Introduction To Python Programming

Introduction

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

- 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 beginnerlevel programmers and supports the development of a wide range of applications from simple text processing to WWW browsers to games.

Features of Python

- Easy-to-learn: Python has few keywords, simple structure, and a clearly defined syntax. This allows a 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. Apart from the above-mentioned features, Python has a big list of good features. A few are listed below-
- 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.

First Python Program

Let us execute the 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 3.3.2 (default, Dec 10 2013, 11:35:01)

[GCC 4.6.3] on Linux

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

>>>

On Windows:

Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015, 22:43:06) [MSC v.1600
32 bit (Intel)] on win32

Type "copyright", "credits" or "license()" for more information.

>>>
```

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

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

If you are running the older version of Python (Python 2.x), use of parenthesis as inprint function is optional. 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.

Let us write a simple Python program in a script. Python files have the extension.py. Type the following source code in a test.py file

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

We assume that you have the Python interpreter set in PATH variable. Now, try to run this program as follows

On Linux

```
$ python test.py
```

This produces the following result

```
Hello, Python!
```

On Windows

```
C:\Python34>Python test.py
```

This produces the following result

```
Hello, Python!
```

Let us try another way to execute a Python script in Linux.

Here is the modified test.py

```
file- #!/usr/bin/python3

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

We assume that you have Python interpreter available in the /usr/bin directory.

Now, try to run this program as follows-

```
$ chmod +x test.py

        # This is to make file executable

$./test.py
```

This produces the following result

Python Character Set

A "character class", or a "character set", is a set of characters put in square brackets. The regex engine matches only one out of several characters in the character class or character set. We place the characters we want to match between square brackets. If you want to match any vowel, we use the character set [aeiou].

A character class or set matches only a single character. The order of the characters inside a character class or set does not matter. The results are identical.

We use a hyphen inside a character class to specify a range of characters. [0-9] matches a single digit between 0 and 9. Similarly for uppercase and lowercase letters we have the character class [A-Za-z].

Normally, a Python source file must be entirely made up of characters from the ASCII set (character codes between 0 and 127). However, you may choose to tell Python that in a certain source file you are using a character set that is a superset of ASCII. In this case, Python allows that specific source file to contain characters outside the ASCII set, but only in comments and string literals.

Token

Python breaks each logical line into a sequence of elementary lexical components known as tokens. Each token corresponds to a substring of the logical line. The normal token types are identifiers, keywords, operators, delimiters, and literals, as covered in the following sections. You may freely use whitespace between tokens to separate them. Some whitespace separation is necessary between logically adjacent identifiers or keywords; otherwise, Python would parse them as a single, longer identifier.

Example

printx is a single identifier; to write the keyword print followed by the identifier x, you need to insert some whitespace (e.g., print x).

Identifiers

An 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). Case is significant in Python: lowercase and uppercase letters are distinct. Python does not allow punctuation characters such as @, $, and % within identifiers.

Normal Python style is to start class names with an uppercase letter and all other identifiers with a lowercase letter. Starting an identifier with a single leading underscore indicates by convention that the identifier is meant to be 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. The identifier _ (a single underscore) is special in interactive interpreter sessions: the interpreter binds _ to the result of the last expression statement it has evaluated interactively, if any.

Example

MyVariable, var1, var2.

Keywords

Well simply, python keywords are the words that are reserved. That means you can't use them as name of any entities like variables, classes and functions.

So you might be thinking what are these keywords for. They are for defining the syntax and structures of Python language.

You should know there are 33 keywords in Python programming language as of writing this tutorial. Although the number can vary in course of time. Also keywords in Python is case sensitive. So they are to be written as it is. Here is a list of all keywords in python programming.

Example

```
>>>help()

Welcome to Python 3.6's help utility!

help> keywords

Here is a list of the Python keywords.  Enter any keyword to get more help.
```

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

```
help>
```

Literals

Python Literals can be defined as data that is given in a variable or constant.

Python supports the following literals:

1. String literals:

String literals can be formed by enclosing a text in the quotes. We can use both single as well as double quotes to create a string.

Example

"Aman" , '12345'

Types of Strings:

There are two types of Strings supported in Python:

a) Single-line String- Strings that are terminated within a single-line are known as Single line Strings.

Example

```
text1='hello'
```

b) Multi-line String - A piece of text that is written in multiple lines is known as multiple lines string.

There are two ways to create multiline strings:

1) Adding black slash at the end of each line.

Example

```
text1='hello\
user'
print(text1)
```

Output

```
hellouser
```

2) Using triple quotation marks:-

Example

```
str2="""welcome
to
SSSIT"""
print (str2)
```

Output

```
welcome
to
SSSIT
```

2. Numeric literals:

Numeric Literals are immutable. Numeric literals can belong to following four different numerical types.

Int(signed integers)	Long(long integers)	float(floating point)	Complex(complex)
Numbers(can be both positive and negative) with no fractional part.eg: 100	Integers of unlimited size followed by lowercase or uppercase L eg: 87032845L	Real numbers with both integer and fractional part eg: -26.2	In the form of a+bj where a forms the real part and b forms the imaginary part of the complex number. eg: 3.14j

Example

Numeric Literals

```
x = 0b10100 #Binary Literals

y = 100 #Decimal Literal

z = 0o215 #Octal Literal

u = 0x12d #Hexadecimal Literal
```

Float Literal

```
float_1 = 100.5

float_2 = 1.5e2
```

Complex Literal

```
a = 5+3.14j

print(x, y, z, u)

print(float_1, float_2)

print(a, a.imag, a.real)
```

Output

```
20 100 141 301

100.5 150.0

(5+3.14j) 3.14 5.0
```

3. Boolean literals:

A Boolean literal can have any of the two values: True or False.

Example

Boolean Literals

```
x = (1 == True)

y = (2 == False)

z = (3 == True)

a = True + 10

b = False + 10

print("x is", x)

print("y is", y)

print("z is", z)

print("a:", a)

print("b:", b)
```

Output

```
x is True

y is False

z is False

a: 11

b: 10
```

4. Special literals.

Python contains one special literal i.e., **None.**

None is used to specify to that field that is not created. It is also used for the end of lists in Python.

Example

Special Literals

```
val1=10

val2=None

print(val1)

print(val2)
```

Output

```
10

None
```

5. Literal Collections.

Python provides the four types of literal collection such as List literals, Tuple literals, Dict literals, and Set literals.

List:

List contains items of different data types. Lists are mutable i.e., modifiable.

The values stored in List are separated by comma(,) and enclosed within square brackets([]). We can store different types of data in a List.

Example

List literals

```
list=['John',678,20.4,'Peter']
list1=[456,'Andrew']

print(list)

print(list + list1)
```

Output

```
['John', 678, 20.4, 'Peter']

['John', 678, 20.4, 'Peter', 456, 'Andrew']
```

Dictionary:

Python dictionary stores the data in the key-value pair.

It is enclosed by curly-braces {} and each pair is separated by the commas(,).

Example

```
dict = {'name': 'Pater', 'Age':18,'Roll_nu':101}

print(dict)
```

Output

```
{'name': 'Pater', 'Age': 18, 'Roll_nu': 101}
```

Tuple:

Python tuple is a collection of different data-type. It is immutable which means it cannot be modified after creation.

It is enclosed by the parentheses () and each element is separated by the comma(,).

Example

```
tup = (10,20,"Dev",[2,3,4])

print(tup)
```

Output

```
(10, 20, 'Dev', [2, 3, 4])
```

Set:

Python set is the collection of the unordered dataset.

It is enclosed by the {} and each element is separated by the comma(,).

Example

```
set = {'apple','grapes','guava','papaya'}

print(set)
```

Output

```
{'guava', 'apple', 'papaya', 'grapes'}
```

Delemeter

A delimiter is a sequence of one or more characters used to specify the boundary between separate, independent regions in plain text or other data streams. An example of a delimiter is the comma character, which acts as a field delimiter in a sequence of comma-separated values.

The following tokens serve as delimiters:

(	)	[	]	{	}	@
,	:	.	`	=	;	
+=	-=	*=	/=	//=	%=	
&=	|=	^=	>>=	<<=	**=	

Operators

Operators are used to perform operations on variables and values.

Python divides the operators in the following groups:

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

Arithmetic Operators

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x – y

*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y
//	Floor division	x // y

Assignment Operators

Assignment operators and augmented assignment operators are used to assign values to variables:

An augmented assignment is generally used to replace a statement where an operator takes a variable as one of its arguments and then assigns the result back to the same variable. A simple example is x += 1 which is expanded to x = x + (1) . Similar constructions are often available for various binary operators.

Operator	Example	Same As
=	x = 5	x = 5
+=	x += 3	x = x + 3
-=	x -= 3	x = x - 3
*=	x *= 3	x = x * 3
/=	x /= 3	x = x / 3
%=	x %= 3	x = x % 3
//=	x //= 3	x = x // 3
**=	x **= 3	x = x ** 3
&=	x &= 3	x = x & 3

|=	x |= 3	x = x | 3
^=	x ^= 3	x = x ^ 3
>>=	x >>= 3	x = x >> 3
<<=	x <<= 3	x = x << 3

Comparison Operators

Comparison operators are used to compare two values:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Logical Operators

Logical operators are used to combine conditional statements:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is	x < 5 or x < 4

	true	
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Identity Operators

Identity operators are used to compare the objects, not if they are equal, but if they are actually the same object, with the same memory location:

Operator	Description	Example
is	Returns True if both variables are the same object	x is y
is not	Returns True if both variables are not the same object	x is not y

Bitwise Operators

Bitwise operators are used to compare (binary) numbers:

Operator	Name	Description
&	AND	Sets each bit to 1 if both bits are 1
|	OR	Sets each bit to 1 if one of two bits is 1
^	XOR	Sets each bit to 1 if only one of two bits is 1
~	NOT	Inverts all the bits
<<	Zero fill left shift	Shift left by pushing zeros in from the right and let the leftmost bits fall off
>>	Signed right shift	Shift right by pushing copies of the leftmost bit in from the left, and let the rightmost bits fall off

Comments

Single line Comments

Single-line comments are created simply by beginning a line with the hash (#) character, and they are automatically terminated by the end of line.

Example

```
#This is a comment in Python
```

Multiline Comments

Comments that span multiple lines – used to explain things in more detail – are created by adding a delimiter (“””) on each end of the comment.

Example

```
"""
This would be a multiline comment
in Python that spans several lines and
describes your code or anything you want it to be.
"""
```

Variables

Variables are containers for storing data values.

Unlike other programming languages, Python has no command for declaring a variable.

A variable is created the moment you first assign a value to it.

Variable Names

A variable can have a short name (like x and y) or a more descriptive name (age, carname, total_volume). Rules for Python variables:

- A variable name must start with a letter or the underscore character
- A variable name cannot start with a number
- A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and _)
- Variable names are case-sensitive (age, Age and AGE are three different variables)

```
#Legal variable names:
myvar = "Silicon"
my_var = "Silicon"
_my_var = "Silicon"
myVar = "Silicon"
MYVAR = "Silicon"
```

```
myvar2 = "Silicon"

#Illegal variable names:
2myvar = "Silicon"
my-var = "Silicon"
my var = "Silicon"
```

Concept of L-value and R-value

Lvalue and Rvalue refer to the left and right side of the assignment operator.

The **Lvalue** (pronounced: L value) concept refers to the requirement that the operand on the left side of the assignment operator is modifiable, usually a variable. **Rvalue** concept pulls or fetches the value of the expression or operand on the right side of the assignment operator. Some examples:

```
age = 39
```

The value 39 is pulled or fetched (Rvalue) and stored into the variable named age (Lvalue); destroying the value previously stored in that variable.

```
voting_age = 18

age = voting_age
```

If the expression has a variable or named constant on the right side of the assignment operator, it would pull or fetch the value stored in the variable or constant. The value 18 is pulled or fetched from the variable named voting_age and stored into the variable named age.

```
age < 17
```

If the expression is a test expression or Boolean expression, the concept is still an Rvalue one. The value in the identifier named age is pulled or fetched and used in the relational comparison of less than.

```
JACK_BENNYS_AGE = 39

JACK_BENNYS_AGE = 65;
```

This is illegal because the identifier JACK_BENNYS_AGE does not have Lvalue properties. It is not a modifiable data object, because it is a constant.

Some uses of the Lvalue and Rvalue can be confusing in languages that support increment and decrement operators. Consider:

```
oldest = 55

age = oldest++
```

Postfix increment says to use my existing value then when you are done with the other operators; increment me. Thus, the first use of the oldest variable is an Rvalue context where the existing value of 55 is pulled or fetched and then assigned to the variable age; an Lvalue context. The second use of the oldest variable is an Lvalue context wherein the value of the oldest is incremented from 55 to 56.

Data types in Python

Data types are the classification or categorization of data items. Data types represent a kind of value which determines what operations can be performed on that data. Numeric, non-numeric and Boolean (true/false) data are the most used data types. However, each programming language has its own classification largely reflecting its programming philosophy.

Python has the following data types built-in by default, in these categories:

Text Type:	str
Numeric Types:	int, float, complex
Sequence Types:	list, tuple, range
Mapping Type:	dict
Set Types:	set, frozenset
Boolean Type:	bool
Binary Types:	bytes, bytearray, memoryview

Numeric

A numeric value is any representation of data which has a numeric value. Python identifies three types of numbers:

- **Integer:** Positive or negative whole numbers (without a fractional part)
- **Float:** Any real number with a floating point representation in which a fractional component is denoted by a decimal symbol or scientific notation
- **Complex number:** A number with a real and imaginary component represented as x+yj. x and y are floats and j is -1(square root of -1 called an imaginary number)

Boolean

Data with one of two built-in values True or False. Notice that 'T' and 'F' are capital. true and false are not valid booleans and Python will throw an error for them.

Sequence Type

A sequence is an ordered collection of similar or different data types. Python has the following built-in sequence data types:

- **String**: A string value is a collection of one or more characters put in single, double or triple quotes.
- **List** : A list object is an ordered collection of one or more data items, not necessarily of the same type, put in square brackets.
- **Tuple**: A Tuple object is an ordered collection of one or more data items, not necessarily of the same type, put in parentheses.

Dictionary

A dictionary object is an unordered collection of data in a key:value pair form. A collection of such pairs is enclosed in curly brackets. For example: {1:"Steve", 2:"Bill", 3:"Ram", 4: "Farha"}

type() function

Python has an in-built function **type()** to ascertain the data type of a certain value. For example, enter type(1234) in Python shell and it will return <class 'int'>, which means 1234 is an integer value. Try and verify the data type of different values in Python shell, as shown below.

```
>>> type(1234)
<class 'int'>
>>> type(55.50)
<class 'float'>
>>> type(6+4j)
<class 'complex'>
>>> type("hello")
<class 'str'>
>>> type([1,2,3,4])
<class 'list'>
>>> type((1,2,3,4))
<class 'tuple'>
>>> type({1:"one", 2:"two", 3:"three"})
<class 'dict'>
```

Mutable and Immutable Objects

Data objects of the above types are stored in a computer's memory for processing. Some of these values can be modified during processing, but contents of others can't be altered once they are created in the memory.

Number values, strings, and tuple are immutable, which means their contents can't be altered after creation.

On the other hand, collection of items in a List or Dictionary object can be modified. It is possible to add, delete, insert, and rearrange items in a list or dictionary. Hence, they are mutable objects.

Accepting Input from console

What is Console in Python? Console (also called Shell) is basically a command line interpreter that takes input from the user i.e one command at a time and interprets it. If it is error free then it runs the command and gives required output otherwise shows the error message.

Here we write command and to execute the command just press enter key and your command will be interpreted.
For coding in Python you must know the basics of the console used in Python.

The primary prompt of the python console is the three greater than symbols

```
>>>
```

You are free to write the next command on the shell only when after executing the first command these prompts have appeared. The Python Console accepts command in Python which you write after the prompt.

Accepting Input from Console

User enters the values in the Console and that value is then used in the program as it was required.

To take input from the user we make use of a built-in function *input()*.

```
# input
input1 = input()

# output
print(input1)
```

We can also type cast this input to integer, float or string by specifying the input() function inside the type.

Typecasting the input to Integer:

There might be conditions when you might require integer input from user/Console, the following code takes two input(integer/float) from console and typecasts them to integer then prints the sum.

```
# input
num1 = int(input())
num2 = int(input())

# printing the sum in integer
print(num1 + num2)
```

Typecasting the input to Float:

To convert the input to float the following code will work out.

```
# input
num1 = float(input())
num2 = float(input())

# printing the sum in float
print(num1 + num2)
```

Typecasting the input to String:

All kind of input can be converted to string type whether they are float or integer. We make use of keyword str for typecasting.

```
# input
string = str(input())

# output
print(string)
```

Assignment statements

An assignment statement evaluates the expression list (remember that this can be a single expression or a comma-separated list, the latter yielding a tuple) and assigns the single resulting object to each of the target lists, from left to right.

The assignment statement: name = expression. The purpose of Python's assignment statement is to associate names with values in your program. It is the only statement that does not start with a keyword. An assignment statement is a line containing at least one single equal sign (=) that is not inside parentheses.

Example

```
x = 5
```

Expressions

Expressions are representations of value. They are different from statement in the fact that statements do something while expressions are representation of value. For example any string is also an expressions since it represents the value of the string as well.

Python has some advanced constructs through which you can represent values and hence these constructs are also called expressions.

How to create an expressions

Python expressions only contain identifiers, literals, and operators. So, what are these?

Identifiers: Any name that is used to define a class, function, variable module, or object is an identifier. **Literals**: These are language-independent terms in Python and should exist independently in any programming language. In Python, there are the string literals, byte literals, integer literals, floating point literals, and imaginary literals. **Operators**: In Python you can implement the following operations using the corresponding tokens.

Operator	Token
add	+
subtract	-
multiply	*
power	**
Integer Division	/

Operator	Token
remainder	%
decorator	@
Binary left shift	<<
Binary right shift	>>
and	&
or	\
Binary Xor	^
Binary ones complement	~
Less than	<
Greater than	>
Less than or equal to	<=
Greater than or equal to	>=
Check equality	==
Check not equal	!=

Precedence of operators in Python

There can be more than one operator in an expression. To evaluate these types of expressions there is a rule of precedence in Python. It guides the order in which these operations are carried out. For example, multiplication has higher precedence than subtraction.

The following table lists all operators from highest precedence to lowest.

Operator	Description
**	Exponentiation (raise to the power)
~ + -	Complement, unary plus and minus (method names for the last two are +@ and -@)
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>><<	Right and left bitwise shift
&	Bitwise 'AND'td>
^ |	Bitwise exclusive `OR' and regular `OR'
<= <>>=	Comparison operators
<> == !=	Equality operators
= %= /= //= -= += *= **=	Assignment operators
is is not	Identity operators
in not in	Membership operators
not or and	Logical operators

Operator precedence affects how an expression is evaluated.

Example

x = 7 + 3 * 2; here, x is assigned 13, not 20 because operator * has higher precedence than +, so it first multiplies 3*2 and then adds into 7.

Here, operators with the highest precedence appear at the top of the table, those with the lowest appear at the bottom.

Example

```
a =20
b =10
c =15
d =5
e =0

e =(a + b)* c / d       #( 30 * 15 ) / 5
print("Value of (a + b) * c / d is "),  e

e =((a + b)* c)/ d     # (30 * 15 ) / 5
print("Value of ((a + b) * c) / d is "),  e

e =(a + b)*(c / d);# (30) * (15/5)
print("Value of (a + b) * (c / d) is "),  e

e = a +(b * c)/d;#  20 + (150/5)
print("Value of a + (b * c) / d is "),  e
```

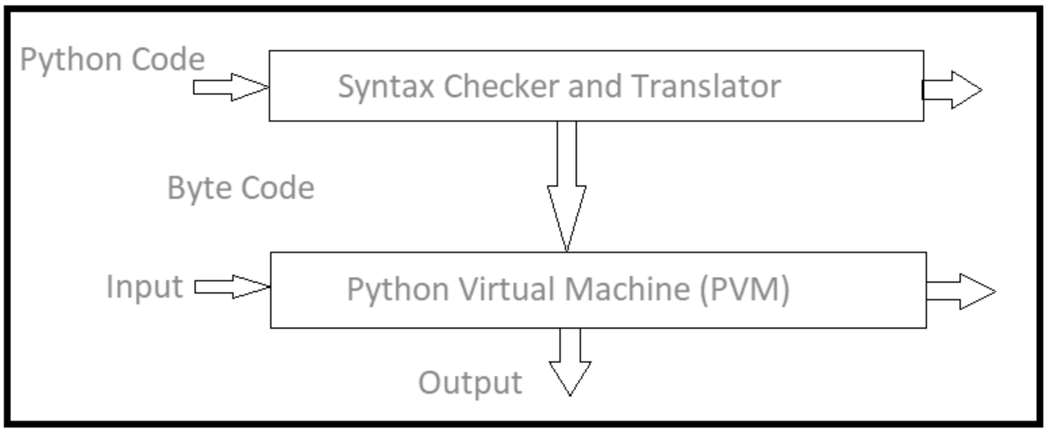
When you execute the above program, it produces the following result −

```
Value of (a + b) * c / d is
Value of ((a + b) * c) / d is
Value of (a + b) * (c / d) is
Value of a + (b * c) / d is
```

Execution of a Program

Internal working of Python

Python is an object oriented programming language like Java. Python is called an interpreted language. Python uses code modules that are interchangeable instead of a single long list of instructions that was standard for functional programming languages. The standard implementation of python is called “cpython”. It is the default and widely used implementation of the Python. Python doesn’t convert its code into machine code, something that hardware can understand. It actually converts it into something called byte code. So within python, compilation happens, but it’s just not into a machine language. It is into byte code and this byte code can’t be understood by CPU. So we need actually an interpreter called the python virtual machine. The python virtual machine executes the byte codes.



The Python interpreter performs following tasks to execute a Python program :

- **Step 1 :** The interpreter reads a python code or instruction. Then it verifies that the instruction is well formatted, i.e. it checks the syntax of each line.If it encounters any error, it immediately halts the translation and shows an error message.
- **Step 2 :** If there is no error, i.e. if the python instruction or code is well formatted then the interpreter translates it into its equivalent form in intermediate language called “Byte code”.Thus, after successful execution of Python script or code, it is completely translated into Byte code.
- **Step 3 :** Byte code is sent to the Python Virtual Machine(PVM).Here again the byte code is executed on PVM.If an error occurs during this execution then the execution is halted with an error message.

To run a python script, the following ways can be used.

- The operating system command-line or terminal
- The Python interactive mode
- The IDE or text editor you like best
- The file manager of your system, by double-clicking on the icon of your script

Types of error in python

Syntax error

Python will find these kinds of errors when it tries to parse your program, and exit with an error message without running anything. Syntax errors are like spelling or grammar mistakes in a language like English.

Syntax errors are the most basic type of error. They arise when the Python parser is unable to understand a line of code. Syntax errors are almost always fatal, i.e. there is almost never a way to successfully execute a piece of code containing syntax errors.

Example

```
>>>print "hello"
SyntaxError: Missing parentheses in call to 'print'. Did you mean print("hello")?
```

Run-Time error

If a program is free of syntax errors, it will be run by the Python interpreter. However, the program may exit if it encounters a runtime error – a problem that went undetected when the program was parsed, but is only revealed when the code is executed.

Some examples of Python Runtime errors –

- division by zero
- performing an operation on incompatible types
- using an identifier which has not been defined
- accessing a list element, dictionary value or object attribute which doesn't exist
- trying to access a file which doesn't exist

Logical error

Logical errors are the most difficult to fix. They occur when the program runs without crashing, but produces an incorrect result. The error is caused by a mistake in the program's logic. You won't get an error message, because no syntax or runtime error has occurred. You will have to find the problem on your own by reviewing all the relevant parts of your code – although some tools can flag suspicious code which looks like it could cause unexpected behaviour.

Sometimes there can be absolutely nothing wrong with your Python implementation of an algorithm – the algorithm itself can be incorrect. However, more frequently these kinds of errors are caused by programmer carelessness. Here are some examples of mistakes which lead to logical errors:

- using the wrong variable name
- indenting a block to the wrong level
- using integer division instead of floating-point division
- getting operator precedence wrong
- making a mistake in a Boolean expression
- off-by-one, and other numerical errors

If you misspell an identifier name, you may get a runtime error or a logical error, depending on whether the misspelled name is defined.

A common source of variable name mix-ups and incorrect indentation is frequent copying and pasting of large blocks of code. If you have many duplicate lines with minor differences, it's very easy to miss a necessary change when you are editing your pasted lines. You should always try to factor out excessive duplication using functions and loops – we will look at this in more detail later.

Conditional Statement in Python

If condition

This is the simplest example of a conditional statement. The syntax is:

```
if(condition):
        indented Statement Block
```

The block of lines indented the same amount after the colon (:) will be executed whenever the condition is TRUE.
Python supports the usual logical conditions from mathematics:

- Equals: a == b
- Not Equals: a != b
- Less than: a < b
- Less than or equal to: a <= b
- Greater than: a > b
- Greater than or equal to: a >= b

These conditions can be used in several ways, most commonly in "if statements" and loops.

An "if statement" is written by using the if keyword.

Example

```
a = 33
b = 200
if b > a:
  print("b is greater than a")
```

Output

```
b is greater than a
```

if-else condition

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

```
a = 200
b = 33
if b > a:
        print("b is greater than a")
else:
        print("b is not greater than a")
```

Output

```
b is not greater than a
```

if-elif-else condition

The elif keyword is pythons way of saying "if the previous conditions were not true, then try this condition". We can also put else after elif condition.

Example

```
a = 33
b = 33
if b > a:
        print("b is greater than a")
elif a == b:
        print("a and b are equal")
else:
        print("a is greater than b")
```

Output

```
a and b are equal
```

Example Programs

Program to find absolute value of a number.

```
value=-50
if  value< 0:
        absl= -value;
else:
        absl = value;
print("Absolute value=",absl)
```

Output

```
Absolute value=50
```

Program to sort three numbers inputted from keyboard.

```
first = int(input("Enter the first number: "))
second = int(input("Enter the second number: "))
third = int(input("Enter the third number: "))

if first > second and first > third:
        if second > third:
                print("After sorting numbers are : ",first,second,third)
        else:
                print("After sorting numbers are : ",first,third,second)
if second > first and second > third:
        if first > third:
                print("After sorting numbers are : ",second,first,third)
        else:
                print("After sorting numbers are : ",second,third,first)
if third > second and third > first:
        if second > first:
                print("After sorting numbers are : ",third,second,first)
        else:
                print("After sorting numbers are : ",third,first,second)
```

Output

```
Enter the first number: 34

Enter the second number: 21

Enter the third number: 54

After sorting numbers are: 54,34,21
```

Program to find divisibility of a number

```
a = int(input("Enter the first number: "))
b = int(input("Enter the second number: "))
if (a % b) == 0:
        print("The first number is divisible by the second number.")
else:
        print("The first number is not divisible by the second number.")
```

Output

```
Enter the first number: 68

Enter the second number: 17

The first number is divisible by the second number.
```

Iterative computation and control flow

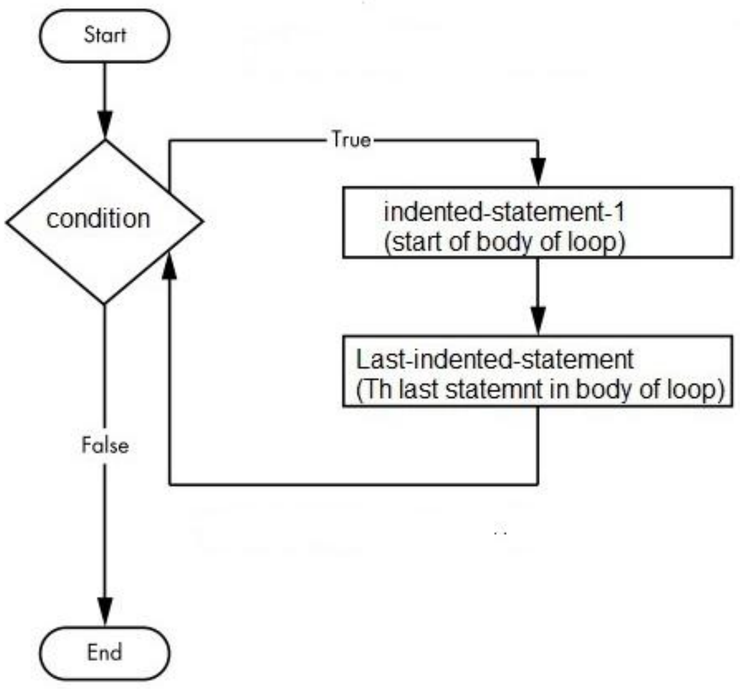
for-in loop

Syntax:

```
for iterator_var in sequence:

        statement(s)
```

Flowchart



Example

Print the natural numbers within n.

```
n=8
for i in range(n):
        print(i+1)
```

or we can write

```
n = 8
for i in range(1, n+1):
        print(i)
```

Output

```
1
2
3
4
5
6
7
8
```

Here i value starts from 0.

Example

```
colors = ["red", "green", "blue", "purple"]

i = 0

for i in range(len(colors)):

        print(colors[i])

        i += 1
```

Output

```
red

green

blue

purple
```

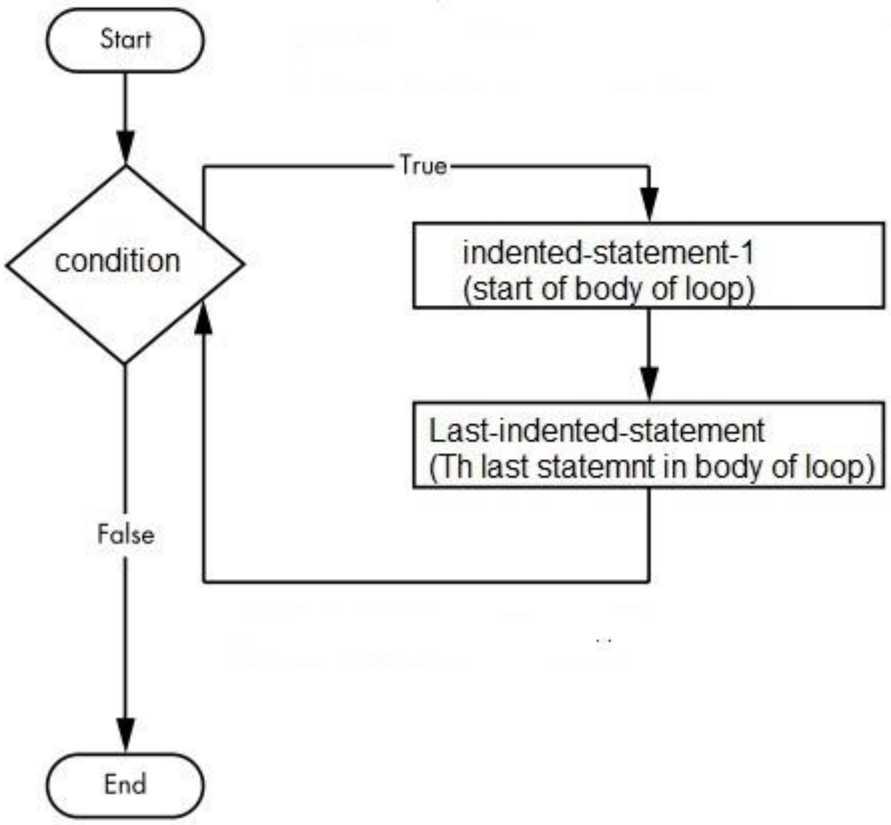
While Loop

Syntax:

```
while expression:

        statement(s)
```

Flowchart



Example

Print the sum of n natural numbers.

```
n=int(input("Enter value of n: "))
i=1
sum=0
while(i<=n):
   sum=sum+i
   i=i+1
print("Sum of n natural numbers is :",sum)
```

Output

```
Enter value of n: 5
Sum of n natural numbers is : 15
```

Example

```
colors = ["red", "green", "blue", "purple"]
i = 0
while i<len(colors):
        print(colors[i])
        i += 1
```

Output

```
red

green

blue

purple
```

Simple Programs

Python program to find compound interest.

```
p = float(input("Enter principal amount : "))
R = float(input("Enter annual rate of interest : "))
t = int(input("Enter time in years : "))
n = int(input("Enter number of compounding periods per year : "))
# compound_Interest=P * pow((1 + R/(100 * n)), n*t)
x=1+(R/(100*n))
Res=1
for i in range(1,n*t):
   Res=Res*x
print("Compound interest =",(p*Res)-p)
print("Total amount with interest =",(p*Res))
```

Output

```
Enter principal amount : 10000
Enter annual rate of interest : 8
Enter time in years : 5
Enter number of compounding periods per year : 4
Compound interest = 4568.111725277988
Total amount with interest = 14568.111725277988
```

Python program to find factorial of a positive number.

```
num = int(input("Enter a number: "))
factorial = 1

# check if the number is negative, positive or zero
if num < 0:
        print("Sorry, factorial does not exist for negative numbers")
elifnum == 0:
        print("The factorial of 0 is 1")
else:
        for i in range(1,num + 1):
                factorial = factorial*i
print("The factorial is",factorial)
```

Output

```
Enter a number: 5

The factorial is 120
```