**Fundamental of Python**

# Python Variables

Variable is a name that is used to refer to memory location. Python variable is also known as an identifier and used to hold value.

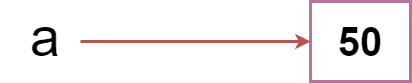
## **Identifier Naming**

Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

* The first character of the variable must be an alphabet or underscore ( \_ ).
* All the characters except the first character may be an alphabet of lower-case(a-z), upper-case (A-Z), underscore, or digit (0-9).
* Identifier name must not contain any white-space, or special character (!, @, #, %, ^, &, \*).
* Identifier name must not be similar to any keyword defined in the language.
* Identifier names are case sensitive; for example, my name, and MyName is not the same.
* Examples of valid identifiers: a123, \_n, n\_9, etc.
* Examples of invalid identifiers: 1a, n%4, n 9, etc.

Let's understand the following example

1. a = 50



## **Variable Names**

We have already discussed how to declare the valid variable. Variable names can be any length can have uppercase, lowercase (A to Z, a to z), the digit (0-9), and underscore character(\_). Consider the following example of valid variables names.

1. name = "Devansh"
2. age = 20
3. marks = 80.50
5. print(name)
6. print(age)
7. print(marks)

**Output:**

Devansh

20

80.5

Consider the following valid variables name.

1. name = "A"
2. Name = "B"
3. naMe = "C"
4. NAME = "D"
5. n\_a\_m\_e = "E"
6. \_name = "F"
7. name\_ = "G"
8. \_name\_ = "H"
9. na56me = "I"
11. print(name,Name,naMe,NAME,n\_a\_m\_e, NAME, n\_a\_m\_e, \_name, name\_,\_name, na56me)

**Output:**

A B C D E D E F G F I

## **Multiple Assignment**

Python allows us to assign a value to multiple variables in a single statement, which is also known as multiple assignments.

We can apply multiple assignments in two ways, either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Consider the following example.

**1. Assigning single value to multiple variables**

**Eg:**

1. x=y=z=50
2. print(x)
3. print(y)
4. print(z)

**Output:**

50

50

50

**2. Assigning multiple values to multiple variables:**

**Eg:**

1. a,b,c=5,10,15
2. print a
3. print b
4. print c

**Output:**

5

10

15

## **Standard data types**

A variable can hold different types of values. For example, a person's name must be stored as a string whereas its id must be stored as an integer.

Python provides various standard data types that define the storage method on each of them. The data types defined in Python are given below.

1. [Numbers](https://www.javatpoint.com/python-data-types#numbers)
2. [Sequence Type](https://www.javatpoint.com/python-data-types#SequenceType)
3. [Boolean](https://www.javatpoint.com/python-data-types#Boolean)
4. [Set](https://www.javatpoint.com/python-data-types#Set)
5. [Dictionary](https://www.javatpoint.com/python-data-types#dictionary)



# Python Operators

The operator can be defined as a symbol which is responsible for a particular operation between two operands. Operators are the pillars of a program on which the logic is built in a specific programming language. Python provides a variety of operators, which are described as follows.

* Arithmetic operators
* Comparison operators
* Assignment Operators
* Logical Operators
* Bitwise Operators
* Membership Operators
* Identity Operators

## **Arithmetic Operators**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| + (Addition) | It is used to add two operands. For example, if a = 20, b = 10 => a+b = 30 |
| - (Subtraction) | It is used to subtract the second operand from the first operand. If the first  operand is less than the second operand, the value results negative.  For example, if a = 20, b = 10 => a - b = 10 |
| / (divide) | It returns the quotient after dividing the first operand by the second operand.  For example, if a = 20, b = 10 => a/b = 2.0 |
| \* (Multiplication) | It is used to multiply one operand with the other.  For example, if a = 20, b = 10 => a \* b = 200 |
| % (reminder) | It returns the reminder after dividing the first operand by the second operand.  For example, if a = 20, b = 10 => a%b = 0 |
| \*\* (Exponent) | It is an exponent operator represented as it calculates the first operand power  to the second operand. |
| //(Floor division) | It gives the floor value of the quotient produced by dividing the two operands. |

Arithmetic operators are used to perform arithmetic operations between two operands. It includes + (addition), - (subtraction), \*(multiplication), /(divide), %(reminder), //(floor division), and exponent (\*\*) operators.

## **Comparison operator**

Comparison operators are used to comparing the value of the two operands and returns Boolean true or false accordingly. The comparison operators are described in the following table.

|  |  |  |  |
| --- | --- | --- | --- |
| **Operator** | **Description** | | |
| == | | If the value of two operands is equal, then the condition becomes true. |
| != | | If the value of two operands is not equal, then the condition becomes true. |
| <= | | If the first operand is less than or equal to the second operand, then the condition  becomes true. |
| >= | | If the first operand is greater than or equal to the second operand, then the condition  becomes true. |
| > | | If the first operand is greater than the second operand, then the condition becomes true. |
| **<** | | If the first operand is less than the second operand, then the condition becomes true. |

## **Assignment Operators**

The assignment operators are used to assign the value of the right expression to the left operand. The assignment operators are described in the following table.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| = | It assigns the value of the right expression to the left operand. |
| += | It increases the value of the left operand by the value of the right operand and assigns  the modified value back to left operand. For example, if a = 10, b = 20 => a+ = b  will be equal to a = a+ b and therefore, a = 30. |
| -= | It decreases the value of the left operand by the value of the right operand and assigns  the modified value back to left operand. For example, if a = 20, b = 10 => a- = b  will be equal to a = a- b and therefore, a = 10. |
| \*= | It multiplies the value of the left operand by the value of the right operand and assigns  the modified value back to then the left operand.  For example, if a = 10, b = 20 => a\* = b will be equal to a = a\* b and therefore, a = 200. |
| %= | It divides the value of the left operand by the value of the right operand and assigns  the reminder back to the left operand.  For example, if a = 20, b = 10 => a % = b will be equal to a = a % b  and therefore, a = 0. |
| \*\*= | a\*\*=b will be equal to a=a\*\*b, for example, if a = 4, b =2, a\*\*=b will  assign 4\*\*2 = 16 to a. |
| //= | A//=b will be equal to a = a// b, for example, if a = 4, b = 3, a//=b will assign  4//3 = 1 to a. |

## **Bitwise Operators**

The bitwise operators perform bit by bit operation on the values of the two operands. Consider the following example.

**For example,**

1. **if** a = 7
2. b = 6
3. then, binary (a) = 0111
4. binary (b) = 0110
6. hence, a & b = 0011
7. a | b = 0111
8. a ^ b = 0100
9. ~ a = 1000

|  |  |
| --- | --- |
| **Operator** | **Description** |
| & (binary and) | If both the bits at the same place in two operands are 1, then 1 is copied to the result.  Otherwise, 0 is copied. |
| | (binary or) | The resulting bit will be 0 if both the bits are zero; otherwise, the resulting bit will be 1. |
| ^ (binary xor) | The resulting bit will be 1 if both the bits are different; otherwise, the resulting bit  will be 0. |
| ~ (negation) | It calculates the negation of each bit of the operand, i.e., if the bit is 0, the resulting  bit will be 1 and vice versa. |
| << (left shift) | The left operand value is moved left by the number of bits present in  the right operand. |
| >> (right shift) | The left operand is moved right by the number of bits present in the right operand. |

## **Logical Operators**

The logical operators are used primarily in the expression evaluation to make a decision. Python supports the following logical operators.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| and | If both the expression are true, then the condition will be true. If a and b are the two  expressions, a → true, b → true => a and b → true. |
| or | If one of the expressions is true, then the condition will be true. If a and b are the  two expressions, a → true, b → false => a or b → true. |
| not | If an expression **a** is true, then not (a) will be false and vice versa. |

## **Membership Operators**

Python membership operators are used to check the membership of value inside a Python data structure. If the value is present in the data structure, then the resulting value is true otherwise it returns false.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| in | It is evaluated to be true if the first operand is found in the second operand  (list, tuple, or dictionary). |
| not in | It is evaluated to be true if the first operand is not found in the second operand  (list, tuple, or dictionary). |

## **Identity Operators**

The identity operators are used to decide whether an element certain class or type.

|  |  |
| --- | --- |
| **Operator** | **Description** |
| is | It is evaluated to be true if the reference present at both sides point to the same object. |
| is not | It is evaluated to be true if the reference present at both sides do not point to the  same object. |

## **Operator Precedence**

|  |  |
| --- | --- |
| **Operator** | **Description** |
| \*\* | The exponent operator is given priority over all the others used in the expression. |
| ~ + - | The negation, unary plus, and minus. |
| \* / % // | The multiplication, divide, modules, reminder, and floor division. |
| + - | Binary plus, and minus |
| >> << | Left shift. and right shift |
| & | Binary and. |
| ^ | | Binary xor, and or |
| <= < > >= | Comparison operators (less than, less than equal to, greater than, greater then  equal to). |
| <> == != | Equality operators. |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

The precedence of the operators is essential to find out since it enables us to know which operator should be evaluated first. The precedence table of the operators in Python is given below.

# Python Comments

Python Comment is an essential tool for the programmers. Comments are generally used to explain the code. We can easily understand the code if it has a proper explanation. A good programmer must use the comments because in the future anyone wants to modify the code as well as implement the new module; then, it can be done easily.

In the other programming language such as C++, It provides the // for single-lined comment and /\*.... \*/ for multiple-lined comment, but Python provides the single-lined Python comment. To apply the comment in the code we use the hash(#) at the beginning of the statement or code.

Let's understand the following example.

1. # This is the print statement
2. **print**("Hello Python")

## **Multiline Python Comment**

We must use the hash(#) at the beginning of every line of code to apply the multiline Python comment. Consider the following example.

1. # First line of the comment
2. # Second line of the comment
3. # Third line of the comment

**Example:**

1. # Variable a holds value 5
2. # Variable b holds value 10
3. # Variable c holds sum of a and b
4. # Print the result
5. a = 5
6. b = 10
7. c = a+b
8. **print**("The sum is:", c)

**Output:**

The sum is: 15

## **Docstrings Python Comment**

The docstring comment is mostly used in the module, function, class or method. It is a documentation Python string. We will explain the class/method in further tutorials.

**Example:**

1. **def** intro():
2. """
3. This function prints Hello Joseph
4. """
5. **print**("Hi Joseph")
6. intro()

**Output:**

Hello Joseph

## **Python indentation**

Python indentation uses to define the block of the code. The other programming languages such as C, C++, and Java use curly braces {}, whereas Python uses an indentation. Whitespaces are used as indentation in Python.

Indentation uses at the beginning of the code and ends with the unintended line. That same line indentation defines the block of the code (body of a function, loop, etc.)

Generally, four whitespaces are used as the indentation. The amount of indentation depends on user, but it must be consistent throughout that block.

1. **for** i **in** range(5):
2. **print**(i)
3. **if**(i == 3):
4. **break**

**Python If-else statements**

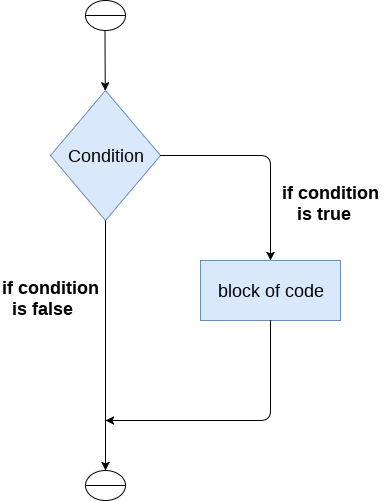
Decision making is the most important aspect of almost all the programming languages. As the name implies, decision making allows us to run a particular block of code for a particular decision. Here, the decisions are made on the validity of the particular conditions. Condition checking is the backbone of decision making.

In python, decision making is performed by the following statements.

|  |  |
| --- | --- |
| **Statement** | **Description** |
|  |  |
| If Statement | The if statement is used to test a specific condition. If the condition is true,  a block of code (if-block) will be executed. |
| If - else Statement | The if-else statement is similar to if statement except the fact that,  it also provides the block of the code for the false case of the  condition to be checked. If the condition provided in the if statement  is false, then the else statement will be executed. |
| Nested if Statement | Nested if statements enable us to use if ? else statement inside an  outer if statement. |

## **The if statement**

The if statement is used to test a particular condition and if the condition is true, it executes a block of code known as if-block. The condition of if statement can be any valid logical expression which can be either evaluated to true or false.



The syntax of the if-statement is given below.

1. **if** expression:
2. statement

### **Example 1**

1. num = int(input("enter the number?"))
2. **if** num%2 == 0:
3. **print**("Number is even")

**Output:**

enter the number?10

Number is even

### **Example 2 : Program to print the largest of the three numbers.**

1. a = int(input("Enter a? "));
2. b = int(input("Enter b? "));
3. c = int(input("Enter c? "));
4. **if** a>b **and** a>c:
5. **print**("a is largest");
6. **if** b>a **and** b>c:
7. **print**("b is largest");
8. **if** c>a **and** c>b:
9. **print**("c is largest");

**Output:**

Enter a? 100

Enter b? 120

Enter c? 130

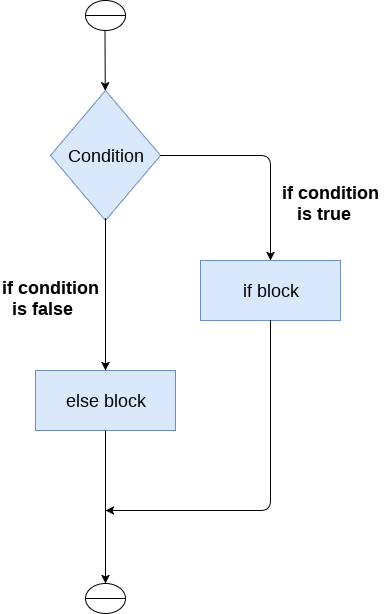
c is largest

## **The if-else statement**

The if-else statement provides an else block combined with the if statement which is executed in the false case of the condition.

If the condition is true, then the if-block is executed. Otherwise, the else-block is executed.

**Flowchart:**



The syntax of the if-else statement is given below.

1. **if** condition:
2. #block of statements
3. **else**:
4. #another block of statements (else-block)

### **Example 1 : Program to check whether a person is eligible to vote or not.**

1. age = int (input("Enter your age? "))
2. **if** age>=18:
3. **print**("You are eligible to vote !!");
4. **else**:
5. **print**("Sorry! you have to wait !!");

**Output:**

Enter your age? 90

You are eligible to vote !!

### **Example 2: Program to check whether a number is even or not.**

1. num = int(input("enter the number?"))
2. **if** num%2 == 0:
3. **print**("Number is even...")
4. **else**:
5. **print**("Number is odd...")

**Output:**

enter the number?10

Number is even

## **The elif statement**

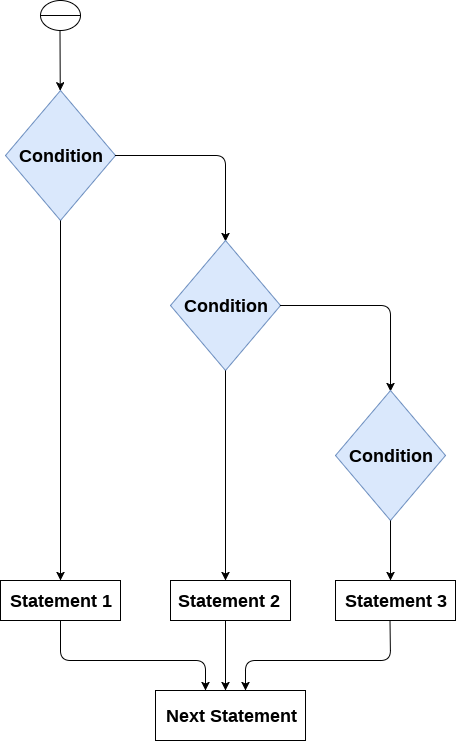
The elif statement enables us to check multiple conditions and execute the specific block of statements depending upon the true condition among them. We can have any number of elif statements in our program depending upon our need. However, using elif is optional.

The elif statement works like an if-else-if ladder statement in C. It must be succeeded by an if statement.

The syntax of the elif statement is given below.

1. **if** expression 1:
2. # block of statements
4. **elif** expression 2:
5. # block of statements
7. **elif** expression 3:
8. # block of statements
10. **else**:
11. # block of statements

**Flow chart:**



### **Example 1**

1. number = int(input("Enter the number?"))
2. **if** number==10:
3. **print**("number is equals to 10")
4. **elif** number==50:
5. **print**("number is equal to 50");
6. **elif** number==100:
7. **print**("number is equal to 100");
8. **else**:
9. **print**("number is not equal to 10, 50 or 100");

**Output:**

Enter the number?15

number is not equal to 10, 50 or 100

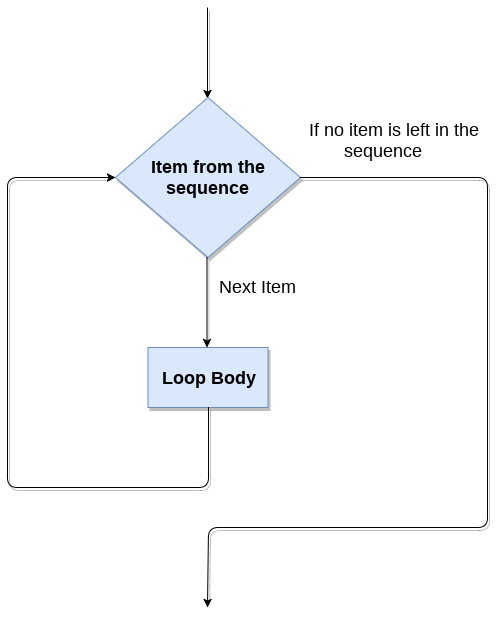
# Python for loop

The for **loop in Python** is used to iterate the statements or a part of the program several times. It is frequently used to traverse the data structures like list, tuple, or dictionary.

The syntax of for loop in python is given below.

1. **for** iterating\_var **in** sequence:
2. statement(s)

**The for loop flowchart**



### **For loop Using Sequence**

**Example-1: Iterating string using for loop**

1. str = "Python"
2. **for** i **in** str:
3. **print**(i)

**Output:**

P

y

t

h

o

n

**Example- 2: Program to print the table of the given number .**

1. list = [1,2,3,4,5,6,7,8,9,10]
2. n = 5
3. **for** i **in** list:
4. c = n\*i
5. **print**(c)

**Output:**

5

10

15

20

25

30

35

40

45

50s

**Example-4: Program to print the sum of the given list.**

1. list = [10,30,23,43,65,12]
2. sum = 0
3. **for** i **in** list:
4. sum = sum+i
5. **print**("The sum is:",sum)

**Output:**

The sum is: 183

### **For loop Using range() function**

**The range() function**

The **range()** function is used to generate the sequence of the numbers. If we pass the range(10), it will generate the numbers from 0 to 9. The syntax of the range() function is given below.

**Syntax:**

1. range(start,stop,step size)

* The start represents the beginning of the iteration.
* The stop represents that the loop will iterate till stop-1. The **range(1,5)** will generate numbers 1 to 4 iterations. It is optional.
* The step size is used to skip the specific numbers from the iteration. It is optional to use. By default, the step size is 1. It is optional.

Consider the following examples:

**Example-1: Program to print numbers in sequence.**

1. **for** i **in** range(10):
2. **print**(i,end = ' ')

**Output:**

0 1 2 3 4 5 6 7 8 9

**Example - 2: Program to print table of given number.**

1. n = int(input("Enter the number "))
2. **for** i **in** range(1,11):
3. c = n\*i
4. **print**(n,"\*",i,"=",c)

**Output:**

Enter the number 10

10 \* 1 = 10

10 \* 2 = 20

10 \* 3 = 30

10 \* 4 = 40

10 \* 5 = 50

10 \* 6 = 60

10 \* 7 = 70

10 \* 8 = 80

10 \* 9 = 90

10 \* 10 = 100

**Example-3: Program to print even number using step size in range().**

1. n = int(input("Enter the number "))
2. **for** i **in** range(2,n,2):
3. **print**(i)

**Output:**

Enter the number 20

2

4

6

8

10

12

14

16

18

We can also use the **range()** function with sequence of numbers. The **len()** function is combined with range() function which iterate through a sequence using indexing. Consider the following example.

1. list = ['Peter','Joseph','Ricky','Devansh']
2. **for** i **in** range(len(list)):
3. **print**("Hello",list[i])

**Output:**

Hello Peter

Hello Joseph

Hello Ricky

Hello Devansh

## **Nested for loop in python**

Python allows us to nest any number of for loops inside a **for** loop. The inner loop is executed n number of times for every iteration of the outer loop. The syntax is given below.

**Syntax**

1. **for** iterating\_var1 **in** sequence:  #outer loop
2. **for** iterating\_var2 **in** sequence:  #inner loop
3. #block of statements
4. #Other statements

## **Example- 1: Nested for loop**

1. # User input for number of rows
2. rows = int(input("Enter the rows:"))
3. # Outer loop will print number of rows
4. **for** i **in** range(0,rows+1):
5. # Inner loop will print number of Astrisk
6. **for** j **in** range(i):
7. **print**("\*",end = '')
8. **print**()

**Output:**

Enter the rows:5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

### **Example-2: Program to number pyramid.**

1. rows = int(input("Enter the rows"))
2. **for** i **in** range(0,rows+1):
3. **for** j **in** range(i):
4. **print**(i,end = '')
5. **print**()

**Output:**

1

22

333

4444

55555

# Python While loop

The Python while loop allows a part of the code to be executed until the given condition returns false. It is also known as a pre-tested loop.

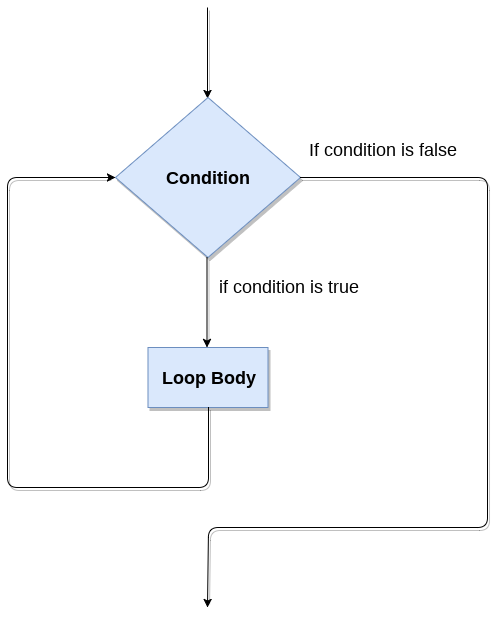
It can be viewed as a repeating if statement. When we don't know the number of iterations then the while loop is most effective to use.

The syntax is given below.

1. **while** expression:
2. statements

Here, the statements can be a single statement or a group of statements. The expression should be any valid Python expression resulting in true or false. The true is any non-zero value and false is 0.

**While loop Flowchart**



## **Loop Control Statements**

We can change the normal sequence of **while** loop's execution using the loop control statement. When the while loop's execution is completed, all automatic objects defined in that scope are demolished. Python offers the following control statement to use within the while loop.

### **Example-1: Program to print 1 to 10 using while loop**

1. i=1
2. #The **while** loop will iterate until condition becomes **false**.
3. While(i<=10):
4. print(i)
5. i=i+1

**Output:**

1

2

3

4

5

6

7

8

9

10

### **Example -2: Program to print table of given numbers.**

1. i=1
2. number=0
3. b=9
4. number = **int**(input("Enter the number:"))
5. **while** i<=10:
6. print("%d X %d = %d \n"%(number,i,number\*i))
7. i = i+1

**Output:**

Enter the number:10

10 X 1 = 10

10 X 2 = 20

10 X 3 = 30

10 X 4 = 40

10 X 5 = 50

10 X 6 = 60

10 X 7 = 70

10 X 8 = 80

10 X 9 = 90

10 X 10 = 100

## **Infinite while loop**

If the condition is given in the while loop never becomes false, then the while loop will never terminate, and it turns into the **infinite while loop.**

Any **non-zero** value in the while loop indicates an **always-true** condition, whereas zero indicates the always-false condition. This type of approach is useful if we want our program to run continuously in the loop without any disturbance.

### **Example 1**

1. **while** (1):
2. print("Hi! we are inside the infinite while loop")

**Output:**

Hi! we are inside the infinite while loop

Hi! we are inside the infinite while loop

### **Example 2**

1. var = 1
2. **while**(var != 2):
3. i = **int**(input("Enter the number:"))
4. print("Entered value is %d"%(i))

**Output:**

Enter the number:10

Entered value is 10

Enter the number:10

Entered value is 10

Enter the number:10

Entered value is 10

Infinite time

# Python break statement

The break is a keyword in python which is used to bring the program control out of the loop. The break statement breaks the loops one by one, i.e., in the case of nested loops, it breaks the inner loop first and then proceeds to outer loops. In other words, we can say that break is used to abort the current execution of the program and the control goes to the next line after the loop.

The break is commonly used in the cases where we need to break the loop for a given condition.

The syntax of the break is given below.

1. #loop statements
2. **break**;

## **Example 1**

1. list =[1,2,3,4]
2. count = 1;
3. **for** i **in** list:
4. **if** i == 4:
5. **print**("item matched")
6. count = count + 1;
7. **break**
8. **print**("found at",count,"location");

**Output:**

item matched

found at 2 location

## **Example 2**

1. str = "python"
2. **for** i **in** str:
3. **if** i == 'o':
4. **break**
5. **print**(i);

**Output:**

p

y

t

h

## **Example 3: break statement with while loop**

1. i = 0;
2. **while** 1:
3. **print**(i," ",end=""),
4. i=i+1;
5. **if** i == 10:
6. **break**;
7. **print**("came out of while loop");

**Output:**

0 1 2 3 4 5 6 7 8 9 came out of while loop

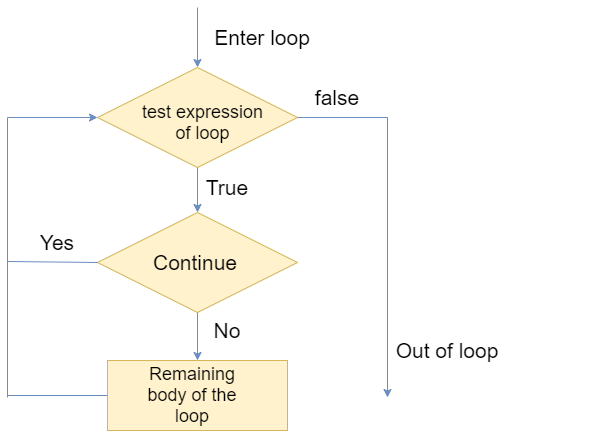
# Python continue Statement

The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.The continue statement in Python is used to bring the program control to the beginning of the loop. The continue statement skips the remaining lines of code inside the loop and start with the next iteration. It is mainly used for a particular condition inside the loop so that we can skip some specific code for a particular condition.

### **Syntax**

1. #loop statements
2. **continue**
3. #the code to be skipped

### **Flow Diagram**



Consider the following examples.

### **Example 1**

1. i = 0
2. **while**(i < 10):
3. i = i+1
4. **if**(i == 5):
5. **continue**
6. **print**(i)

**Output:**

1

2

3

4

6

7

8

9

10

Observe the output of above code, the value 5 is skipped because we have provided the **if condition** using with **continue statement** in while loop. When it matched with the given condition then control transferred to the beginning of the while loop and it skipped the value 5 from the code.

Let's have a look at another example:

### **Example 2**

1. str = "JavaTpoint"
2. **for** i **in** str:
3. **if**(i == 'T'):
4. **continue**
5. **print**(i)

**Output:**

J

a

v

a

p

o

i

n

t

# Python Pass

In Python, the pass keyword is used to execute nothing; it means, when we don't want to execute code, the pass can be used to execute empty. It is the same as the name refers to. It just makes the control to pass by without executing any code. If we want to bypass any code pass statement can be used.

It is beneficial when a statement is required syntactically, but we want we don't want to execute or execute it later. The difference between the comments and pass is that, comments are entirely ignored by the Python interpreter, where the pass statement is not ignored.

Suppose we have a loop, and we do not want to execute right this moment, but we will execute in the future. Here we can use the pass.

Consider the following example.

**Example - Pass statement**

1. # pass is just a placeholder for
2. # we will adde functionality later.
3. values = {'P', 'y', 't', 'h','o','n'}
4. **for** val **in** values:
5. **pass**

**Example - 2:**

1. **for** i **in** [1,2,3,4,5]:
2. **if**(i==4):
3. **pass**
4. **print**("This is pass block",i)
5. **print**(i)

**Output:**

1. 1
2. 2
3. 3
4. This **is** **pass** block 4
5. 4
6. 5

# Python List

A list in Python is used to store the sequence of various types of data. Python lists are mutable type its mean we can modify its element after it created. However, Python consists of six data-types that are capable to store the sequences, but the most common and reliable type is the list.

A list can be defined as a collection of values or items of different types. The items in the list are separated with the comma (,) and enclosed with the square brackets [].

A list can be define as below

1. L1 = ["John", 102, "USA"]
2. L2 = [1, 2, 3, 4, 5, 6]

IIf we try to print the type of L1, L2, and L3 using type() function then it will come out to be a list.

1. **print**(type(L1))
2. **print**(type(L2))

**Output:**

<class 'list'>

<class 'list'>

### **Characteristics of Lists**

The list has the following characteristics:

* The lists are ordered.
* The element of the list can access by index.
* The lists are the mutable type.
* The lists are mutable types.
* A list can store the number of various elements.

Let's check the first statement that lists are the ordered.

1. a = [1,2,"Peter",4.50,"Ricky",5,6]
2. b = [1,2,5,"Peter",4.50,"Ricky",6]
3. a ==b

**Output:**

False

Both lists have consisted of the same elements, but the second list changed the index position of the 5th element that violates the order of lists. When compare both lists it returns the false.

Lists maintain the order of the element for the lifetime. That's why it is the ordered collection of objects.

1. a = [1, 2,"Peter", 4.50,"Ricky",5, 6]
2. b = [1, 2,"Peter", 4.50,"Ricky",5, 6]
3. a == b

**Output:**

True

## **List indexing and splitting**

The indexing is processed in the same way as it happens with the strings. The elements of the list can be accessed by using the slice operator [].

The index starts from 0 and goes to length - 1. The first element of the list is stored at the 0th index, the second element of the list is stored at the 1st index, and so on.

# Python Dictionary

Python Dictionary is used to store the data in a key-value pair format. The dictionary is the data type in Python, which can simulate the real-life data arrangement where some specific value exists for some particular key. It is the mutable data-structure. The dictionary is defined into element Keys and values.

* Keys must be a single element
* Value can be any type such as list, tuple, integer, etc.

In other words, we can say that a dictionary is the collection of key-value pairs where the value can be any Python object. In contrast, the keys are the immutable Python object, i.e., Numbers, string, or tuple.

## **Creating the dictionary**

The dictionary can be created by using multiple key-value pairs enclosed with the curly brackets {}, and each key is separated from its value by the colon (:).The syntax to define the dictionary is given below.

**Syntax:**

1. Dict = {"Name": "Tom", "Age": 22}

In the above dictionary **Dict**, The keys **Name** and **Age** are the string that is an immutable object.

Let's see an example to create a dictionary and print its content.

1. Employee = {"Name": "John", "Age": 29, "salary":25000,"Company":"GOOGLE"}
2. **print**(type(Employee))
3. **print**("printing Employee data .... ")
4. **print**(Employee)

**Output**

<class 'dict'>

Printing Employee data ....

{'Name': 'John', 'Age': 29, 'salary': 25000, 'Company': 'GOOGLE'}

Python provides the built-in function **dict()** method which is also used to create dictionary. The empty curly braces {} is used to create empty dictionary.

1. # Creating an empty Dictionary
2. Dict = {}
3. **print**("Empty Dictionary: ")
4. **print**(Dict)
6. # Creating a Dictionary
7. # with dict() method
8. Dict = dict({1: 'Java', 2: 'T', 3:'Point'})
9. **print**("\nCreate Dictionary by using  dict(): ")
10. **print**(Dict)
12. # Creating a Dictionary
13. # with each item as a Pair
14. Dict = dict([(1, 'Devansh'), (2, 'Sharma')])
15. **print**("\nDictionary with each item as a pair: ")
16. **print**(Dict)

**Output:**

Empty Dictionary:

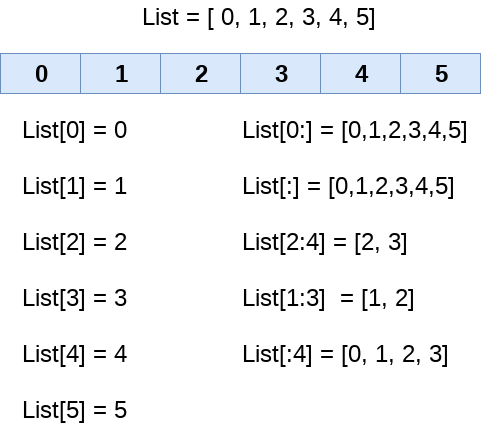
{}

Create Dictionary by using dict():

{1: 'Java', 2: 'T', 3: 'Point'}

Dictionary with each item as a pair:

{1: 'Devansh', 2: 'Sharma'}



We can get the sub-list of the list using the following syntax.

1. list\_varible(start:stop:step)

* The **start** denotes the starting index position of the list.
* The **stop** denotes the last index position of the list.
* The **step** is used to skip the nth element within a **start:stop**

Consider the following example:

1. list = [1,2,3,4,5,6,7]
2. **print**(list[0])
3. **print**(list[1])
4. **print**(list[2])
5. **print**(list[3])
6. # Slicing the elements
7. **print**(list[0:6])
8. # By default the index value is 0 so its starts from the 0th element and go for index -1.
9. **print**(list[:])
10. **print**(list[2:5])
11. **print**(list[1:6:2])

**Output:**

1

2

3

4

[1, 2, 3, 4, 5, 6]

[1, 2, 3, 4, 5, 6, 7]

[3, 4, 5]

[2, 4, 6]

Unlike other languages, Python provides the flexibility to use the negative indexing also. The negative indices are counted from the right. The last element (rightmost) of the list has the index -1; its adjacent left element is present at the index -2 and so on until the left-most elements are encountered.

Python Lists

Let's have a look at the following example where we will use negative indexing to access the elements of the list.

1. list = [1,2,3,4,5]
2. **print**(list[-1])
3. **print**(list[-3:])
4. **print**(list[:-1])
5. **print**(list[-3:-1])

**Output:**

5

[3, 4, 5]

[1, 2, 3, 4]

[3, 4]

# Python Tuple

Python Tuple is used to store the sequence of immutable Python objects. The tuple is similar to lists since the value of the items stored in the list can be changed, whereas the tuple is immutable, and the value of the items stored in the tuple cannot be changed.

## **Creating a tuple**

A tuple can be written as the collection of comma-separated (,) values enclosed with the small () brackets. The parentheses are optional but it is good practice to use. A tuple can be defined as follows.

1. T1 = (101, "Peter", 22)
2. T2 = ("Apple", "Banana", "Orange")
3. T3 = 10,20,30,40,50
5. print(type(T1))
6. print(type(T2))
7. print(type(T3))

**Output:**

<class 'tuple'>

<class 'tuple'>

<class 'tuple'>

### **Example - 1**

1. tuple1 = (10, 20, 30, 40, 50, 60)
2. print(tuple1)
3. count = 0
4. **for** i in tuple1:
5. print("tuple1[%d] = %d"%(count, i))
6. count = count+1

**Output:**

(10, 20, 30, 40, 50, 60)

tuple1[0] = 10

tuple1[1] = 20

tuple1[2] = 30

tuple1[3] = 40

tuple1[4] = 50

tuple1[5] = 60

### **Example - 2**

1. tuple1 = tuple(input("Enter the tuple elements ..."))
2. print(tuple1)
3. count = 0
4. **for** i in tuple1:
5. print("tuple1[%d] = %s"%(count, i))
6. count = count+1

**Output:**

Enter the tuple elements ...123456

('1', '2', '3', '4', '5', '6')

tuple1[0] = 1

tuple1[1] = 2

tuple1[2] = 3

tuple1[3] = 4

tuple1[4] = 5

tuple1[5] = 6

A tuple is indexed in the same way as the lists. The items in the tuple can be accessed by using their specific index value.

We will see all these aspects of tuple in this section of the tutorial.

## **Tuple indexing and slicing**

The indexing and slicing in the tuple are similar to lists. The indexing in the tuple starts from 0 and goes to length(tuple) - 1.

The items in the tuple can be accessed by using the index [] operator. Python also allows us to use the colon operator to access multiple items in the tuple.

Consider the following image to understand the indexing and slicing in detail.

# Python Set

A Python set is the collection of the unordered items. Each element in the set must be unique, immutable, and the sets remove the duplicate elements. Sets are mutable which means we can modify it after its creation.

Unlike other collections in Python, there is no index attached to the elements of the set, i.e., we cannot directly access any element of the set by the index. However, we can print them all together, or we can get the list of elements by looping through the set.

## **Creating a set**

The set can be created by enclosing the comma-separated immutable items with the curly braces {}. Python also provides the set() method, which can be used to create the set by the passed sequence.

### **Example 1: Using curly braces**

1. Days = {"Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"}
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

**Output:**

{'Friday', 'Tuesday', 'Monday', 'Saturday', 'Thursday', 'Sunday', 'Wednesday'}

<class 'set'>

looping through the set elements ...

Friday

Tuesday

Monday

Saturday

Thursday

Sunday

Wednesday

### **Example 2: Using set() method**

1. Days = set(["Monday", "Tuesday", "Wednesday", "Thursday", "Friday", "Saturday", "Sunday"])
2. **print**(Days)
3. **print**(type(Days))
4. **print**("looping through the set elements ... ")
5. **for** i **in** Days:
6. **print**(i)

**Output:**

{'Friday', 'Wednesday', 'Thursday', 'Saturday', 'Monday', 'Tuesday', 'Sunday'}

<class 'set'>

looping through the set elements ...

Friday

Wednesday

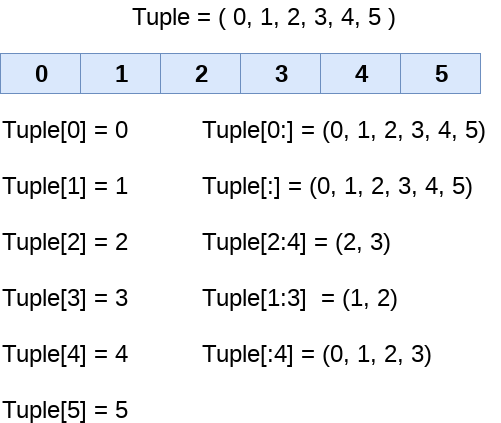
Thursday

Saturday

Monday

Tuesday

Sunday



Consider the following example:

1. tup = (1,2,3,4,5,6,7)
2. print(tup[0])
3. print(tup[1])
4. print(tup[2])
5. # It will give the IndexError
6. print(tup[8])

**Output:**

1

2

3

tuple index out of range

In the above code, the tuple has 7 elements which denote 0 to 6. We tried to access an element outside of tuple that raised an **IndexError**.

1. tuple = (1,2,3,4,5,6,7)
2. #element 1 to end
3. print(tuple[1:])
4. #element 0 to 3 element
5. print(tuple[:4])
6. #element 1 to 4 element
7. print(tuple[1:5])
8. # element 0 to 6 and take step of 2
9. print(tuple[0:6:2])

**Output:**

(2, 3, 4, 5, 6, 7)

(1, 2, 3, 4)

(1, 2, 3, 4)

(1, 3, 5)

## **Negative Indexing**

The tuple element can also access by using negative indexing. The index of -1 denotes the rightmost element and -2 to the second last item and so on.

The elements from left to right are traversed using the negative indexing. Consider the following example:

1. tuple1 = (1, 2, 3, 4, 5)
2. print(tuple1[-1])
3. print(tuple1[-4])
4. print(tuple1[-3:-1])
5. print(tuple1[:-1])
6. print(tuple1[-2:])

**Output:**

5

2

(3, 4)

(1, 2, 3, 4)

(4, 5)

# Python Function

Functions are the most important aspect of an application. A function can be defined as the organized block of reusable code, which can be called whenever required.

Python allows us to divide a large program into the basic building blocks known as a function. The function contains the set of programming statements enclosed by {}. A function can be called multiple times to provide reusability and modularity to the Python program.

The Function helps to programmer to break the program into the smaller part. It organizes the code very effectively and avoids the repetition of the code. As the program grows, function makes the program more organized.

Python provide us various inbuilt functions like **range()** or **print()**. Although, the user can create its functions, which can be called user-defined functions.

There are mainly two types of functions.

* **User-define functions** - The user-defined functions are those define by the **user** to perform the specific task.
* **Built-in functions** - The built-in functions are those functions that are **pre-defined** in Python.

In this tutorial, we will discuss the user define functions.

## **Advantage of Functions in Python**

There are the following advantages of Python functions.

* Using functions, we can avoid rewriting the same logic/code again and again in a program.
* We can call Python functions multiple times in a program and anywhere in a program.
* We can track a large Python program easily when it is divided into multiple functions.
* Reusability is the main achievement of Python functions.
* However, Function calling is always overhead in a Python program.

### **Creating a Function**

Python provides the **def** keyword to define the function. The syntax of the define function is given below.

**Syntax:**

1. **def** my\_function(parameters):
2. function\_block
3. **return** expression

Let's understand the syntax of functions definition.

* The **def** keyword, along with the function name is used to define the function.
* The identifier rule must follow the function name.
* A function accepts the parameter (argument), and they can be optional.
* The function block is started with the colon (:), and block statements must be at the same indentation.
* The **return** statement is used to return the value. A function can have only one **return**

### **Function Calling**

In Python, after the function is created, we can call it from another function. A function must be defined before the function call; otherwise, the Python interpreter gives an error. To call the function, use the function name followed by the parentheses.

Consider the following example of a simple example that prints the message "Hello World".

1. #function definition
2. **def** hello\_world():
3. **print**("hello world")
4. # function calling
5. hello\_world()

**Output:**

hello world

## **The return statement**

The return statement is used at the end of the function and returns the result of the function. It terminates the function execution and transfers the result where the function is called. The return statement cannot be used outside of the function.

**Syntax**

1. **return** [expression\_list]

It can contain the expression which gets evaluated and value is returned to the caller function. If the return statement has no expression or does not exist itself in the function then it returns the **None** object.

Consider the following example:

### **Example 1**

1. # Defining function
2. **def** sum():
3. a = 10
4. b = 20
5. c = a+b
6. **return** c
7. # calling sum() function in print statement
8. **print**("The sum is:",sum())

**Output:**

The sum is: 30

In the above code, we have defined the function named **sum,** and it has a statement **c = a+b,** which computes the given values, and the result is returned by the return statement to the caller function.

### **Example 2 Creating function without return statement**

1. # Defining function
2. **def** sum():
3. a = 10
4. b = 20
5. c = a+b
6. # calling sum() function in print statement
7. **print**(sum())

**Output:**

None

In the above code, we have defined the same function without the return statement as we can see that the **sum()** function returned the **None** object to the caller function.

# Python Lambda Functions

Python Lambda function is known as the anonymous function that is defined without a name. Python allows us to not declare the function in the standard manner, i.e., by using the **def** keyword. Rather, the anonymous functions are declared by using the **lambda** keyword. However, Lambda functions can accept any number of arguments, but they can return only one value in the form of expression.

The anonymous function contains a small piece of code. It simulates inline functions of C and C++, but it is not exactly an inline function.

The syntax to define an anonymous function is given below.

### **Syntax**

1. **lambda** arguments: expression

It can accept any number of arguments and has only one expression. It is useful when the function objects are required.

Consider the following example of the lambda function.

### **Example 1**

1. # a is an argument and a+10 is an expression which got evaluated and returned.
2. x = **lambda** a:a+10
3. # Here we are printing the function object
4. **print**(x)
5. **print**("sum = ",x(20))

**Output:**

<function <lambda> at 0x0000019E285D16A8>

sum = 30

In the above example, we have defined the **lambda a: a+10** anonymous function where **a** is an argument and **a+10** is an expression. The given expression gets evaluated and returned the result. The above lambda function is same as the normal function.

1. **def** x(a):
2. **return** a+10
3. **print**(sum = x(10))