**TechLamp Python Classroom Notes :**

**Lecture 1 :**

**Python** is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Like Perl, Python source code is also available under the GNU General Public License (GPL). This **tutorial** gives enough understanding on **Python programming** language.

Why to Learn Python?

**Python** is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable. It uses English keywords frequently where as other languages use punctuation, and it has fewer syntactical constructions than other languages.

**Python** is a MUST for students and working professionals to become a great Software Engineer specially when they are working in Web Development Domain. I will list down some of the key advantages of learning Python:

* **Python is Interpreted** − Python is processed at runtime by the interpreter. You do not need to compile your program before executing it. This is similar to PERL and PHP.
* **Python is Interactive** − You can actually sit at a Python prompt and interact with the interpreter directly to write your programs.
* **Python is Object-Oriented** − Python supports Object-Oriented style or technique of programming that encapsulates code within objects.
* **Python is a Beginner's Language** − Python is a great language for the beginner-level programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Characteristics of Python

Following are important characteristics of **Python Programming** −

* It supports functional and structured programming methods as well as OOP.
* It can be used as a scripting language or can be compiled to byte-code for building large applications.
* It provides very high-level dynamic data types and supports dynamic type checking.
* It supports automatic garbage collection.
* It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

## Applications of Python

As mentioned before, Python is one of the most widely used language over the web. I'm going to list few of them here:

* **Easy-to-learn** − Python has few keywords, simple structure, and a clearly defined syntax. This allows the student to pick up the language quickly.
* **Easy-to-read** − Python code is more clearly defined and visible to the eyes.
* **Easy-to-maintain** − Python's source code is fairly easy-to-maintain.
* **A broad standard library** − Python's bulk of the library is very portable and cross-platform compatible on UNIX, Windows, and Macintosh.
* **Interactive Mode** − Python has support for an interactive mode which allows interactive testing and debugging of snippets of code.
* **Portable** − Python can run on a wide variety of hardware platforms and has the same interface on all platforms.
* **Extendable** − You can add low-level modules to the Python interpreter. These modules enable programmers to add to or customize their tools to be more efficient.
* **Databases** − Python provides interfaces to all major commercial databases.
* **GUI Programming** − Python supports GUI applications that can be created and ported to many system calls, libraries and windows systems, such as Windows MFC, Macintosh, and the X Window system of Unix.
* **Scalable** − Python provides a better structure and support for large programs than shell scripting.

## First Python Program

Let us execute programs in different modes of programming.

### Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt −

$ python

Python 2.4.3 (#1, Nov 11 2010, 13:34:43)

[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2

Type "help", "copyright", "credits" or "license" for more information.

>>>

Type the following text at the Python prompt and press the Enter −

>>> print("Hello, Python!")

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!");**. However in Python version 2.4.3, this produces the following result −

Hello, Python!

### Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

## Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (\_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, $, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers −

* Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
* Starting an identifier with a single leading underscore indicates that the identifier is private.
* Starting an identifier with two leading underscores indicates a strongly private identifier.
* If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

Based on the data type of a variable, the interpreter allocates memory and decides what can be stored in the reserved memory. Therefore, by assigning different data types to variables, you can store integers, decimals or characters in these variables.

## Assigning Values to Variables

Python variables do not need explicit declaration to reserve memory space. The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

The operand to the left of the = operator is the name of the variable and the operand to the right of the = operator is the value stored in the variable. For example −

#!/usr/bin/python

counter = 100 # An integer assignment

miles = 1000.0 # A floating point

name = "John" # A string

print(counter)

print(miles)

print(name)

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result −

100

1000.0

John

## Multiple Assignment

Python allows you to assign a single value to several variables simultaneously. For example −

a = b = c = 1

Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example −

a,b,c = 1,2,"john"

Here, two integer objects with values 1 and 2 are assigned to variables a and b respectively, and one string object with the value "john" is assigned to the variable c.

## Standard Data Types

The data stored in memory can be of many types. For example, a person's age is stored as a numeric value and his or her address is stored as alphanumeric characters. Python has various standard data types that are used to define the operations possible on them and the storage method for each of them.

Python has five standard data types −

* Numbers
* String
* List
* Tuple
* Dictionary

## Python Numbers

Number data types store numeric values. Number objects are created when you assign a value to them. For example −

var1 = 1

var2 = 10

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is −

del var1[,var2[,var3[....,varN]]]]

You can delete a single object or multiple objects by using the del statement. For example −

del var

del var\_a, var\_b

Python supports four different numerical types −

* int (signed integers)
* long (long integers, they can also be represented in octal and hexadecimal)
* float (floating point real values)
* complex (complex numbers)

### Examples

Here are some examples of numbers −

|  |  |  |  |
| --- | --- | --- | --- |
| **int** | **long** | **float** | **complex** |
| 10 | 51924361L | 0.0 | 3.14j |
| 100 | -0x19323L | 15.20 | 45.j |
| -786 | 0122L | -21.9 | 9.322e-36j |
| 080 | 0xDEFABCECBDAECBFBAEl | 32.3+e18 | .876j |
| -0490 | 535633629843L | -90. | -.6545+0J |
| -0x260 | -052318172735L | -32.54e100 | 3e+26J |
| 0x69 | -4721885298529L | 70.2-E12 | 4.53e-7j |

* Python allows you to use a lowercase l with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.
* A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x and y are the real numbers and j is the imaginary unit.

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Class room Code1 : <Basics.py>

Notepad version: [..\Documents\Basics.txt](../Documents/Basics.txt)

Class room Code2: <comparison_operators.py> :

Notepad version: [..\Documents\comparison\_operators.txt](../Documents/comparison_operators.txt)

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Python Strings

Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.

The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator. For example −

#!/usr/bin/python

str = 'Hello World!'

print(str) # Prints complete string

print(str[0]) # Prints first character of the string

print(str[2:5]) # Prints characters starting from 3rd to 5th

print(str[2:]) # Prints string starting from 3rd character

print(str \* 2) # Prints string two times

print(str + "TEST") # Prints concatenated string

This will produce the following result −

Hello World!

H

llo

llo World!

Hello World!Hello World!

Hello World!TEST

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Classroom Codes : <string_operations.py>

Notepad version : [..\Documents\string\_operations.txt](../Documents/string_operations.txt)

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type.

The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator. For example −

#!/usr/bin/python

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tinylist = [123, 'john']

print(list) # Prints complete list

print(list[0]) # Prints first element of the list

print(list[1:3]) # Prints elements starting from 2nd till 3rd

print(list[2:]) # Prints elements starting from 3rd element

print(tinylist \* 2) # Prints list two times

print(list + tinylist) # Prints concatenated lists

This produce the following result −

['abcd', 786, 2.23, 'john', 70.2]

abcd

[786, 2.23]

[2.23, 'john', 70.2]

[123, 'john', 123, 'john']

['abcd', 786, 2.23, 'john', 70.2, 123, 'john']

Creating a list is as simple as putting different comma-separated values between square brackets. For example −

list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5 ];

list3 = ["a", "b", "c", "d"]

Similar to string indices, list indices start at 0, and lists can be sliced, concatenated and so on.

Accessing Values in Lists

To access values in lists, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

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

#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

list2 = [1, 2, 3, 4, 5, 6, 7 ];

print("list1[0]: ", list1[0])

print("list2[1:5]: ", list2[1:5])

When the above code is executed, it produces the following result −

list1[0]: physics

list2[1:5]: [2, 3, 4, 5]

Updating Lists

You can update single or multiple elements of lists by giving the slice on the left-hand side of the assignment operator, and you can add to elements in a list with the append() method. For example −

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

#!/usr/bin/python

list = ['physics', 'chemistry', 1997, 2000];

print("Value available at index 2 : ")

print(list[2])

list[2] = 2001;

print("New value available at index 2 : ")

print(list[2])

**Note** − append() method is discussed in subsequent section.

When the above code is executed, it produces the following result −

Value available at index 2 :

1997

New value available at index 2 :

2001

Delete List Elements

To remove a list element, you can use either the del statement if you know exactly which element(s) you are deleting or the remove() method if you do not know. For example −

[Live Demo](http://tpcg.io/k3xK1x)

#!/usr/bin/python

list1 = ['physics', 'chemistry', 1997, 2000];

print(list1)

del list1[2];

print("After deleting value at index 2 : ")

print(list1)

When the above code is executed, it produces following result −

['physics', 'chemistry', 1997, 2000]

After deleting value at index 2 :

['physics', 'chemistry', 2000]

**Note** − remove() method is discussed in subsequent section.

Basic List Operations

Lists respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new list, not a string.

In fact, lists respond to all of the general sequence operations we used on strings in the prior chapter.

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len([1, 2, 3]) | 3 | Length |
| [1, 2, 3] + [4, 5, 6] | [1, 2, 3, 4, 5, 6] | Concatenation |
| ['Hi!'] \* 4 | ['Hi!', 'Hi!', 'Hi!', 'Hi!'] | Repetition |
| 3 in [1, 2, 3] | True | Membership |
| for x in [1, 2, 3]: print x, | 1 2 3 | Iteration |

Indexing, Slicing, and Matrixes

Because lists are sequences, indexing and slicing work the same way for lists as they do for strings.

Assuming following input −

L = ['spam', 'Spam', 'SPAM!']

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | SPAM! | Offsets start at zero |
| L[-2] | Spam | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

Built-in List Functions & Methods

Python includes the following list functions −

|  |  |
| --- | --- |
| **Sr.No.** | **Function with Description** |
| 1 | [cmp(list1, list2)](https://www.tutorialspoint.com/python/list_cmp.htm)  Compares elements of both lists. |
| 2 | [len(list)](https://www.tutorialspoint.com/python/list_len.htm)  Gives the total length of the list. |
| 3 | [max(list)](https://www.tutorialspoint.com/python/list_max.htm)  Returns item from the list with max value. |
| 4 | [min(list)](https://www.tutorialspoint.com/python/list_min.htm)  Returns item from the list with min value. |
| 5 | [list(seq)](https://www.tutorialspoint.com/python/list_list.htm)  Converts a tuple into list. |

Python includes following list methods

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [list.append(obj)](https://www.tutorialspoint.com/python/list_append.htm)  Appends object obj to list |
| 2 | [list.count(obj)](https://www.tutorialspoint.com/python/list_count.htm)  Returns count of how many times obj occurs in list |
| 3 | [list.extend(seq)](https://www.tutorialspoint.com/python/list_extend.htm)  Appends the contents of seq to list |
| 4 | [list.index(obj)](https://www.tutorialspoint.com/python/list_index.htm)  Returns the lowest index in list that obj appears |
| 5 | [list.insert(index, obj)](https://www.tutorialspoint.com/python/list_insert.htm)  Inserts object obj into list at offset index |
| 6 | [list.pop(obj=list[-1])](https://www.tutorialspoint.com/python/list_pop.htm)  Removes and returns last object or obj from list |
| 7 | [list.remove(obj)](https://www.tutorialspoint.com/python/list_remove.htm)  Removes object obj from list |
| 8 | [list.reverse()](https://www.tutorialspoint.com/python/list_reverse.htm)  Reverses objects of list in place |
| 9 | [list.sort([func])](https://www.tutorialspoint.com/python/list_sort.htm)  Sorts objects of list, use compare func if given |

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Classroom Codes : <list_op.py>

Notepad version: [..\Documents\list\_op.txt](../Documents/list_op.txt)

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.

The main differences between lists and tuples are: Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated. Tuples can be thought of as **read-only** lists. For example −

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

tinytuple = (123, 'john')

print(tuple) # Prints the complete tuple

print(tuple[0]) # Prints first element of the tuple

print(tuple[1:3]) # Prints elements of the tuple starting from 2nd till 3rd

print(tuple[2:]) # Prints elements of the tuple starting from 3rd element

print(tinytuple \* 2) # Prints the contents of the tuple twice

print(tuple + tinytuple) # Prints concatenated tuples

This produce the following result −

('abcd', 786, 2.23, 'john', 70.2)

abcd

(786, 2.23)

(2.23, 'john', 70.2)

(123, 'john', 123, 'john')

('abcd', 786, 2.23, 'john', 70.2, 123, 'john')

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists −

#!/usr/bin/python

tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )

list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]

tuple[2] = 1000 # Invalid syntax with tuple

list[2] = 1000 # Valid syntax with list

Creating a tuple is as simple as putting different comma-separated values. Optionally you can put these comma-separated values between parentheses also. For example −

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5 );

tup3 = "a", "b", "c", "d";

The empty tuple is written as two parentheses containing nothing −

tup1 = ();

To write a tuple containing a single value you have to include a comma, even though there is only one value −

tup1 = (50,);

Like string indices, tuple indices start at 0, and they can be sliced, concatenated, and so on.

Accessing Values in Tuples

To access values in tuple, use the square brackets for slicing along with the index or indices to obtain value available at that index. For example −

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

#!/usr/bin/python

tup1 = ('physics', 'chemistry', 1997, 2000);

tup2 = (1, 2, 3, 4, 5, 6, 7 );

print("tup1[0]: ", tup1[0]);

print("tup2[1:5]: ", tup2[1:5]);

When the above code is executed, it produces the following result −

tup1[0]: physics

tup2[1:5]: [2, 3, 4, 5]

Updating Tuples

Tuples are immutable which means you cannot update or change the values of tuple elements. You are able to take portions of existing tuples to create new tuples as the following example demonstrates −

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

#!/usr/bin/python

tup1 = (12, 34.56);

tup2 = ('abc', 'xyz');

# Following action is not valid for tuples

# tup1[0] = 100;

# So let's create a new tuple as follows

tup3 = tup1 + tup2;

print(tup3);

When the above code is executed, it produces the following result −

(12, 34.56, 'abc', 'xyz')

Delete Tuple Elements

Removing individual tuple elements is not possible. There is, of course, nothing wrong with putting together another tuple with the undesired elements discarded.

To explicitly remove an entire tuple, just use the **del** statement. For example −

[Live Demo](http://tpcg.io/ANv9OB)

#!/usr/bin/python

tup = ('physics', 'chemistry', 1997, 2000);

print(tup);

del tup;

print("After deleting tup : ");

print(tup);

This produces the following result. Note an exception raised, this is because after **del tup** tuple does not exist any more −

('physics', 'chemistry', 1997, 2000)

After deleting tup :

Traceback (most recent call last):

File "test.py", line 9, in <module>

print tup;

NameError: name 'tup' is not defined

Basic Tuples Operations

Tuples respond to the + and \* operators much like strings; they mean concatenation and repetition here too, except that the result is a new tuple, not a string.

In fact, tuples respond to all of the general sequence operations we used on strings in the prior chapter −

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| len((1, 2, 3)) | 3 | Length |
| (1, 2, 3) + (4, 5, 6) | (1, 2, 3, 4, 5, 6) | Concatenation |
| ('Hi!',) \* 4 | ('Hi!', 'Hi!', 'Hi!', 'Hi!') | Repetition |
| 3 in (1, 2, 3) | True | Membership |
| for x in (1, 2, 3): print x, | 1 2 3 | Iteration |

Indexing, Slicing, and Matrixes

Because tuples are sequences, indexing and slicing work the same way for tuples as they do for strings. Assuming following input −

L = ('spam', 'Spam', 'SPAM!')

|  |  |  |
| --- | --- | --- |
| **Python Expression** | **Results** | **Description** |
| L[2] | 'SPAM!' | Offsets start at zero |
| L[-2] | 'Spam' | Negative: count from the right |
| L[1:] | ['Spam', 'SPAM!'] | Slicing fetches sections |

No Enclosing Delimiters

Any set of multiple objects, comma-separated, written without identifying symbols, i.e., brackets for lists, parentheses for tuples, etc., default to tuples, as indicated in these short examples −

[Live Demo](http://tpcg.io/TO4E3c)

#!/usr/bin/python

Print('abc', -4.24e93, 18+6.6j, 'xyz');

x, y = 1, 2;

print("Value of x , y : ", x,y);

When the above code is executed, it produces the following result −

abc -4.24e+93 (18+6.6j) xyz

Value of x , y : 1 2

Built-in Tuple Functions

Python includes the following tuple functions −

|  |  |
| --- | --- |
| **Sr.No.** | **Function with Description** |
| 1 | [cmp(tuple1, tuple2)](https://www.tutorialspoint.com/python/tuple_cmp.htm)  Compares elements of both tuples. |
| 2 | [len(tuple)](https://www.tutorialspoint.com/python/tuple_len.htm)  Gives the total length of the tuple. |
| 3 | [max(tuple)](https://www.tutorialspoint.com/python/tuple_max.htm)  Returns item from the tuple with max value. |
| 4 | [min(tuple)](https://www.tutorialspoint.com/python/tuple_min.htm)  Returns item from the tuple with min value. |
| 5 | [tuple(seq)](https://www.tutorialspoint.com/python/tuple_tuple.htm)  Converts a list into tuple. |

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Classroom Codes : <tuple_op.py>

## Notepad version: [..\Documents\tuple\_op.txt](../Documents/tuple_op.txt)

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Lecture 2:

## Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object.

Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example −

#!/usr/bin/python

dict = {}

dict['one'] = "This is one"

dict[2] = "This is two"

tinydict = {'name': 'john','code':6734, 'dept': 'sales'}

print(dict['one']) # Prints value for 'one' key

print(dict[2]) # Prints value for 2 key

print(tinydict) # Prints complete dictionary

print(tinydict.keys()) # Prints all the keys

print(tinydict.values()) # Prints all the values

This produce the following result −

This is one

This is two

{'dept': 'sales', 'code': 6734, 'name': 'john'}

['dept', 'code', 'name']

['sales', 6734, 'john']

Dictionaries have no concept of order among elements. It is incorrect to say that the elements are "out of order"; they are simply unordered.

Each key is separated from its value by a colon (:), the items are separated by commas, and the whole thing is enclosed in curly braces. An empty dictionary without any items is written with just two curly braces, like this: {}.

Keys are unique within a dictionary while values may not be. The values of a dictionary can be of any type, but the keys must be of an immutable data type such as strings, numbers, or tuples.

Accessing Values in Dictionary

To access dictionary elements, you can use the familiar square brackets along with the key to obtain its value. Following is a simple example −

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

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print("dict['Name']: ", dict['Name'])

print("dict['Age']: ", dict['Age'])

When the above code is executed, it produces the following result −

dict['Name']: Zara

dict['Age']: 7

If we attempt to access a data item with a key, which is not part of the dictionary, we get an error as follows −

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

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

print("dict['Alice']: ", dict['Alice'])

When the above code is executed, it produces the following result −

dict['Alice']:

Traceback (most recent call last):

File "test.py", line 4, in <module>

print "dict['Alice']: ", dict['Alice'];

KeyError: 'Alice'

Updating Dictionary

You can update a dictionary by adding a new entry or a key-value pair, modifying an existing entry, or deleting an existing entry as shown below in the simple example −

[Live Demo](http://tpcg.io/porcg2)

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

dict['Age'] = 8; # update existing entry

dict['School'] = "DPS School"; # Add new entry

print("dict['Age']: ", dict['Age'])

print("dict['School']: ", dict['School'])

When the above code is executed, it produces the following result −

dict['Age']: 8

dict['School']: DPS School

Delete Dictionary Elements

You can either remove individual dictionary elements or clear the entire contents of a dictionary. You can also delete entire dictionary in a single operation.

To explicitly remove an entire dictionary, just use the **del** statement. Following is a simple example −

[Live Demo](http://tpcg.io/mbvT73)

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Class': 'First'}

del dict['Name']; # remove entry with key 'Name'

dict.clear(); # remove all entries in dict

del dict ; # delete entire dictionary

print("dict['Age']: ", dict['Age'])

print("dict['School']: ", dict['School'])

This produces the following result. Note that an exception is raised because after **del dict** dictionary does not exist any more −

dict['Age']:

Traceback (most recent call last):

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

print "dict['Age']: ", dict['Age'];

TypeError: 'type' object is unsubscriptable

**Note** − del() method is discussed in subsequent section.

Properties of Dictionary Keys

Dictionary values have no restrictions. They can be any arbitrary Python object, either standard objects or user-defined objects. However, same is not true for the keys.

There are two important points to remember about dictionary keys −

**(a)** More than one entry per key not allowed. Which means no duplicate key is allowed. When duplicate keys encountered during assignment, the last assignment wins. For example −

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

#!/usr/bin/python

dict = {'Name': 'Zara', 'Age': 7, 'Name': 'Manni'}

print("dict['Name']: ", dict['Name']0

When the above code is executed, it produces the following result −

dict['Name']: Manni

**(b)** Keys must be immutable. Which means you can use strings, numbers or tuples as dictionary keys but something like ['key'] is not allowed. Following is a simple example −

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

#!/usr/bin/python

dict = {['Name']: 'Zara', 'Age': 7}

print("dict['Name']: ", dict['Name'])

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 3, in <module>

dict = {['Name']: 'Zara', 'Age': 7};

TypeError: unhashable type: 'list'

Built-in Dictionary Functions & Methods

Python includes the following dictionary functions −

|  |  |
| --- | --- |
| **Sr.No.** | **Function with Description** |
| 1 | [cmp(dict1, dict2)](https://www.tutorialspoint.com/python/dictionary_cmp.htm)  Compares elements of both dict. |
| 2 | [len(dict)](https://www.tutorialspoint.com/python/dictionary_len.htm)  Gives the total length of the dictionary. This would be equal to the number of items in the dictionary. |
| 3 | [str(dict)](https://www.tutorialspoint.com/python/dictionary_str.htm)  Produces a printable string representation of a dictionary |
| 4 | [type(variable)](https://www.tutorialspoint.com/python/dictionary_type.htm)  Returns the type of the passed variable. If passed variable is dictionary, then it would return a dictionary type. |

Python includes following dictionary methods −

|  |  |
| --- | --- |
| **Sr.No.** | **Methods with Description** |
| 1 | [dict.clear()](https://www.tutorialspoint.com/python/dictionary_clear.htm)  Removes all elements of dictionary *dict* |
| 2 | [dict.copy()](https://www.tutorialspoint.com/python/dictionary_copy.htm)  Returns a shallow copy of dictionary *dict* |
| 3 | [dict.fromkeys()](https://www.tutorialspoint.com/python/dictionary_fromkeys.htm)  Create a new dictionary with keys from seq and values *set* to *value*. |
| 4 | [dict.get(key, default=None)](https://www.tutorialspoint.com/python/dictionary_get.htm)  For *key* key, returns value or default if key not in dictionary |
| 5 | [dict.has\_key(key)](https://www.tutorialspoint.com/python/dictionary_has_key.htm)  Returns *true* if key in dictionary *dict*, *false* otherwise |
| 6 | [dict.items()](https://www.tutorialspoint.com/python/dictionary_items.htm)  Returns a list of *dict*'s (key, value) tuple pairs |
| 7 | [dict.keys()](https://www.tutorialspoint.com/python/dictionary_keys.htm)  Returns list of dictionary dict's keys |
| 8 | [dict.setdefault(key, default=None)](https://www.tutorialspoint.com/python/dictionary_setdefault.htm)  Similar to get(), but will set dict[key]=default if *key* is not already in dict |
| 9 | [dict.update(dict2)](https://www.tutorialspoint.com/python/dictionary_update.htm)  Adds dictionary *dict2*'s key-values pairs to *dict* |
| 10 | [dict.values()](https://www.tutorialspoint.com/python/dictionary_values.htm)  Returns list of dictionary *dict*'s values |

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Classroom Codes :

## Notepad version:

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Data Type Conversion

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function.

There are several built-in functions to perform conversion from one data type to another. These functions return a new object representing the converted value.

|  |  |
| --- | --- |
| **Sr.No.** | **Function & Description** |
| 1 | **int(x [,base])**  Converts x to an integer. base specifies the base if x is a string. |
| 2 | **long(x [,base] )**  Converts x to a long integer. base specifies the base if x is a string. |
| 3 | **float(x)**  Converts x to a floating-point number. |
| 4 | **complex(real [,imag])**  Creates a complex number. |
| 5 | **str(x)**  Converts object x to a string representation. |
| 6 | **repr(x)**  Converts object x to an expression string. |
| 7 | **eval(str)**  Evaluates a string and returns an object. |
| 8 | **tuple(s)**  Converts s to a tuple. |
| 9 | **list(s)**  Converts s to a list. |
| 10 | **set(s)**  Converts s to a set. |
| 11 | **dict(d)**  Creates a dictionary. d must be a sequence of (key,value) tuples. |
| 12 | **frozenset(s)**  Converts s to a frozen set. |
| 13 | **chr(x)**  Converts an integer to a character. |
| 14 | **unichr(x)**  Converts an integer to a Unicode character. |
| 15 | **ord(x)**  Converts a single character to its integer value. |
| 16 | **hex(x)**  Converts an integer to a hexadecimal string. |
| 17 | **oct(x)**  Converts an integer to an octal string. |

**Decision Making :**

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce TRUE or FALSE as outcome. You need to determine which action to take and which statements to execute if outcome is TRUE or FALSE otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages −



Python programming language assumes any **non-zero** and **non-null** values as TRUE, and if it is either **zero** or **null**, then it is assumed as FALSE value.

Python programming language provides following types of decision making statements. Click the following links to check their detail.

|  |  |
| --- | --- |
| **Sr.No.** | **Statement & Description** |
| 1 | [if statements](https://www.tutorialspoint.com/python/python_if_statement.htm)  An **if statement** consists of a boolean expression followed by one or more statements. |
| 2 | [if...else statements](https://www.tutorialspoint.com/python/python_if_else.htm)  An **if statement** can be followed by an optional **else statement**, which executes when the boolean expression is FALSE. |
| 3 | [nested if statements](https://www.tutorialspoint.com/python/nested_if_statements_in_python.htm)  You can use one **if** or **else if** statement inside another **if** or **else if** statement(s). |

Let us go through each decision making briefly −

Single Statement Suites

If the suite of an **if** clause consists only of a single line, it may go on the same line as the header statement.

Here is an example of a **one-line if** clause −

[Live Demo](http://tpcg.io/6fSJOo)

#!/usr/bin/python

var = 100

if ( var == 100 ) : print("Value of expression is 100")

print("Good bye!")

When the above code is executed, it produces the following result −

Value of expression is 100

Good bye!

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Classroom Codes :

## Notepad version

## Loops :

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement −



Python programming language provides following types of loops to handle looping requirements.

|  |  |
| --- | --- |
| **Sr.No.** | **Loop Type & Description** |
| 1 | WHILE Loop :  Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body. |
| 2 | FOR Loop :  Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable. |
|  |  |

Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed.

Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly

|  |  |
| --- | --- |
| **Sr.No.** | **Control Statement & Description** |
| 1 | Break :  Terminates the loop statement and transfers execution to the statement immediately following the loop. |
| 2 | Continue :  Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating. |
| 3 | Pass :  The pass statement in Python is used when a statement is required syntactically but you do not want any command or code to execute. |

## While loop :

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.

### Syntax

The syntax of a **while** loop in Python programming language is −

while expression:

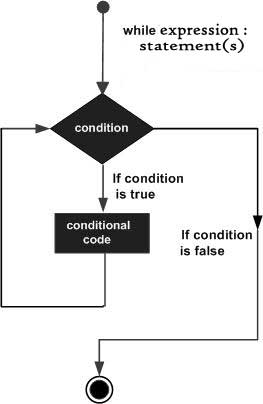
statement(s)

Here, **statement(s)** may be a single statement or a block of statements. The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true.

When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.

### Flow Diagram



Here, key point of the while loop is that the loop might not ever run. When the condition is tested and the result is false, the loop body will be skipped and the first statement after the while loop will be executed.

### Example

#!/usr/bin/python

count = 0

while (count < 9):

print('The count is:', count)

count = count + 1

print("Good bye!")

When the above code is executed, it produces the following result −

The count is: 0

The count is: 1

The count is: 2

The count is: 3

The count is: 4

The count is: 5

The count is: 6

The count is: 7

The count is: 8

Good bye!

The block here, consisting of the print and increment statements, is executed repeatedly until count is no longer less than 9. With each iteration, the current value of the index count is displayed and then increased by 1.

## The Infinite Loop

A loop becomes infinite loop if a condition never becomes FALSE. You must use caution when using while loops because of the possibility that this condition never resolves to a FALSE value. This results in a loop that never ends. Such a loop is called an infinite loop.

An infinite loop might be useful in client/server programming where the server needs to run continuously so that client programs can communicate with it as and when required.

#!/usr/bin/python

var = 1

while var == 1 : # This constructs an infinite loop

num = raw\_input("Enter a number :")

print("You entered: ", num)

print("Good bye!")

When the above code is executed, it produces the following result −

Enter a number :20

You entered: 20

Enter a number :29

You entered: 29

Enter a number :3

You entered: 3

Enter a number between :Traceback (most recent call last):

File "test.py", line 5, in <module>

num = raw\_input("Enter a number :")

KeyboardInterrupt

Above example goes in an infinite loop and you need to use CTRL+C to exit the program.

## Using else Statement with While Loop

Python supports to have an **else** statement associated with a loop statement.

* If the **else** statement is used with a **while** loop, the **else** statement is executed when the condition becomes false.

The following example illustrates the combination of an else statement with a while statement that prints a number as long as it is less than 5, otherwise else statement gets executed.

#!/usr/bin/python

count = 0

while count < 5:

print(count, " is less than 5")

count = count + 1

else:

print(count, " is not less than 5")

When the above code is executed, it produces the following result −

0 is less than 5

1 is less than 5

2 is less than 5

3 is less than 5

4 is less than 5

5 is not less than 5

## Single Statement Suites

Similar to the **if** statement syntax, if your **while** clause consists only of a single statement, it may be placed on the same line as the while header.

Here is the syntax and example of a **one-line while** clause −

#!/usr/bin/python

flag = 1

while (flag): print('Given flag is really true!')

print("Good bye!")

It is better not try above example because it goes into infinite loop and you need to press CTRL+C keys to exit.

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Classroom Codes :

## Notepad version

**For loop :**

It has the ability to iterate over the items of any sequence, such as a list or a string.

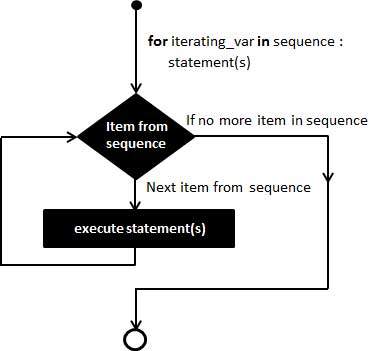
### Syntax

for iterating\_var in sequence:

statements(s)

If a sequence contains an expression list, it is evaluated first. Then, the first item in the sequence is assigned to the iterating variable *iterating\_var*. Next, the statements block is executed. Each item in the list is assigned to *iterating\_var*, and the statement(s) block is executed until the entire sequence is exhausted.

### Flow Diagram



### Example

#!/usr/bin/python

for letter in **'Python':** # First Example

print('Current Letter :', letter)

fruits = ['banana', 'apple', 'mango']

for fruit in **fruits**: # Second Example

print('Current fruit :', fruit)

print ("Good bye!")

When the above code is executed, it produces the following result −

Current Letter : P

Current Letter : y

Current Letter : t

Current Letter : h

Current Letter : o

Current Letter : n

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

## Iterating by Sequence Index

An alternative way of iterating through each item is by index offset into the sequence itself. Following is a simple example −

#!/usr/bin/python

fruits = ['banana', 'apple', 'mango']

for index in range(len(fruits)):

print('Current fruit :', fruits[index])

print("Good bye!")

When the above code is executed, it produces the following result −

Current fruit : banana

Current fruit : apple

Current fruit : mango

Good bye!

Here, we took the assistance of the len() built-in function, which provides the total number of elements in the tuple as well as the range() built-in function to give us the actual sequence to iterate over.

## Using else Statement with For Loop

Python supports to have an else statement associated with a loop statement

* If the **else** statement is used with a **for** loop, the **else** statement is executed when the loop has exhausted iterating the list.

The following example illustrates the combination of an else statement with a for statement that searches for prime numbers from 10 through 20.

#!/usr/bin/python

for num in range(10,20): #to iterate between 10 to 20

for i in range(2,num): #to iterate on the factors of the number

if num%i == 0: #to determine the first factor

j=num/i #to calculate the second factor

print('%d equals %d \* %d' % (num,i,j))

break #to move to the next number, the #first FOR

else: # else part of the loop

print(num, 'is a prime number')

break

When the above code is executed, it produces the following result −

10 equals 2 \* 5

11 is a prime number

12 equals 2 \* 6

13 is a prime number

14 equals 2 \* 7

15 equals 3 \* 5

16 equals 2 \* 8

17 is a prime number

18 equals 2 \* 9

19 is a prime number

## \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

## Classroom Codes :

## Notepad version

## Functions :

A function is a block of organized, reusable code that is used to perform a single, related action. Functions provide better modularity for your application and a high degree of code reusing.

As you already know, Python gives you many built-in functions like print(), etc. but you can also create your own functions. These functions are called *user-defined functions.*

Defining a Function

You can define functions to provide the required functionality. Here are simple rules to define a function in Python.

* Function blocks begin with the keyword **def** followed by the function name and parentheses ( ( ) ).
* Any input parameters or arguments should be placed within these parentheses. You can also define parameters inside these parentheses.
* The first statement of a function can be an optional statement - the documentation string of the function or *docstring*.
* The code block within every function starts with a colon (:) and is indented.
* The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

Syntax

def functionname( parameters ):

"function\_docstring"

function\_suite

return [expression]

By default, parameters have a positional behavior and you need to inform them in the same order that they were defined.

Example

The following function takes a string as input parameter and prints it on standard screen.

def printme( str ):

"This prints a passed string into this function"

Print(str)

return

Calling a Function

Defining a function only gives it a name, specifies the parameters that are to be included in the function and structures the blocks of code.

Once the basic structure of a function is finalized, you can execute it by calling it from another function or directly from the Python prompt. Following is the example to call printme() function −

[Live Demo](http://tpcg.io/3mrnYY)

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

Print(str)

return;

# Now you can call printme function

printme("I'm first call to user defined function!")

printme("Again second call to the same function")

When the above code is executed, it produces the following result −

I'm first call to user defined function!

Again second call to the same function

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.

#!/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]

Function Arguments

You can call a function by using the following types of formal arguments −

* Required arguments
* Keyword arguments
* Default arguments
* Variable-length arguments

Required arguments

Required arguments are the arguments passed to a function in correct positional order. Here, the number of arguments in the function call should match exactly with the function definition.

To call the function *printme()*, you definitely need to pass one argument, otherwise it gives a syntax error as follows −

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

Print(str)

return;

# Now you can call printme function

printme()

When the above code is executed, it produces the following result −

Traceback (most recent call last):

File "test.py", line 11, in <module>

printme();

TypeError: printme() takes exactly 1 argument (0 given)

Keyword arguments

Keyword arguments are related to the function calls. When you use keyword arguments in a function call, the caller identifies the arguments by the parameter name.

This allows you to skip arguments or place them out of order because the Python interpreter is able to use the keywords provided to match the values with parameters. You can also make keyword calls to the *printme()* function in the following ways −

#!/usr/bin/python

# Function definition is here

def printme( str ):

"This prints a passed string into this function"

Print(str)

return;

# Now you can call printme function

printme( str = "My string")

When the above code is executed, it produces the following result −

My string

The following example gives more clear picture. Note that the order of parameters does not matter.

#!/usr/bin/python

# Function definition is here

def printinfo( name, age ):

"This prints a passed info into this function"

Print("Name: ", name)

Print("Age ", age)

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Default arguments

A default argument is an argument that assumes a default value if a value is not provided in the function call for that argument. The following example gives an idea on default arguments, it prints default age if it is not passed −

#!/usr/bin/python

# Function definition is here

def printinfo( name, age = 35 ):

"This prints a passed info into this function"

Print("Name: ", name)

Print("Age ", age)

return;

# Now you can call printinfo function

printinfo( age=50, name="miki" )

printinfo( name="miki" )

When the above code is executed, it produces the following result −

Name: miki

Age 50

Name: miki

Age 35

Variable-length arguments

You may need to process a function for more arguments than you specified while defining the function. These arguments are called *variable-length* arguments and are not named in the function definition, unlike required and default arguments.

Syntax for a function with non-keyword variable arguments is this −

def functionname([formal\_args,] \*var\_args\_tuple ):

"function\_docstring"

function\_suite

return [expression]

An asterisk (\*) is placed before the variable name that holds the values of all nonkeyword variable arguments. This tuple remains empty if no additional arguments are specified during the function call. Following is a simple example −

#!/usr/bin/python

# Function definition is here

def printinfo( arg1, \*vartuple ):

"This prints a variable passed arguments"

Print("Output is: ")

Print(arg1)

for var in vartuple:

print(var)

return;

# Now you can call printinfo function

printinfo( 10 )

printinfo( 70, 60, 50 )

When the above code is executed, it produces the following result −

Output is:

10

Output is:

70

60

50

The *Anonymous* Functions

These functions are called anonymous because they are not declared in the standard manner by using the *def* keyword. You can use the *lambda* keyword to create small anonymous functions.

* Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.
* An anonymous function cannot be a direct call to print because lambda requires an expression
* Lambda functions have their own local namespace and cannot access variables other than those in their parameter list and those in the global namespace.
* Although it appears that lambda's are a one-line version of a function, they are not equivalent to inline statements in C or C++, whose purpose is by passing function stack allocation during invocation for performance reasons.

Syntax

The syntax of *lambda* functions contains only a single statement, which is as follows −

lambda [arg1 [,arg2,.....argn]]:expression

Following is the example to show how *lambda* form of function works −

#!/usr/bin/python

# Function definition is here

sum = lambda arg1, arg2: arg1 + arg2;

# Now you can call sum as a function

Print("Value of total : ", sum( 10, 20 ))

Print("Value of total : ", sum( 20, 20 ))

When the above code is executed, it produces the following result −

Value of total : 30

Value of total : 40

The *return* Statement

The statement return [expression] exits a function, optionally passing back an expression to the caller. A return statement with no arguments is the same as return None.

All the above examples are not returning any value. You can return a value from a function as follows −

#!/usr/bin/python

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2

print("Inside the function : ", total)

return total;

# Now you can call sum function

total = sum( 10, 20 );

print("Outside the function : ", total)

When the above code is executed, it produces the following result −

Inside the function : 30

Outside the function : 30

Scope of Variables

All variables in a program may not be accessible at all locations in that program. This depends on where you have declared a variable.

The scope of a variable determines the portion of the program where you can access a particular identifier. There are two basic scopes of variables in Python −

* Global variables
* Local variables

Global vs. Local variables

Variables that are defined inside a function body have a local scope, and those defined outside have a global scope.

This means that local variables can be accessed only inside the function in which they are declared, whereas global variables can be accessed throughout the program body by all functions. When you call a function, the variables declared inside it are brought into scope. Following is a simple example −

#!/usr/bin/python

total = 0; # This is global variable.

# Function definition is here

def sum( arg1, arg2 ):

# Add both the parameters and return them."

total = arg1 + arg2; # Here total is local variable.

Print("Inside the function local total : ", total)

return total;

# Now you can call sum function

sum( 10, 20 );

print("Outside the function global total : ", total)

When the above code is executed, it produces the following result −

Inside the function local total : 30

Outside the function global total : 0