folders. Without a common specification in place for storing folders, certain S3 client tools will build directory structures that are not compatible with one another. In my example, the folders I created in the AWS Management Console were not compatible with s3fs.

Unfortunately, the only solution for now is to adopt a single S3 tool exclusively to modify the contents of a bucket. For the S3 buckets I intend to mount, I will only be using s3fs against the contents of those buckets.

(server side copies are not possible - due to how FUSE orchestrates the low level instructions, the file must first be downloaded to the client and then uploaded to the new location)

Conclusions

s3fs: safe, efficient storage of medium-large files. Perfect for backup / archiving purposes.

Summary:

- ✓ Cloud computing is the way forward in IT industry, there is no going back now. Majority of the people in IT industry some or the other way are consuming Cloud based services.
- ✓ Amazon Web Services is the market leader ind Public domain. That is because its very innovative and every now and then comes up with new services and features.
- ✓ If you are using AWS you should be checking its news all the time.
- ✓ https://aws.amazon.com/new/
- ✓ AWS services can be broadly divied into three categoris, SysOps, Developers and DevOps services. There major push is in Developers services now and majoriy of AWS services are Developers services, currently.
- ✓ But Most widely used services are SysOps services though, like Ec2, RDS, Route53, VPC etc.
- ✓ AWS services can be accessed by Commad lines, scripts and configuration management tools. This is called as programatic access and is used for automation.
- ✓ S3cmd, s3fs, AWSCli, are some command line tools to manage AWS services.

Conclusions:

In DevOps field you would be using lots cloud services and AWS is most widely used among them.

There are other Cloud Providers like Microsoft Azure, Google Cloud & Rackspace.

After learning AWS services you should work on automating those services through your scripts or tools which we will see in later chapters.

VI Python Scripting.

1. Python Introduction

Python is a high-level, interpreted, interactive and object-oriented scripting language. Python is designed to be highly readable.

So why is python tutorial in this book?

When we were writing bash script for doing automation, slowly we increased the complexity of our code. Our code became more and more complex. Also, we do not have any other feature in bash scripting apart from just automating Linux tasks.

Python is among the easiest programming language out there.

Being easy it gives us lots of feature and libraries which extends the power of python. Python is very extensible. We can use python to run Linux automation, windows automation, Cloud automation and several others. It's very versatile in nature.

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.

Installing python.

Windows Installation

Here are the steps to install Python on Windows machine.

Open a Web browser and go to http://www.python.org/download/. Follow the link for the Windows installer python-XYZ.msi file where XYZ is the version you need to install.

To use this installer python-XYZ.msi, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.

Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

After installation we have to setup Python PATH in the system environment variable.

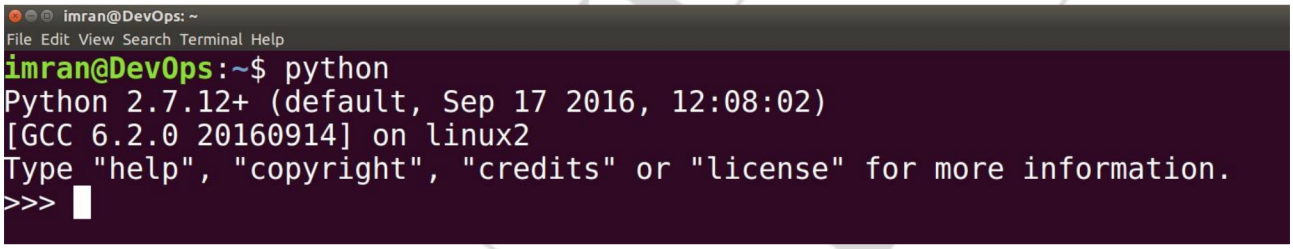
Linux Installation

Linux system by default comes with python installed so it is not required to take any action for Linux system.

2. Basic Syntax

Python Interactive programming

Open Linux shell => Type python and hit enter, this will drop you into the python shell/interpreter.



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

Python scripting

Create a file named hello.py, **py** stands for python and is extension of python script file.

Add below mentioned content and save file.

```
#!/usr/bin/python

print "Hello, DevOps!"
```

Line 1- #! is the shebang charect as we know from bash scripting, Interpreter path is /usr/bin/python. You can find python interpreter path in your Linux system by giving "which python". 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.

Line 2- print is the python command which prints messages.

Executing python script.

Now, try to run this program as follows:

$ python hello.py

or make the hello.py executable by giving command “chmod u+x hello.py” and execute as below.

$./hello.py

This produces the following result:

Hello, DevOps!

Lines and Indentation

Beware of spaces!

Python program statements are written one above the other, as shown in the following example:

```
print “Python is a scripting language.”
print "We are using it to learn automation."
print "Python is very extensible."
```

The 3 line program above is correct and will run producing the expected output.

The program below will generate a syntax error

because of the space character inserted at the start of the second line.

```
print “Python is a scripting language.”
   print "We are using it to learn automation."
print "Python is very extensible."
```

Python evaluate the second print statement as a block of code inside first print statement. So its thinking seond print is under first print statement but its a separate statement.

Python provides no braces to indicate blocks of code for class and function definitions

or flow control. Blocks of code are denoted by line indentation, which is rigidly

enforced.

The if statement

The if statement starts with the keyword "if"

followed by a test case (e.g. x == 30), followed by a colon (:).

```
x = 30
if x == 30:
  print ‘x is not equal to 30’
else:
  print ‘x is not equal to 30’
```

Under the if statement, the statements to be run if the condition is true are entered.

after typing a few space characters. In the example above, two space characters are used.

Any statements to be run after the else: part of the program also have to be spaced out to the ***same level*** by typing 2 space characters before the statement.

Spacing out a program in this way is known as indenting.

Indentation is part of the syntax of the Python language.

The number of spaces in the indentation is variable, but all statements within the block must be indented the same amount. For example:

```
if True:
      print "True"
else:
      print "False"
```

However, the following block generates an error:

```
if True:
      print "Answer"
      print "True"
else:
      print "Answer"
    print "False"
```

Thus, in Python all the continuous lines indented with same number of spaces would

form a block.

Note: Do not try to understand the logic at this point of time. Just make sure you

understood various blocks even if they are without braces.

Quotation in Python

We can use quotes in python for enclosing the text/strings. There are three types of Quotes in python.

- Single Quote (')

  Anything inside single quotes is consider as literal string and will not be evaluated by python.

- Double (")

  Double quotes can be used when you have a variable in the string that needs to be evaluated by python using string formating(%).

- Triple (''' or """)

  This is used to write a multiline string or also called as paragraph string.

For example:

```
word = 'word'
sentence = "This is a sentence."
paragraph = """This is a paragraph. It is
made up of multiple lines and sentences."""
```

Comments in Python

A hash sign (#) is a comment symbol in python. Python ignores all the text after # and does not evaluate it. All characters after the # and up to the end of the physical line are part of the comment and the

Python interpreter ignores them.

```
#!/usr/bin/python
# First comment
```

```
print "Hello, Python!";
# second comment
```

This produces the following result:

```
Hello, Python!
```

You can type a comment on the same line after a statement or expression:

```
name = "fuzi island" # This is a comment
```

You can comment multiple lines as follows:

```
# This is a comment.
# This is a comment, too.
#' This is a comment, too.
# I said that already.
```

Multiline Comment.

We can use paragraph string to create a multiline comment. Everything inside paragraph string would be ignored by Python.

```
#!/usr/bin/python
num = 84
"""This is multiline comment,
Anything enclosed in this will not be evaluated
by python"""
```

3. Variable Types

Variable are temporary storage for data in memory. There are different kinds of data, by assigning different data types to variables, you can store integers, decimals, or characters in these variables.

Variable Assignment.

We don’t need to define variable types in python like other programming language. Variable declaration happens automatically when we 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
mynumber = 86                  # A integer variable assignment
myfloatnumber = 86. 11         # A float variable assignment
myname = “imran”               # A string variable assignment
```

```
print mynumber
print myfloatnumber
print myname
```

Here 86. 86.11 & imran are values assigned to variables names mynumber, myfloatnumber & myname respectively. Execute the above script and check the output.

Multiple Assignment

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

example:

```
x = y = z= 86
```

Here, an integer object is created with the value 86, and all three variables are assigned to the same memory location.

You can also assign multiple objects to multiple variables.

For example:

```
x,y,z = 86, 11, “imran”
```

Here, two integer objects with values 86 and 11 are assigned to variables x and y respectively, and one string object with the value "imran" is assigned to the variable z.

Python 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

Strings

Python strings are simple text data enclosed in single or double quotes

```
stringvar = 'sample string'

strvar2 = 'string#2'
```

Python allows for either pairs of single or double quotes.

```
stringvar = "sample string"

strvar2 = "string#2"
```

String slicing can also be done and stored into a variable, slice means subset or part of the string.

Subsets of strings can be done by 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

teststr = 'Hello DevOps!'

print teststr # Prints complete string

print teststr[0] # Prints first character of the string

print teststr[0:5] # Prints characters starting from 0th to 5th

print teststr[6:12] # Prints characters starting from 6th to 12th

print teststr[3:] # Prints string starting from 3rd character

print teststr * 2 # Prints string two times

print teststr + " Hola" # Prints concatenated string
```

This will produce the following result:

```
Hello DevOps!

H

Hello

DevOps

llo DevOps!

Hello DevOps!Hello DevOps!


Hello DevOps! Hola
```

Lists

Lists are also knows a array of python. It can store multiple values of diffrent datatype into a variable. A list contains items separated by commas and enclosed within square brackets ([]). Its similar to arrays in C but 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
list1 = [ 'ansible', 1111 , 'puppet', 86.96, 'git', 'aws']
list2 = ['miner', 'boy']
print list1 # Prints complete list
print list1[0] # Prints first element of the list
print list1[-1] # Prints last element of the list
print list1[2:4] # Prints elements starting from 2nd till 4th
print list1[3:] # Prints elements starting from 3rd element
print list2 * 2 # Prints list two times
```

This produces the following result:

```
['ansible', 1111, 'puppet', 86.96, 'git', 'aws']
ansible
aws
['puppet', 86.96]
[86.96, 'git', 'aws']
['miner', 'boy', 'miner', 'boy']
['ansible', 1111, 'puppet', 86.96, 'git', 'aws', 'miner', 'boy']
```

Tuples

Tuple is another type of array it looks and feel exactly like a list but tuple are immutable that means you can't edit a tuple just like you can't edit the content in a cd rom. A tuple consists of a

number of values separated by commas. Tuples are enclosed in parenthesis ().Tuples can be thought of as read-only lists. For example:

```
#!/usr/bin/python
tuple1 = ( 'ansible', 1111, 'puppet', 86.96, 'git', 'aws')
tuple2 = ('miner', 'boy')
print tuple1 # Prints complete list
print tuple1[0] # Prints first element of the list
print tuple1[1:3] # Prints elements starting from 2nd till 3rd
print tuple1[2:] # Prints elements starting from 3rd element
print tuple2 * 2 # Prints list two times
print tuple1 + tuple2 # Prints concatenated lists
```

This produces the following result:

```
('ansible', 1111, 'puppet', 86.96, 'git', 'aws')
ansible
(1111, 'puppet')
('puppet', 86.96, 'git', 'aws')
('miner', 'boy', 'miner', 'boy')
('ansible', 1111, 'puppet', 86.96, 'git', 'aws', 'miner', 'boy')
```

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 = ('ansible', 1111, 'puppet', 86.96, 'git', 'aws'
list = ['ansible', 1111, 'puppet', 86.96, 'git', 'aws']
tuple[5] = 'terra' # Invalid syntax with tuple
list[5] = 'terra' # Valid syntax with list
```

Dictionary

In dictionary data in stored in format of key-value pair unlike list or tuple where we just have values. You can think it a any normal english dictionary where word in the key and its meaning is its value.

Python's dictionaries are kind of hash table type. 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

dict1 = {}

dict1['profile'] = "hitman"

dict1['passcode'] = "zeus"

dict2 = {'name': 'imran','code':47, 'dept': 'hitman'}

print dict1['profile'] # Prints value for 'profile' key

print dict1['passcode'] # Prints value for 'passcode' key

print dict2 # Prints complete dictionary

print dict2.keys() # Prints all the keys

print dict2.values() # Prints all the values
```

This produces the following result:

```
hitman

zeus

{'dept': 'hitman', 'code': 47, 'name': 'imran'}

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

['hitman', 47, 'imran']
```

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

4. Python Operators

Operators are the constructs which can manipulate the value of operands.
Consider the expression 4 + 5 = 9. Here, 4 and 5 are called operands and + is called operator.

Types of Operators

Python language supports the following types of operators.

- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

Python Arithmetic Operators

Assume variable x holds 50 and variable y holds 60, then:

+ Addition Adds values on either side of the operator.
x + y = 110

- Subtraction Subtracts right hand operand from left hand operand.
x – y = -10

* Multiplication Multiplies values on either side of the operator
x * y = 200

/ Division Divides left hand operand by right hand operand
x / y = 2

% Modulus Divides left hand operand by right hand operand and returns remainder
x % y = 0

** Exponent Performs exponential (power) calculation on operators
x**y =10 to the power 20

Python Comparison Operators
These operators compare the values on either sides of them and decide the relation among them. They are also called Relational operators.

```
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.
>>> a = 30
>>> b = 40
>>> a == b
False
>>> a != b
True
>>> a > b
False
>>> a < b
True
>>> a >= b
False
>>> a <= b
True
>>>
```

Example

```
#!/usr/bin/python

a = 2

b = 30


if ( a == b ):

  print "a is equal to b"

else:

  print "a is not equal to b"


if ( a != b ):

  print "a is not equal to b"

else:

  print "a is equal to b"


if ( a < b ):

  print "a is less than b"

else:

  print "a is not less than b"


if ( a > b ):

  print "a is greater than b"

else:
```

```
print "a is not greater than b"
```

Once executed it return below output.

a is not equal to b
a is not equal to b
a is less than b
a is not greater than b

Python Logical Operators

There are following logical operators supported by Python language.
If variable a holds 40 and variable b holds 50 then:
Operator Description Example

If both the operands are true then condition
becomes true. (a and b) is true.

If any of the two operands are non-zero
then condition becomes true. (a or b) is true.

Used to reverse the logical state of its
operand. Not (a and b) is false.

Python Membership Operators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators as explained below:

Evaluates to true if it finds a variable in
the specified sequence and false
otherwise.

```
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.
>>> a = 'alpha'
>>> b = ['test', 'alpha', 'team']
>>> a in b
True
>>> c = 'beta'
>>> c in b
False
>>>
```

Example

```
#!/usr/bin/python

a = 10

b = 20

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


if ( a in list ):
  print "a is available in the given list"
else:
  print "a is not available in the given list"


if ( b not in list ):
  print "b is not available in the given list"
else:
  print "b is available in the given list"


a = 2
if ( a in list ):
  print "a is available in the given list"
else:
```

When executed gives below result.

```
a is not available in the given list
b is not available in the given list
a is available in the given list
```

5. Decision Making

Decision making is there in every programming language. Based on a test case the program decides to execute statements/commands or not. In python we use if, elif and else keywords to make decisions.

Syntax

```
if test case:
    code
```

If the test case is true then the code in if block gets executed.

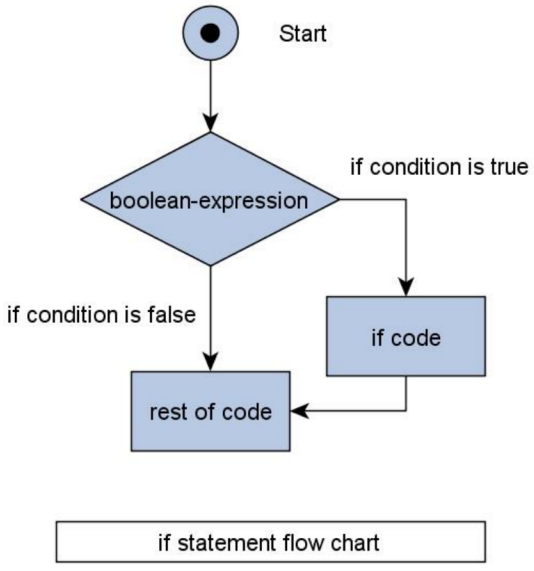
If the text expression is false, the statement(s) is not executed.As we have seen previously in indentation section to create a block of code underneath any statement we indent the code or we give spaces to it.

For example, the print statement comes under the if block.

```
if True:
   print "Indented three spaces"
```

In python 0 is interpreted as False and any non zero value is True.

Python if Statement Flowchart



if statement flow chart

```
#!/usr/bin/python

# If the number is positive, we print an appropriate message

num = 30
if num == 30:
    print "Number is equal to 30."
print "This is always printed. Rest of the code."



num = 82
```

```
if num < 100:
    print num, "is less than 100."
print "This is also always printed.Rest of the code."
```

Python if/else Statement

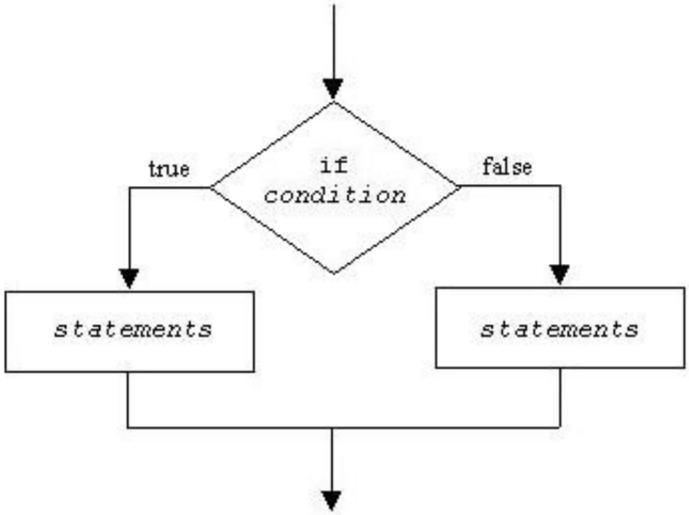
Syntax of if/else

```
if test case:
    Body of if
else:
    Body of else
```

The if/else statement evaluates test case and will execute body of if only when test condition is true.

If the condition is false body of else is executed.

Python if. Else Flowchart




```
#!/usr/bin/python
# Program checks
if the number is
positive or negative
# And displays an appropriate message

num = -4

if num >= 0:
    print "Positive or Zero"
else:
    print "Negative number"
```

Python if/elif/else

Syntax of if/elif/else

```
if test case:

    Body of if

elif test case:

    Body of elif

else:

    Body of else
```

If the condition is false else block gets executed but what if we want to evaluate few more condition and then decide the execution. We can use **elif** to put more condtions in out decision making process its short form of else if.

If the condition for **if** is **False**, it checks the condition of the next **elif** block and so on.

If all the conditions are **False**, body of **else** is executed.

The **if** block can have only one **`else`** block. But it can have multiple **`elif`** blocks.

Flowchart of if...elif...else

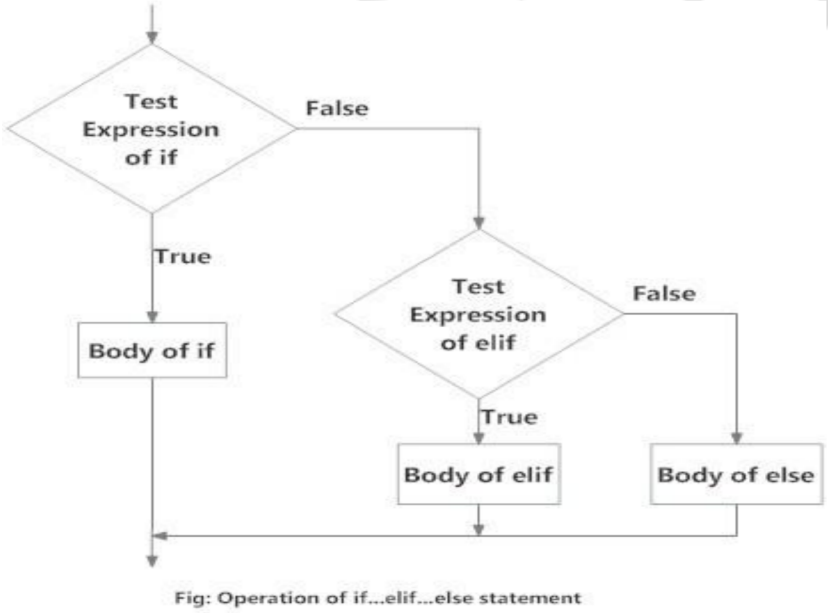


Fig: Operation of if...elif...else statement

```
#!/usr/bin/python

# In this program,

# we check if the number is greater than or

# less than 10
```

```
num = 84



if num > 10:
    print "Greater than 10"
elif num < 10:
    print "Less than 10"
else:
   print "Value out of range"
```

6. Loops

Loops are used when you have a block of code that you want to repeat it for fixed number of time or until a condition is not satisfied.

Python provides us with two loops

While loop: Repeat the block of code untill the while condition is true.

```
while test_case:
    Body of while
```

For loop: Repeat the block of code for a number of time.

```
for variable in sequence:
        Body of for
```

For loops are traditionally used when you have a block of code which you want to repeat a fixed number of times. The Python *for* statement iterates over the members of a sequence in order, executing the block each time. Contrast the *for* statement with the "while" loop, used when a condition needs to be checked each iteration, or to repeat a block of code forever. For example:

For loop from 0 to 2, therefore running 3 times.

```
for x in range(0, 3):
    print "We're on time %d" % (x)
```

While loop from 1 to infinity, therefore running forever.

```
x = 1
while True:
    print "To infinity and beyond! We're getting close, on %d now!" % (x)
    x += 1
```

While Loops

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.

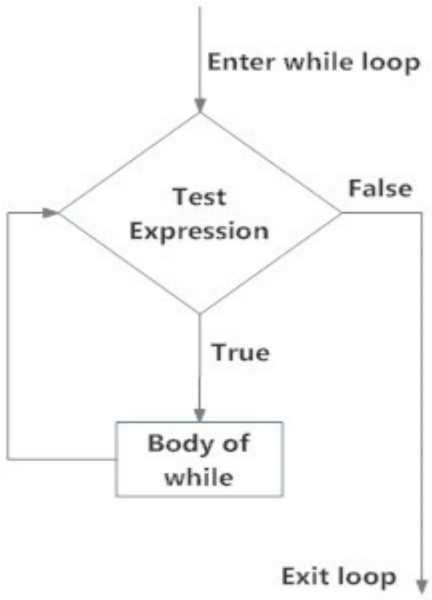


Fig: operation of while loop

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.