
Naming Conventions in Python

Prepared By: Bhesh Raj Pokhrel

Identifier in Python

- A Python identifier is a **name used to identify** a **variable**, function, class, **module**, or other objects. This name can be
 - class name
 - package name
 - variable name
 - ***module-name***
 - function name
 - method name

Note:

A **module** is simply a **Python file** (.py) that contains **functions, classes, or variables** we can reuse in other programs.

In Python, **module-name** refers to the **name of the file** (without the .py extension) that use in program with **import** keyword.

Example:

Module:

```
# math_utils.py
def add(a, b):
    return a + b
```

Module-Name:

```
import math_utils

result = math_utils.add(3, 4)
print(result)
```

Rules to define identifiers in Python:

- The only **allowed characters** to write identifier in python are,
 - **Alphabets** can be either **lowercase** or **uppercase**.
 - **Digits** (0 to 9)
 - **Underscore** symbol (`_`)
 - Usage of any **other symbol** (like `$`, `!`, `#`, `@`) will result in **Syntax Error**.
- Identifiers **allow digits**, but identifiers **should not start with digits**.
- Identifiers are **case-sensitive**.
- We **cannot use keywords** as identifiers.
- **Spaces are not allowed** between identifiers.

Examples of Identifiers in Python:

- **435student** → **invalid** (Should not start with a digit)
- **student564** → valid
- **student565info** → valid
- **\$tudent** → **invalid** (Should not start with symbol)
- **_student_info** → valid (Can start with underscore)
- **class** → **invalid** (**Keyword** can't be used as identifier)
- **def** → **invalid** (**Keyword** can't be used as identifier)

Identifiers: Naming Convention

1. class	a. class names should start with uppercase and remaining letters are in lowercase. b. If a name has multiple words, then every inner word should start with an uppercase letter. Example: StudentInfo. c. INFO: This rule is applicable for classes created by users only; the in-built class names used all are in lowercase.
2. package 3. module 4. variable 5. function 6. method	a. Names should be in lower case. b. If a name has multiple words, then separating words with underscore () is good practice. Example: one_two
7. Non-public instance variables	a. Non-public instance variables should begin with underscore (), we can say private data. Example: _one
8. Constants	a. Constants names should be written in all capital letters. b. If a name has multiple words, then separating words with underscore () is good practice. Example: ONE_TWO
9. Non-accessible entities	a. There are some variables and functions which should be accessed from outside. b. Those variables and function names should start with double underscores (dud) symbols. Example: __init__

Imports: Convention

- Imports should usually be on **separate lines**.
- Imports should be grouped in the **following order**:
 1. **Standard** library imports.
 2. **Related third party** imports.
 3. **Local application/library** specific imports.
- Absolute imports are recommended,

```
import mypkg.sibling  
from mypkg import sibling  
from mypkg.sibling import example
```

Example

```
import math
import random

print("Absolute:", abs(-7))
print("Square root:", math.sqrt(16))
print("Factorial:", math.factorial(4))
print("Power:", pow(2, 5))
# Random integer between 1 and 10
print(random.randint(1, 10))
# Random float between 1 and 5
print(random.uniform(1, 5))
```

Indentation in Python

- In Python, we **need to group the statements** by using **indentation**.
 - Indentation keeps the **group of statements separate**.
 - The recommended indentation is **4 spaces**.
 - We must follow the **order of indentation** otherwise we will get ***IndentationError***

valid Indentation

Code:

```
print("statement one")
print("statement two")
print("statement three")
```

Invalid Indentation

Code:

```
print("statement one")
    print("statement two")
print("statement three")
```

Output: **IndentationError: unexpected indent**

Invalid Indentation

Code:

```
print("statement one")
    print("statement two")
        print("statement three")
```

Comments in Python:

1. Comments are **useful to describe** the code in an easy way.
 2. Python ignores comments while running the program.
 3. To comment python code, we can use hash **symbol #**
 4. For **multi-line** comments, we can use **single/double** triple quotation marks.
- **Comments Example in Python:**

```
#This is basic python program  
#to print hello world  
print("Hello World in python");
```

Variable in Python

- All the **data which we create in the program** will be **saved in some memory location** on the system.
- **Variable is the name** that we **give to the memory location** which holds some data.
- With a variable, we can **store the data, access the data**, and also manipulate the data.
- **To summarize, a variable is a**
 - Name
 - Refers to a value
 - Hold some data
 - Name of a memory location

Properties of a variable in Python:

- **Type:** Each and every programming language has data types and each variable should belong to one of the data types.
- **Scope:** In any programming language, scope refers to the place or limit or boundaries or the part of the program within which the variable is accessible. Generally, the two types of scopes defined are the global scope and local scope.
- **Value:** Each and every variable in a program holds the value of some type that can be accessed or modified during the due flow of the program.
- **Location:** The data is stored in the memory location. The variable is the name that we give to that location. Each memory location has some address associated with it.
- **Lifetime:** The lifetime of a variable refers to the time period for which the memory location associated with the variable stores the data.

Creating a variable in Python

- Generally, in many programming languages, **while defining or creating a variable**, we **need the data type** and variable name and value. For example, in **C language** variable creation
- **int i = 10;**
- Python is a **dynamically typed** programming language. There is **no need of specifying the data type** for a variable. The program **automatically takes the data type** dynamically during the creation. In python, to create a variable we **just need to specify**
 - **Name** of the variable
 - **Assign a value** to the variable
- **Syntax to create variable in python:** **name_of_the_variable = value**
- **Note:** We can print **text messages** along with variables for better understanding. For that.
 - Text message we should keep in within double-quotes.
 - Text message and variable name should be separated by comma symbol.
 - **age = 16**
print("My age is sweet", age)

Invalid cases for variables:

- While defining variables in python,
 1. Variable names should be on the **left side**.
 2. The value should be on the right side.
 3. Violating this will result in a syntax error

Example 5: (creating a variable with keyword name)

Code:

```
if = 10  
print(if)
```

Output: **SyntaxError: invalid syntax**

Example 4: (creating variable in wrong direction)

Code:

```
10 = emp_id  
print(emp_id)
```

Output: **SyntaxError: can't assign to literal**

Variable names should not be keywords, else will result in an error.

Multiple variables in a single line in Python:

- We can assign **multiple variables** to **multiple values** in a single line.
- During the **assignment**, the variable name on the left-hand side **should be equal to** the values on the right-hand side. If not it results in an error.

Example 6: (assigning multiple variables in single line)

Code:

```
a,b,c = 10,20,30
```

```
print(a,b,c)
```

Output: 10 20 30

Example 7: (unequal items on either side)

Code:

```
a,b,c = 10,20
```

```
print(a,b,c)
```

Output: **ValueError: not enough values to unpack (expected 3, got 2)**

Example 8: (unequal items on either side)

Code:

```
a,b = 10,20,30
```

```
print(a,b,c)
```

Output: **ValueError: too many values to unpack (expected 2)**

Data Types in Python

- The **data** can be **categorized into different types** like integers, float numbers, boolean, string, etc.
- The **classification**, which states the **type of data stored in the variable** is called data type.
- Generally, data types can be classified into **two types**:

Built-in data types in Python:

Numeric types (int, Float, Complex)

bool

None

Str

Bytes

Bytearray

Tuple

List

Range

Set

dict

User-defined data types: Data types that are created by the programmer. Eg. **class, module, array** etc

Example: Data Types

```
#datatypes in python
```

```
age=24
```

```
weight=57.5
```

```
name="Ram Kumar"
```

```
married=True
```

```
marks=[50,40,60,70,80]
```

```
print("Type of Age :",type(age))
```

```
print("Type of weight : ", type(weight))
```

```
print("Type of Name :", type(name))
```

```
print("Type of Married :",type(married))
```

```
print("Type of marks :",type(marks))
```

```
C:\python_proj>py DataTypes.py
```

```
Type of Age : <class 'int'>
```

```
Type of weight : <class 'float'>
```

```
Type of Name : <class 'str'>
```

```
Type of Married : <class 'bool'>
```

```
Type of marks : <class 'list'>
```

Example: Bytearray

```
x = [10, 20, 100, 40, 15]
y = bytes(x)

print(type(y))

print("First index value: ", y[0]);

for num in y:
    print(num)
```

```
<class 'bytes'>
First index value:  10
10
20
100
40
15
```

Range Data Type in Python:

- We can create a **range of values** by using the **range()** function. The range data type represents a **sequence of numbers**. The range data type is **immutable** means we **cannot modify** it once it is created. Range data type values can be printed by using for loop.
- **Example: Creating a range of values using a range**

```
a=range(3,10)
print(a)
for x in a:
    print(x,end=" ")
```

```
range(3, 10)
3 4 5 6 7 8 9
```

- **Note:** A list of values can be created using the range.

Example: Creating a list

```
a=list(range(1, 10))
print(a)
```

Output: **[1, 2, 3, 4, 5, 6, 7, 8, 9]**

Assignment:

- Write a program to calculate the simple interest.
- Write a program to calculate area of circle.
- Write a program to print 1 to 100.

Operators in Python

- In programming languages, **an operator is a symbol** that is applied to some **operands** (usually variables), **to perform** certain actions or operations. For Example: **`c=a+b;`**

BASIC CLASSIFICATION OF OPERATORS IN PYTHON

- **Unary Operator** – If the **operator acts on a single operand** then it's called the unary operator. For example, in **'-5'** the operator **'-'** is used to make the (single) operand **'5'** a negative value hence called the unary operator.
- **Binary Operator** – If the **operator acts on two operands** then it's called a binary operator. For example, **+** operators need two variables(operands) to perform some operation, hence called binary operator.
- **Ternary Operator** – If the **operator acts on three operands** then it's called a ternary operator. In general, the ternary operator is a precise way of writing conditional statements.

Types Of Operators in Python:

- **Arithmetic Operators:** The operators which are **used to perform arithmetic operations** like addition, subtraction, division etc. They are: **+, -, *, /, %, **, //**
- **Relational Operators:** The operators which are **used to check for some relation** like **greater than** or **less than** etc.. between operands are called relational operators. They are: **<, >, <=, >=, ==, !=**
- **Logical Operators:** The operators which do some **logical operation on the operands** and return **True or False** are called logical operators. The logical operations can be **'AND', 'OR', 'NOT'** etc.
- **Assignment Operators:** The operators which are **used for assigning values** to variables. **'='** is the assignment operator.
- **Unary Minus Operator:** The operator **'-'** is called the Unary minus operator. It is used to **negate the number**.
- **Membership Operators:** The operators that are **used to check whether a particular variable is part of another variable or not**. They are **'in'** and **'not in'**
- **Identity Operators:** The operators which are **used to check for identity**. They are **'is'** and **'is not'**.

ARITHMETIC OPERATORS IN PYTHON:

- These operators are **used to perform that basic mathematical** operation as done in every programming language. Let's understand them with some examples. Let's assume, **a = 20 and b = 12**

Operator	Meaning	Example	Result
+	Addition	a+b	32
-	Subtraction	a-b	8
*	Multiplication	a*b	240
/	Division (Quotient of the division)	a/b	1.6666666666666667
%	Modulus (Remainder of division)	a%b	8
**	Exponent operator	a**b	4096000000000000
//	Integer division(gives only integer quotient)	a//b	1

Note: Division operator / always performs **floating-point arithmetic**, so it returns a float value. **Floor division (//)** can perform **both** floating-point and integral as well,

- If values are **int type**, the result is **int type**.
- If **at least one value** is float type, then the **result is of float type**.

Example: Arithmetic Operators

```
a = 20
b = 12
print(a+b)
print(a-b)
print(a*b)
print(a/b)
print(a%b)
print(a**b)
print(a//b)
#Example Floor Division
print(12//5)
print(12.0//5)
```

```
32
8
240
1.6666666666666667
8
409600000000000000
1
2
2.0
```

RELATIONAL OPERATORS IN PYTHON:

- These are **used to compare two values** for some relation and **return True or False** depending on the relation. Let's assume, a = 13 and b = 5

Operator	Example	Result
>	a>b	True
>=	a>=b	True
<	a<b	False
<=	a<=b	False
==	a==b	False
!=	a!=b	True

Example: Relational Operator

```
a = 13  
b = 5  
print(a>b)  
print(a>=b)  
print(a<b)  
print(a<=b)  
print(a==b)  
print(a!=b)
```

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

LOGICAL OPERATORS IN PYTHON

- In python, there are **three types of logical operators**. They are **and, or, not**.
- **Note:** In logical operators, False indicates **0(zero)** and True indicates **non-zero value**. Logical operators on boolean types

Operator	Example	Meaning
and	x and y	If x is False, returns x, else y.
or	x or y	If x is True, returns x, else y
not	not x	If x is False, returns True, else False

ASSIGNMENT OPERATORS IN PYTHON

- By using these operators, we **can assign values to variables**. '**=**' is the assignment operator used in python.
- There are some **compound operators (short-hand operators)** which are the combination of some arithmetic and assignment operators (**+=, -=, *=, /=, %=, **=, //=**). Assume that, a = 13 and b = 5

Operator	Example	Equal to	Result
=	x = a + b	x = a + b	18
+=	a += 5	a = a + 5	18
-=	a -= 5	a = a - 5	8

MEMBERSHIP OPERATORS IN PYTHON

- Membership operators are used to checking whether an element is present in a sequence of elements are not.
- Here, the **sequence** means **strings, list, tuple, dictionaries**, etc.
- There are **two membership operators** available in python i.e. **'in'** and **'not in'**.
- **'in' operator:** The in operators **returns True** if **element is found** in the collection of sequences. **returns False** if **not found**
- **'not in' operator:** The **not-in operator** **returns True** if the **element is not found** in the collection of sequence. **returns False** in found

Example: Membership Operator

```
text = "Welcome to python programming"
names = ["Kp", "Harka", "Balén", "Trump"]

print("Welcome" in text)
print("Harka" in text)
print("Kp" not in text)

print("Trump" in names)
print("Hari" in names)
print("Hema" not in names)
```

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

IDENTITY OPERATOR IN PYTHON

- This operator **compares the memory location(address) to two elements or variables or objects.**
- With these operators, we will be able to know whether the **two objects are pointing to the same location or not.**
- The **memory location of the object** can be seen using the **id()** function. For Example:

```
a = 25  
b = 25  
print(id(a))  
print(id(b))
```

```
10105856  
10105856
```

- **Types of Identity Operators in Python:**
- There are two identity operators in python, is and is not.
- **is:**
 - **A is B** returns **True**, if **both A and B are pointing to the same address.**
 - **A is B** returns **False**, if **both A and B are not pointing to the same address.**
- **is not:**
 - **A is not B** returns **True**, if both A and B are **not pointing to the same object.**
 - **A is not B** returns **False**, if both A and B are **pointing to the same object.**

Example: Identity Operators in Python

```
a = 25
b = 25
c=30
print(a is b)
print("address of a:",id(a))
print("address of b:",id(b))
print(a is c)
print("address of c:",id(c))
```

```
True
address of a: 140708560520872
address of b: 140708560520872
False
address of c: 140708560521032
```

Note: The **'is'** and **'is not'** operators are **not comparing** the **values** of the objects. They **compare the memory locations (address)** of the objects. If we want to compare the value of the objects, we should use the **relational operator '=='**.

In Python:

- **Small integers** between -5 and 256 are **cached** (integer caching) by the interpreter.
- So if two variables are assigned the **same value within that range**, they point to the **same memory location**.

Input and Output in Python

- A predefined function **input()** is available in python **to take input** from the keyboard during runtime and function **print()** is available **to display result in console**.
- **Input()** function takes a **value from the keyboard** and **returns it as a string type**. Based on the requirement **we can convert from string to other types**.

```
#program to add two numbers  
num1=input("Enter first Number : ")  
num2=input("Enter second Number :")  
  
num3=int(num1)+int(num2)  
  
print("num3 ",num3)
```

Convert to other data types from a string in Python

- We can **convert the string to other data types** using some **inbuilt functions**..

Function	Converts to
int(x)	Integer
float(x)	Floating-point
str(x)	String
bool(x)	Boolean
list(x)	List
tuple(x)	Tuple

eval() function in python

- This is an **in-built function** available in python, which takes the **strings as an input**.
- The strings which we pass to it **should, generally, be expressions**.
- The **eval()** function takes the **expression** in the form of a string and **evaluates it** and returns the result.

Examples,

- `eval('10 + 10') → 20`
- `eval('10 * 10') → 100`
- `eval('10 and 10') → 10`
- `eval('10 or 10') → 10`
- `eval('0 and 10') → 0`
- `eval('10 or 0') → 10`
- `eval('10 / 10') → 1.0`
- `eval('10 // 10') → 1`
- `eval('10 >= 10') → True`

Example: eval() function example in python

```
sum = eval(input("Enter expression: "))  
print(sum)
```

Output:

```
Enter expression: 4+5*6  
34
```

Command Line Arguments in Python:

- As we know to execute the python file we have to use command as “**python filename.py**”. This command we are executing thorough the **command prompt/terminal**.
- The **script is invoked** with this command and the execution starts.
- While invoking the script **we can pass the arguments to it**, which are called command-line arguments. Arguments are nothing **but parameters or values to the variables**.
- The command with the arguments will look like: **python filename.py 10 20**
- Our command is ‘**python**’ and the ‘**filename.py**’ ‘**10**’ ‘**20**’ are arguments to it.
- In order to use these arguments in our script/code, we should do the following **import from sys import argv**
- After including the above **import statement** in our file, the arguments which we pass from the command are available in an attribute called ‘**argv**’. argv will be a list of elements that can be accessed using indexes.
- argv[0] will always be the filename.py
- argv[1] will be 10 in this case
- argv[2] will be 20 in this case

Example: Command line arguments in python

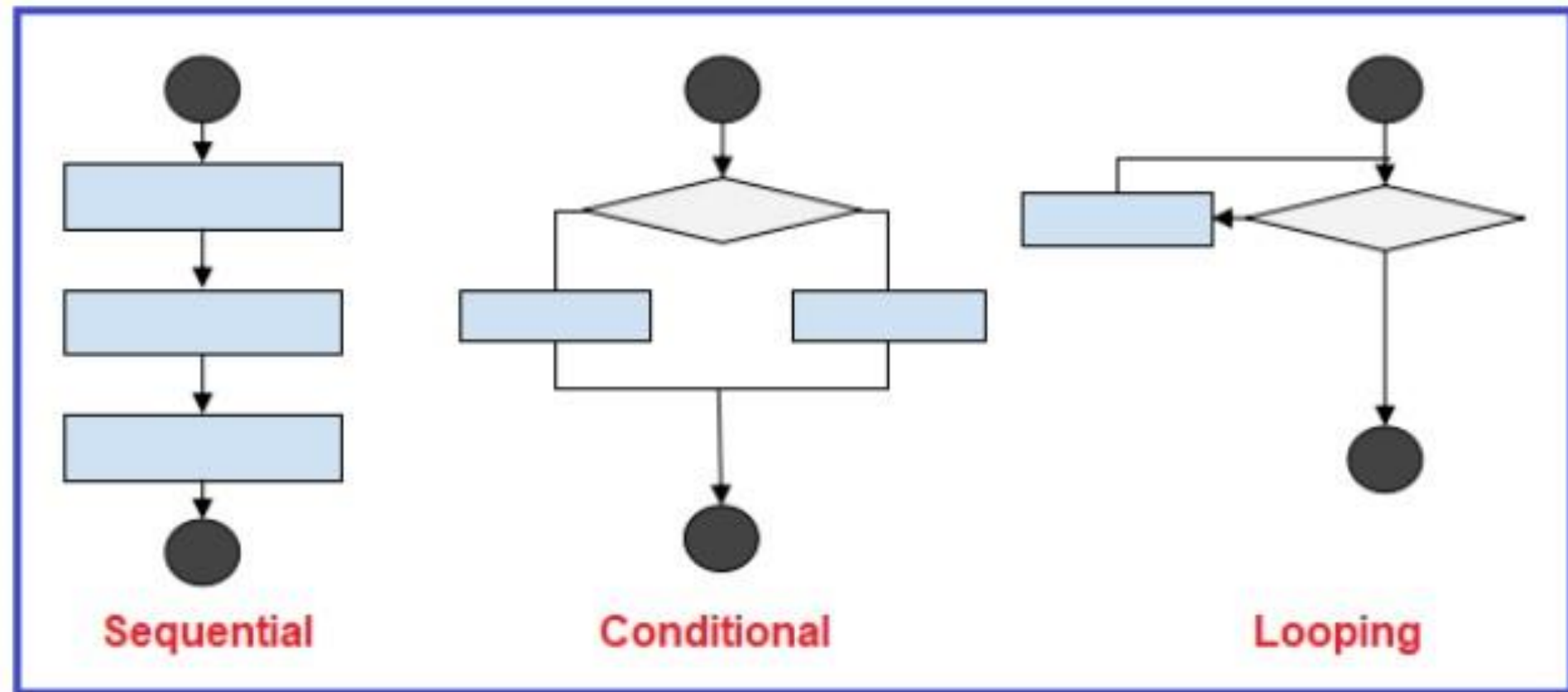
```
#import only argv variable  
#from sys module  
from sys import argv  
print(argv[0])  
print(argv[1])  
print(argv[2])
```

```
#import sys module  
import sys  
print(sys.argv[0])  
print(sys.argv[1])  
print(sys.argv[2])
```

Control Flow Statements in Python

- In programming languages, flow control means the **order in which the statements or instructions**, that we write, **get executed**.
- There are, generally, **three ways** in which the statements will be executed. They are,

1. **Sequential**
2. **Conditional**
3. **Looping**



Sequential Statement:

- In this type of execution flow, the statements are **executed one after the other sequentially**. By using sequential statements, we can develop simple programs

```
print("one")  
print("two")  
print("three")
```

Output:

```
one  
two  
three
```

Conditional Statement:

- Conditional statements are also called **decision-making statements**.
- Here the statements are executed **based on the condition**. As shown above in the flow graph, if the condition is **true** then **one set of statements** are executed, and if **false** then the **other set**.
- There are **three types** of conditional statements in python. They are as follows:
 1. **if statement**
 2. **if-else statement**
 3. **if elif else**
- **Note:** Python **does not have a built-in switch or case statement** like C, C++, or Java. However, Python 3.10 introduced the **match-case** statement (official switch alternative).

if statement

- The if statement contains an **expression/condition**. As per the syntax **colon (:) is mandatory** otherwise it throws a **syntax error**.
- The condition gives the result as a **bool type**, either **True** or **False**. If the condition result is **True**, then **if block statements** will be executed. If the condition result is **False**, then if block statements **won't execute**.

Syntax:

if condition:

 if block statements

out of if block statements

```
num = 1
print("num==1 condition gives: ", (num==1))
if num == 1:
    print("if block statements executed")
```

Output:

```
num==1 condition gives: True
if block statements executed
```

if-else statement in python:

- The if statement **contains an expression/condition**. As per the syntax **colon (:) is mandatory** after the condition and after the else otherwise it throws a syntax error. T
- The condition gives the result as **bool type**, which means either True or False. If the condition result is **True**, then **if block** statements will be executed. If the condition result is **False**, then **else block** statements will be executed.

Syntax

```
if condition:  
    if block statements  
else:  
    else block statements
```

```
num = 1  
print("num==1 condition gives: ", (num==1))  
if num == 1:  
    print("if block statements executed")
```

Output:

```
num==1 condition gives: True  
if block statements executed
```

if elif else statement in python:

- If one condition becomes **True** then the set of statements associated with that particular condition are executed and the execution comes out without checking other conditions.

Syntax

```
if condition1:  
    if block statements  
elif condition2:  
    elif block1 statements  
elif condition3:  
    elif block2 statements  
else:  
    else block statements
```

```
print("Please enter the values from 0 to 4")  
x=int(input("Enter a number: "))  
  
if x==0:  
    print("You entered:", x)  
elif x==1:  
    print("You entered:", x)  
elif x==2:  
    print("You entered:", x)  
elif x==3:  
    print("You entered:", x)  
elif x==4:  
    print("You entered:", x)  
else:  
    print("Beyond the range than specified")
```

Output1:

```
Please enter the values from 0 to 4  
Enter a number: 3  
You entered: 3
```

Match case in python

- The match-case statement in Python, introduced in **Python 3.10**, is a powerful control flow structure for pattern matching, similar to the **switch-case** statement found in other languages **but more expressive** and flexible.

How It Works

- The **match keyword** is followed by an **expression** whose value is matched against multiple case patterns.
- The **first case pattern** that matches the expression triggers execution of its associated code block.
- If no patterns match, a **wildcard case (case _:)** can be used as a **default**.

Basic Syntax

python

```
match expression:
    case pattern1:
        # code block for pattern1
    case pattern2:
        # code block for pattern2
    case _:
        # default code block (matches anything)
```

Match case in python

- **Key Features**
- **Structural Pattern Matching:** Can match complex data structures like sequences, tuples, dictionaries, and even class instances.
- **Multiple Patterns:** Use | (or) to match multiple values in one case.
- **Guards:** Use **if conditions** within cases to refine matches.
- **Wildcard _:** Acts as a catch-all default case.
- **Extracting Values:** Can bind parts of the matched data to variables for use inside the case block.

#Match case Simple Example

```
def check_number(x):  
    match x:  
        case 10:  
            print("It's 10")  
        case 20:  
            print("It's 20")  
        case _:  
            print("It's neither 10 nor 20")  
  
check_number(10) # Output: It's 10  
check_number(30) # Output: It's neither 10 nor 20
```

Match case in python

```
#Example of Multiple Patterns with | Operator
def check_number(x):
    match x:
        case 1 | 2 | 3:
            print("x is 1, 2, or 3")
        case 4 | 5:
            print("x is 4 or 5")
        case _:
            print("x is something else")

check_number(2) # Output: x is 1, 2, or 3
check_number(5) # Output: x is 4 or 5
check_number(10) # Output: x is something else
```

```
#Example with Tuples and Guards
point = (0, 4)

match point:
    case (0, 0):
        print("Origin")
    case (0, y):
        print(f"Y={y} on y-axis")
    case (x, 0):
        print(f"X={x} on x-axis")
    case (x, y) if x == y:
        print("On the line x=y")
    case _:
        print("Somewhere else")
```

Looping(iteration)

- A set of statements are **executed repeatedly** until the condition becomes false. Once the **condition becomes false** then the execution comes out of that loop.
- There are two types of loops available in python. They are:
 1. **while loop**
 2. **for loop**
- **Note:** Python **does not have a built-in do...while loop** like C, C++, or Java. However, you can **simulate** a do...while loop using a while True loop with a break.

While loop in Python:

- The while loop contains an **expression/condition**. As per the syntax **colon (:) is mandatory** otherwise it throws a syntax error.
- The condition gives the **result as bool type**, either **True or False**.
- The loop keeps on executing the statements **until** the condition becomes **False**.

Syntax
while condition:
statements

```
x=10
while (x>=10) and (x<=20):
    print(x)
    x+=2
print("End")
```

Parts of while loop in Python:

Initialization: This is the **first part** of the while loop. Before entering the condition section, some initialization is required

Condition: Once the initializations are done, then it will **go for condition checking** which is the heart of the while loop. The condition is checked and if it returns **True**, then **execution enters the loop** for executing the statements inside.

Increment/Decrement section: This is the section where **the iterator is increased** or decreased. Generally, we use **arithmetic operators** in the loop for this section

for loop in python:

- Basically, a for loop is **used to iterate elements one by one** from **sequences** like string, list, tuple, etc.
- This loop can be **easily understood** when compared to the while loop. While iterating elements from the sequence we can perform operations on every element.

Syntax

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

```
x = [10, 20, 30, "Python"]  
for i in x:  
    print(i)
```

- **Note:** the classic **C-style for loop** (for(i = 0; i < n; i++)) is **not supported** in Python.

Classic C-style for loop

```
for (int i = 0; i < 5; i++) {  
    printf("%d\n", i);  
}
```

Equivalent Python Loop

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

NESTED LOOPS IN PYTHON

- If a loop is written **inside another loop** then it is called a **nested loop**. Example.

```
rows = range(1, 5)
for x in rows:
    for star in range(1, x+1):
        print('*', end=' ')
    print()
```

LOOPS with ELSE block in Python:

- In Python, both **for** and **while** loops can include an **else block**, which executes when the **loop completes normally** (i.e., without being interrupted by a **break** statement).
- This is **distinct** from the **else clause in if statements**. The **else block** runs **only if the** loop finishes its iterations **without encountering a break statement**.
- **Note:** These loops with **else blocks** are best suitable in the scenarios, where we **need to search an element in the list**. When the element is **not present in the list** then the **else part will be** executed saying no such element in the list.

Example

```
values = range(5)
for x in values:
    print("item is available")
else:
    print("sorry boss, item is not available")
```

Output:

```
item is available
item is available
item is available
item is available
item is available
sorry boss, item is not available
```

```
group = [1, 2, 3, 4]
search = int(input("Enter the element in search: "))
for element in group:
    if search == element:
        print("element found in group")
        break
else:
    print("Element not found")
```

Output1:

```
Enter the element in search: 2
element found in group
```

Output2:

```
Enter the element in search: 6
Element not found
```

CONTINUE statement in Python

- The continue statement when encountered **will skip** the **current iteration** and the loop proceeds for the next iteration without executing the statements, which are followed by after the continue statement.

```
cart=[10, 20, 500, 700, 50, 60]
for item in cart:
    if item>=500:
        continue
    print("Item: ",item)
```

Output:

```
Item: 10
Item: 20
Item: 50
Item: 60
```

PASS statement in Python:

- The pass statement is a **placeholder**. It **does nothing** when executed. We use it when Python **expects a statement**, but you **don't want to write any code there yet**.

```
num = [10, 20, 30, 400, 500, 600]
for i in num:
    if i < 100:
        print("pass statement executed")
        pass
    else:
        print("Give festival coupon for these guys who bought: ", i)
```

```
for i in range(5):
    if i == 3:
        pass # Do nothing when i is 3
    print(i)
```

Output:

```
pass statement executed
pass statement executed
pass statement executed
Give festival coupon for these guys who bought: 600
```

RETURN statement in Python:

- The return statement in Python is **used inside a function to exit the function** and send a **value back to the caller**.
- When a return statement is executed, the **function terminates immediately**, and the specified value is returned. If no value is specified, the function returns **None by default**.
- **Key Points**
 - The return statement **ends the function execution**.
 - It can return any Python **object: numbers, strings, lists, dictionaries, tuples, or even other functions**.
 - You can **return multiple values** by separating them **with commas**; Python returns **them as a tuple**.
 - If no return is specified, the **function returns None**.

Example: Return Statement

```
#Returning a single value:  
def add(a, b):  
    return a + b  
  
result = add(3, 5)  
print(result)  # Output: 8
```

```
#Returning multiple values (as a tuple):  
def get_name_age():  
    name = "Alice"  
    age = 30  
    return name, age  
  
n, a = get_name_age()  
print(n)  # Output: Alice  
print(a)  # Output: 30
```

```
#Returning a function from another function:  
def outer():  
    def inner():  
        return "Hello from inner function!"  
    return inner  
  
func = outer()  
print(func())  # Output: Hello from inner function!
```

Practical Session
