

Python

Overview of Python

Python is a general-purpose interpreted, interactive, object-oriented, and high-level programming language. It was created by Guido van Rossum during 1985- 1990. Python is a programming language. Python can be used on server to create web applications.

- **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.**

The Python interpreter is easily extended with new functions and data types implemented in C or C++ (or other languages callable from C). Python is also suitable as an extension language for customizable applications.

Python Features

Python's features include –

- **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.

Who uses python today...

- Python is being applied in real revenue-generating products by real companies. For instance:
- Google makes extensive use of Python in its web search system, and employs Python's creator.
- Intel, Cisco, Hewlett-Packard, Seagate, Qualcomm, and IBM use Python for hardware testing.
- ESRI uses Python as an end-user customization tool for its popular GIS mapping products.
- The YouTube video sharing service is largely written in Python

Why do people use Python...?

The following primary factors cited by Python users seem to be these:

- Python is object-oriented

Structure supports such concepts as polymorphism, operation overloading, and multiple inheritance.

.

- It's free (open source)

Downloading and installing Python is free and easy Source code is easily accessible

- It's powerful

- Dynamic typing
- Built-in types and tools
- Library utilities
- Third party utilities (e.g. Numeric, NumPy, SciPy) - Automatic memory management

- It's portable

- Python runs virtually every major platform used today
- As long as you have a compatible Python interpreter installed, Python programs will run in exactly the same manner, irrespective of platform.

Apart from the above-mentioned features, Python has a big list of good features, 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.

Python is available on Linux and Mac OSX

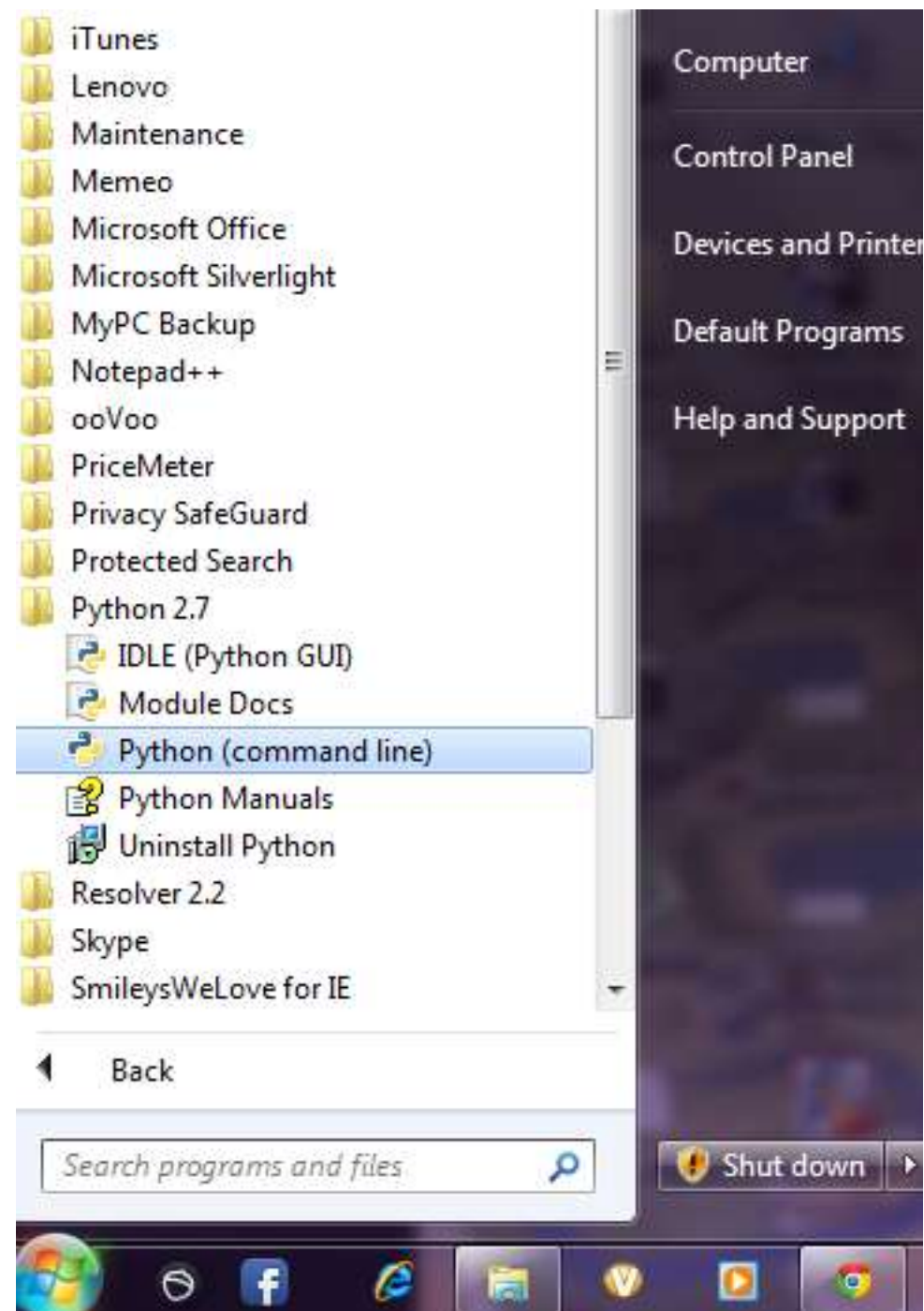
Open a terminal window and type "python" to find out if it is already installed and which version is installed.

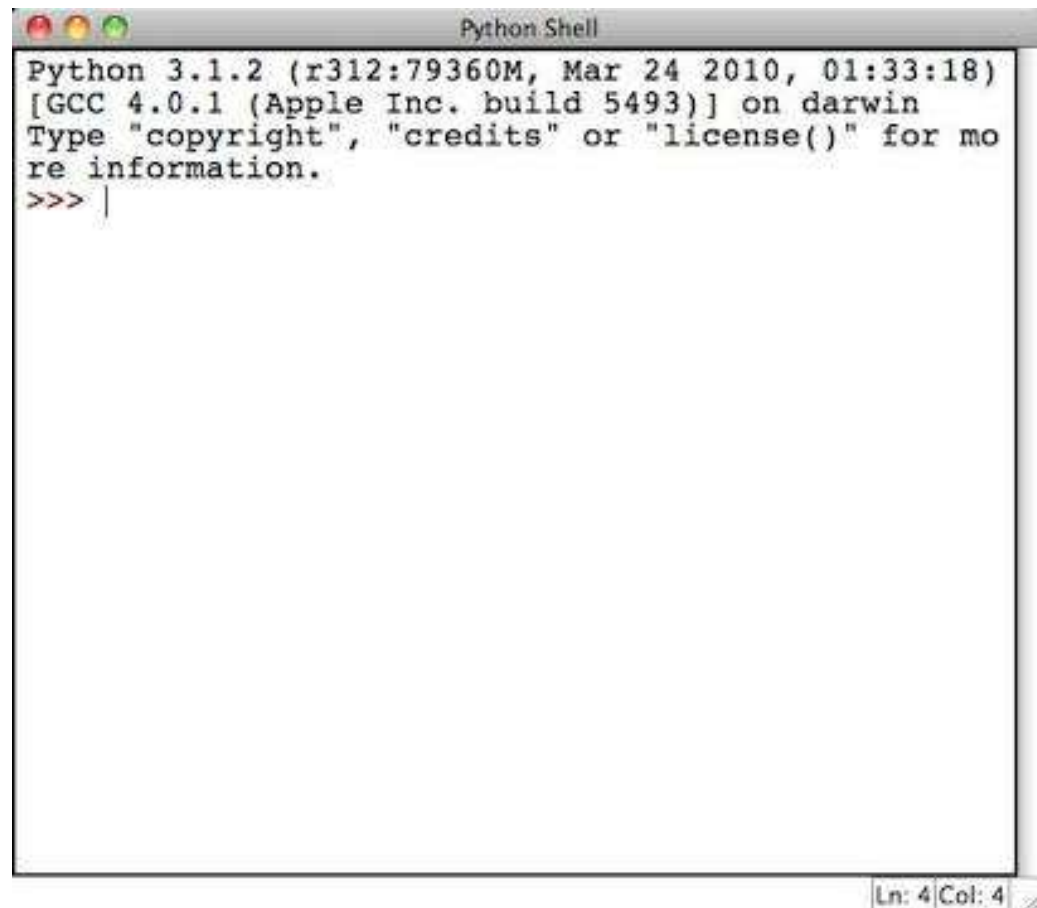
- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion
- Python has also been ported to the Java and .NET virtual machines

Installing Python

- Python is pre-installed on most Unix systems, including Linux and MAC OS X
- But for in Windows Operating Systems , user can download from the <https://www.python.org/downloads/> - from the above link download latest version of python IDE and install, recent version is 3.4.1 but most of them uses version 2.7.7 only

After installing the Python Ver#3.6 go to start menu then click on python 3.6 in that one you can select python (command line) it is prompt with >>>





```
Python 3.1.2 (r312:79360M, Mar 24 2010, 01:33:18)
[GCC 4.0.1 (Apple Inc. build 5493)] on darwin
Type "copyright", "credits" or "license()" for more
>>> |
```

Ln: 4 Col: 4

Running Python

Once you're inside the Python interpreter, type in commands at will.

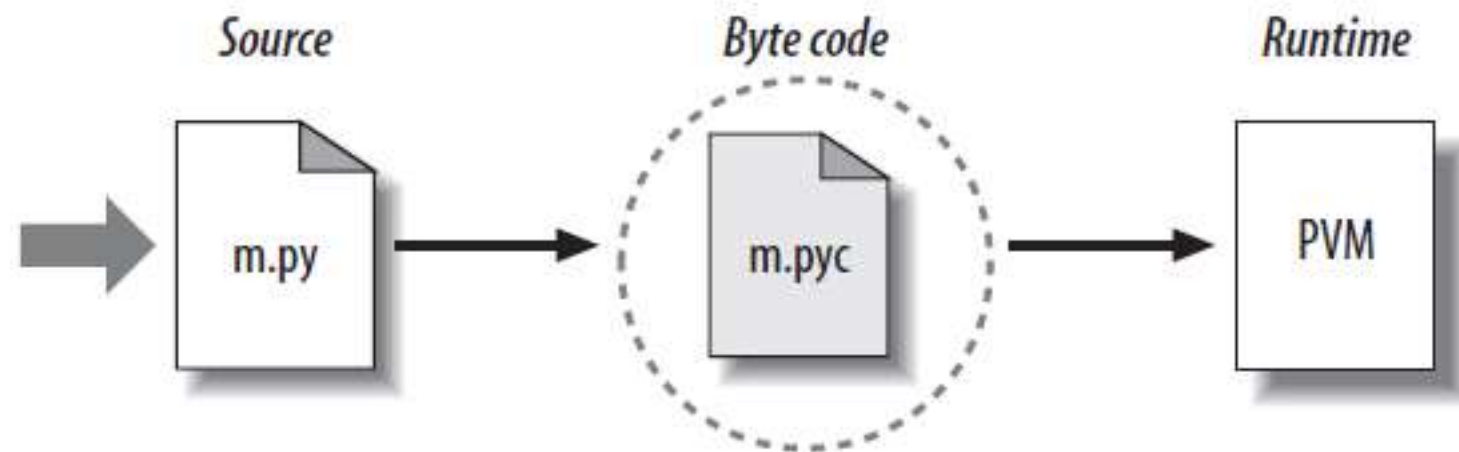
- **Examples:**

```
>>> print ('Hello world')
```

```
Hello world
```

Python Code Execution

- Python's traditional runtime execution model: source code you type is translated to byte code, which is then run by the Python Virtual Machine. Your code is automatically compiled, but then it is interpreted.



Source code extension is .py

Byte code extension is .pyc (compiled python code)

Installing Python

Unix and Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to <https://www.python.org/downloads/>.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.

Windows Installation

Here are the steps to install Python on Windows machine.

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

Setting up PATH

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.

The **path** variable is named as PATH in Unix or Path in Windows (Unix is case sensitive; Windows is not).

Setting path at Windows

To add the Python directory to the path for a particular session in Windows

—

At the command prompt — type `path %path%;C:\Python` and press Enter.

Note — C:\Python is the path of the Python directory

Running Python

There are three different ways to start Python –

Interactive Interpreter

```
$python # Unix/Linux
```

or

```
python% # Unix/Linux
```

or

```
C:> python # Windows/DOS
```

Script from the Command-line

A Python script can be executed at command line by invoking the interpreter on your application

```
$python script.py # Unix/Linux
```

or

```
python% script.py # Unix/Linux
```

or

```
C: >python script.py # Windows/DOS
```

Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- Unix – IDLE is the very first Unix IDE for Python.
- Windows – PythonWin is the first Windows interface for Python and is an IDE with a GUI.
- Macintosh – The Macintosh version of Python along with the IDLE IDE is available from the main website

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 3.6.4 (v3.6.4:d48ecebad5, Dec 18 2017, 21:07:28)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>>
```

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

```
>>> print ("hello world")
hello world
```

Script Mode Programming

Type the following source code in a test.py file –

```
print "Hello, Python!"
```

We assume that you have Python interpreter set in PATH variable.

```
$ python test.py
Hello, Python!
```

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.

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.

Reserved Words

The following list shows the Python keywords. These are reserved words and you cannot use them as constant or variable or any other identifier names.

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return
del	import	try
elif	in	while
else	is	with
except	lambda	yield

Lines and Indentation

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

```
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"
```

Multi-Line Statements

```
days = ['Monday', 'Tuesday', 'Wednesday',
        'Thursday', 'Friday']
```

Quotation in Python

Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.

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

Comments in Python

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

Using Blank Lines

A line containing only whitespace, possibly with a comment, is known as a blank line and Python totally ignores it.

Waiting for the User

The following line of the program displays the prompt, the statement saying “Press the enter key to exit”, and waits for the user to take action –

```
#!/usr/bin/python
```

```
raw_input("\n\nPress the enter key to exit.")
```

or

```
input("\n\nPress the enter key to exit.")
```

Multiple Statements on a Single Line

The semicolon (;) allows multiple statements on the single line given that neither statement starts a new code block.

```
import sys; x = 'foo'; sys.stdout.write(x + '\n')
```

Multiple Statement Groups as Suites

A group of individual statements, which make a single code block are called suites in Python. Compound or complex statements, such as if, while, def, and class require a header line and a suite.

```
if expression :  
    suite  
elif expression :  
    suite  
else :  
    suite
```

Command Line Arguments

```
$ python -h
usage: python [option] ... [-c cmd | -m mod | file | -] [arg] ...
Options and arguments (and corresponding environment variables):
-c cmd : program passed in as string (terminates option list)
-d      : debug output from parser (also PYTHONDEBUG=x)
-E      : ignore environment variables (such as PYTHONPATH)
-h      : print this help message and exit
```

Variables

Variables are nothing but reserved memory locations to store values.

Creating Variables

```
x = 10
y = "python"
print(x)
print(y)
```

Output:
10
python

```
x = 2 # x is of type int
x = "hello" # x is now of type str
print(x)
```

Output:
hello

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)

Output Variables

The Python `print` statement is often used to output variables.

To combine both text and a variable, Python uses the `+` character:

Ex:1 `x = "awesome"`
`print("Python is " + x)`
or

Output:
Python is awesome

Ex:2 `x = "Python is "`
`y = "awesome"`
`z = x + y`
`print(z)`

Output:
15

EX:3
`x = 5`
`y = 10`
`print(x + y)`

Ex:4
`x = 5`
`y = "John"`
`print(x + y)`

Output:
TypeError: unsupported operand type(s) for +: 'int' and 'str'

Multiple Assignment

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

`a=b=c=1` #a,b,c has 1 value

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

`a=1`

`b=2`

`c="john"`

Standard Data Types

Python has five standard data types –

- Numbers
- String
- List
- Tuple
- Dictionary

Python Numbers

There are three numeric types in Python:

- int
- float
- complex

Ex:

```
x=1 #int
```

```
y=2.8 #float
```

```
z=1j #complex
```

```
print(type(x))
```

```
print(type(y))
```

```
print(type(z))
```

Output:

```
<class 'int'>
```

```
<class 'float'>
```

```
<class 'complex'>
```

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
80	0xDEFA BCECBDAECBFBAEI	32.3+e18	.876j
-0x260	-052318172735L	-3.25E+101	3e+26J

Python Strings

```
str='Hello World!'
print (str)
print (str[0])
print (str[2:5])
print (str[2:])
print (str * 2)
print (str + "Test")
```

```
#print complete string
#print first character of the string
#print character starting from 3rd to 5th
#print string starting from 3rd character
#print string two times
#print concatenated string
```

Output:

```
Hello World!
H
llo
llo World!
Hello World!Hello World!
Hello World!TEST
```

Python Lists

```
list1 = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
print (list1)           # Prints complete list
print (list1[0])        # Prints first element of the list
print (list1[1:3])      # Prints elements starting from 2nd till 3rd
print (list1[2:])       # Prints elements starting from 3rd element
print (tinylist * 2)    # Prints list two times
print (list + tinylist) # Prints concatenated lists
```

Output:

```
['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']
```

Python Tuples

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.

```
tuple1= ( `abcd` , 546 , 5.43, `sugar`, 23.5)
```

```
tuple2 = (123 , `sugar`)
```

```
print tuple1
```

```
print tuple1[0]
```

```
print tuple1[1:3]
```

```
print tuple2 * 2
```

```
print tuple1 + tuple2
```

```
#print complete list
```

```
#print first element of the tuple
```

```
#print elements starting from 2nd till 3rd
```

```
#print list two times
```

```
#prints concatenated lists
```

Output:

```
(`abcd` , 546 , 5.43, `sugar`, 23.5)
```

```
abcd
```

```
(546 , 5.43)
```

```
(5.43, `sugar` , 23.5)
```

```
(123, `sugar` , 123, `sugar`)
```

```
(`abcd`, 546, 5.43, `sugar`, 23.5, 123, `sugar`)
```

Python Dictionary

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.

```
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
```

Output:

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.

eval() Parameters

Syntax: `eval(expression, globals = None, Locals = None)`

`expression` : This string as parsed and evaluated as a Python expression

`globals` : a dictionary

`locals` : a mapping object

```
x=1
```

```
print (eval('x+1'))
```

```
print(x+1)
```

Python 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

Python Arithmetic Operators

Operator	Name	Example
<code>+</code>	Addition	<code>x + y</code>
<code>-</code>	Subtraction	<code>x - y</code>
<code>*</code>	Multiplication	<code>x * y</code>
<code>/</code>	Division	<code>x / y</code>
<code>%</code>	Modulus	<code>x % y</code>
<code>**</code>	Exponentiation	<code>x ** y</code>
<code>//</code>	Floor division	<code>x // y</code>

Python Assignment Operators

Assignment operators are used to assign values to variables:

Operator	Example	Same As
<code>=</code>	<code>x = 5</code>	<code>x = 5</code>
<code>+=</code>	<code>x += 3</code>	<code>x = x + 3</code>
<code>-=</code>	<code>x -= 3</code>	<code>x = x - 3</code>
<code>*=</code>	<code>x *= 3</code>	<code>x = x * 3</code>
<code>/=</code>	<code>x /= 3</code>	<code>x = x / 3</code>
<code>%=</code>	<code>x %= 3</code>	<code>x = x % 3</code>
<code>//=</code>	<code>x //= 3</code>	<code>x = x // 3</code>
<code>**=</code>	<code>x **= 3</code>	<code>x = x ** 3</code>

Python 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

Python Logical Operators

Logical operators are used to combine conditional statements:

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

Python 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

Python Membership Operators

Python’s membership operators test for membership in a sequence, such as strings, lists, or tuples.

in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects.

is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here is results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here is not results in 1 if id(x) is not equal to id(y).

Python - Decision Making

1	<u>if statements</u> An if statement consists of a boolean expression followed by one or more statements.
2	<u>if...else statements</u> An if statement can be followed by an optional else statement, which executes when the boolean expression is FALSE.
3	<u>nested if statements</u> You can use one if or else if statement inside another if or else if statement(s).

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

#If statement without indentation will raise an error

```
a = 33
b = 200
if b > a:
    print("b is greater than a") # you will get an error
```

Elif

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

Else

```
a = 200
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")
```

One line if statement:

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

One line if else statement:

```
print("A") if a > b else print("B")
```

One line if else statement, with 3 conditions:

```
print("A") if a > b else print("=") if a == b else print("B")
```

And

```
if a > b and c > a:  
    print("Both conditions are True")
```

Or

```
if a > b or a > c:  
    print("At least one of the conditions are True")
```

Python Loops

Python has two primitive loop commands:

- **while** loops
- **for** loops

The while Loop

```
i = 1  
while i < 6:  
    print(i)  
    i += 1
```


The break Statement

```
i = 1
while i < 6:
    print(i)
    if i == 3:
        break
    i += 1
```

The continue Statement

```
i = 0
while i < 6:
    i += 1
    if i == 3:
        continue
    print(i)
```

#Note number 3 is missing in the output

Python For Loops

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

Looping Through a String

```
for x in "banana":
    print(x)
```

#Loop through the letters in the word "banana":

The break Statement

Exit the loop when **x** is "banana":

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    print(x)
    if x == "banana":
        break
```

Example

Exit the loop when **x** is "banana", but this time the break comes before the print:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        break
    print(x)
```

The continue Statement

Do not print banana:

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
    if x == "banana":
        continue
    print(x)
```

The range() Function

The `range()` function returns a sequence of numbers, starting from 0 by default, and increments by 1 (by default), and ends at a specified number.

```
for x in range(6):  
    print(x)  
for x in range(2, 6):  
    print(x)  
for x in range(2, 30, 3):  
    print(x)
```

Else in For Loop

The `else` keyword in a `for` loop specifies a block of code to be executed when the loop is finished:

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):  
    print(x)  
else:  
    print("Finally finished!")
```

Nested Loops

```
adj = ["red", "big", "tasty"]  
fruits = ["apple", "banana", "cherry"]
```

```
for x in adj:  
    for y in fruits:  
        print(x, y)
```

Number Type Conversion

- Type `int(x)` to convert `x` to a plain integer.
- Type `long(x)` to convert `x` to a long integer.
- Type `float(x)` to convert `x` to a floating-point number.
- Type `complex(x)` to convert `x` to a complex number with real part `x` and imaginary part zero.
- Type `complex(x, y)` to convert `x` and `y` to a complex number with real part `x` and imaginary part `y`. `x` and `y` are numeric expressions

Recursion

Python also accepts function recursion, which means a defined function can call itself.

```
def tri_recursion(k):  
    if(k>0):  
        result = k+tri_recursion(k-1)  
        print(result)  
    else:  
        result = 0  
    return result  
  
print("\n\nRecursion Example Results")  
tri_recursion(6)
```

Accessing Values in Strings

```
var1 = 'Hello World!'
var2 = "Python Programming"
```

```
print "var1[0]: ", var1[0]
print "var2[1:5]: ", var2[1:5]
```

Output:

```
var1[0]: H
var2[1:5]: ytho
```

Updating Strings

```
var1 = 'Hello World!'
print "Updated String :- ", var1[:6] + 'Python'
```

Updated String :- Hello Python

String Special Operators

+	Concatenation - Adds values on either side of the operator	a + b will give HelloPython
*	Repetition - Creates new strings, concatenating multiple copies of the same string	a*2 will give -HelloHello
[]	Slice - Gives the character from the given index	a[1] will give e
[:]	Range Slice - Gives the characters from the given range	a[1:4] will give ell

String Formatting Operator

```
print "My name is %s and weight is %d kg!" % ('Ram', 21)
```

My name is Ram and weight is 21 kg!

Triple Quotes

```
para_str = """this is a long string that is made up of  
several lines and non-printable characters such as  
TAB ( \t ) and they will show up that way when displayed.  
NEWLINEs within the string, whether explicitly given like  
this within the brackets [ \n ], or just a NEWLINE within  
the variable assignment will also show up.  
"""  
print para_str
```

Output:

this is a long string that is made up of
several lines and non-printable characters such as
TAB () and they will show up that way when displayed.
NEWLINEs within the string, whether explicitly given like
this within the brackets [
], or just a NEWLINE within
the variable assignment will also show up.

Python Collections

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

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

Python Lists

A list is a collection which is ordered and changeable.

```
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]
```

```
list1[0]: physics  
list2[1:5]: [2, 3, 4, 5]
```

Updating Lists

```
list = ['physics', 'chemistry', 1997, 2000];  
#print "Value available at index 2 : "  
#print(list[2])
```

```
list[2] = 2001;  
print(list[2])
```

Output:
2001

Delete List Elements

```
list1 = ['physics', 'chemistry', 1997, 2000];  
del list1[2];  
print(list1)
```

Output:

```
['physics', 'chemistry', 2000]
```


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

Built-in List Functions & Methods

Sr.No.	Function with Description
1	<u>cmp(list1, list2)</u> Compares elements of both lists.
2	<u>len(list)</u> Gives the total length of the list.
3	<u>max(list)</u> Returns item from the list with max value.

4	<u>min(list)</u> Returns item from the list with min value.
5	<u>list(seq)</u> Converts a sequence of elements into list.

Sr.No.	Methods with Description
1	<u>list.append(obj)</u> Appends object obj to list
2	<u>list.count(obj)</u> Returns count of how many times obj occurs in list
3	<u>list.extend(seq) #[10,11,12]</u> Appends the contents of seq to list
4	<u>list.index(obj)</u> Returns the lowest index in list that obj appears
5	<u>list.insert(index, obj) #(3,13)</u> Inserts object obj into list at offset index

6

list.pop(obj=list[-1])

Removes and returns last object or obj from list

7

list.remove(obj)

Removes object obj from list

8

list.reverse()

Reverses objects of list in place

9

list.sort([func])

Sorts objects of list, use compare func if given

Python - Tuples

A tuple is a sequence of immutable Python objects. The differences between tuples and lists are, the tuples cannot be changed unlike lists and tuples use parentheses, whereas lists use square brackets.

Accessing Values in Tuples

```
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];
```

```
tup1[0]: physics  
tup2[1:5]: [2,3,4,5]
```

Updating Tuples

```
tup1= ('physics', 'chemistry' );  
tup2 = (1,2);  
tup3 = tup1 +tup2;  
print tup3;
```

```
('physics','chemistry',1,2)
```

Loop Through a Tuple

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

Tuple Length

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

Add Items

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

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

Remove Items

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

The tuple() Constructor

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

Tuple Methods

Method	Description
--------	-------------

<u>count()</u>	Returns the number of times a specified value occurs in a tuple
----------------	---

<u>index()</u>	Searches the tuple for a specified value and returns the position of where it was found
----------------	---

Set

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

```
thisset = {"apple", "banana", "cherry"}
```

```
for x in thisset:  
    print(x)
```

```
thisset = {"apple", "banana", "cherry"}  
print("banana" in thisset) #output: True
```

Add Items

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

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

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

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

Get the Length of a Set

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


Remove Item

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

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

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

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

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

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

The **clear()** method empties the set:

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

The **del** keyword will delete the set completely:

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

The **set()** Constructor

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

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

Dictionary

A dictionary is a collection which is unordered, changeable and indexed.

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

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

Output:
Mustang

Change Values

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

Loop Through a Dictionary

```
thisdict = {  
    "brand": "Ford",  
    "model": "Mustang",  
    "year": 1964  
}  
for x in thisdict:  
    print(thisdict[x])
```

or

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

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

or

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

Print the number of items in the dictionary:

```
print(len(thisdict))
```

Adding Items

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

Removing Items

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

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

The **popitem()** method removes the last inserted item

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

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

The dict() Constructor

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

