In while loop first the test case is evaluated, if the test case is True then the body of while loop gets executed and repeated untill the test case evaluates to false. After every iteration the test case is evaluated.

Python interprets any non-zero value as T**rue**. None and 0 are interpreted as False.

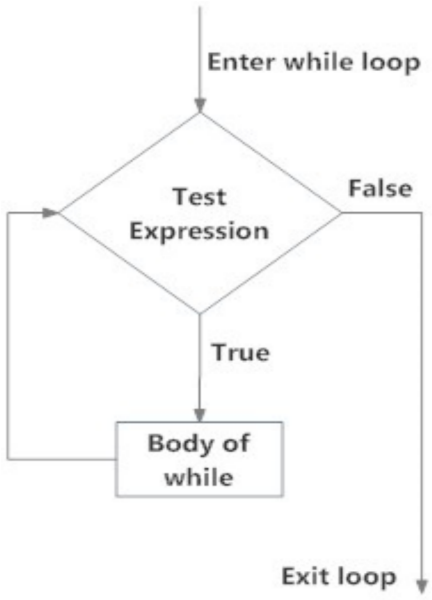


Fig: operation of while loop

Flowchart of while Loop

Example

```
#!/usr/bin/python


count = 0
while (count < 12):
   print 'The count is:', count
   count = count + 1


print "Good bye!"
```

For loops

For loop can iterate over any sequences like list, tuple or string which are indexed. Every element in these datatypes are indexed can be iterated over. This process of iterating over a sequence is also called as traversing or traversal in nature.

```
for variable1 in sequence:
        Body of for
```

Here, **variable1** is the variable that takes the value of the item inside the sequence on each iteration.

Loop continues until we reach the last item in the sequence. The body of for loop is separated from the rest of the code using indentation.

Flowchart of for Loop

Fig: operation of for loop

```
#!/usr/bin/python

for word in 'DevOps!':   # First Example
  print 'Current Word :', word


planets = ['Mercury', 'Venus','Earth']
for planet in planets:  # Second Example
  print 'Current planet :', planet
print "Bye!"
```

Iterating over range, range and len function.

```
imran@DevOps:~/.../word$ python
Python 2.7.12+ (default, Sep 17 2016, 12:08:02)
[GCC 6.2.0 20160914] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> fruits = ['banana', 'apple', 'mango']
>>> len(fruits)
3
>>> range(len(fruits))
[0, 1, 2]
>>> range(6)
[0, 1, 2, 3, 4, 5]
```

range() function return a list and for loop can iterate over a list.

```
#!/usr/bin/python

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

for index in range(len(fruits)):

  print 'Current fruit :', fruits[index]

print "Good bye!"
```

Break and Continue

Using **for loops** and **while loops** in Python allow you to automate and repeat tasks in an efficient manner.

But sometimes you wish to interept or change the flow or exit from the loop while its executing.

In Python, break and continue statements can alter the flow of a normal loop.

The break and continue statements are used in these cases.

Python break statement

The `break` statement provides you with the opportunity to exit out of a loop when an external condition is triggered, this external condition could be if block which evaluates to true somehere in the iteration and use break keyword to exit from the loop.

Flowchart of break

The working of break statement in for loop and while loop is shown below.

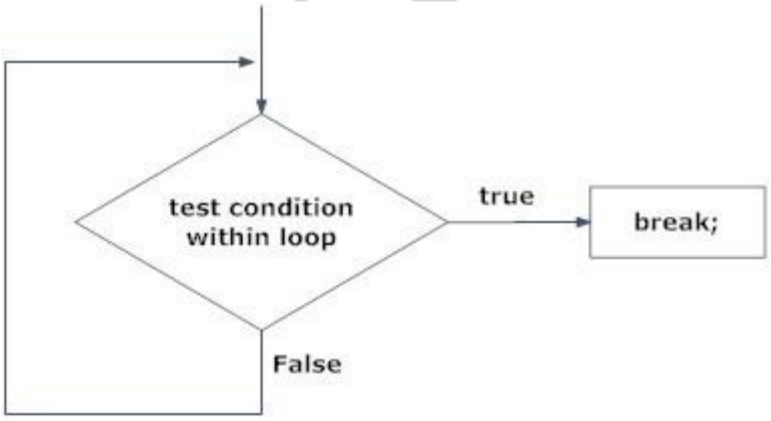
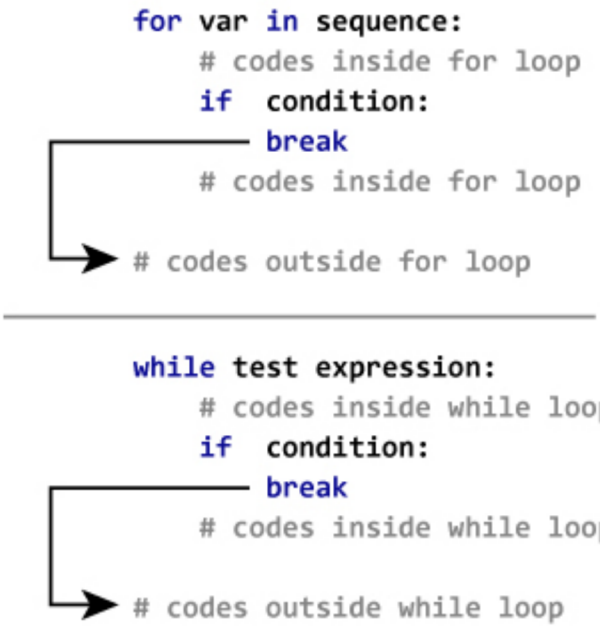


Figure: Flowchart of break statement




```
#!/usr/bin/python

for word in 'Python':
# First Example
  if word == 't':
    break
  print 'Current Letter :', word


a = 10


# Second Example
while a > 0:
  print 'Current variable value :', a
  a = a -1
  if a == 5:
    break


print "Execution completed!"
```

```
Current Letter : P

Current Letter : y
```

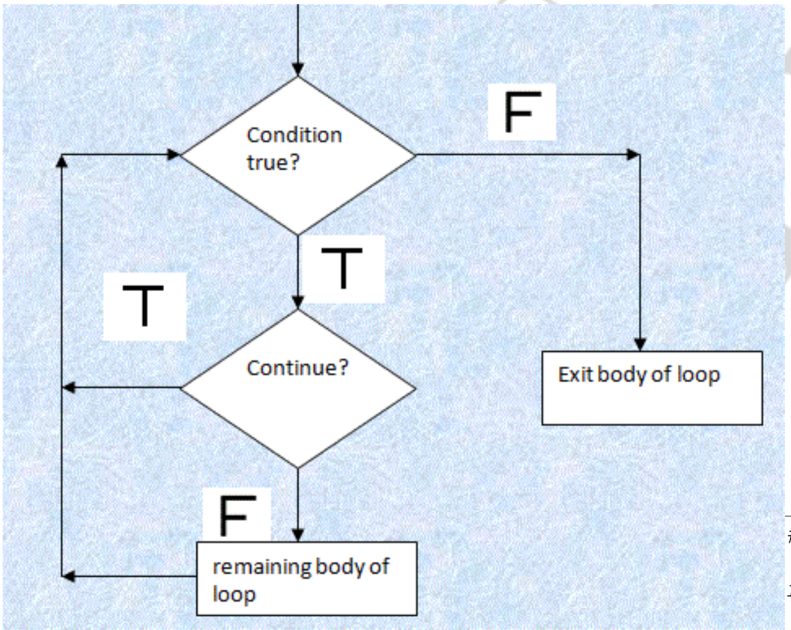
```
Current variable value : 10
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Execution completed!
```

Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

Flowchart of continue

The working of continue statement in for and while loop is shown below.




```
#!/usr/bin/python
for word in 'Python':
# First Example
  if word == 't':
    continue
  print 'Current Letter :', word

a = 10

# Second Example
while a > 0:
```

```
  a = a -1
  if a == 5:
    continue
  print 'Current variable value :', a

print "Execution completed!"
```

```
Current Letter : P
Current Letter : y
Current Letter : h
Current Letter : o
Current Letter : n
Current variable value : 9
Current variable value : 8
Current variable value : 7
Current variable value : 6
Current variable value : 4
Current variable value : 3
Current variable value : 2
Current variable value : 1
Current variable value : 0
Execution completed!
```

7. Built In Methods/Functions

Python has numerous built in methods that extends its capabilities. Listed below are few of the useful built in methods on all the DataType.

Check Python docs for the list of all the methods and its uses.

String

Capitalize: Capitalizes first letter of string.

```
str = "this is string example....wow!!!";
print "str.capitalize() : ", str.capitalize()
```

find: Determine if str occurs in string or in a substring.

```
str1 = "this is string example....wow!!!";

str2 = "exam";

print str1.find(str2);

print str1.find(str2, 10);

print str1.find(str2, 40);
```

join: Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.

```
str = "-";

seq = ("a", "b", "c"); # This is sequence of strings.

print str.join( seq );
```

rstrip: Removes all trailing whitespace of string.

```
a = "Hello      "

a.rstrip()

Hello
```

split: Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.

```
str = "Line1-abcdef \nLine2-abc \nLine4-abcd";

print str.split( );

['Line1-abcdef', 'Line2-abc', 'Line4-abcd']
```

Lists

append: Adds element at the end of the list

```
aList = [123, 'xyz', 'zara', 'abc'];

aList.append( 2009 );

print "Updated List : ", aList;
```

extend: Combine two list together

```
aList = [123, 'xyz', 'zara', 'abc', 123];

bList = [2009, 'manni'];

aList.extend(bList)

print "Extended List : ", aList ;
```

insert: Insert a element into the specified index value.

```
aList = [123, 'xyz', 'zara', 'abc']

aList.insert( 3, 2009)

print "Final List : ", aList
```

pop: Removes element from list at a specified index value, default is last element.

```
aList = [123, 'xyz', 'zara', 'abc'];

print "A List : ", aList.pop();

print "B List : ", aList.pop(2);
```

Dictionary

Update dictionary elements

```
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'];
```

delete dictionary elements

```
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'];
```

8. 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.

We have already used some built in functions like range(), len(), type() etc. We will learn now how to write our own functions.

Functions rules

Function block begins with the keyword def followed by function name and parantheses.
Syntax

```
def func_name()
```

If any arguments has to passed to the function it should be defined under paranthesis.
Syntax

```
def func_name(arg1, arg2, arg3)
```

Code block starts with a colon :
Syntax

```
def func_name(arg1,arg2):
   code block
```

Keyword “return” in the code block exits the function and can return some value, if no value is returned default return value is “None”.
Syntax

```
def func_name(arg1,arg2):
   code block
   return variable
def func_name(arg1,arg2):
   code block
   return # returns value as None
```

Defining & Calling a function.

```
#!/usr/bin/python


def print_func( var1 ):
  "This prints a passed string into this function"
  print var1
  return


# Calling print_func
print_func("Testing function calling.")


# Printing & Calling print_func
```

```
print print_func("Testing function calling.")



# Storing print_func returned value


returnedvalue =  print_func("Testing function calling.")
print returnedvalue
```

Output

```
Testing function calling.
Testing function calling.
None
Testing function calling.
None
```

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:
Age
miki
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:
Age
50
Name:
Age
miki
miki
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 non keyword 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 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 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
```

9. Modules

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the `import` command.

We use modules to break down large programs into small manageable and organized files. Furthermore, modules provide reusability of code.

We can define our most used functions in a module and import it, instead of copying their definitions into different programs.

Example script name is modexa.py

```
imran@DevOps:/tmp$ cat modexa.py

def multi(a, b):

  c = a * b

  return c


def hello():

  print "Hello from modexa module."
```

Importing a module and calling methods inside it.

```
imran@DevOps:/tmp$ python
Python 2.7.12+ (default, Sep 17 2016, 12:08:02)
[GCC 6.2.0 20160914] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import modexa
>>> print modexa.multi(3,4)
12
>>> modexa.hello()
Hello from modexa module.
```

The from...import Statement

Python's from statement lets you import specific attributes from a module into the

current namespace. The from...import has the following syntax:

```
from modname import name1[, name2[, ... nameN]]
```

For example, to import the function fibonacci from the module fib, use the following statement:

```
from fib import fibonacci
```

This statement does not import the entire module fib into the current namespace; it just introduces the item fibonacci from the module fib into the global symbol table of the importing module.

The from...import * Statement:

It is also possible to import all names from a module into the current namespace by using the following import statement:

```
from modname import *
```

This provides an easy way to import all the items from a module into the current

The dir( ) Function

The dir() built-in function returns a sorted list of strings containing the names defined by a module.

The list contains the names of all the modules, variables and functions that are defined in a module. Following is a simple example:

```
#!/usr/bin/python

# Import built-in module math

import math

content = dir(math)

print content;
```

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

```
['__doc__', '__file__', '__name__', 'acos', 'asin', 'atan',

'atan2', 'ceil', 'cos', 'cosh', 'degrees', 'e', 'exp',
```

```
'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log',
'log10', 'modf', 'pi', 'pow', 'radians', 'sin', 'sinh',
'sqrt', 'tan', 'tanh']
```

Here, the special string variable __name__ is the module's name, and __file__ is the.

filename from which the module was loaded.

10. Python for OS tasks.

Python is very widely used as a scripting language to automate day to day system admin tasks or even a full scale Orchestration of multiple systems.

The OS module in Python provides a way of using operating system dependent functionality.

The functions that the OS module provides allows you to interface with the underlying operating system that Python is running on. (Windows, Mac or Linux.

You can find important information about your location or about the process.

Before we start, make sure that you have imported the OS module "*import os*"

OS Functions explained`

```
os.system("ls")      # Executing a shell command

os.chdir()       # Move focus to a different directory

os.getcwd()      # Returns the current working directory

os.getpid()      # Returns the real process ID of the current process

os.chmod()       # Change the mode of path to the numeric mode

os.chown()       # Change the owner and group id

os.getsize()     # Get the size of a file



os.uname()       # Return information about the current operating system
```

```
os.listdir(path)# List of the entries in the directory given by path


os.path.exists()# Check if a path exists


os.mkdir(path)  # Create a directory named path with numeric mode mode


os.remove(path) # Remove (delete) the file path


os.rmdir(path)  # Remove (delete) the directory path


os.makedirs(path)# Recursive directory creation function


os.removedirs(path) # Remove directories recursively


os.rename(src, dst) # Rename the file or directory src to dst
```

OS Functions Examples

```
imran@DevOps:.../pysys$ ls
devdir  sometext.txt  testdir  testfile
imran@DevOps:.../pysys$ python
Python 2.7.12+ (default, Sep 17 2016, 12:08:02)
[GCC 6.2.0 20160914] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import os
>>> print os.getcwd()
/tmp/pysys
>>> os.system("ls -l")
total 12
drwxrwxr-x 2 imran imran 4096 Jun  1 05:41 devdir
-rw-rw-r-- 1 imran imran   29 Jun  1 05:33 sometext.txt
drwxrwxr-x 2 imran imran 4096 Jun  1 05:33 testdir
-rw-rw-r-- 1 imran imran    0 Jun  1 05:44 testfile
0
>>> print os.getpid()
23405
>>> os.chmod("/tmp/pysys/testfile", 700)
>>> if os.path.exists("/tmp/pysys/testfile"):
...     os.mkdir("/tmp/pysys/testdir123")
...
```

Check for directory or file.

```
path = "/tmp/pysys"

if os.path.isdir(path):
```

```
  print "It is a directory."
if os.path.isfile(path):
  print "It is  a file."
```

Creates a directory

```
print os.mkdir('devopsdir', 0750)
```

Remove a directory with os.rmdir()

```
print os.rmdir('directory')
```

Rename a file with os.rename()

```
print os.rename('/path/to/old/file', '/path/to/new/file')
```

Move focus to a different directory with os.chdir()

```
print os.chdir('/tmp')
```

Print out all directories, sub-directories and files with os.walk()

```
for root, dirs, files in os.walk("/tmp"):
    print root
    print dirs
    print files
```

Returns a list of all files in a directory with os.listdir()

```
for filename in os.listdir("/tmp"):
    print "This is inside /tmp", filename
```

Get the size of a file with os.path.getsize()

```
path.getsize("/tmp/file.txt")
```

11. Fabric for automation

What is Fabric?

As the README says:

Fabric is a Python (2.5-2.7) library and command-line tool for streamlining the use of SSH for application deployment or systems administration tasks.

More specifically, Fabric is:

A tool that lets you execute **arbitrary Python functions** via the **command line**;

A library of subroutines (built on top of a lower-level library) to make executing shell commands over SSH **easy** and **Pythonic**.

Naturally, most users combine these two things, using Fabric to write and execute Python functions, or **tasks**, to automate interactions with remote servers

Installing Fabric

Fabric module does not comes inbuilt in python but can be installed with python package manager like “pip”.**pip** is a package management system used to install and manage software packages written in **Python**.

First we have to install pip.

Do I need to install pip?

pip is already installed if you're using Python 2 >=2.7.9 or Python 3 >=3.4 binaries downloaded from python.org, but you'll need to upgrade pip.

Installing with get-pip.py

To install pip, securely download https://bootstrap.pypa.io/get-pip.py

Then run the following:

```
python get-pip.py
```

Installing fabric with pip.

```
pip install fabric
```

Fabfile or Fab scripts

Fabric will load fabfile.py and run the functions that we defined in it.

Create a python script named fabfile.py and define functions in it.

For Example:

```
File Edit View Search Terminal Help
imran@DevOps:.../fabric-scripts$ cat fabfile.py
def hello():
  print("Hello DevOps!")
imran@DevOps:.../fabric-scripts$ fab hello
Hello DevOps!

Done.
```

Fab commands are executed from bash shell, when fab command is executed it looks for fabfile.py script locally and call the function inside it.

Task arguments

It's often useful to pass runtime parameters into your tasks, just as you might during regular Python programming.

```
imran@DevOps:.../fabric-scripts$ cat fabfile.py
def hello(name="world"):
  print "Hello", name
imran@DevOps:.../fabric-scripts$ fab hello
Hello world

Done.
imran@DevOps:.../fabric-scripts$ fab hello:devops
Hello devops

Done.
```

Fabric functions

Fabric provides a set of commands in fabric API that are simple but powerful.

With Fabric, you can use simple Fabric calls like

local # execute a local command)

run # execute a remote command on all specific hosts, user-level permissions)

sudo # sudo a command on the remote server)

put # copy over a local file to a remote destination)

get # download a file from the remote server)

prompt # prompt user with text and return the input (like raw_input))

reboot # reboot the remote system, disconnect, and wait for wait seconds)

To test run, sudo, get, put & reboot functions you will need a remote linux system. Could be a vm or cloud instance.

```
imran@DevOps:.../fabric-scripts$ ls
devfile4  fabfile.py  fabfile.pyc  Vagrantfile
```

```
from fabric.api import *


def hello(name="world"):
  print "Hello", name
```