**Python Variables**

* Variable is a name which is used to refer memory location. Variable also known as identifier and used to hold value.
* In Python, we don't need to specify the type of variable because Python is a type infer language and smart enough to get variable type.
* Variable names can be a group of both letters and digits, but they have to begin with a letter or an underscore.
* It is recomended to use lowercase letters for variable name. Rahul and rahul both are two different variables.

## **Identifier Naming**

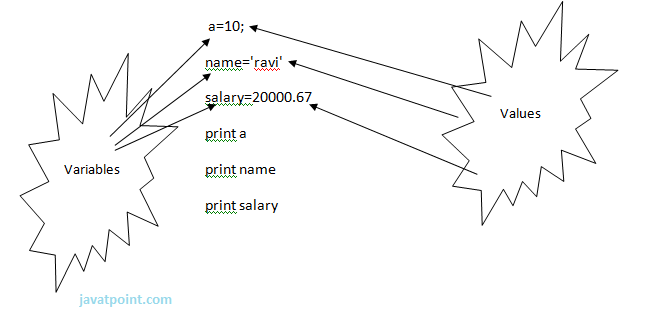
Variables are the example of identifiers. An Identifier is used to identify the literals used in the program. The rules to name an identifier are given below.

* 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.

## **Declaring Variable and Assigning Values**

* Python does not bound us to declare variable before using in the application. It allows us to create variable at required time.
* We don't need to declare explicitly variable in Python. When we assign any value to the variable that variable is declared automatically.
* The equal (=) operator is used to assign value to a variable.

**Eg:**



**Output:**

1. >>>
2. 10
3. ravi
4. 20000.67
5. >>>

## **Multiple Assignment**

* Python allows us to assign a value to multiple variables in a single statement which is also known as multiple assignment.
* We can apply multiple assignments in two ways either by assigning a single value to multiple variables or assigning multiple values to multiple variables. Lets see given examples.

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

**Eg:**

x=y=z=50

**print**(x)

**print** (y)

**print** (z)

**Output:**

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

**Eg:**

a,b,c=5,10,15

**print** (a)

**print** (b)

**print** ©

The values will be assigned in the order in which variables appears.

**Tokens:**

* Tokens can be defined as a punctuator mark, reserved words and each individual word in a statement.
* Token is the smallest unit inside the given program.

There are following tokens in Python:

* Keywords.
* Identifiers.
* Literals.
* Operators.

**Python Data Types**

* Variables can hold values of different data types. Python is a dynamically typed language hence we need not define the type of the variable while declaring it. The interpreter implicitly binds the value with its type.
* Python enables us to check the type of the variable used in the program. Python provides us the **type()** function which returns the type of the variable passed.

Consider the following example to define the values of different data types and checking its type.

a=10

b="Hi Python"

c = 10.5

**print**(type(a));

**print**(type(b));

**print**(type(c));

**Output:**

<type 'int'>

<type 'str'>

<type 'float'>

## **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. [String](https://www.javatpoint.com/python-data-types#string)
3. [List](https://www.javatpoint.com/python-data-types#list)
4. [Tuple](https://www.javatpoint.com/python-data-types#tuple)
5. [Dictionary](https://www.javatpoint.com/python-data-types#dictionary)

### **Numbers**

Number stores numeric values. Python creates Number objects when a number is assigned to a variable. For example;

1. a = 3 , b = 5  #a and b are number objects

Python supports 3 types of numeric data.

1. int (signed integers like 10, 2, 29, etc.)
2. float (float is used to store floating point numbers like 1.9, 9.902, 15.2, etc.)
3. complex (complex numbers like 2.14j, 2.0 + 2.3j, etc.)

A complex number contains an ordered pair, i.e., x + iy where x and y denote the real and imaginary parts respectively).

### **String**

* The string can be defined as the sequence of characters represented in the quotation marks. In python, we can use single, double, or triple quotes to define a string.
* String handling in python is a straightforward task since there are various inbuilt functions and operators provided.

In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+" python" returns "hello python".

The operator \* is known as repetition operator as the operation "Python " \*2 returns "Python Python ".

The following example illustrates the string handling in python.

str1 = 'hello javatpoint' #string str1

str2 = ' how are you' #string str2

**print** (str1[0:2]) #printing first two character using slice operator

**print** (str1[4]) #printing 4th character of the string

**print** (str1\*2) #printing the string twice

**print** (str1 + str2) #printing the concatenation of str1 and str2

**Output:**

he

o

hello javatpointhello javatpoint

hello javatpoint how are you

### **List**

* Lists are similar to arrays in C. However; the list can contain data of different types. The items stored in the list are separated with a comma (,) and enclosed within square brackets [].
* We can use slice [:] operators to access the data of the list. The concatenation operator (+) and repetition operator (\*) works with the list in the same way as they were working with the strings.

Consider the following example.

l  = [1, "hi", "python", 2]

**print** (l[3:]);

**print** (l[0:2]);

**print** (l);

**print** (l + l);

**print** (l \* 3);

**Output:**

[2]

[1, 'hi']

[1, 'hi', 'python', 2]

[1, 'hi', 'python', 2, 1, 'hi', 'python', 2]

[1, 'hi', 'python', 2, 1, 'hi', 'python', 2, 1, 'hi', 'python', 2]

### **Tuple**

A tuple is similar to the list in many ways. Like lists, tuples also contain the collection of the items of different data types. The items of the tuple are separated with a comma (,) and enclosed in parentheses ().

A tuple is a read-only data structure as we can't modify the size and value of the items of a tuple.

Let's see a simple example of the tuple.

t  = ("hi", "python", 2)

**print** (t[1:]);

**print** (t[0:1]);

**print** (t);

**print** (t + t);

**print** (t \* 3);

**print** (type(t))

t[2] = "hi";

**Output:**

('python', 2)

('hi',)

('hi', 'python', 2)

('hi', 'python', 2, 'hi', 'python', 2)

('hi', 'python', 2, 'hi', 'python', 2, 'hi', 'python', 2)

<type 'tuple'>

Traceback (most recent call last):

File "main.py", line 8, in <module>

t[2] = "hi";

TypeError: 'tuple' object does not support item assignment

### **Dictionary**

* Dictionary is an ordered set of a key-value pair of items. It is like an associative array or a hash table where each key stores a specific value. Key can hold any primitive data type whereas value is an arbitrary Python object.
* The items in the dictionary are separated with the comma and enclosed in the curly braces {}.

Consider the following example.

d = {1:'Jimmy', 2:'Alex', 3:'john', 4:'mike'};

**print** (d.keys());

**print** (d.values());

**Output:**

[1, 2, 3, 4]

['Jimmy', 'Alex', 'john', 'mike']

# Python Keywords

Python Keywords are special reserved words which convey a special meaning to the compiler/interpreter. Each keyword have a special meaning and a specific operation. These keywords can't be used as variable. Following is the List of Python Keywords.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| True | False | None | and | as |
| asset | def | class | continue | break |
| else | finally | elif | del | except |
| global | for | if | from | import |
| raise | try | or | return | pass |
| nonlocal | in | not | is | lambda |

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Python Literals**  Literals can be defined as a data that is given in a variable or constant.  Python support the following literals:  **I. String literals:**  String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes for a String.  **Eg:**  "Aman" , '12345'  **Types of Strings:**  There are two types of Strings supported in Python:  a).Single line String- Strings that are terminated within a single line are known as Single line Strings.  **Eg:**   1. >>> text1='hello'   b).Multi line String- A piece of text that is spread along multiple lines is known as Multiple line String.  There are two ways to create Multiline Strings:  **1). Adding black slash at the end of each line.**  **Eg:**  >>> text1='hello\  user'  >>> text1  'hellouser'  >>>  **2).Using triple quotation marks:-**  **Eg:**  >>> str2='''''welcome  to  SSSIT'''  >>> **print** str2  welcome  to  SSSIT  >>>  **II.Numeric literals:**  Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.   |  |  |  |  | | --- | --- | --- | --- | | **Int(signed integers)** | **Long(long integers)** | **float(floating point)** | **Complex(complex)** | | Numbers( can be both positive and negative) with no fractional part.eg: 100 | Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L | Real numbers with both integer and fractional part eg: -26.2 | In the form of a+bj where a forms the real part and b forms the imaginary part of complex number. eg: 3.14j |   **III. Boolean literals:**  A Boolean literal can have any of the two values: True or False.  **IV. Special literals.**  Python contains one special literal i.e., None.  None is used to specify to that field that is not created. It is also used for end of lists in Python.  Eg:  >>> val1=10  >>> val2=None  >>> val1  10  >>> val2  >>> **print** val2  None  >>>  **V.Literal Collections.**  Collections such as tuples, lists and Dictionary are used in Python.  **List:**   * List contain items of different data types. Lists are mutable i.e., modifiable. * The values stored in List are separated by commas(,) and enclosed within a square brackets([]). We can store different type of data in a List. * Value stored in a List can be retrieved using the slice operator([] and [:]). * The plus sign (+) is the list concatenation and asterisk(\*) is the repetition operator.   **Eg:**  >>> list=['aman',678,20.4,'saurav']  >>> list1=[456,'rahul']  >>> list  ['aman', 678, 20.4, 'saurav']  >>> list[1:3]  [678, 20.4]  >>> list+list1  ['aman', 678, 20.4, 'saurav', 456, 'rahul']  >>> list1\*2  [456, 'rahul', 456, 'rahul']  >>> |

# 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 particular programming language. Python provides a variety of operators described as follows.

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

## **Arithmetic operators**

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

Consider the following table for a detailed explanation of 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 result 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 |
| **\* (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 second operand. |
| **// (Floor division)** | It gives the floor value of the quotient produced by dividing the two operands. |

## **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 value of two operands is not equal, 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. |

## **Python 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 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 assign 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 assign 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 assign 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 = 200. |
| %= | It divides the value of the left operand by the value of the right operand and assign the reminder back to 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 operator**

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

**For example,**

**if** a = 7;

   b = 6;

then, binary (a) = 0111

binary (b) = 0011

hence, a & b = 0011

    a | b = 0111

          a ^ b = 0100

   ~ 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 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**

|  |  |
| --- | --- |
| **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 side do not point to the same object. |

## **Operator Precedence**

The precedence of the operators is important 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.

|  |  |
| --- | --- |
| **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 then, less then equal to, greater then, greater then equal to). |
| <> == != | Equality operators. |
| = %= /= //= -= += \*= \*\*= | Assignment operators |
| is is not | Identity operators |
| in not in | Membership operators |
| not or and | Logical operators |

**Python Comments**

* Comments in Python can be used to explain any program code. It can also be used to hide the code as well.
* Comments are the most helpful stuff of any program. It enables us to understand the way, a program works. In python, any statement written along with # symbol is known as a comment. The interpreter does not interpret the comment.
* Comment is not a part of the program, but it enhances the interactivity of the program and makes the program readable.

Python supports two types of comments:

**1) Single Line Comment:**

In case user wants to specify a single line comment, then comment must start with ?#?

**Eg:**

# This is single line comment.

**print** "Hello Python"

**Output:**

Hello Python

**2) Multi Line Comment:**

Multi lined comment can be given inside triple quotes.

**eg:**

''''' This

    Is

   Multipline comment'''

**eg:**

#single line comment

**print** "Hello Python"

'''''This is

multiline comment'''

**Output:**

Hello Python

# Python Decision making (conditional)

* 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.

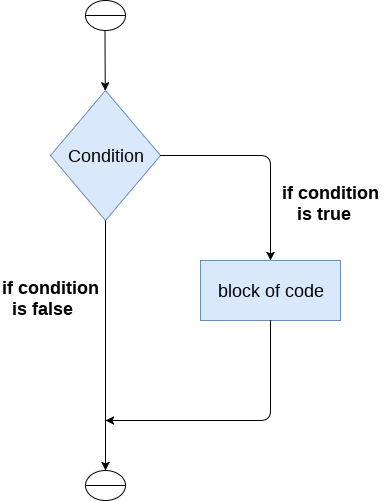
|  |  |
| --- | --- |
| **Statement** | **Description** |
|  |  |
| If Statement | The if statement is used to test a specific condition. If the condition is true, a block of code (if-block) will be executed. |
| If - else Statement | The if-else statement is similar to if statement except the fact that, 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. |

## **Indentation in Python**

* For the ease of programming and to achieve simplicity, python doesn't allow the use of parentheses for the block level code. In Python, indentation is used to declare a block. If two statements are at the same indentation level, then they are the part of the same block.
* Generally, four spaces are given to indent the statements which are a typical amount of indentation in python.
* Indentation is the most used part of the python language since it declares the block of code. All the statements of one block are intended at the same level indentation. We will see how the actual indentation takes place in decision making and other stuff in python.

## **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.

**if** expression:

    statement

**Example 1**

num = int(input("enter the number?"))

**if** num%2 == 0:

**print**("Number is even")

**Output:**

enter the number?10

Number is even

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

a = int(input("Enter a? "));

b = int(input("Enter b? "));

c = int(input("Enter c? "));

**if** a>b **and** a>c:

**print**("a is largest");

**if** b>a **and** b>c:

**print**("b is largest");

**if** c>a **and** c>b:

**print**("c is largest");

**Output:**

Enter a? 100

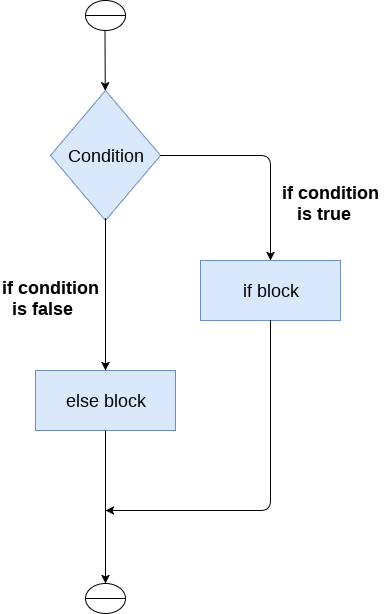
Enter b? 120

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.

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

**if** condition:

    #block of statements

**else**:

    #another block of statements (else-block)

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

age = int (input("Enter your age? "))

**if** age>=18:

**print**("You are eligible to vote !!");

**else**:

**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.**

num = int(input("enter the number?"))

**if** num%2 == 0:

**print**("Number is even...")

**else**:

**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.

**if** expression 1:

   # block of statements

**elif** expression 2:

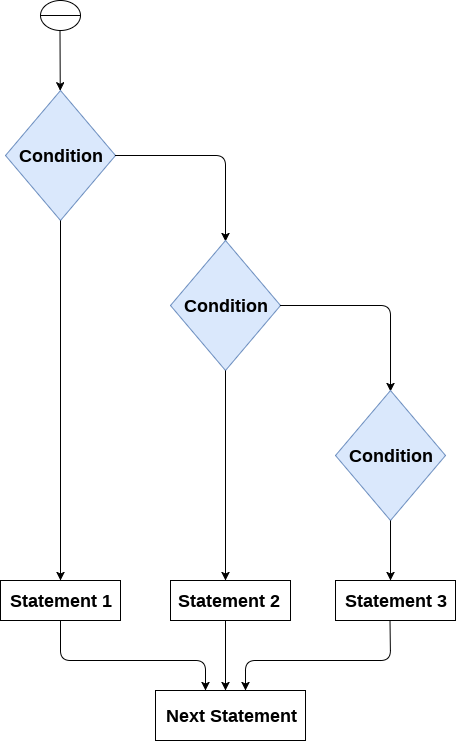
    # block of statements

**elif** expression 3:

  # block of statements

**else**:

  # block of statements



### **Example 1**

number = int(input("Enter the number?"))

**if** number==10:

**print**("number is equals to 10")

**elif** number==50:

**print**("number is equal to 50");

**elif** number==100:

**print**("number is equal to 100");

**else**:

**print**("number is not equal to 10, 50 or 100");

**Output:**

Enter the number?15

number is not equal to 10, 50 or 100

### **Example 2**

marks = int(input("Enter the marks? "))

if marks > 85 **and** marks <= 100:

**print**("Congrats ! you scored grade A ...")

elif marks > 60 **and** marks <= 85:

**print**("You scored grade B + ...")

elif marks > 40 **and** marks <= 60:

**print**("You scored grade B ...")

elif (marks > 30 **and** marks <= 40):

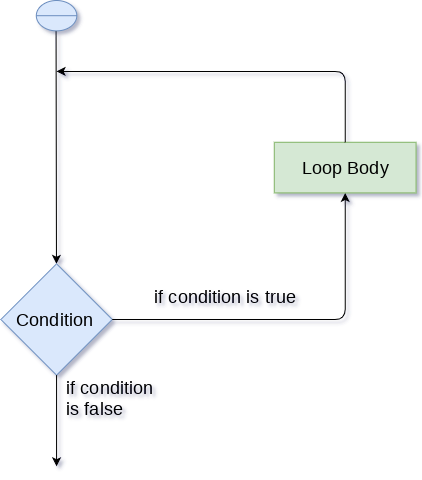
**print**("You scored grade C ...")

else:

**print**("Sorry you are fail ?")

# Python Loops

* The flow of the programs written in any programming language is sequential by default. Sometimes we may need to alter the flow of the program. The execution of a specific code may need to be repeated several numbers of times.
* For this purpose, The programming languages provide various types of loops which are capable of repeating some specific code several numbers of times. Consider the following diagram to understand the working of a loop statement.



## **Why we use loops in python?**

The looping simplifies the complex problems into the easy ones. It enables us to alter the flow of the program so that instead of writing the same code again and again, we can repeat the same code for a finite number of times. For example, if we need to print the first 10 natural numbers then, instead of using the print statement 10 times, we can print inside a loop which runs up to 10 iterations.

## **Advantages of loops**

There are the following advantages of loops in Python.

1. It provides code re-usability.
2. Using loops, we do not need to write the same code again and again.
3. Using loops, we can traverse over the elements of data structures (array or linked lists).

There are the following loop statements in Python.

|  |  |
| --- | --- |
| **Loop Statement** | **Description** |
| for loop | The for loop is used in the case where we need to execute some part of the code until the given condition is satisfied. The for loop is also called as a per-tested loop. It is better to use for loop if the number of iteration is known in advance. |
| while loop | The while loop is to be used in the scenario where we don't know the number of iterations in advance. The block of statements is executed in the while loop until the condition specified in the while loop is satisfied. It is also called a pre-tested loop. |

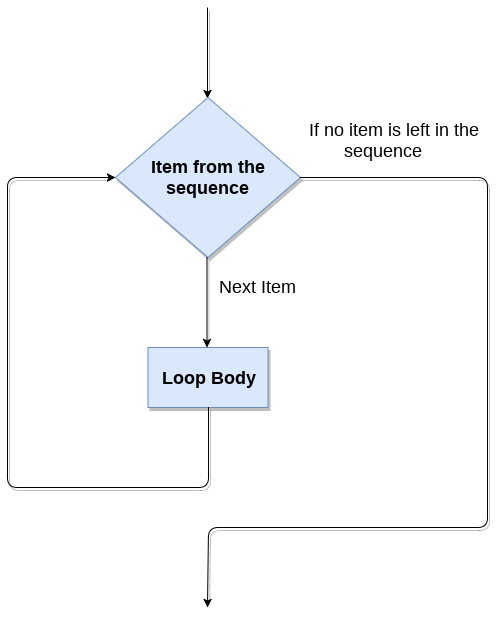
# Python for loop

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.

**for** iterating\_var **in** sequence:

    statement(s)



## **Example**

i=1

n=int(input("Enter the number up to which you want to print the natural numbers?"))

**for** i **in** range(0,10):

**print**(i,end = ' ')

**Output:**

0 1 2 3 4 5 6 7 8 9

## **Python for loop example : printing the table of the given number**

i=1;

num = int(input("Enter a number:"));

**for** i **in** range(1,11):

**print**("%d X %d = %d"%(num,i,num\*i));

**Output:**

Enter a 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

## **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 of the nested for loop in python is given below.

**for** iterating\_var1 **in** sequence:

**for** iterating\_var2 **in** sequence:

        #block of statements

#Other statements

## **Example 1**

n = int(input("Enter the number of rows you want to print?"))

i,j=0,0

**for** i **in** range(0,n):

**print**()

**for** j **in** range(0,i+1):

**print**("\*",end="")

**Output:**

Enter the number of rows you want to print?5

\*

\*\*

\*\*\*

\*\*\*\*

\*\*\*\*\*

## **Using else statement with for loop**

Unlike other languages like C, C++, or Java, python allows us to use the else statement with the for loop which can be executed only when all the iterations are exhausted. Here, we must notice that if the loop contains any of the break statement then the else statement will not be executed.

## **Example 1**

**for** i **in** range(0,5):

**print**(i)

**else**:**print**("for loop completely exhausted, since there is no break.");

In the above example, for loop is executed completely since there is no break statement in the loop. The control comes out of the loop and hence the else block is executed.

**Output:**

0

1

2

3

4

for loop completely exhausted, since there is no break.

## **Example 2**

**for** i **in** range(0,5):

**print**(i)

**break**;

**else**:**print**("for loop is exhausted");

**print**("The loop is broken due to break statement...came out of loop")

In the above example, the loop is broken due to break statement therefore the else statement will not be executed. The statement present immediate next to else block will be executed.

**Output:**

0

The loop is broken due to break statement...came out of loop

# Python while loop

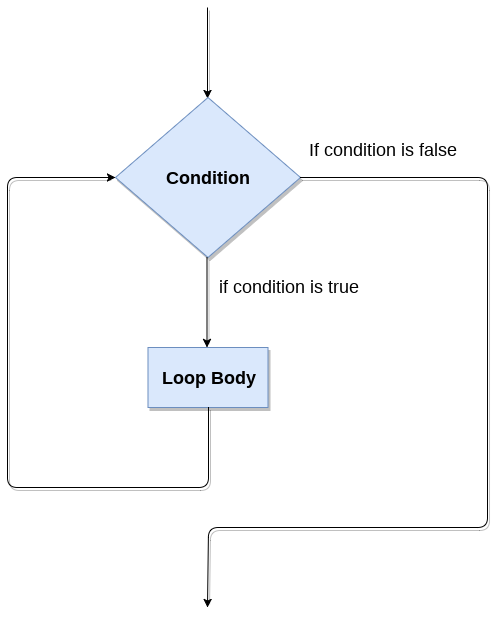
* The while loop is also known as a pre-tested loop. In general, a while loop allows a part of the code to be executed as long as the given condition is true.
* It can be viewed as a repeating if statement. The while loop is mostly used in the case where the number of iterations is not known in advance.

The syntax is given below.

**while** expression:

    statements

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



## **Example 1**

i=1;

**while** i<=10:

**print**(i);

 i=i+1;

**Output:**

1

2

3

4

5

6

7

8

9

10

## **Example 2**

i=1

number=0

b=9

number = int(input("Enter the number?"))

**while** i<=10:

**print**("%d X %d = %d \n"%(number,i,number\*i));

    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 given in the while loop never becomes false then the while loop will never terminate and result into the infinite while loop.

Any non-zero value in the while loop indicates an always-true condition whereas 0 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**

**while** (1):

**print**("Hi! we are inside the infinite while loop");

**Output:**

Hi! we are inside the infinite while loop

(infinite times)

## **Example 2**

var = 1

**while** var != 2:

   i = int(input("Enter the number?"))

**print** ("Entered value is %d"%(i))

**Output:**

Enter the number?102

Entered value is 102

Enter the number?102

Entered value is 102

Enter the number?103

Entered value is 103

Enter the number?103

(infinite loop)

## **Using else with Python while loop**

Python enables us to use the while loop with the while loop also. The else block is executed when the condition given in the while statement becomes false. Like for loop, if the while loop is broken using break statement, then the else block will not be executed and the statement present after else block will be executed.

Consider the following example.

i=1;

**while** i<=5:

**print**(i)

   i=i+1;

**else**:**print**("The while loop exhausted");

**Output:**

1

2

3

4

5

The while loop exhausted

## **Example 2**

i=1;

**while** i<=5:

**print**(i)

   i=i+1;

**if**(i==3):

**break**;

**else**:**print**("The while loop exhausted");

**Output:**

1

2

# 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.

#loop statements

**break**;

**Example 1**

list =[1,2,3,4]

count = 1;

**for** i **in** list:

**if** i == 4:

**print**("item matched")

        count = count + 1;

**break**

**print**("found at",count,"location");

**Output:**

item matched

found at 2 location

## **Example 2**

str = "python"

**for** i **in** str:

**if** i == 'o':

**break**

**print**(i);

**Output:**

p

y

t

h

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

i = 0;

**while** 1:

**print**(i," ",end=""),

    i=i+1;

**if** i == 10:

**break**;

**print**("came out of while loop");

**Output:**

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

## **Example 3**

n=2

**while** 1:

    i=1;

**while** i<=10:

**print**("%d X %d = %d\n"%(n,i,n\*i));

        i = i+1;

    choice = int(input("Do you want to continue printing the table, press 0 for no?"))

**if** choice == 0:

**break**;

    n=n+1

**Output:**

2 X 1 = 2

2 X 2 = 4

2 X 3 = 6

2 X 4 = 8

2 X 5 = 10

2 X 6 = 12

2 X 7 = 14

2 X 8 = 16

2 X 9 = 18

2 X 10 = 20

Do you want to continue printing the table, press 0 for no?1

3 X 1 = 3

3 X 2 = 6

3 X 3 = 9

3 X 4 = 12

3 X 5 = 15

3 X 6 = 18

3 X 7 = 21

3 X 8 = 24

3 X 9 = 27

3 X 10 = 30

Do you want to continue printing the table, press 0 for no?0

# 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 syntax of Python continue statement is given below.

#loop statements

**continue**;

#the code to be skipped

## **Example 1**

i = 0;

**while** i!=10:

**print**("%d"%i);

**continue**;

    i=i+1;

**Output:**

infinite loop

## **Example 2**

i=1; #initializing a local variable

#starting a loop from 1 to 10

**for** i **in** range(1,11):

**if** i==5:

**continue**;

**print**("%d"%i);

**Output:**

1

2

3

4

6

7

8

9

10

## **Pass Statement**

The pass statement is a null operation since nothing happens when it is executed. It is used in the cases where a statement is syntactically needed but we don't want to use any executable statement at its place.

For example, it can be used while overriding a parent class method in the subclass but don't want to give its specific implementation in the subclass.

Pass is also used where the code will be written somewhere but not yet written in the program file.

The syntax of the pass statement is given below.

## **Example**

list = [1,2,3,4,5]

flag = 0

**for** i **in** list:

**print**("Current element:",i,end=" ");

**if** i==3:

**pass**;

**print**("\nWe are inside pass block\n");

        flag = 1;

**if** flag==1:

**print**("\nCame out of pass\n");

        flag=0;

**Output:**

Current element: 1 Current element: 2 Current element: 3

We are inside pass block

Came out of pass

Current element: 4 Current element: 5

Python Pass

In Python, 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 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.

**Python Pass Syntax**

**pass**

**Python Pass Example**

**for** i **in** [1,2,3,4,5]:

**if** i==3:

**pass**

**print** "Pass when value is",i

**print** i,

**Output:**

>>>

1 2 Pass when value **is** 3

3 4 5

>>>

# Python Strings

## **String Literals**

String literals in python are surrounded by either single quotation marks, or double quotation marks.

'hello' is the same as "hello".

You can display a string literal with the print() function:

### **Example**

print("Hello")  
print('Hello')

## **Assign String to a Variable**

Assigning a string to a variable is done with the variable name followed by an equal sign and the string:

### **Example**

a = "Hello"  
print(a)

## **Multiline Strings**

You can assign a multiline string to a variable by using three quotes:

### **Example**

You can use three double quotes:

a = """Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua."""  
print(a)

Or three single quotes:

### **Example**

a = '''Lorem ipsum dolor sit amet,  
consectetur adipiscing elit,  
sed do eiusmod tempor incididunt  
ut labore et dolore magna aliqua.'''  
print(a)

**Note:** in the result, the line breaks are inserted at the same position as in the code.

## **Strings are Arrays**

Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.

However, Python does not have a character data type, a single character is simply a string with a length of 1.

Square brackets can be used to access elements of the string.

### **Example**

Get the character at position 1 (remember that the first character has the position 0):

a = "Hello, World!"  
print(a[1])

## **Slicing**

You can return a range of characters by using the slice syntax.

Specify the start index and the end index, separated by a colon, to return a part of the string.

### **Example**

Get the characters from position 2 to position 5 (not included):

b = "Hello, World!"  
print(b[2:5])

## **Negative Indexing**

Use negative indexes to start the slice from the end of the string:

### **Example**

Get the characters from position 5 to position 1, starting the count from the end of the string:

b = "Hello, World!"  
print(b[-5:-2])

## **String Length**

To get the length of a string, use the len() function.

### **Example**

The len() function returns the length of a string:

a = "Hello, World!"  
print(len(a))

## **String Methods**

Python has a set of built-in methods that you can use on strings.

### **Example**

The strip() method removes any whitespace from the beginning or the end:

a = " Hello, World! "  
print(a.strip()) # returns "Hello, World!"

### **Example**

The lower() method returns the string in lower case:

a = "Hello, World!"  
print(a.lower())

### **Example**

The upper() method returns the string in upper case:

a = "Hello, World!"  
print(a.upper())

### **Example**

The replace() method replaces a string with another string:

a = "Hello, World!"  
print(a.replace("H", "J"))

### **Example**

The split() method splits the string into substrings if it finds instances of the separator:

a = "Hello, World!"  
print(a.split(",")) # returns ['Hello', ' World!']

Learn more about String Methods with our [String Methods Reference](https://www.w3schools.com/python/python_ref_string.asp)

## **Check String**

To check if a certain phrase or character is present in a string, we can use the keywords in or not in.

### **Example**

Check if the phrase "ain" is present in the following text:

txt = "The rain in Spain stays mainly in the plain"  
x = "ain" in txt  
print(x)

### **Example**

Check if the phrase "ain" is NOT present in the following text:

txt = "The rain in Spain stays mainly in the plain"  
x = "ain" not in txt  
print(x)

## **String Concatenation**

To concatenate, or combine, two strings you can use the + operator.

### **Example**

Merge variable a with variable b into variable c:

a = "Hello"  
b = "World"  
c = a + b  
print(c)

### **Example**

To add a space between them, add a " ":

a = "Hello"  
b = "World"  
c = a + " " + b  
print(c)

## **String Format**

As we learned in the Python Variables chapter, we cannot combine strings and numbers like this:

### **Example**

age = 36  
txt = "My name is John, I am " + age  
print(txt)

But we can combine strings and numbers by using the format() method!

The format() method takes the passed arguments, formats them, and places them in the string where the placeholders {} are:

### **Example**

Use the format() method to insert numbers into strings:

age = 36  
txt = "My name is John, and I am {}"  
print(txt.format(age))

The format() method takes unlimited number of arguments, and are placed into the respective placeholders:

### **Example**

quantity = 3  
itemno = 567  
price = 49.95  
myorder = "I want {} pieces of item {} for {} dollars."  
print(myorder.format(quantity, itemno, price))

You can use index numbers {0} to be sure the arguments are placed in the correct placeholders:

### **Example**

quantity = 3  
itemno = 567  
price = 49.95  
myorder = "I want to pay {2} dollars for {0} pieces of item {1}."  
print(myorder.format(quantity, itemno, price))

## **Escape Character**

To insert characters that are illegal in a string, use an escape character.

An escape character is a backslash \ followed by the character you want to insert.

An example of an illegal character is a double quote inside a string that is surrounded by double quotes:

### **Example**

You will get an error if you use double quotes inside a string that is surrounded by double quotes:

txt = "We are the so-called "Vikings" from the north."

To fix this problem, use the escape character \":

### **Example**

The escape character allows you to use double quotes when you normally would not be allowed:

txt = "We are the so-called \"Vikings\" from the north."

Other escape characters used in Python:

|  |  |  |
| --- | --- | --- |
| **Code** | **Result** | **Try it** |
| \' | Single Quote | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_escape2) |
| \\ | Backslash | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_backslash) |
| \n | New Line | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_newline) |
| \r | Carriage Return | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_r) |
| \t | Tab | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_t) |
| \b | Backspace | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_b) |
| \f | Form Feed |  |
| \ooo | Octal value | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_octal) |
| \xhh | Hex value | [Try it »](https://www.w3schools.com/python/showpython.asp?filename=demo_string_hex) |

## **String Methods**

Python has a set of built-in methods that you can use on strings.

**Note:** All string methods returns new values. They do not change the original string.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [capitalize()](https://www.w3schools.com/python/ref_string_capitalize.asp) | Converts the first character to upper case |
| [casefold()](https://www.w3schools.com/python/ref_string_casefold.asp) | Converts string into lower case |
| [center()](https://www.w3schools.com/python/ref_string_center.asp) | Returns a centered string |
| [count()](https://www.w3schools.com/python/ref_string_count.asp) | Returns the number of times a specified value occurs in a string |
| [encode()](https://www.w3schools.com/python/ref_string_encode.asp) | Returns an encoded version of the string |
| [endswith()](https://www.w3schools.com/python/ref_string_endswith.asp) | Returns true if the string ends with the specified value |
| [expandtabs()](https://www.w3schools.com/python/ref_string_expandtabs.asp) | Sets the tab size of the string |
| [find()](https://www.w3schools.com/python/ref_string_find.asp) | Searches the string for a specified value and returns the position of where it was found |
| [format()](https://www.w3schools.com/python/ref_string_format.asp) | Formats specified values in a string |
| format\_map() | Formats specified values in a string |
| [index()](https://www.w3schools.com/python/ref_string_index.asp) | Searches the string for a specified value and returns the position of where it was found |
| [isalnum()](https://www.w3schools.com/python/ref_string_isalnum.asp) | Returns True if all characters in the string are alphanumeric |
| [isalpha()](https://www.w3schools.com/python/ref_string_isalpha.asp) | Returns True if all characters in the string are in the alphabet |
| [isdecimal()](https://www.w3schools.com/python/ref_string_isdecimal.asp) | Returns True if all characters in the string are decimals |
| [isdigit()](https://www.w3schools.com/python/ref_string_isdigit.asp) | Returns True if all characters in the string are digits |
| [isidentifier()](https://www.w3schools.com/python/ref_string_isidentifier.asp) | Returns True if the string is an identifier |
| [islower()](https://www.w3schools.com/python/ref_string_islower.asp) | Returns True if all characters in the string are lower case |
| [isnumeric()](https://www.w3schools.com/python/ref_string_isnumeric.asp) | Returns True if all characters in the string are numeric |
| [isprintable()](https://www.w3schools.com/python/ref_string_isprintable.asp) | Returns True if all characters in the string are printable |
| [isspace()](https://www.w3schools.com/python/ref_string_isspace.asp) | Returns True if all characters in the string are whitespaces |
| [istitle()](https://www.w3schools.com/python/ref_string_istitle.asp) | Returns True if the string follows the rules of a title |
| [isupper()](https://www.w3schools.com/python/ref_string_isupper.asp) | Returns True if all characters in the string are upper case |
| [join()](https://www.w3schools.com/python/ref_string_join.asp) | Joins the elements of an iterable to the end of the string |
| [ljust()](https://www.w3schools.com/python/ref_string_ljust.asp) | Returns a left justified version of the string |
| [lower()](https://www.w3schools.com/python/ref_string_lower.asp) | Converts a string into lower case |
| [lstrip()](https://www.w3schools.com/python/ref_string_lstrip.asp) | Returns a left trim version of the string |
| maketrans() | Returns a translation table to be used in translations |
| [partition()](https://www.w3schools.com/python/ref_string_partition.asp) | Returns a tuple where the string is parted into three parts |
| [replace()](https://www.w3schools.com/python/ref_string_replace.asp) | Returns a string where a specified value is replaced with a specified value |
| [rfind()](https://www.w3schools.com/python/ref_string_rfind.asp) | Searches the string for a specified value and returns the last position of where it was found |
| [rindex()](https://www.w3schools.com/python/ref_string_rindex.asp) | Searches the string for a specified value and returns the last position of where it was found |
| [rjust()](https://www.w3schools.com/python/ref_string_rjust.asp) | Returns a right justified version of the string |
| [rpartition()](https://www.w3schools.com/python/ref_string_rpartition.asp) | Returns a tuple where the string is parted into three parts |
| [rsplit()](https://www.w3schools.com/python/ref_string_rsplit.asp) | Splits the string at the specified separator, and returns a list |
| [rstrip()](https://www.w3schools.com/python/ref_string_rstrip.asp) | Returns a right trim version of the string |
| [split()](https://www.w3schools.com/python/ref_string_split.asp) | Splits the string at the specified separator, and returns a list |
| [splitlines()](https://www.w3schools.com/python/ref_string_splitlines.asp) | Splits the string at line breaks and returns a list |
| [startswith()](https://www.w3schools.com/python/ref_string_startswith.asp) | Returns true if the string starts with the specified value |
| [strip()](https://www.w3schools.com/python/ref_string_strip.asp) | Returns a trimmed version of the string |
| [swapcase()](https://www.w3schools.com/python/ref_string_swapcase.asp) | Swaps cases, lower case becomes upper case and vice versa |
| [title()](https://www.w3schools.com/python/ref_string_title.asp) | Converts the first character of each word to upper case |
| translate() | Returns a translated string |
| [upper()](https://www.w3schools.com/python/ref_string_upper.asp) | Converts a string into upper case |
| [zfill()](https://www.w3schools.com/python/ref_string_zfill.asp) | Fills the string with a specified number of 0 values at the beginning |

## **Python Collections (Arrays)**

There are four collection data types in the Python programming language:

* **List** is a collection which is ordered and changeable. Allows duplicate members.
* **Tuple** is a collection which is ordered and unchangeable. Allows duplicate members.
* **Set** is a collection which is unordered and unindexed. No duplicate members.
* **Dictionary** is a collection which is unordered, changeable and indexed. No duplicate members.

When choosing a collection type, it is useful to understand the properties of that type. Choosing the right type for a particular data set could mean retention of meaning, and, it could mean an increase in efficiency or security.

## **List**

A list is a collection which is ordered and changeable. In Python lists are written with square brackets.

### **Example**

Create a List:

thislist = ["apple", "banana", "cherry"]  
print(thislist)

## **Access Items**

You access the list items by referring to the index number:

### **Example**

Print the second item of the list:

thislist = ["apple", "banana", "cherry"]  
print(thislist[1])

### **Negative Indexing**

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

### **Example**

Print the last item of the list:

thislist = ["apple", "banana", "cherry"]  
print(thislist[-1])

### **Range of Indexes**

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new list with the specified items.

### **Example**

Return the third, fourth, and fifth item:

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[2:5])

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

### **Range of Negative Indexes**

Specify negative indexes if you want to start the search from the end of the list:

### **Example**

This example returns the items from index -4 (included) to index -1 (excluded)

thislist = ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]  
print(thislist[-4:-1])

## **Change Item Value**

To change the value of a specific item, refer to the index number:

### **Example**

Change the second item:

thislist = ["apple", "banana", "cherry"]  
thislist[1] = "blackcurrant"  
print(thislist)

## **Loop Through a List**

You can loop through the list items by using a for loop:

### **Example**

Print all items in the list, one by one:

thislist = ["apple", "banana", "cherry"]  
for x in thislist:  
  print(x)

You will learn more about for loops in our [Python For Loops](https://www.w3schools.com/python/python_for_loops.asp) Chapter.

## **Check if Item Exists**

To determine if a specified item is present in a list use the in keyword:

### **Example**

Check if "apple" is present in the list:

thislist = ["apple", "banana", "cherry"]  
if "apple" in thislist:  
  print("Yes, 'apple' is in the fruits list")

## **List Length**

To determine how many items a list has, use the len() function:

### **Example**

Print the number of items in the list:

thislist = ["apple", "banana", "cherry"]  
print(len(thislist))

## **Add Items**

To add an item to the end of the list, use the append() method:

### **Example**

Using the append() method to append an item:

thislist = ["apple", "banana", "cherry"]  
thislist.append("orange")  
print(thislist)

To add an item at the specified index, use the insert() method:

### **Example**

Insert an item as the second position:

thislist = ["apple", "banana", "cherry"]  
thislist.insert(1, "orange")  
print(thislist)

## **Remove Item**

There are several methods to remove items from a list:

### **Example**

The remove() method removes the specified item:

thislist = ["apple", "banana", "cherry"]  
thislist.remove("banana")  
print(thislist)

### **Example**

The pop() method removes the specified index, (or the last item if index is not specified):

thislist = ["apple", "banana", "cherry"]  
thislist.pop()  
print(thislist)

### **Example**

The del keyword removes the specified index:

thislist = ["apple", "banana", "cherry"]  
del thislist[0]  
print(thislist)

### **Example**

The del keyword can also delete the list completely:

thislist = ["apple", "banana", "cherry"]  
del thislist

### **Example**

The clear() method empties the list:

thislist = ["apple", "banana", "cherry"]  
thislist.clear()  
print(thislist)

## **Copy a List**

You cannot copy a list simply by typing list2 = list1, because: list2 will only be a reference to list1, and changes made in list1 will automatically also be made in list2.

There are ways to make a copy, one way is to use the built-in List method copy().

### **Example**

Make a copy of a list with the copy() method:

thislist = ["apple", "banana", "cherry"]  
mylist = thislist.copy()  
print(mylist)

Another way to make a copy is to use the built-in method list().

### **Example**

Make a copy of a list with the list() method:

thislist = ["apple", "banana", "cherry"]  
mylist = list(thislist)  
print(mylist)

## **Join Two Lists**

There are several ways to join, or concatenate, two or more lists in Python.

One of the easiest ways are by using the + operator.

### **Example**

Join two list:

list1 = ["a", "b" , "c"]  
list2 = [1, 2, 3]  
  
list3 = list1 + list2  
print(list3)

Another way to join two lists are by appending all the items from list2 into list1, one by one:

### **Example**

Append list2 into list1:

list1 = ["a", "b" , "c"]  
list2 = [1, 2, 3]  
  
for x in list2:  
  list1.append(x)  
  
print(list1)

Or you can use the extend() method, which purpose is to add elements from one list to another list:

### **Example**

Use the extend() method to add list2 at the end of list1:

list1 = ["a", "b" , "c"]  
list2 = [1, 2, 3]  
  
list1.extend(list2)  
print(list1)

## **The list() Constructor**

It is also possible to use the list() constructor to make a new list.

### **Example**

Using the list() constructor to make a List:

thislist = list(("apple", "banana", "cherry")) # note the double round-brackets  
print(thislist)

## **List Methods**

Python has a set of built-in methods that you can use on lists.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [append()](https://www.w3schools.com/python/ref_list_append.asp) | Adds an element at the end of the list |
| [clear()](https://www.w3schools.com/python/ref_list_clear.asp) | Removes all the elements from the list |
| [copy()](https://www.w3schools.com/python/ref_list_copy.asp) | Returns a copy of the list |
| [count()](https://www.w3schools.com/python/ref_list_count.asp) | Returns the number of elements with the specified value |
| [extend()](https://www.w3schools.com/python/ref_list_extend.asp) | Add the elements of a list (or any iterable), to the end of the current list |
| [index()](https://www.w3schools.com/python/ref_list_index.asp) | Returns the index of the first element with the specified value |
| [insert()](https://www.w3schools.com/python/ref_list_insert.asp) | Adds an element at the specified position |
| [pop()](https://www.w3schools.com/python/ref_list_pop.asp) | Removes the element at the specified position |
| [remove()](https://www.w3schools.com/python/ref_list_remove.asp) | Removes the item with the specified value |
| [reverse()](https://www.w3schools.com/python/ref_list_reverse.asp) | Reverses the order of the list |
| [sort()](https://www.w3schools.com/python/ref_list_sort.asp) | Sorts the list |

# Python Tuples

## **Tuple**

A tuple is a collection which is ordered and **unchangeable**. In Python tuples are written with round brackets.

### **Example**

Create a Tuple:

thistuple = ("apple", "banana", "cherry")  
print(thistuple)

## **Access Tuple Items**

You can access tuple items by referring to the index number, inside square brackets:

### **Example**

Print the second item in the tuple:

thistuple = ("apple", "banana", "cherry")  
print(thistuple[1])

### **Negative Indexing**

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

### **Example**

Print the last item of the tuple:

thistuple = ("apple", "banana", "cherry")  
print(thistuple[-1])

### **Range of Indexes**

You can specify a range of indexes by specifying where to start and where to end the range.

When specifying a range, the return value will be a new tuple with the specified items.

### **Example**

Return the third, fourth, and fifth item:

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[2:5])

**Note:** The search will start at index 2 (included) and end at index 5 (not included).

Remember that the first item has index 0.

### **Range of Negative Indexes**

Specify negative indexes if you want to start the search from the end of the tuple:

### **Example**

This example returns the items from index -4 (included) to index -1 (excluded)

thistuple = ("apple", "banana", "cherry", "orange", "kiwi", "melon", "mango")  
print(thistuple[-4:-1])

## **Change Tuple Values**

Once a tuple is created, you cannot change its values. Tuples are **unchangeable**, or **immutable** as it also is called.

But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

### **Example**

Convert the tuple into a list to be able to change it:

x = ("apple", "banana", "cherry")  
y = list(x)  
y[1] = "kiwi"  
x = tuple(y)  
  
print(x)

## **Loop Through a Tuple**

You can loop through the tuple items by using a for loop.

### **Example**

Iterate through the items and print the values:

thistuple = ("apple", "banana", "cherry")  
for x in thistuple:  
  print(x)

You will learn more about for loops in our [Python For Loops](https://www.w3schools.com/python/python_for_loops.asp) Chapter.

## **Check if Item Exists**

To determine if a specified item is present in a tuple use the in keyword:

### **Example**

Check if "apple" is present in the tuple:

thistuple = ("apple", "banana", "cherry")  
if "apple" in thistuple:  
  print("Yes, 'apple' is in the fruits tuple")

## **Tuple Length**

To determine how many items a tuple has, use the len() method:

### **Example**

Print the number of items in the tuple:

thistuple = ("apple", "banana", "cherry")  
print(len(thistuple))

## **Add Items**

Once a tuple is created, you cannot add items to it. Tuples are **unchangeable**.

### **Example**

You cannot add items to a tuple:

thistuple = ("apple", "banana", "cherry")  
thistuple[3] = "orange" # This will raise an error  
print(thistuple)

## **Create Tuple With One Item**

To create a tuple with only one item, you have add a comma after the item, unless Python will not recognize the variable as a tuple.

### **Example**

One item tuple, remember the commma:

thistuple = ("apple",)  
print(type(thistuple))  
  
#NOT a tuple  
thistuple = ("apple")  
print(type(thistuple))

## **Remove Items**

**Note:** You cannot remove items in a tuple.

 Tuples are **unchangeable**, so you cannot remove items from it, but you can delete the tuple completely:

### **Example**

The del keyword can delete the tuple completely:

thistuple = ("apple", "banana", "cherry")  
del thistuple  
print(thistuple) #this will raise an error because the tuple no longer exists

## **Join Two Tuples**

To join two or more tuples you can use the + operator:

### **Example**

Join two tuples:

tuple1 = ("a", "b" , "c")  
tuple2 = (1, 2, 3)  
  
tuple3 = tuple1 + tuple2  
print(tuple3)

## **The tuple() Constructor**

It is also possible to use the tuple() constructor to make a tuple.

### **Example**

Using the tuple() method to make a tuple:

thistuple = tuple(("apple", "banana", "cherry")) # note the double round-brackets  
print(thistuple)

## **Tuple Methods**

Python has two built-in methods that you can use on tuples.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [count()](https://www.w3schools.com/python/ref_tuple_count.asp) | Returns the number of times a specified value occurs in a tuple |
| [index()](https://www.w3schools.com/python/ref_tuple_index.asp) | Searches the tuple for a specified value and returns the position of where it was found |

# Python Sets

## **Set**

A set is a collection which is unordered and unindexed. In Python sets are written with curly brackets.

### **Example**

Create a Set:

thisset = {"apple", "banana", "cherry"}  
print(thisset)

**Note:** Sets are unordered, so you cannot be sure in which order the items will appear.

## **Access Items**

You cannot access items in a set by referring to an index, since sets are unordered the items has no index.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

### **Example**

Loop through the set, and print the values:

thisset = {"apple", "banana", "cherry"}  
  
for x in thisset:  
  print(x)

### **Example**

Check if "banana" is present in the set:

thisset = {"apple", "banana", "cherry"}  
  
print("banana" in thisset)

## **Change Items**

Once a set is created, you cannot change its items, but you can add new items.

## **Add Items**

To add one item to a set use the add() method.

To add more than one item to a set use the update() method.

### **Example**

Add an item to a set, using the add() method:

thisset = {"apple", "banana", "cherry"}  
  
thisset.add("orange")  
  
print(thisset)

### **Example**

Add multiple items to a set, using the update() method:

thisset = {"apple", "banana", "cherry"}  
  
thisset.update(["orange", "mango", "grapes"])  
  
print(thisset)

## **Get the Length of a Set**

To determine how many items a set has, use the len() method.

### **Example**

Get the number of items in a set:

thisset = {"apple", "banana", "cherry"}  
  
print(len(thisset))

## **Remove Item**

To remove an item in a set, use the remove(), or the discard() method.

### **Example**

Remove "banana" by using the remove() method:

thisset = {"apple", "banana", "cherry"}  
  
thisset.remove("banana")  
  
print(thisset)

**Note:** If the item to remove does not exist, remove() will raise an error.

### **Example**

Remove "banana" by using the discard() method:

thisset = {"apple", "banana", "cherry"}  
  
thisset.discard("banana")  
  
print(thisset)

**Note:** If the item to remove does not exist, discard() will **NOT** raise an error.

You can also use the pop(), method to remove an item, but this method will remove the last item. Remember that sets are unordered, so you will not know what item that gets removed.

The return value of the pop() method is the removed item.

### **Example**

Remove the last item by using the pop() method:

thisset = {"apple", "banana", "cherry"}  
  
x = thisset.pop()  
  
print(x)  
  
print(thisset)

**Note:** Sets are unordered, so when using the pop() method, you will not know which item that gets removed.

### **Example**

The clear() method empties the set:

thisset = {"apple", "banana", "cherry"}  
  
thisset.clear()  
  
print(thisset)

### **Example**

The del keyword will delete the set completely:

thisset = {"apple", "banana", "cherry"}  
  
del thisset  
  
print(thisset)

## **Join Two Sets**

There are several ways to join two or more sets in Python.

You can use the union() method that returns a new set containing all items from both sets, or the update()method that inserts all the items from one set into another:

### **Example**

The union() method returns a new set with all items from both sets:

set1 = {"a", "b" , "c"}  
set2 = {1, 2, 3}  
  
set3 = set1.union(set2)  
print(set3)

### **Example**

The update() method inserts the items in set2 into set1:

set1 = {"a", "b" , "c"}  
set2 = {1, 2, 3}  
  
set1.update(set2)  
print(set1)

**Note:** Both union() and update() will exclude any duplicate items.

There are other methods that joins two sets and keeps ONLY the duplicates, or NEVER the duplicates, check the full list of set methods in the bottom of this page.

## **The set() Constructor**

It is also possible to use the set() constructor to make a set.

### **Example**

Using the set() constructor to make a set:

thisset = set(("apple", "banana", "cherry")) # note the double round-brackets  
print(thisset)

## **Set Methods**

Python has a set of built-in methods that you can use on sets.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [add()](https://www.w3schools.com/python/ref_set_add.asp) | Adds an element to the set |
| [clear()](https://www.w3schools.com/python/ref_set_clear.asp) | Removes all the elements from the set |
| [copy()](https://www.w3schools.com/python/ref_set_copy.asp) | Returns a copy of the set |
| [difference()](https://www.w3schools.com/python/ref_set_difference.asp) | Returns a set containing the difference between two or more sets |
| [difference\_update()](https://www.w3schools.com/python/ref_set_difference_update.asp) | Removes the items in this set that are also included in another, specified set |
| [discard()](https://www.w3schools.com/python/ref_set_discard.asp) | Remove the specified item |
| [intersection()](https://www.w3schools.com/python/ref_set_intersection.asp) | Returns a set, that is the intersection of two other sets |
| [intersection\_update()](https://www.w3schools.com/python/ref_set_intersection_update.asp) | Removes the items in this set that are not present in other, specified set(s) |
| [isdisjoint()](https://www.w3schools.com/python/ref_set_isdisjoint.asp) | Returns whether two sets have a intersection or not |
| [issubset()](https://www.w3schools.com/python/ref_set_issubset.asp) | Returns whether another set contains this set or not |
| [issuperset()](https://www.w3schools.com/python/ref_set_issuperset.asp) | Returns whether this set contains another set or not |
| [pop()](https://www.w3schools.com/python/ref_set_pop.asp) | Removes an element from the set |
| [remove()](https://www.w3schools.com/python/ref_set_remove.asp) | Removes the specified element |
| [symmetric\_difference()](https://www.w3schools.com/python/ref_set_symmetric_difference.asp) | Returns a set with the symmetric differences of two sets |
| [symmetric\_difference\_update()](https://www.w3schools.com/python/ref_set_symmetric_difference_update.asp) | inserts the symmetric differences from this set and another |
| [union()](https://www.w3schools.com/python/ref_set_union.asp) | Return a set containing the union of sets |
| [update()](https://www.w3schools.com/python/ref_set_update.asp) | Update the set with the union of this set and others |

# Python Dictionaries

## **Dictionary**

A dictionary is a collection which is unordered, changeable and indexed. In Python dictionaries are written with curly brackets, and they have keys and values.

### **Example**

Create and print a dictionary:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
print(thisdict)

## **Accessing Items**

You can access the items of a dictionary by referring to its key name, inside square brackets:

### **Example**

Get the value of the "model" key:

x = thisdict["model"]

There is also a method called get() that will give you the same result:

### **Example**

Get the value of the "model" key:

x = thisdict.get("model")

## **Change Values**

You can change the value of a specific item by referring to its key name:

### **Example**

Change the "year" to 2018:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
thisdict["year"] = 2018

## **Loop Through a Dictionary**

You can loop through a dictionary by using a for loop.

When looping through a dictionary, the return value are the keys of the dictionary, but there are methods to return the values as well.

### **Example**

Print all key names in the dictionary, one by one:

for x in thisdict:  
  print(x)

### **Example**

Print all values in the dictionary, one by one:

for x in thisdict:  
  print(thisdict[x])

### **Example**

You can also use the values() function to return values of a dictionary:

for x in thisdict.values():  
  print(x)

### **Example**

Loop through both keys and values, by using the items() function:

for x, y in thisdict.items():  
  print(x, y)

## **Check if Key Exists**

To determine if a specified key is present in a dictionary use the in keyword:

### **Example**

Check if "model" is present in the dictionary:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
if "model" in thisdict:  
  print("Yes, 'model' is one of the keys in the thisdict dictionary")

## **Dictionary Length**

To determine how many items (key-value pairs) a dictionary has, use the len() method.

### **Example**

Print the number of items in the dictionary:

print(len(thisdict))

## **Adding Items**

Adding an item to the dictionary is done by using a new index key and assigning a value to it:

### **Example**

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
thisdict["color"] = "red"  
print(thisdict)

## **Removing Items**

There are several methods to remove items from a dictionary:

### **Example**

The pop() method removes the item with the specified key name:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
thisdict.pop("model")  
print(thisdict)

### **Example**

The popitem() method removes the last inserted item (in versions before 3.7, a random item is removed instead):

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
thisdict.popitem()  
print(thisdict)

### **Example**

The del keyword removes the item with the specified key name:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
del thisdict["model"]  
print(thisdict)

### **Example**

The del keyword can also delete the dictionary completely:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
del thisdict  
print(thisdict) #this will cause an error because "thisdict" no longer exists.

### **Example**

The clear() keyword empties the dictionary:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
thisdict.clear()  
print(thisdict)

## **Copy a Dictionary**

You cannot copy a dictionary simply by typing dict2 = dict1, because: dict2 will only be a reference to dict1, and changes made in dict1 will automatically also be made in dict2.

There are ways to make a copy, one way is to use the built-in Dictionary method copy().

### **Example**

Make a copy of a dictionary with the copy() method:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
mydict = thisdict.copy()  
print(mydict)

Another way to make a copy is to use the built-in method dict().

### **Example**

Make a copy of a dictionary with the dict() method:

thisdict = {  
  "brand": "Ford",  
  "model": "Mustang",  
  "year": 1964  
}  
mydict = dict(thisdict)  
print(mydict)

## **Nested Dictionaries**

A dictionary can also contain many dictionaries, this is called nested dictionaries.

### **Example**

Create a dictionary that contain three dictionaries:

myfamily = {  
  "child1" : {  
    "name" : "Emil",  
    "year" : 2004  
  },  
  "child2" : {  
    "name" : "Tobias",  
    "year" : 2007  
  },  
  "child3" : {  
    "name" : "Linus",  
    "year" : 2011  
  }  
}

Or, if you want to nest three dictionaries that already exists as dictionaries:

### **Example**

Create three dictionaries, than create one dictionary that will contain the other three dictionaries:

child1 = {  
  "name" : "Emil",  
  "year" : 2004  
}  
child2 = {  
  "name" : "Tobias",  
  "year" : 2007  
}  
child3 = {  
  "name" : "Linus",  
  "year" : 2011  
}  
  
myfamily = {  
  "child1" : child1,  
  "child2" : child2,  
  "child3" : child3  
}

## **The dict() Constructor**

It is also possible to use the dict() constructor to make a new dictionary:

### **Example**

thisdict = dict(brand="Ford", model="Mustang", year=1964)  
# note that keywords are not string literals  
# note the use of equals rather than colon for the assignment  
print(thisdict)

## **Dictionary Methods**

Python has a set of built-in methods that you can use on dictionaries.

|  |  |
| --- | --- |
| **Method** | **Description** |
| [clear()](https://www.w3schools.com/python/ref_dictionary_clear.asp) | Removes all the elements from the dictionary |
| [copy()](https://www.w3schools.com/python/ref_dictionary_copy.asp) | Returns a copy of the dictionary |
| [fromkeys()](https://www.w3schools.com/python/ref_dictionary_fromkeys.asp) | Returns a dictionary with the specified keys and values |
| [get()](https://www.w3schools.com/python/ref_dictionary_get.asp) | Returns the value of the specified key |
| [items()](https://www.w3schools.com/python/ref_dictionary_items.asp) | Returns a list containing a tuple for each key value pair |
| [keys()](https://www.w3schools.com/python/ref_dictionary_keys.asp) | Returns a list containing the dictionary's keys |
| [pop()](https://www.w3schools.com/python/ref_dictionary_pop.asp) | Removes the element with the specified key |
| [popitem()](https://www.w3schools.com/python/ref_dictionary_popitem.asp) | Removes the last inserted key-value pair |
| [setdefault()](https://www.w3schools.com/python/ref_dictionary_setdefault.asp) | Returns the value of the specified key. If the key does not exist: insert the key, with the specified value |
| [update()](https://www.w3schools.com/python/ref_dictionary_update.asp) | Updates the dictionary with the specified key-value pairs |
| [values()](https://www.w3schools.com/python/ref_dictionary_values.asp) | Returns a list of all the values in the dictionary |

# Python Functions

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 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.

In other words, we can say that the collection of functions creates a program. The function is also known as procedure or subroutine in other programming languages.

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

## **Advantage of functions in python**

There are the following advantages of C functions.

* By using functions, we can avoid rewriting same logic/code again and again in a program.
* We can call python functions any number of times in a program and from any place 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**

In python, we can use **def** keyword to define the function. The syntax to define a function in python is given below.

**def** my\_function():

    function-suite

**return** <expression>

The function block is started with the colon (:) and all the same level block statements remain at the same indentation.

A function can accept any number of parameters that must be the same in the definition and function calling.

## **Function calling**

In python, a function must be defined before the function calling otherwise the python interpreter gives an error. Once the function is defined, we can call it from another function or the python prompt. To call the function, use the function name followed by the parentheses.

A simple function that prints the message "Hello Word" is given below.

**def** hello\_world():

**print**("hello world")

hello\_world()

**Output:**

hello world

## **Parameters in function**

The information into the functions can be passed as the parameters. The parameters are specified in the parentheses. We can give any number of parameters, but we have to separate them with a comma.

Consider the following example which contains a function that accepts a string as the parameter and prints it.

## **Example 1**

#defining the function

**def** func (name):

**print**("Hi ",name);

#calling the function

func("Ayush")

## **Example 2**

#python function to calculate the sum of two variables

#defining the function

**def** sum (a,b):

**return** a+b;

#taking values from the user

a = int(input("Enter a: "))

b = int(input("Enter b: "))

#printing the sum of a and b

**print**("Sum = ",sum(a,b))

**Output:**

Enter a: 10

Enter b: 20

Sum = 30

## **Call by reference in Python**

In python, all the functions are called by reference, i.e., all the changes made to the reference inside the function revert back to the original value referred by the reference.

However, there is an exception in the case of mutable objects since the changes made to the mutable objects like string do not revert to the original string rather, a new string object is made, and therefore the two different objects are printed.

## **Example 1 Passing Immutable Object (List)**

#defining the function

**def** change\_list(list1):

    list1.append(20);

    list1.append(30);

**print**("list inside function = ",list1)

#defining the list

list1 = [10,30,40,50]

#calling the function

change\_list(list1);

**print**("list outside function = ",list1);

**Output:**

list inside function = [10, 30, 40, 50, 20, 30]

list outside function = [10, 30, 40, 50, 20, 30]

## **Example 2 Passing Mutable Object (String)**

#defining the function

**def** change\_string (str):

    str = str + " Hows you";

**print**("printing the string inside function :",str);

string1 = "Hi I am there"

#calling the function

change\_string(string1)

**print**("printing the string outside function :",string1)

**Output:**

printing the string inside function : Hi I am there Hows you

printing the string outside function : Hi I am there

## **Types of arguments**

There may be several types of arguments which can be passed at the time of function calling.

1. Required arguments
2. Keyword arguments
3. Default arguments
4. Variable-length arguments

## **Required Arguments**

Till now, we have learned about function calling in python. However, we can provide the arguments at the time of function calling. As far as the required arguments are concerned, these are the arguments which are required to be passed at the time of function calling with the exact match of their positions in the function call and function definition. If either of the arguments is not provided in the function call, or the position of the arguments is changed, then the python interpreter will show the error.

Consider the following example.

## **Example 1**

#the argument name is the required argument to the function func

**def** func(name):

    message = "Hi "+name;

**return** message;

name = input("Enter the name?")

**print**(func(name))

**Output:**

Enter the name?John

Hi John

## **Example 2**

#the function simple\_interest accepts three arguments and returns the simple interest accordingly

**def** simple\_interest(p,t,r):

**return** (p\*t\*r)/100

p = float(input("Enter the principle amount? "))

r = float(input("Enter the rate of interest? "))

t = float(input("Enter the time in years? "))

**print**("Simple Interest: ",simple\_interest(p,r,t))

**Output:**

Enter the principle amount? 10000

Enter the rate of interest? 5

Enter the time in years? 2

Simple Interest: 1000.0

## **Example 3**

#the function calculate returns the sum of two arguments a and b

**def** calculate(a,b):

**return** a+b

calculate(10) # this causes an error as we are missing a required arguments b.

**Output:**

TypeError: calculate() missing 1 required positional argument: 'b'

## **Keyword arguments**

Python allows us to call the function with the keyword arguments. This kind of function call will enable us to pass the arguments in the random order.

The name of the arguments is treated as the keywords and matched in the function calling and definition. If the same match is found, the values of the arguments are copied in the function definition.

Consider the following example.

## **Example 1**

#function func is called with the name and message as the keyword arguments

**def** func(name,message):

**print**("printing the message with",name,"and ",message)

func(name = "John",message="hello") #name and message is copied with the values John and hello respectively

**Output:**

printing the message with John and hello

## **Example 2 providing the values in different order at the calling**

#The function simple\_interest(p, t, r) is called with the keyword arguments the order of arguments doesn't matter in this case

**def** simple\_interest(p,t,r):

**return** (p\*t\*r)/100

**print**("Simple Interest: ",simple\_interest(t=10,r=10,p=1900))

**Output:**

Simple Interest: 1900.0

If we provide the different name of arguments at the time of function call, an error will be thrown.

Consider the following example.

## **Example 3**

#The function simple\_interest(p, t, r) is called with the keyword arguments.

**def** simple\_interest(p,t,r):

**return** (p\*t\*r)/100

**print**("Simple Interest: ",simple\_interest(time=10,rate=10,principle=1900)) # doesn't find the exact match of the name of the arguments (keywords)

**Output:**

TypeError: simple\_interest() got an unexpected keyword argument 'time'

The python allows us to provide the mix of the required arguments and keyword arguments at the time of function call. However, the required argument must not be given after the keyword argument, i.e., once the keyword argument is encountered in the function call, the following arguments must also be the keyword arguments.

Consider the following example.

## **Example 4**

**def** func(name1,message,name2):

**print**("printing the message with",name1,",",message,",and",name2)

func("John",message="hello",name2="David") #the first argument is not the keyword argument

**Output:**

printing the message with John , hello ,and David

The following example will cause an error due to an in-proper mix of keyword and required arguments being passed in the function call.

## **Example 5**

**def** func(name1,message,name2):

**print**("printing the message with",name1,",",message,",and",name2)

func("John",message="hello","David")

**Output:**

SyntaxError: positional argument follows keyword argument

## **Default Arguments**

Python allows us to initialize the arguments at the function definition. If the value of any of the argument is not provided at the time of function call, then that argument can be initialized with the value given in the definition even if the argument is not specified at the function call.

## **Example 1**

**def** printme(name,age=22):

**print**("My name is",name,"and age is",age)

printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function

**Output:**

My name is john and age is 22

## **Example 2**

**def** printme(name,age=22):

**print**("My name is",name,"and age is",age)

printme(name = "john") #the variable age is not passed into the function however the default value of age is considered in the function

printme(age = 10,name="David") #the value of age is overwritten here, 10 will be printed as age

**Output:**

My name is john and age is 22

My name is David and age is 10

## **Variable length Arguments**

In the large projects, sometimes we may not know the number of arguments to be passed in advance. In such cases, Python provides us the flexibility to provide the comma separated values which are internally treated as tuples at the function call.

However, at the function definition, we have to define the variable with \* (star) as \*<variable - name >.

Consider the following example.

## **Example**

**def** printme(\*names):

**print**("type of passed argument is ",type(names))

**print**("printing the passed arguments...")

**for** name **in** names:

**print**(name)

printme("john","David","smith","nick")

**Output:**

type of passed argument is <class 'tuple'>

printing the passed arguments...

john

David

smith

nick

## **Scope of variables**

The scopes of the variables depend upon the location where the variable is being declared. The variable declared in one part of the program may not be accessible to the other parts.

In python, the variables are defined with the two types of scopes.

1. Global variables
2. Local variables

The variable defined outside any function is known to have a global scope whereas the variable defined inside a function is known to have a local scope.

Consider the following example.

## **Example 1**

**def** print\_message():

    message = "hello !! I am going to print a message." # the variable message is local to the function itself

**print**(message)

print\_message()

**print**(message) # this will cause an error since a local variable cannot be accessible here.

**Output:**

hello !! I am going to print a message.

File "/root/PycharmProjects/PythonTest/Test1.py", line 5, in

print(message)

NameError: name 'message' is not defined

## **Example 2**

**def** calculate(\*args):

    sum=0

**for** arg **in** args:

        sum = sum +arg

**print**("The sum is",sum)

sum=0

calculate(10,20,30) #60 will be printed as the sum

**print**("Value of sum outside the function:",sum) # 0 will be printed

**Output:**

The sum is 60

Value of sum outside the function: 0

**Working With Functions: Return Values**

* Most functions take in arguments, perform some processing and then return a value to the caller. In Python this is achieved with the return statement.

**def** square(n):

**return** n\*n

two\_squared = square(2)

*# or print it as before*

**print**(square(2))

* Python also has the ability to return multiple values from a function call, something missing from many other languages. In this case the return values should be a comma-separated list of values and Python then constructs a *tuple* and returns this to the caller, e.g.

**def** square(x,y):

**return** x\*x, y\*y

t = square(2,3)

**print**(t) *# Produces (4,9)*

*# Now access the tuple with usual operations*

* An alternate syntax when dealing with multiple return values is to have Python "unwrap" the tuple into the variables directly by specifying the same number of variables on the left-hand side of the assignment as there are returned from the function, e.g.

**def** square(x,y):

**return** x\*x, y\*y

xsq, ysq = square(2,3)

**print**(xsq) *# Prints 4*

**print**(ysq) *# Prints 9*

*# Tuple has vanished!*

## **Passing a List as a Parameter**

You can send any data types of parameter to a function (string, number, list, dictionary etc.), and it will be treated as the same data type inside the function.

E.g. if you send a List as a parameter, it will still be a List when it reaches the function:

### **Example**

def my\_function(food):  
  for x in food:  
    print(x)  
  
fruits = ["apple", "banana", "cherry"]  
  
my\_function(fruits)

## **Keyword Arguments**

You can also send arguments with the key = value syntax.

This way the order of the arguments does not matter.

### **Example**

def my\_function(child3, child2, child1):  
  print("The youngest child is " + child3)  
  
my\_function(child1 = "Emil", child2 = "Tobias", child3 = "Linus")

## **Pass by reference vs value**

All parameters (arguments) in the Python language are passed by reference. It means if you change what a parameter refers to within a function, the change also reflects back in the calling function. For example −

[Live Demo](http://tpcg.io/ZwGczd)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist.append([1,2,3,4]);

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

Here, we are maintaining reference of the passed object and appending values in the same object. So, this would produce the following result −

Values inside the function: [10, 20, 30, [1, 2, 3, 4]]

Values outside the function: [10, 20, 30, [1, 2, 3, 4]]

There is one more example where argument is being passed by reference and the reference is being overwritten inside the called function.

[Live Demo](http://tpcg.io/xgrZcs)

#!/usr/bin/python

# Function definition is here

def changeme( mylist ):

"This changes a passed list into this function"

mylist = [1,2,3,4]; # This would assig new reference in mylist

print "Values inside the function: ", mylist

return

# Now you can call changeme function

mylist = [10,20,30];

changeme( mylist );

print "Values outside the function: ", mylist

The parameter *mylist* is local to the function changeme. Changing mylist within the function does not affect *mylist*. The function accomplishes nothing and finally this would produce the following result −

Values inside the function: [1, 2, 3, 4]

Values outside the function: [10, 20, 30]

**import** turtle  
obj=turtle.Turtle()  
obj.pensize(5)  
obj.pencolor(**"red"**)  
**def** myfun(x,y):  
 obj.goto(x,y)  
 **for** i **in** range(10):  
 obj.circle(5\*i)  
 obj.circle(-5\*i)  
myfun(0,0)  
myfun(100,0)  
myfun(200,0)  
turtle.done()

## What is recursion in Python?

Recursion is the process of defining something in terms of itself.

A physical world example would be to place two parallel mirrors facing each other. Any object in between them would be reflected recursively.

## Python Recursive Function

We know that in Python, a [function](https://www.programiz.com/python-programming/function) can call other functions. It is even possible for the function to call itself. These type of construct are termed as recursive functions.

# An example of a recursive function to

# find the factorial of a number

def calc\_factorial(x):

"""This is a recursive function

to find the factorial of an integer"""

if x == 1:

return 1

else:

return (x \* calc\_factorial(x-1))

num = 4

print("The factorial of", num, "is", calc\_factorial(num))

Each function call multiples the number with the factorial of number 1 until the number is equal to one. This recursive call can be explained in the following steps.

1. calc\_factorial(4) # 1st call with 4
2. 4 \* calc\_factorial(3) # 2nd call with 3
3. 4 \* 3 \* calc\_factorial(2) # 3rd call with 2
4. 4 \* 3 \* 2 \* calc\_factorial(1) # 4th call with 1
5. 4 \* 3 \* 2 \* 1 # return from 4th call as number=1
6. 4 \* 3 \* 2 # return from 3rd call
7. 4 \* 6 # return from 2nd call
8. 24 # return from 1st call

Our recursion ends when the number reduces to 1. This is called the base condition.

Every recursive function must have a base condition that stops the recursion or else the function calls itself infinitely.

## Advantages of Recursion

1. Recursive functions make the code look clean and elegant.
2. A complex task can be broken down into simpler sub-problems using recursion.
3. Sequence generation is easier with recursion than using some nested iteration.

## Disadvantages of Recursion

1. Sometimes the logic behind recursion is hard to follow through.
2. Recursive calls are expensive (inefficient) as they take up a lot of memory and time.
3. Recursive functions are hard to debug.

### What is Recursion?

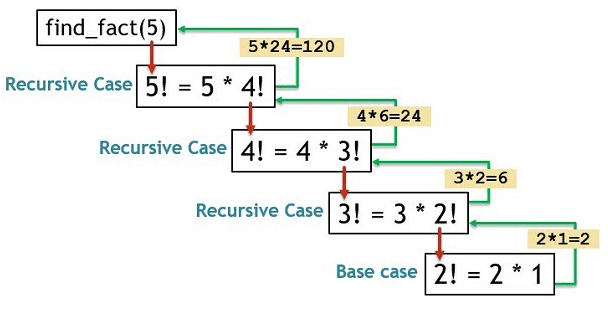
As stated in the introduction, [recursion](https://en.wikipedia.org/wiki/Recursion) involves a process calling itself in the definition. A recursive function generally has two components:

* The **base case** which is a condition that determines when the recursive function should stop
* The call to itself

actorial(5)  
=5\*factorial(4)  
=5\*4\*factorial(3)  
=5\*4\*3\*factorial(2)  
=5\*4\*3\*2\*factorial(1)  
=5\*4\*3\*2\*1  
=5\*4\*3\*2  
=5\*4\*6  
=5\*24  
=120

## **What is Recursion in Python?**

Recursion is the process of determining something in terms of itself. We know that in Python, any function can call any other function, a function can also call itself. These types of functions which call itself till the certain condition is not met are termed as recursive functions.



When a function calls itself, its called Recursion. It will be easier for those who have seen the movie Inception. Leonardo had a dream, in that dream he had another dream, in that dream he had yet another dream, and that goes on. So it's like there is a function called dream(), and we are just calling it in itself.

function dream()

print "Dreaming"

dream()

Recursion is useful in solving problems which can be broken down into smaller problems of the **same kind**. But when it comes to solving problems using Recursion there are several things to be taken care of. Let's take a simple example and try to understand those. Following is the pseudo code of finding factorial of a given number X using recursion.

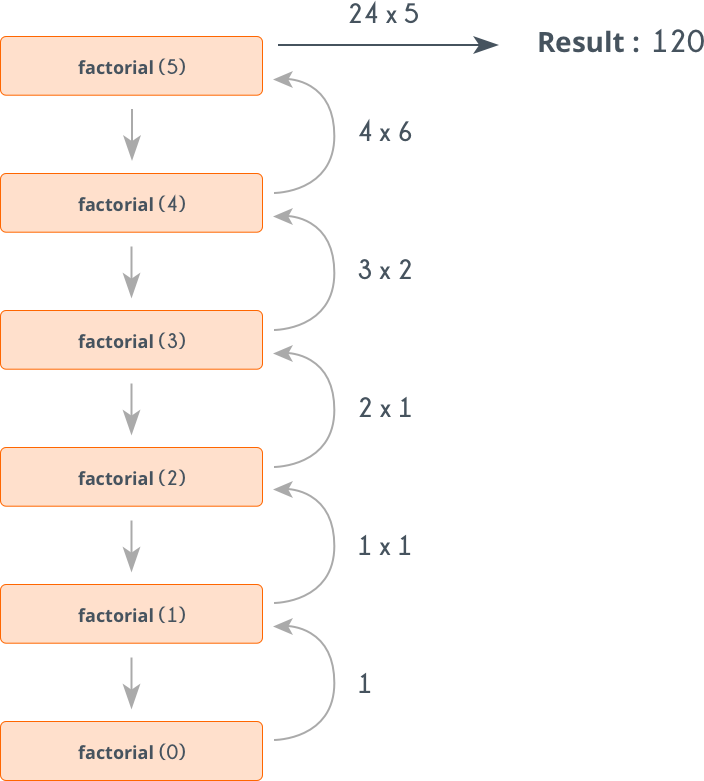
function factorial(x)

if x is 0 // base case

return 1

return x\*factorial(x-1) // break into smaller problem(s)

The following image shows how it works for factorial(5).



We can observe that this implementation does a lot of repeated work (see the following recursion tree). So this is a bad implementation for nth Fibonacci number.

fib(5)

/ \

fib(4) fib(3)

/ \ / \

fib(3) fib(2) fib(2) fib(1)

/ \ / \ / \

fib(2) fib(1) fib(1) fib(0) fib(1) fib(0)

/ \

fib(1) fib(0)