control flow statements in Python allow a program to execute certain instructions conditionally, based on whether a specified condition evaluates to True or False. They enable branching in the flow of execution.

**Characteristics:**

1. The program can choose between two or more paths of execution.
2. Conditions are evaluated using relational or logical expressions.
3. Common decision statements in Python include: if, if-else, if-elif-else, nested-if-else.

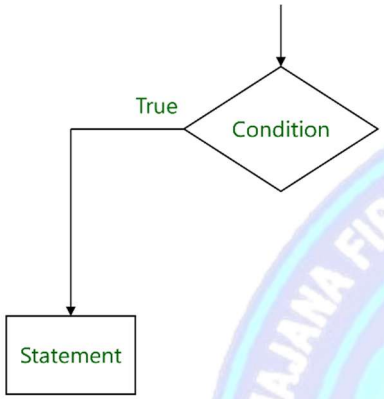
**Types of Decision control flow statements are**

**If statement (Single way*)***

The if statement is a **single-way decision-making control flow statement** in Python. in which:

- Executes a block of code only if the specified condition is **True**.
- Skips the block if the condition is **False**.

**Flow chart:**




**Syntax:**

```
if condition:
    statements
```

**Example:**

```
a = 3
if a > 1:
    print(a,"is greater than 1")
```

**Output:**

```
3 is greater than 1
```

**Explanation:**

```
1. a is initialized with 3.
2. The condition a > 1 is checked.
3. Since 3 > 1 is True, the code inside the if block runs.
4. If the condition were False (e.g., a = 0), nothing inside the if block would
execute.
```

**1. Program to read a number and print a message if it is positive**

```
number = int(input("Enter the number:"))
if number > 0:
    print(number,"is positive")
```

**Output:**

```
Enter the number:2
 2 is positive
```

**Explanation:**

```
1. The program prompts the user to
enter a number.
2. The input is converted from
string to integer using int().
3. The if condition checks whether
the number is greater than 0.
4. If the number is positive, the
print statement inside the if
block runs.
5. If the number is 0 or negative,
nothing is printed because there
is no else block.
```

**2. WAP to check voting eligibility using if statement**

```
age  =  int(input("Enter the your age:"))
if age  >= 18 :
    print("Eligible for voting")
```

**Output:**

```
Enter the number:18
Eligible for voting
```

**3. WAP to check student is passed by marks**

```
marks  =  int(input("Enter the marks:"))
if marks  >= 35 :
    print("student is passed")
```

**Output:**

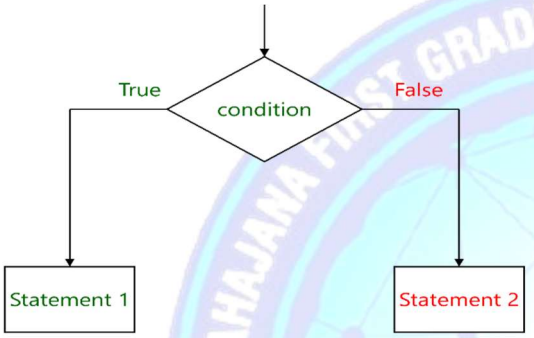
```
Enter the number:35
student is passed
```

**if-else statement**

It is a two-way decision-making control flow statement in which

- The if condition is true, the code inside **if** block executes.
- The if condition is false, the code inside **else** block executes.

**Flow chart**




**Syntax:**

```
if condition:
    statement 1
else:
    statement 2
```

**1. Program to check if a given number is even or odd**

```
number = int(input("Enter a number: "))  # Take input from user and
                                            convert it to integer

if number % 2 == 0:                    # Check if the number is divisible by 2
    print(number, "is even")           # If divisible by 2, print that the
                                                  number is even
else:
    print(number, "is odd")                 # If not divisible by 2, print
                                                 that the number is odd
```

**Output: - Case 1: Even number**

```
Enter a number: 10
10 is even
```

**Output: - Case 2: Odd number**

```
Enter a number: 9
9 is odd
```

**2. Program to check bank loan eligibility check based on CIBIL score**

```
cibil = int(input("Enter your CIBIL score: ")) # Take CIBIL score as input
if cibil >= 750:                                  # Condition for  eligibility
    print("You are eligible for a loan")          # Eligible if CIBIL >= 750
else:
    print("You are not eligible for a loan") # Not eligible if CIBIL < 750
```

**Output: - Case 1: Eligible**

```
Enter your CIBIL score: 750
You are eligible for a loan
```

**Output: - Case 2: Not eligible**

```
Enter your CIBIL score: 600
You are not eligible for a loan
```

**3. program to check mobile data usage warning**

```
data_used = float(input("Enter your data usage in GB: "))  # Take data
                                                           usage as input

warning_limit = 1.5                                  # Warning threshold

if data_used > warning_limit:                         # Condition for warning
    print("Warning! You have exceeded your data limit.")   # Display
                                                  warning if usage is above limit
else:
    print("Your data usage is within the limit.")         # No warning if
                                                 usage is below or equal to limit
```

**Output: - Case 1: within the limit**

```
Enter your data usage in GB: 1
Your data usage is within the limit.
```

**Output: - Case 2 exceeded your data limit**

```
Enter your data usage in GB: 2
Warning! You have exceeded your data limit.
```

**4. program to check C1 component eligibility based on attendance percentage (>=75%)**

```
attendance = int(input("Enter attendance percentage: "))
if attendance >= 75:
    print("Eligible for C1 component")
else:
    print("Not eligible for C1 component")
```

**Output: - Case 1: Eligible**

```
Enter attendance percentage: 76
Eligible for C1 component
```

**Output: -Case 2: Not eligible**

```
Enter attendance percentage: 56
Not eligible for C1 component
```

**5. Check for free delivery in online shopping (> 500)**

```
amount = int(input("Enter purchase amount: "))
if amount > 500:
    print("Free delivery available")
else:
    print("No free delivery")
```

**Output: - Case 1: Free delivery**

```
Enter purchase amount: 600
Free delivery available
```

**Output: - Case 2: No free delivery**

```
Enter purchase amount: 400
No free delivery
```

**6. Check if given character is vowel or consonant:**

```
char = input("Enter only alphabet character: ")
if char in 'aeiouAEIOU':
    print("Vowel")
else:
    print("Consonant")
```

**Output: - Case 1: Vowel**

```
Enter only alphabet character: a
Vowel
```

**Output: -: Case 2 Consonant**

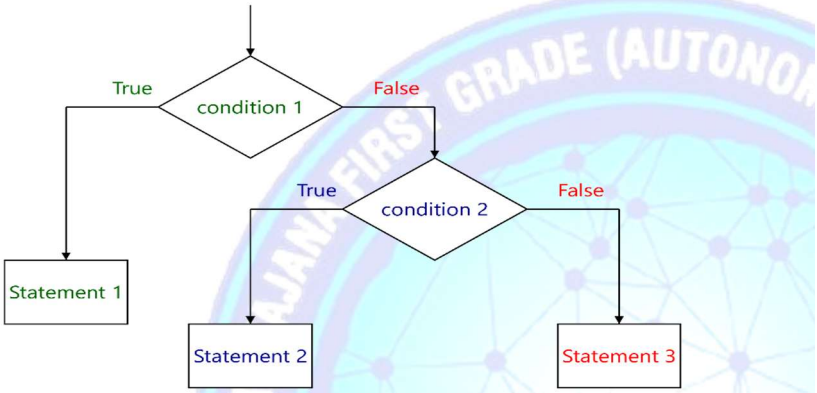
```
Enter only alphabet character: r
Consonant
```

**if-elif-else statement**

it is a multi-way decision control flow statement in which

- **if** condition is true, the code inside **if** block executes.
- **If** condition is false then **elif** condition is checked, if **elif** false then go to **else** statement.

**Flow chart**




**Syntax:**

```
if condition 1:
    statement 1
elif condition 2:
    statement 2
.....
.....
.....
elif condition n:
    statement n

else:
    last statement
```

**1. WAP to find number is negative, positive, or zero**

```
n = int(input("Enter the number: "))
if n > 0:
    print(n, "is positive number")
elif n < 0:
    print(n, "is negative number")
else:
    print("Entered Number is Zero")
```

**Output: - Case 1: Positive**

```
Enter the number: 2
2 is positive number
```

**Output: - Case 2: Negative**

```
Enter the number: -2
-2 is negative number
```

**Output: - Case 3: Zero**

```
Enter the number: 0
Entered Number is Zero
```

**2. WAP to find grade based on marks (0-100)**

```
marks = int(input("Enter the marks: "))
if marks >= 90:
    print("your grade is 'A'")
elif marks >= 80:
    print("your grade is 'B'")
elif marks >= 70:
    print("your grade is 'C'")
elif marks >= 60:
    print("your grade is 'D'")
elif marks >= 35:
```

**Output: - Case 1: A grade**

```
Enter the marks: 95
your grade is 'A'
```

**Output: - Case 2: B grade**

```
Enter the marks: 82
your grade is 'B'
```

**Output: - Case 3: C grade**

```
Enter the marks: 73
your grade is 'C'
```

**Output: - Case: 4 D grade**

```
Enter the marks: 61
your grade is 'D'
```

```
    print("your grade is 'pass'")
else:
    print("your grade is 'fail'")
```

**Output: - Case 5: Pass grade**

```
Enter the marks: 40
your grade is 'pass'
```

**Output: - Case 6: Fail**

```
Enter the marks: 20
your grade is 'fail'
```

**3. WAP to check largest of 3 numbers**

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
if a > b and a > c:
    print(a, "is large")
elif b > a and b > c:
    print(b, "is large")
else:
    print(c, "is large")
```

**Output: -Case 1: A is large**

```
Enter first number: 1
Enter second number: 2
Enter third number: 3
3 is large
```

**Output: - Case 2: B is large**

```
Enter first number: 3
Enter second number: 2
Enter third number: 1
3 is large
```

**Output: - Case 3: C is large**

```
Enter first number: 2
Enter second number:
3
Enter third number: 1
3 is large
```

**4. WAP to find largest of four numbers**

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
d = int(input("Enter fourth number: "))
if a > b and a > c and a > d:
    print(a, "is largest")
elif b > a and b > c and b > d:
    print(b, "is largest")
elif c > a and c > b and c > d:
    print(c, "is largest")
else:
    print(d, "is largest")
```

**Output: - Case 1: A is large**

```
Enter first number: 10
Enter second number: 25
Enter third number: 15
Enter fourth number: 5
25 is largest
```

**Output: - Case 2: B is large**

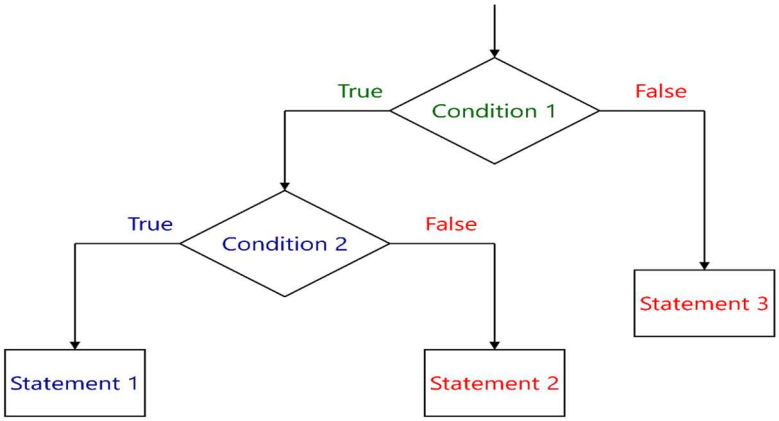
```
Enter first number: 12
Enter second number: 22
Enter third number: 45
Enter fourth number: 35
45 is largest
```

**Nested if statements**

It is a decision-making control flow statement in which an if statement is placed inside another if statement.

- If the outer if condition is true, then the inner if condition is checked.
- If the inner if condition is also true, then the code inside the inner block executes.
- If the inner if condition is false, its corresponding else block executes (if present).
- If the outer if condition is false, then the outer else block executes (if present).

**Flow chart**




**Syntax:**

```
if condition 1:
    if condition 2:
        statement 1
    else:
        statement 2
else:
    statement 3
```

**1. WAP to user id and password authentication using nested if statment**

```
user_id = input("Enter user id: ")
password = input("Enter password: ")
if user_id == "admin"
   if  password == "admin123":
      print("login successsful")
   else:
    print("Incorrect password")
else:
   print("Invalid User_id")
```

**Output: -Case 1: Correct user id and password**

```
Enter user id: admin
Enter password: admin123
login successful
```

**Output: - Case 2: Correct user id but wrong password**

```
Enter user id: admin
Enter password: 12345
Incorrect password
```

**Output: - Case 3: Wrong user id**

```
Enter user id: guest
Enter password: admin123
Invalid User_id
```

**2. WAP to check largest of 3 numbers using nested if**

```
a = int(input("Enter first number: "))
b = int(input("Enter second number: "))
c = int(input("Enter third number: "))
if a > b:
    if a > c:
        print(a,"is large")
    else:
        print(c,"is large")
else:
    if b > c:
        print(b,"is large")
    else:
        print(c,"is large")
```

**Output: - Case 1: a is largest**

```
Enter first number: 10
Enter second number: 5
Enter third number: 3
10 is large
```

**Output: - Case 2: b is largest**

```
Enter first number: 4
Enter second number: 12
Enter third number: 8
12 is large
```

**Output: - Case 3: c is largest**

```
Enter first number: 7
Enter second number: 9
Enter third number: 15
15 is large
```

**Looping Statements (Iterative Statements)**

Looping Statements" are control statements in which a block of code is executed repeatedly until a given condition becomes false or for a specified number of times in a range

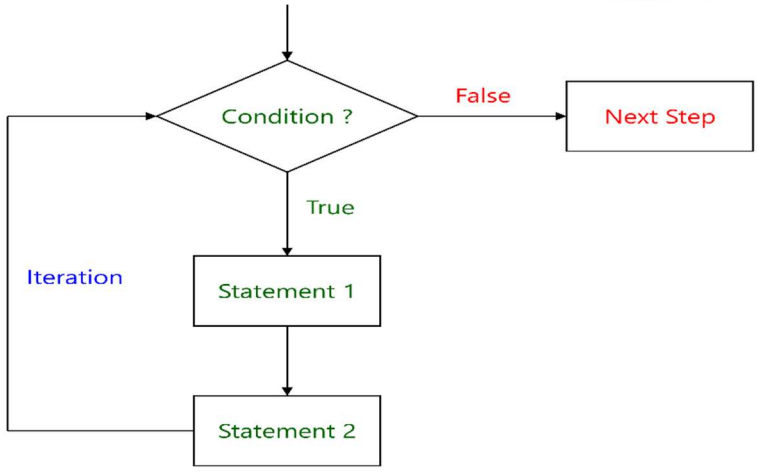
**Let's discuss looping statements one by one in detail.**

**While Statement (while loop)**

The while statement is a looping control flow statement used to execute a block of code repeatedly, as long as the given condition evaluates to True.

- If the while condition is **True**, then statements inside while loop execute **again** and **again** until condition becomes False**.**
- If the condition becomes **False**, the loop stops, and the program continues with the next statement after the loop.
- If the while condition is **False**, then statements inside loop doesn't executes.

**Flow Chart**




**Syntax:**

```
while condition:
    statement 1
    statement 2
else:
    next statement
```

**Working of while loop**

1. **Start**
   - The program begins execution and enters the while loop.
2. **Condition Check**
   - The loop first checks the **condition** (a logical expression).
   - If the **condition is True**, the program enters the loop body.
   - If the **condition is False**, the loop is terminated, and control moves to the **Next Step** after the loop.
3. **True Path (Condition is True)**
   - When the condition is True, the loop executes **Statement 1**, then **Statement 2**, or more statements inside the loop.
   - After executing the statements, control goes back to check the condition again.
4. **Iteration**
   - Each time the condition is checked and the loop body executes, it is called an **iteration**.
   - The process of re-checking the condition and executing statements continues until the condition becomes False.
5. **False Path (Condition is False)**
   - When the condition finally evaluates to False, the loop **terminates**.
   - The program then moves to the **Next Step** (the statement that comes immediately after the loop).

**1. Program to Print Numbers from 1 to 10**

```
i = 1

while i <= 10:

    print(i)

    i += 1

print("completed")
```

**Output**

```
1
2
3
4
5
6
7
8
9
10
completed
```

**Explanation:**

1. The loop **starts with i = 1**.
2. As long as i is **less than or equal to 10**, the loop will keep running.
3. In each step (iteration), the current value of i is shown (printed), and then i increases by 1.
4. This continues for values 1, 2, 3, …, 10.
5. When **i** becomes 11, the condition is no longer true, so the loop stops.
6. After the loop ends, the program displays **"completed"**.

**2. Program to display multiplication table**

```
n = int(input("Enter the Number: "))
i=1
while i<=10:
    m=n*i
    print(n,"*",i,"=",m)
    i=i+1
```

**Output:**

```
Enter the Number: 5
5 * 1 = 5
5 * 2 = 10
5 * 3 = 15
5 * 4 = 20
5 * 5 = 25
5 * 6 = 30
5 * 7 = 35
5 * 8 = 40
5 * 9 = 45
5 * 10 = 50
```

**Explanation of the Program**

1. The program first asks the user to **enter a number**.
   Example: if the user enters 5.
2. It starts with i = 1.
3. The while loop runs **as long as i <= 10**.
   - This ensures the table is printed from 1 to 10.
4. Inside the loop:
   - It calculates m = n * i (the multiplication result).
   - Then it prints the result in the format:
   - n * i = m

**Example:** 5 * 1 = 5.

5. After printing, it increases i by 1.
6. This process continues until i becomes 11, then the condition is false and the loop stops.

**3. Program to reverse countdown of numbers from 5 to 1**

```
i = 5
while i >= 1:
    print(i)
    i -= 1

print("completed")
```

**Output:**

```
5
4
3
2
1
completed
```

**Explanation**

1. The program starts with i = 5.
2. The **condition** i >= 1 is checked.
   - If i is greater than or equal to 1, the loop continues.
3. Inside the loop:
   - It prints the current value of i.
   - Then decreases i by 1 (i -= 1).
4. This repeats: 5 → 4 → 3 → 2 → 1.
5. When i becomes 0, the condition is False, so the loop ends.

**Break statement (break)**

The **break statement** is a **loop control statement** used to **immediately exit** a loop, even if the loop condition is still true.

- Once break is executed, the control jumps **out of the loop** and continues with the statement after the loop.

**Syntax:**

```
while condition:
    if condition:
        break
    statements
```

**Program**

```
i = 1
while i <= 10:
    if i == 6:
        break   # loop stops when i = 6
    print(i)
    i += 1
```

**Output:**

```
1
2
3
4
5
```

**Explanation**

1. The variable i starts with **1**.
2. The while loop condition i <= 10 is **True**, so the loop begins.
3. Inside the loop:
   - If i = = 6, the break statement executes → the loop **stops immediately**.
   - Otherwise, it prints the value of i.
   - Then i increases by 1 (i += 1).
4. The loop continues until either:
   - i becomes greater than 10, or
   - i becomes 6 (because of break).

**Continue statement (continue)**

The **continue statement** is a loop control statement that **skips the current iteration** and moves to the **next iteration** of the loop.

- Unlike break, it does **not stop the loop**.
- It just **ignores the remaining code** inside the loop for the current iteration and goes back to check the condition again.

**Syntax:**

```
while condition:
    if condition:
        continue
    statements
```

**Program**

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        continue   # skip when i = 3
    print(i)
```

**Output:**

```
1
2
4
5
```

**Explanation:**

1. The variable i starts with 0.
2. The while loop checks the condition i < 5. If true, the loop runs.
3. Inside the loop:
   - First, i increases by 1 (i += 1).
   - If i == 3, the continue statement executes → this **skips the print(i) line** for that round and goes back to the next loop iteration.
   - Otherwise, it prints the value of i.
4. The loop continues until i reaches 5.

**Pass statement (pass)**

- The **pass statement** is a placeholder that **does nothing** when executed.
- The **pass statement** is a **placeholder** that does nothing when executed.
- It is used when a statement is **syntactically required** but no action is needed.
- The program **continues execution normally** after encountering pass.
- Python requires some code inside a block (like inside a loop, if, or function), but if you don't want any action to be taken, you can use pass.

**Syntax:**

```
Control statements:
    pass
```

**Syntax:**

```
looping statements:
    pass
```

**Syntax:**

```
def Function_name( ) :
    pass
```

**Program**

```
i = 0
while i < 5:
    i += 1
    if i == 3:
        pass    # Do nothing when i = 3
    print(i)
```

**Output:**

```
1
2
3
4
5
```

**Explanation:**

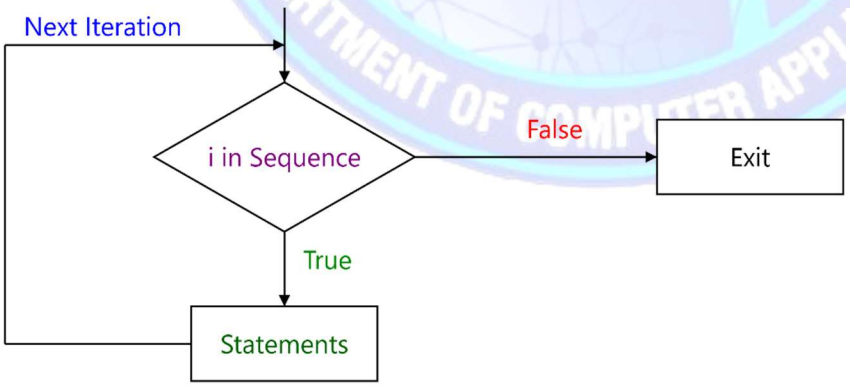
1. **i** starts at 0.
2. The loop runs as long as **i < 5**.
3. Each time, **i** increases by **1**.
4. When **i == 3**, the pass statement executes → but it **does nothing**, so the loop just continues normally.
5. All values are printed:

**For Statement (for loop)**

The **for statement** is a **looping control flow statement** used to iterate over a sequence (like a list, tuple, string, or range) and execute a block of code for **each item in that sequence**.

- If the sequence has elements, the loop variable takes each element one by one, and the statements inside loop executes.
- When all elements are processed, the loop stops, and execution continues with the next statement after the loop.
- If the sequence is empty, the loop body does not execute.

**Flow chart**




**Syntax:**

```
for variable in sequence:
    statements
```

**Working of For Loop**

1. **Start**
   • The program begins execution and enters the for loop.
2. **Sequence Check**
   • The loop takes the first element from the sequence (list, tuple, string, range, etc.) and assigns it to the loop variable.
   • If the sequence has no elements (empty), the loop body is skipped, and control moves to the next statement after the loop.

3. **Execution of statements**
   • When the loop variable gets a value from the sequence, statements inside the loop executes once.
4. **Iteration**
   • After executing the statements, the loop moves to the next element in the sequence.
   • Each time the loop variable takes a new value and the statement inside loop executes, it is called an **iteration**.
   • This process continues until all elements in the sequence are exhausted.
5. **End of Loop**
   • When there are no more elements in the sequence, the loop terminates.
   • The program then moves to the **Next Step** (the statement immediately after the loop).

**1. program**

```
ch = "BCA AI"
for i in ch:
    print(i)
```

**Output:**

```
B
C
A

A
I
```

**Explanation**

1. A string ch is defined with the value "BCA AI".
2. The for loop goes through the string character by character.
3. In each iteration:
   - The loop variable i takes the current character.
   - print(i) prints that character.
4. This continues until all characters in the string are printed.

**Range function [ range( ) ]**

The range( ) function in Python is a **built-in function** used to generate a sequence of numbers. It's often used with **loops**, especially for loops, to iterate over a sequence of numbers.

**Syntax:**

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

- ***start*** *(optional) – the starting number of the sequence (default is 0)*
- ***stop*** *(required) – the number at which the sequence ends (not included)*
- ***step*** *(optional) – the difference between each number in the sequence (default is 1)*

**Example Program 1**

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

**Output:**

```
0
1
2
3
4
```

**Explanation**

1. **range(5)**
   - This generates a sequence of numbers from **0 up to 4** (5 is not included).
   - So internally, Python creates: 0, 1, 2, 3, 4.
2. **for i in range(5):**
   - The for loop iterates over each number in the sequence generated by range(5).
   - In the first iteration, i = 0, then i = 1, and so on until i = 4.
3. **print(i)**
   - This prints the current value of i in each iteration.

**Example Program 2**

```
for i in range(2, 6):
    print(i)
```

**Output:**

```
2
3
4
5
```

**Explanation**

1. **range(2, 6)**
   - Generates a sequence of numbers starting from 2 up to 5 (6 is not included).
   - So the sequence is: 2, 3, 4, 5.
2. **for i in range(2, 6):**
   - The for loop iterates over each number in the sequence.
   - i takes the values 2, 3, 4, 5 in successive iterations.
3. **print(i)**
   - Prints the current value of i in each iteration.

**Example Program 3**

```
for i in range(1,10,2):
    print(i)
```

**Output:**

```
1
3
5
7
9
```

**Example Program**

```
for i in range(-5,0):
    print(i)
```

**Output:**

```
-5
-4
-3
-2
-1
```

We can use **break** and **continue** statements in **for loops** to control the flow of iteration.

**Program example of for and break statements**

```
for i in range(1, 6):
    if i == 4:
        break
    print(i)
```

**Output:**

```
1
2
3
```

**Program example of for and continue statements**

```
for i in range(1, 6):
    if i == 4:
        continue
    print(i)
```

**Output:**

```
1
2
3
5
```

**Exit function [ exit( ) ]**

The exit() function is helping used to **terminate the entire program execution immediately**. It is used directly in interactive sessions with the built-in exit() function.

**Syntax:**

```
statement 1
exit( )
statement 2
```

**Syntax:**

```
statement 1
exit("error message")
statement 2
```

**Example:**

```
print("Program starts")
exit()  # Terminates the program
print("This line will not be executed")
```

**Output:**

```
Program starts
```

**Explanation:**

- The program prints "Program starts".
- **exit()** stops the program immediately.
- The last print statement is **never executed**.

**Errors**

Errors are problems in a program that prevent it from running correctly.
Errors occur when the program **violates the rules of the programming language** or performs an **invalid operation**.

**Types of error**

**1. Syntax Error (Parsing Error)**

- A syntax error occurs when the programmer writes code **without following rules of python grammar.**
- Python **detects syntax errors before running the program**, so the program cannot execute until the error is fixed.
- These errors are also called **parsing error.**
- **Examples**

```
print("Hello World"   # Missing closing parenthesis
   Error Message: SyntaxError: unexpected EOF while parsing
```

```
if 5 > 2
    print("5 is greater")  # Missing colon
Error Message: SyntaxError: expected ':'
```

**2. Semantic Error (Logical Error)**

- Semantic errors occur when a **program runs without crashing** but produces incorrect results due to a mistake in the **program's logic**.
- Python does not show any error message for semantic errors; the output is just wrong.
- These errors are also called as **logical errors.**
- **Examples**

```
# Program to calculate area of a rectangle
length = 5
width = 10
area = length + width  # Mistake: should be length * width
print("Area of rectangle:", area)
```

**Output:**

```
Area of rectangle: 15 but wrong
```

**3. Runtime Errors (Exceptions)**

- Runtime errors are error **occurs while the program is executing**, even if the **syntax is correct**.
- Python shows an error message when a runtime error occurs.
- These errors are also called as **exceptions.**
- Examples include **division by zero**, accessing an invalid index, or using an undefined variable.

```
x = int(input("Enter the value of x: "))
y = 0
print(x / y)  # Division by zero
print("This line will not execute")
```

**Output**

```
Enter the value of x: 10
Traceback (most recent call last):
  File "<stdin>", line 3, in
<module>
ZeroDivisionError: division by zero
```

**Explanation**

1. **x = int(input("Enter the value of x: "))**
   - The program asks the user to enter a number.
   - Suppose the user enters 10, then x = 10.
2. **y = 0**
   - y is set to 0.
3. **print(x / y)**
   - Dividing by zero is **not allowed** in Python.
   - Python raises a **ZeroDivisionError**.
4. **print("This line will not execute")**
   - This line is **never executed** because the program **stops immediately** at the division by zero.

**1. ZeroDivisionError**

Occurs when a number is divided by zero.

```
x = 10
y = 0
print(x / y)  # Division by zero
```

**Output:**
```
ZeroDivisionError: division by zero
```

**2. ValueError**
Occurs when an operation receives a value of the correct type but inappropriate value.
```
num = int("abc")  # Cannot convert string to integer
```
**Output:**
```
ValueError: invalid literal for int() with base 10: 'abc'
```

**3. IndexError**
Occurs when trying to access an element outside the range of a list or tuple.
```
lst = [1, 2, 3]
print(lst[5])
```
**Output:**
```
IndexError: list index out of range
```

**4. KeyError**
Occurs when trying to access a key that doesn't exist in a dictionary.
```
my_dict = {"name": "Alice"}
print(my_dict["age"])
```
**Output:**
```
KeyError: 'age'
```

**5. NameError**
Occurs when using a variable that has not been defined.
```
print(a)
```
**Output:**
```
NameError: name 'a' is not defined
```

**6. TypeError**
Occurs when an operation is applied to an object of inappropriate type.
```
x = 5 + "10"  # Cannot add int and string
```
**Output:**
```
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Some errors, like **logical errors** and **syntax errors**, are easy to fix because we know the program's logic and Python syntax. But **exceptions** are **runtime errors** that happen while the program is running. We can handle these errors using **exception handling** so the program does not crash.

**Exception Handling**

**Exception Handling** is way of handling **runtime errors without stopping the execution of program.**

- Handles runtime errors.
- Prevents the program from crashing.
- Execute the program even **if runtime errors** occur.

**Syntax**
```
try:
    statement 1 # Code that may cause an exception
except ExceptionType:
    statement 2 # Handle the exception
except ExceptionType:
    statement 3 # Executes if no exception occurs
finally:
   statement 4 # Executes always, whether exception occurs or not
```
**Explanation**

**try block**

The **try block** contains the code that **might cause** an exception.

- The code inside try block executes first.
- If no error occurs → the rest of the code runs normally.
- If an error occurs → control is transferred to the except block.

**except block**

The **except block** is used to **catch and handle** exceptions.

- Prevents the program from **crashing**.
- Can handle specific exceptions or use a general handler and can have more than two except block will possible.

**finally block**

The **finally block** contains code that **always executes, whether an exception occurs or not**.

**Example: 1**

```
try:
    x = int(input("Enter the value of x: "))   # risky code
    y = int(input("Enter the value of y: "))   # risky code
    print("Result:", x / y)   # may cause ZeroDivisionError
except ZeroDivisionError:
    print("Error: Division by zero is not allowed")   # handles division by
zero
finally:
    print("finally block executed")   # runs always
```

**Output:**

**Case 1: Normal Execution (No Exception)**

```
Enter the value of x: 10
Enter the value of y: 2
Result: 5.0
finally block executed
```

If you don't know what type of error may occur, type). This will catch any kind of exception.

**Case 2: Division by Zero (Exception occurs)**

```
Enter the value of x: 10
Enter the value of y: 0
Error: Division by zero is not allowed
finally block executed
```

**Example: 2**

```
try:
    x = int(input("Enter the value of x: "))
    y = int(input("Enter the value of y: "))
    print(x / y)
except Exception as e:   # catches any error
    print("An error occurred:", e)
finally:
    print("finally block executed")
```

**Outputs**

**Case 1: Normal execution**

```
Enter the value of x: 10
Enter the value of y: 2
5.0
finally block executed
```

**Case 2: Division by zero**

```
Enter the value of x: 10
Enter the value of y: 0
An error occurred: division by zero
finally block executed
```

**Case 3: Invalid input**

```
Enter the value of x: abc
An error occurred: invalid literal for int() with base 10: 'abc'
finally block executed
```