UNIT – 1

Definition:

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

- **Python is 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.

History of Python



- Python was developed by **Guido van Rossum** in the late eighties and early nineties at the National Research Institute for Mathematics and Computer Science in the Netherlands.
- Python is derived from many other languages, including ABC, Modula-3, C, C++, Algol-68, SmallTalk, Unix shell, and other scripting languages.
- At the time when he began implementing Python, Guido van Rossum was also reading the published scripts from "Monty Python's Flying Circus" (a BBC comedy series from the seventies, in the unlikely case you didn't know). It occurred to him that he needed a name that was short, unique, and slightly mysterious, so he decided to call the language Python.
- Python is now maintained by a core development team at the institute, although **Guido van Rossum** still holds a vital role in directing its progress.
- Python 1.0 was released on **20 February, 1991.**
- Python 2.0 was released on **16 October 2000** and had many major new features, including a cycle detecting garbage collector and support for Unicode. With this release the development process was changed and became more transparent and community-backed.
- Python 3.0 (which early in its development was commonly referred to as Python 3000 or py3k), a major, backwards-incompatible release, was released on **3 December 2008** after a long period of testing. Many of its major features have been back ported to the backwards-compatible Python 2.6.x and 2.7.x version series.
- In January 2017 Google announced work on a Python 2.7 to go transcompiler, which The Register speculated was in response to Python 2.7's planned end-of-life.

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.
- **Scalable:** Python provides a better structure and support for large programs than shell scripting.

Need of Python Programming

- **Software quality**

  Python code is designed to be *readable*, and hence reusable and maintainable—much more so than traditional scripting languages. The uniformity of Python code makes it easy to understand, even if you did not write it. In addition, Python has deep support for more advanced *software reuse* mechanisms, such as object-oriented (OO) and function programming.

- ***Developer productivity***

  Python boosts developer productivity many times beyond compiled or statically typed languages such as C, C++, and Java. Python code is typically *one-third to* less to debug, and less to maintain after the fact. Python programs also run immediately, without the lengthy compile and link steps required by some other tools, further boosting programmer speed. *Program portability* Most Python programs run unchanged on *all major computer platforms*. Porting Python code between Linux and Windows, for example, is usually just a matter of copying a script‘s code between machines.

- ***Support libraries***

  Python comes with a large collection of prebuilt and portable functionality, known as the *standard library*. This library supports an array of application-level programming tasks, from text pattern matching to network scripting. In addition, Python can be extended with both home grown libraries and a vast collection of third-party application support software. Python‘s *third-party domain* offers tools for website construction, numeric programming, serial port access, game development, and much more (see ahead for a sampling).

- ***Component integration***

Python scripts can easily communicate with other parts of an application, using a variety of integration mechanisms. Such integrations allow Python to be used as a product *customization and extension* tool. Today, Python code can invoke C and C++ libraries, can be called from C and C++ programs, can integrate with Java and .NET components, can communicate over frameworks such as COM and Silverlight, can interface with devices over serial ports, and can interact over networks with interfaces like SOAP, XML-RPC, and CORBA. It is not a standalone tool.

- ***Enjoyment***

Because of Python‘s ease of use and built-in toolset, it can make the act of programming *more pleasure than chore*. Although this may be an intangible benefit, its effect on productivity is an important asset. Of these factors, the first two (quality and productivity) are probably the most compelling benefits to most Python users, and merit a fuller description.

- **It's Object-Oriented**

Python is an object-oriented language, from the ground up. Its class model supports advanced notions such as polymorphism, operator overloading, and multiple inheritance; yet in the context of Python's dynamic typing, object-oriented programming (OOP) is remarkably easy to apply. Python's OOP nature makes it ideal as a scripting tool for object-oriented systems languages such as C++ and Java. For example, Python programs can subclass (specialized) classes implemented in C++ or Java.

- **It's Free**

Python is freeware—something which has lately been come to be called *open source software.* As with Tcl and Perl, you can get the entire system for free over the Internet. There are no restrictions on copying it, embedding it in your systems, or shipping it with your products. In fact, you can even sell Python, if you're so inclined. But don't get the wrong idea: "free" doesn't mean "unsupported". On the contrary, the Python online community responds to user queries with a speed that most commercial software vendors would do well to notice.

- **It's Portable**

Python is written in portable ANSI C, and compiles and runs on virtually every major platform in use today. For example, it runs on UNIX systems, Linux, MS-DOS, MS-Windows (95, 98, NT), Macintosh, Amiga, Be-OS, OS/2, VMS, QNX, and more. Further, Python programs are automatically compiled to portable *bytecode,* which runs the same on any platform with a compatible version of Python installed (more on this in the section "It's easy to use"). What that means is that Python programs that use the core language run the same on UNIX, MS-Windows, and any other system with a Python interpreter.

- **It's Powerful**

From a features perspective, Python is something of a hybrid. Its tool set places it between traditional scripting languages (such as Tcl, Scheme, and Perl), and systems

languages (such as C, C++, and Java). Python provides all the simplicity and ease of use of a scripting language, along with more advanced programming tools typically found in systems development languages.

- **Automatic memory management**

Python automatically allocates and reclaims ("garbage collects") objects when no longer used, and most grow and shrink on demand; Python, not you, keeps track of low-level memory details.

- **Programming-in-the-large support**

Finally, for building larger systems, Python includes tools such as modules, classes, and exceptions; they allow you to organize systems into components, do OOP, and handle events gracefully.

- **It's Mixable**

Python programs can be easily "glued" to components written in other languages. In technical terms, by employing the Python/C integration APIs, Python programs can be both extended by (called to) components written in C or C++, and embedded in (called by) C or C++ programs. That means you can add functionality to the Python system as needed and use Python programs within other environments or systems.

- **It's Easy to Use**

For many, Python's combination of rapid turnaround and language simplicity make programming more fun than work. To run a Python program, you simply type it and run it. There are no intermediate compile and link steps (as when using languages such as C or C++). As with other interpreted languages, Python executes programs immediately, which makes for both an interactive programming experience and rapid turnaround after program changes. Strictly speaking, Python programs are compiled (translated) to an intermediate form called *bytecode,* which is then run by the interpreter.

- **It's Easy to Learn**

This brings us to the topic of this book: compared to other programming languages, the core Python language is amazingly easy to learn. In fact In fact, you can expect to be coding significant Python programs in a matter of days (and perhaps in just hours, if you're already an experienced programmer).

- **Internet Scripting**

Python comes with standard Internet utility modules that allow Python programs to communicate over sockets, extract form information sent to a server-side CGI script, parse HTML, transfer files by FTP, process XML files, and much more. There are also a number of peripheral tools for doing Internet programming in Python. For instance, the HTMLGen and pythondoc systems generate HTML files from Python class-based descriptions, and the JPython system mentioned above provides for seamless Python/Java integration.

- **Database Programming**

Python's standard pickle module provides a simple object-persistence system: it allows programs to easily save and restore entire Python objects to files. For more traditional database demands, there are Python interfaces to Sybase, Oracle, Informix,

ODBC, and more. There is even a portable SQL database API for Python that runs the same on a variety of underlying database systems, and a system named *gadfly* that implements an SQL database for Python programs.

- **Image Processing, AI, Distributed Objects, Etc.**

  Python is commonly applied in more domains than can be mentioned here. But in general, many are just instances of Python's component integration role in action. By adding Python as a frontend to libraries of components written in a compiled language such as C, Python becomes useful for scripting in a variety of domains. For instance, image processing for Python is implemented as a set of library components implemented in a compiled language such as C, along with a Python frontend layer on top used to configure and launch the compiled components.

Who Uses Python Today?

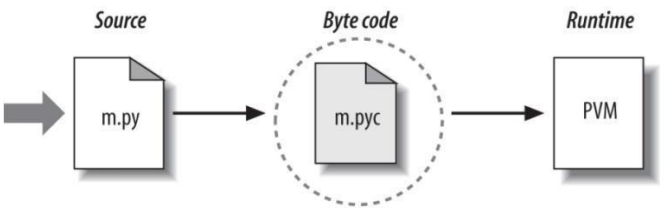
1. *Google* makes extensive use of Python in its web search systems.
2. The popular *YouTube* video sharing service is largely written in Python.
3. The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
4. The *Raspberry Pi* single-board computer promotes Python as its educational language.
5. The widespread *BitTorrent* peer-to-peer file sharing system began its life as a Python program.
6. Google‘s *App Engine* web development framework uses Python as an application language.
7. *Maya*, a powerful integrated 3D modeling and animation system, provides a Python scripting API.
8. *Intel, Cisco*, *Hewlett-Packard*, *Seagate*, *Qualcomm*, and *IBM* use Python for hardware testing.
9. *NASA*, *Los Alamos*, *Fermilab*, *JPL*, and others use Python for scientific programming tasks.

Byte code Compilation:

Python first compiles your source code (the statements in your file) into a format known as byte code. Compilation is simply a translation step, and byte code is a lower-level, platform independent representation of your source code. Roughly, Python translates each of your source statements into a group of byte code instructions by decomposing them into individual steps. This byte code translation is performed to speed execution —byte code can be run much more quickly than the original source code statements in your text file.

The Python Virtual Machine:

Once your program has been compiled to byte code (or the byte code has been loaded from existing *.pyc* file), it is shipped off for execution to something generally known as the python virtual machine (PVM).



Applications of Python:

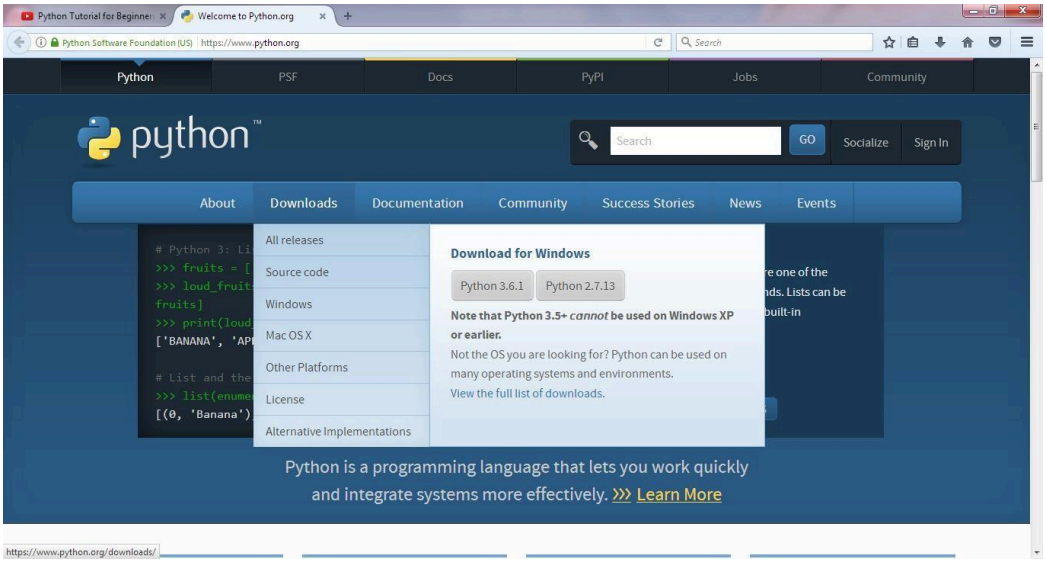
1. Systems Programming
2. GUIs
3. Internet Scripting
4. Component Integration
5. Database Programming
6. Rapid Prototyping
7. Numeric and Scientific Programming

What Are Python's Technical Strengths?

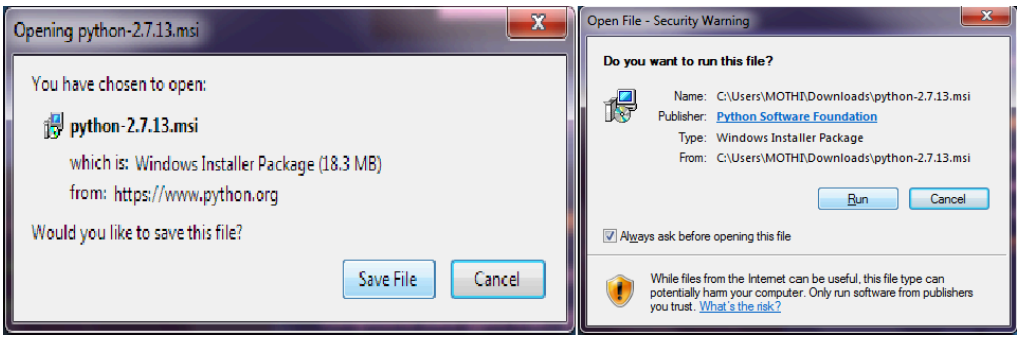
1. It's Object-Oriented and Functional
2. It's Free
3. It's Portable
4. It's Powerful
5. It's Mixable
6. It's Relatively Easy to Use
7. It's Relatively Easy to Learn

Download and installation Python software:

Step 1: Go to website www.python.org and click downloads select version which you want.



Step 2: Click on **Python 2.7.13** and download. After download open the file.


Opening python-2.7.13.msi
You have chosen to open:
python-2.7.13.msi
which is: Windows Installer Package (18.3 MB)
from: https://www.python.org
Would you like to save this file?
Save File
Cancel
Open File - Security Warning
Do you want to run this file?
Name: C:\Users\MOTHI\Downloads\python-2.7.13.msi
Publisher: Python Software Foundation
Type: Windows Installer Package
From: C:\Users\MOTHI\Downloads\python-2.7.13.msi
Run
Cancel
Always ask before opening this file
While files from the Internet can be useful, this file type can potentially harm your computer. Only run software from publishers you trust. What's the risk?

Step 3: Click on **Next** to continue.



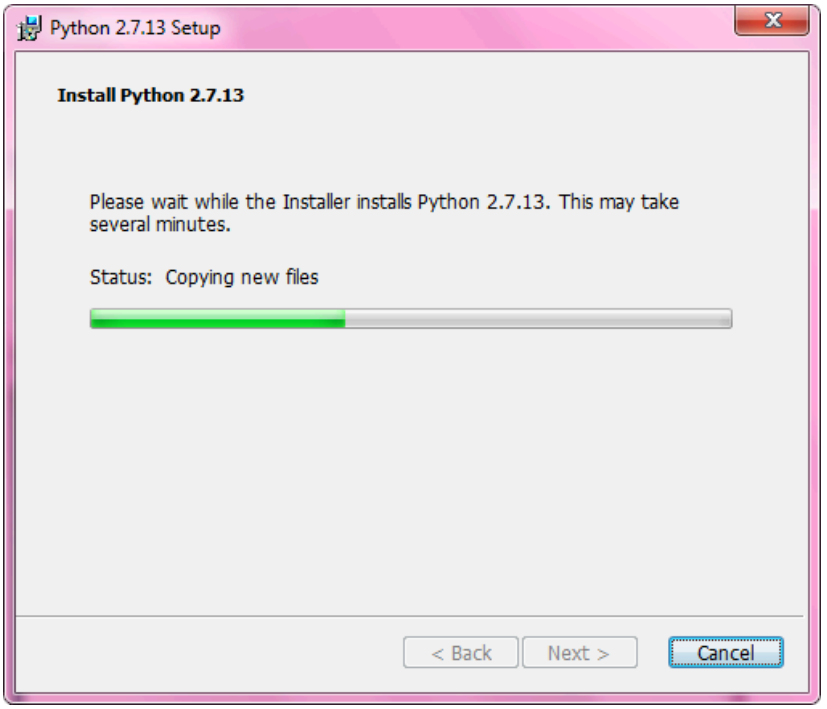
Step 4: After installation location will be displayed. The Default location is **C:\Python27.** Click on next to continue.



Step 5: After the python interpreter and libraries are displayed for installation. Click on Next to continue.



Step 6: The installation has been processed.

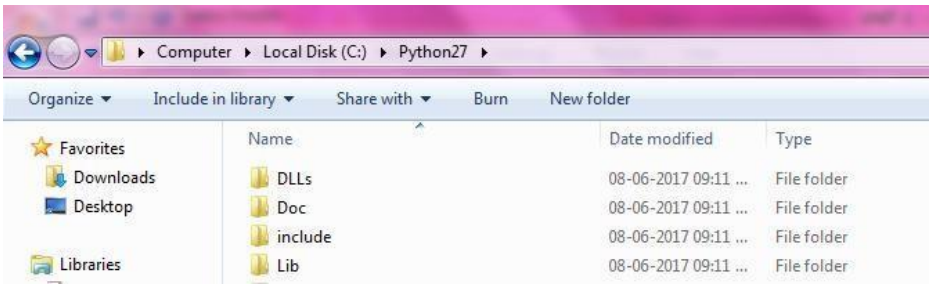


Step 7: Click the **Finish** to complete the installation.

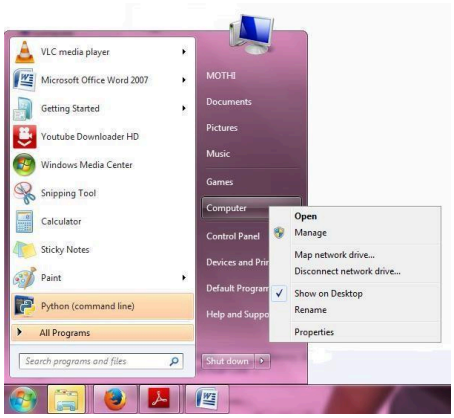


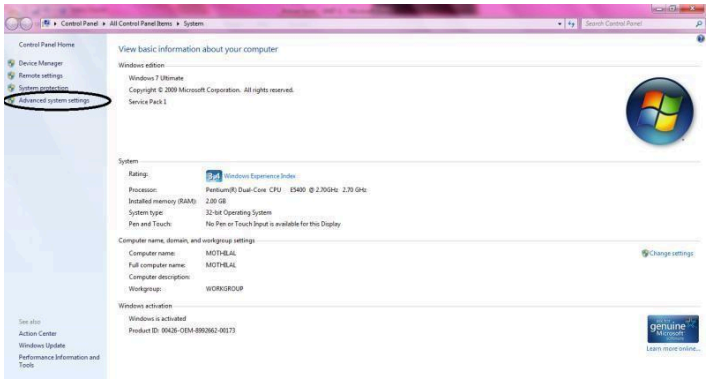
Setting up PATH to python:

- Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.
- 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.
- Copy the Python installation location C:\Python27

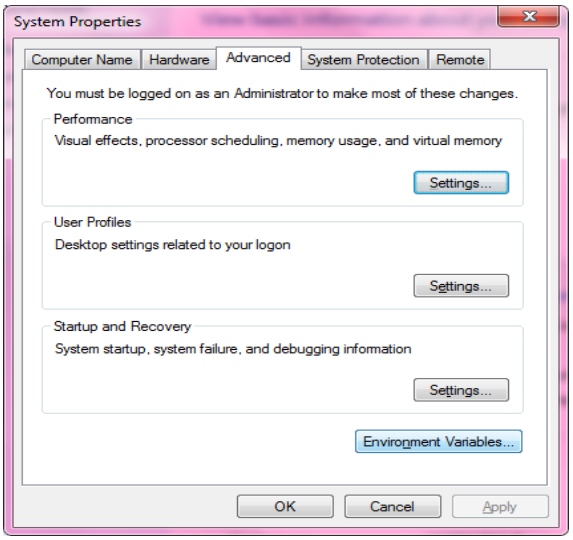


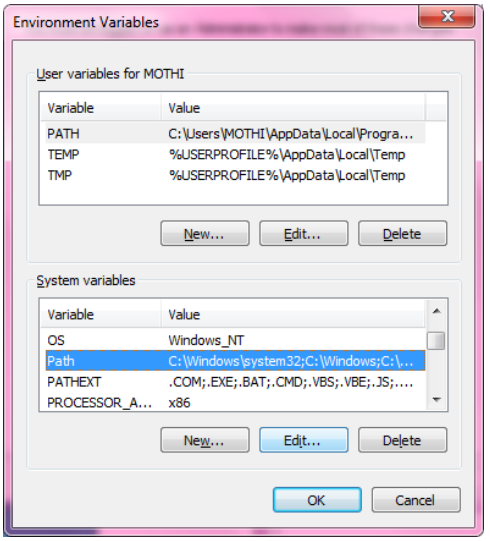
- Right-click the My Computer icon on your desktop and choose **Properties**. And then select **Advanced System properties.**



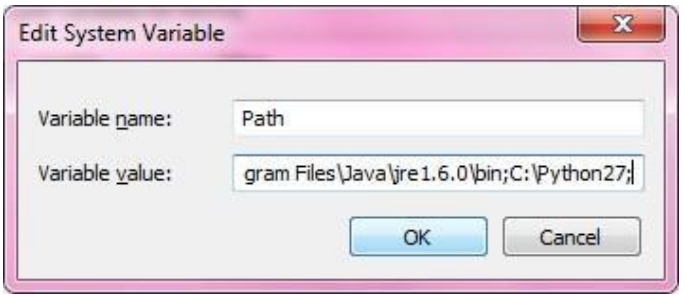


- Goto **Environment Variables** and go to **System Variables** select **Path** and click on **Edit.**





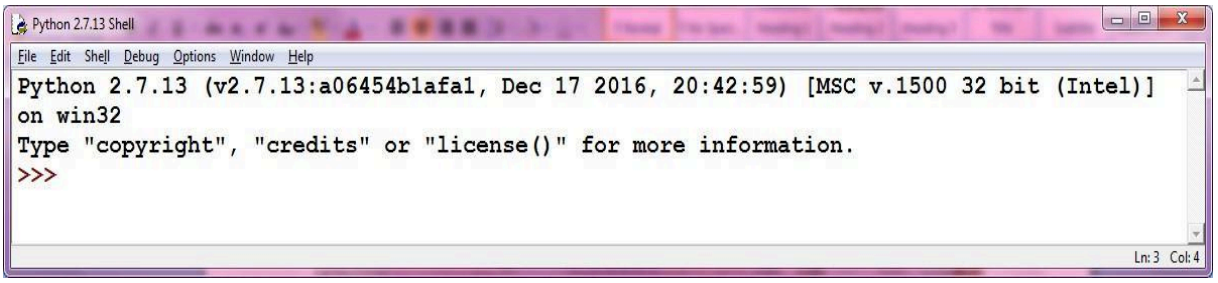
- Add semicolon (;) at end and copy the location **C:\Python27** and give semicolon (;) and click OK.


Edit System Variable
Variable name:
Path
Variable value:
gram Files\Java\jre1.6.0\bin;C:\Python27;
OK
Cancel

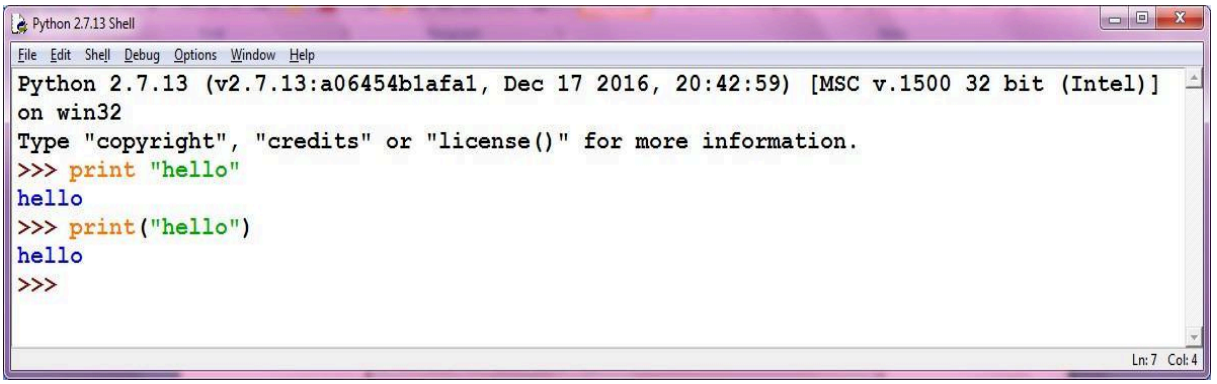
Running Python:

a. Running Python Interpreter:

Python comes with an interactive interpreter. When you type python in your shell or command prompt, the python interpreter becomes active with a >>> prompt and waits for your commands.

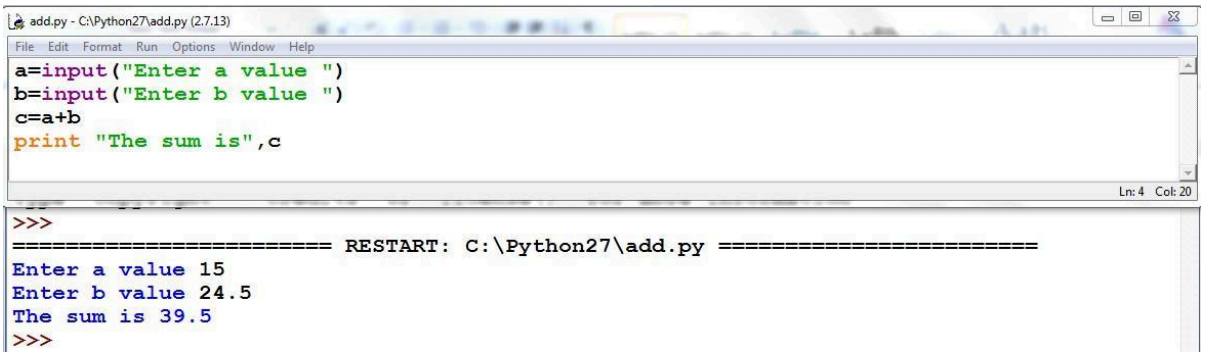


Now you can type any valid python expression at the prompt. Python reads the typed expression, evaluates it and prints the result.



b. Running Python Scripts in IDLE:

- Goto **File** menu click on New File (CTRL+N) and write the code and save add.py a=input("Enter a value ")
  b=input("Enter b value ")
  c=a+b
  print "The sum is",c
- And run the program by pressing F5 or Run ☾ Run Module.



c. Running python scripts in Command Prompt:

- Before going to run we have to check the PATH in environment variables.
- Open your text editor, type the following text and save it as hello.py.

  print "hello"
- And run this program by calling python hello.py. Make sure you change to the directory where you saved the file before doing it.



Variables:

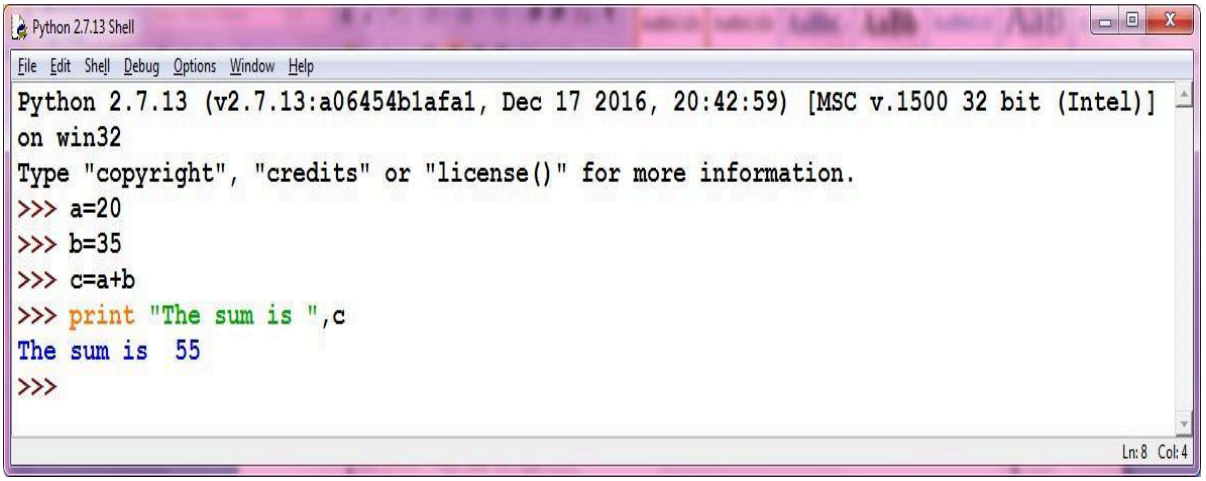
Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

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

Assigning Values to Variables

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

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



Multiple Assignments to variables:

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

For example –

a = b = c = 1

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

For example –

a, b, c = 1, 2.5, ”mothi”

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

KEYWORDS

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. All the Python keywords contain lowercase letters only.

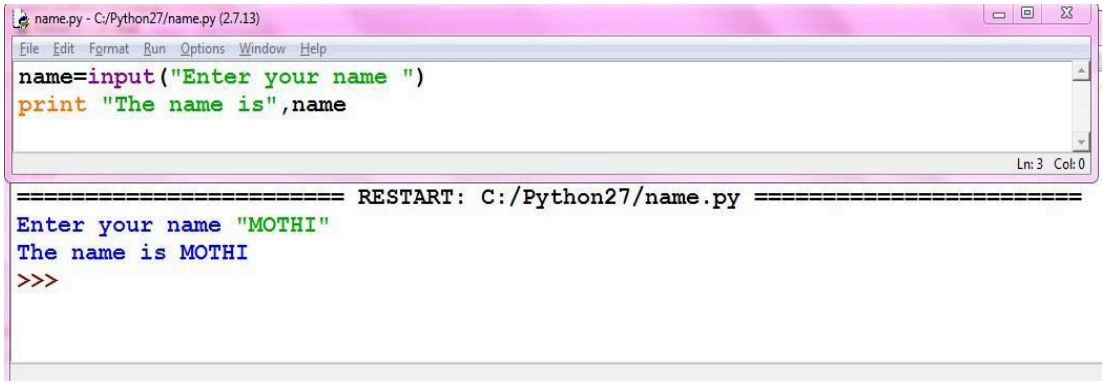
and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

INPUT Function:

To get input from the user you can use the input function. When the input function is called the program stops running the program, prompts the user to enter something at the keyboard by printing a string called the prompt to the screen, and then waits for the user to press the Enter key. The user types a string of characters and presses enter. Then the input function returns that string and Python continues running the program by executing the next statement after the input statement.

Python provides the function input(). input has an optional parameter, which is the prompt string.

For example,



OUTPUT function:

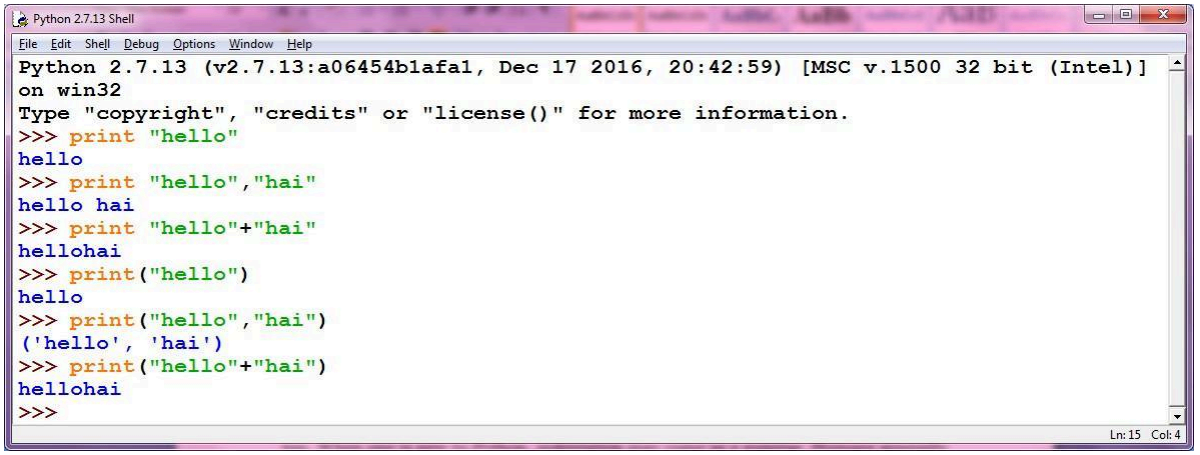
We use the print() function or print keyword to output data to the standard output device (screen). This function prints the object/string written in function.

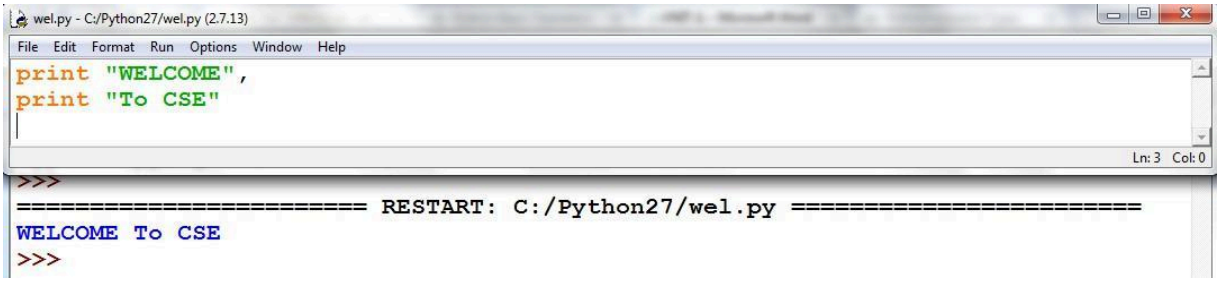
The actual syntax of the print() function is

print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)

Here, objects is the value(s) to be printed.

The sep separator is used between the values. It defaults into a space character. After all values are printed, end is printed. It defaults into a new line (\n).

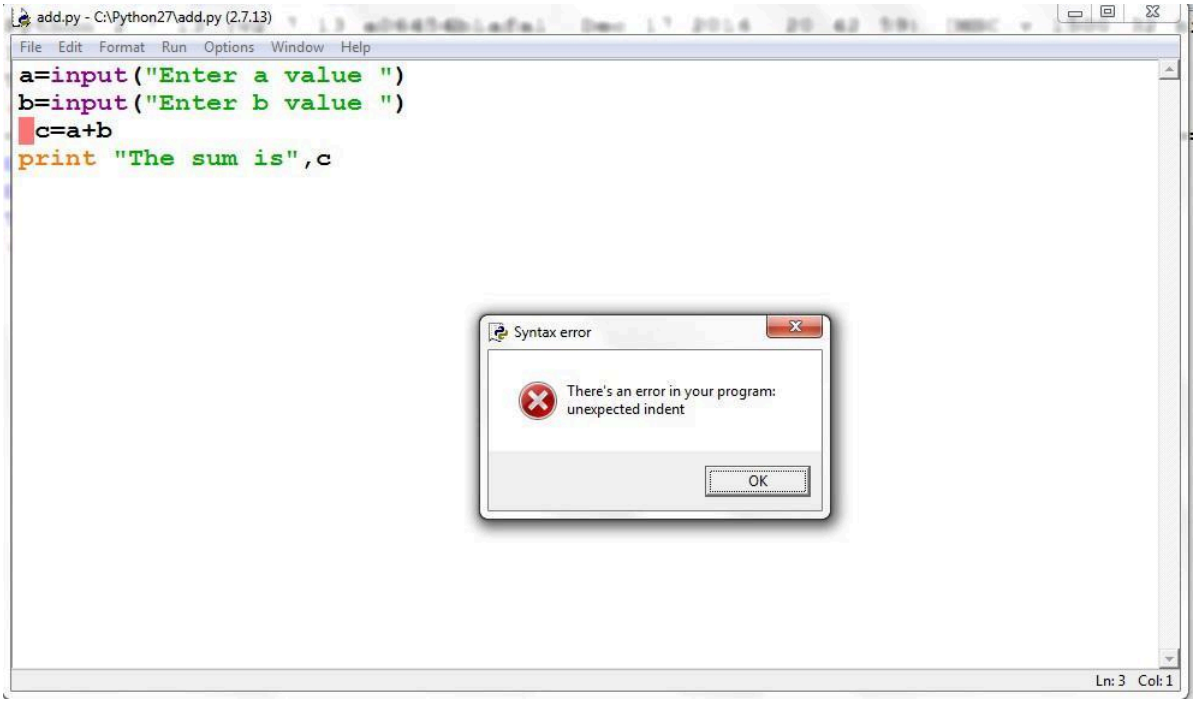




Indentation

Code blocks are identified by indentation rather than using symbols like curly braces. Without extra symbols, programs are easier to read. Also, indentation clearly identifies which block of code a statement belongs to. Of course, code blocks can consist of single statements, too. When one is new to Python, indentation may come as a surprise. Humans generally prefer to avoid change, so perhaps after many years of coding with brace delimitation, the first impression of using pure indentation may not be completely positive. However, recall that two of Python's features are that it is simplistic in nature and easy to read.

Python does not support braces to indicate blocks of code for class and function definitions or flow control. Blocks of code are denoted by line indentation. All the continuous lines indented with same number of spaces would form a block. Python strictly follow indentation rules to indicate the blocks.



Standard Data Types:

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

Python has five standard data types:

- Numbers
- String
- Boolean
- List
- Tuple
- Set
- Dictionary

Python Numbers:

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

Python supports four different numerical types:

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

Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of an ordered pair of real floating-point numbers denoted by x + yj, where x is the real part and b is the imaginary part of the complex number.

For example:

Program:

```
a = 3
b = 2.65
c = 98657412345L
d = 2+5j
print "int is",a
print "float is",b
print "long is",c
print "complex is",d
```

Output:

```
int is 3
float is 2.65
long is 98657412345
complex is (2+5j)
```

Python Strings:

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

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

Program:

```
str ="WELCOME"
print str # Prints complete string
print str[0] # Prints first character of the string
print str[2:5] # Prints characters starting from 3rd to 5th
print str[2:] # Prints string starting from 3rd character
print str * 2 # Prints string two times
print str + "CSE" # Prints concatenated string
```

Output:

```
WELCOME
W
LCO
LCOME
WELCOMEWELCOM
E WELCOMECSE
```

Built-in String methods for Strings:

SNO	Method Name	Description
1	capitalize()	Capitalizes first letter of string.
2	center(width, fillchar)	Returns a space-padded string with the original string centered to a total of width columns.

3	count(str, beg= 0,end=len(string))	Counts how many times str occurs in string or in a substring of string if starting index beg and ending index end are given.
4	decode(encoding='UTF-8',errors='strict')	Decodes the string using the codec registered for encoding. Encoding defaults to the default string encoding.
5	encode(encoding='UTF-8',errors='strict')	Returns encoded string version of string; on error, default is to raise a Value Error unless errors is given with 'ignore' or 'replace'.
6	endswith(suffix, beg=0, end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) ends with suffix; returns true if so and false otherwise.
7	expandtabs(tabsize=8)	Expands tabs in string to multiple spaces; defaults to 8 spaces per tab if tabsize not provided.
8	find(str, beg=0 end=len(string))	Determine if str occurs in string or in a substring of string if starting index beg and ending index end are given returns index if found and -1 otherwise.
9	index(str, beg=0, end=len(string))	Same as find(), but raises an exception if str not found.
10	isalnum()	Returns true if string has at least 1 character and all characters are alphanumeric and false otherwise.
11	isalpha()	Returns true if string has at least 1 character and all characters are alphabetic and false otherwise.
12	isdigit()	Returns true if string contains only digits and false otherwise.
13	islower()	Returns true if string has at least 1 cased character and all cased characters are in lowercase and false otherwise.
14	isnumeric()	Returns true if a unicode string contains only numeric characters and false otherwise.
15	isspace()	Returns true if string contains only whitespace characters and false otherwise.
16	istitle()	Returns true if string is properly "titlecased" and false otherwise.
17	isupper()	Returns true if string has at least one cased character and all cased characters are in uppercase and false otherwise.
18	join(seq)	Merges (concatenates) the string representations of elements in sequence seq into a string, with separator string.
19	len(string)	Returns the length of the string.
20	ljust(width[, fillchar])	Returns a space-padded string with the original string left-justified to a total of width columns.

21	lower()	Converts all uppercase letters in string to lowercase.
22	lstrip()	Removes all leading whitespace in string.
23	maketrans()	Returns a translation table to be used in translates function.
24	max(str)	Returns the max alphabetical character from the string str.
25	min(str)	Returns min alphabetical character from the string str.
26	replace(old, new [, max])	Replaces all occurrences of old in string with new or at most max occurrences if max given.
27	rfind(str, beg=0,end=len(string))	Same as find(), but search backwards in string.
28	rindex(str, beg=0, end=len(string))	Same as index(), but search backwards in string.
29	rjust(width,[, fillchar])	Returns a space-padded string with the original string right-justified to a total of width columns.
30	rstrip()	Removes all trailing whitespace of string.
31	split(str="", num=string.count(str))	Splits string according to delimiter str (space if not provided) and returns list of substrings; split into at most num substrings if given.
32	splitlines (num=string.count('\n'))	Splits string at all (or num) NEWLINEs and returns a list of each line with NEWLINEs removed.
33	startswith(str, beg=0,end=len(string))	Determines if string or a substring of string (if starting index beg and ending index end are given) starts with substring str; returns true if so and false otherwise.
34	strip([chars])	Performs both lstrip() and rstrip() on string.
35	swapcase()	Inverts case for all letters in string.
36	title()	Returns "titlecased" version of string, that is, all words begin with uppercase and the rest are lowercase.
37	translate(table, deletechars="")	Translates string according to translation table str(256 chars), removing those in the del string.
38	upper()	Converts lowercase letters in string to uppercase.
39	zfill (width)	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, zfill() retains any sign given (less one zero).
40	isdecimal()	Returns true if a unicode string contains only decimal characters and false otherwise.

Example:

```
str1="welcome"
print "Capitalize function---",str1.capitalize()
print str1.center(15,"*")
print "length is",len(str1)
```

```
print "count function---",str1.count('e',0,len(str1))
print "endswith function---",str1.endswith('me',0,len(str1))
print "startswith function---",str1.startswith('me',0,len(str1))
print "find function---",str1.find('e',0,len(str1))
str2="welcome2017"
print "isalnum
function---",str2.isalnum() print "isalpha
function---",str2.isalpha() print "islower
function---",str2.islower() print "isupper
function---",str2.isupper() str3="
welcome"
print "lstrip function---",str3.lstrip()
str4="77777777cse777777";
print "lstrip function---",str4.lstrip('7')
print "rstrip function---",str4.rstrip('7')
print "strip function---",str4.strip('7')
str5="welcome to java"
print "replace function---",str5.replace("java","python")
```

Output:

```
Capitalize function--- Welcome
****welcome****
length is 7
count function--- 2
endswith function--- True
startswith function--- False
find function--- 1
isalnum function--- True
isalpha function--- False
islower function--- True
isupper function--- False
lstrip function--- welcome
lstrip function--- cse777777
rstrip function--- 77777777cse
strip function--- cse
replace function--- welcome to python
```

Python Boolean:

Booleans are identified by True or False.

Example:

Program:

```
a = True
b = False
print a
print b
```

Output:

```
True
False
```

Data Type Conversion:

Sometimes, you may need to perform conversions between the built-in types. To convert between types, you simply use the type name as a function. For example, it is not possible to perform "2"+4 since one operand is integer and the other is string type. To perform this we have convert string to integer i.e., **int("2") + 4 = 6**.

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

Function	Description
int(x [,base])	Converts x to an integer.
long(x [,base])	Converts x to a long integer.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
eval(str)	Evaluates a string and returns an object.
tuple(s)	Converts s to a tuple.
list(s)	Converts s to a list.
set(s)	Converts s to a set.
dict(d)	Creates a dictionary, d must be a sequence of (key, value) tuples.
frozenset(s)	Converts s to a frozen set.
chr(x)	Converts an integer to a character.
unichr(x)	Converts an integer to a Unicode character.
ord(x)	Converts a single character to its integer value.
hex(x)	Converts an integer to a hexadecimal string.
oct(x)	Converts an integer to an octal string.

Types of Operators:

Python language supports the following types of operators.

- Arithmetic Operators **+, -, *, /, %, **, //**
- Comparison (Relational) Operators **= =, ! =, < >, <, >, <=, >=**
- Assignment Operators **=, +=, -=, *=, /=, %=, **=, //=**
- Logical Operators **and, or, not**
- Bitwise Operators **&, |, ^, ~,<<, >>**
- Membership Operators **in, not in**
- Identity Operators **is, is not**

Arithmetic Operators:

Some basic arithmetic operators are +, -, *, /, %, **, and //. You can apply these operators on numbers as well as variables to perform corresponding operations.

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	a + b = 30
- Subtraction	Subtracts right hand operand from left hand operand.	a – b = -10

* Multiplication	Multiplies values on either side of the operator	a * b = 200
/ Division	Divides left hand operand by right hand operand	b / a = 2
% Modulus	Divides left hand operand by right hand operand and returns remainder	b % a = 0
** Exponent	Performs exponential (power) calculation on operators	a**b =10 to the power 20
// Floor Division	The division of operands where the result is the quotient in which the digits after the decimal point are removed.	9//2 = 4 and 9.0//2.0 = 4.0

Program:

```
a = 21
b = 10
print "Addition is", a + b
print "Subtraction is ", a - b
print "Multiplication is ", a * b
print "Division is ", a / b
print "Modulus is ", a % b
a = 2
b = 3
print "Power value is ", a ** b
a = 10
b = 4
print "Floor Division is ", a // b
```

Output:

```
Addition is 31
Subtraction is 11
Multiplication is 210
Division is 2
Modulus is 1
Power value is 8
Floor Division is 2
```

Comparison (Relational) Operators

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

Operator	Description	Example
= =	If the values of two operands are equal, then the condition becomes true.	(a == b) is not true.
!=	If values of two operands are not equal, then condition becomes true.	(a != b) is true.
< >	If values of two operands are not equal, then condition becomes true.	(a <> b) is true. This is similar to != operator.

>	If the value of left operand is greater than the value of right operand, then condition becomes true.	(a > b) is not true.
<	If the value of left operand is less than the value of right operand, then condition becomes true.	(a < b) is true.
> =	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	(a >= b) is not true.
< =	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	(a <= b) is true.

Example:

```
a=20
b=30
if a < b:
   print "b is big"
elif a > b:
   print "a is big"
else:
print "Both are equal"
```

Output:

b is bigAssignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

Example:

```
a=82
b=27
a += b
print a
a=25
b=12
a  -= b
print a
a=24
b=4
```

```
a *= b
print a
a=4
b=6
a **= b
print a
```

Output:

```
109
13
96
4096
```

Logical Operators

Operator	Description	Example
And Logical AND	If both the operands are true then condition becomes true.	(a and b) is true.
Or Logical OR	If any of the two operands are non-zero then condition becomes true.	(a or b) is true.
not Logical NOT	Used to reverse the logical state of its operand.	Not (a and b) is false.

Example:

```
a=20
b=1
0
c=30
if a >= b and a >= c:
   print "a is big"
elif b >= a and b >= c:
   print "b is big"
else:
   print "c is big"
```

Output:

```
c is big
```

Bitwise Operators

Operator	Description	Example
& Binary AND	Operator copies a bit to the result if it exists in both operands.	(a & b) = 12 (means 0000 1100)
| Binary OR	It copies a bit if it exists in either operand.	(a | b) = 61 (means 0011 1101)

^ Binary XOR	It copies the bit if it is set in one operand but not both.	(a ^ b) = 49 (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	(~a) = -61 (means 1100 0011 in 2's complement form due to a signed binary number.
<< Binary Left Shift	The left operands value is moved left by the number of bits specified by the right operand.	a << 2 = 240 (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	a >> 2 = 15 (means 0000 1111)

Membership Operators

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

Operator	Description	Example
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.

Example:

```
a = 3
list = [1, 2, 3, 4, 5 ];
if ( a in list ):
        print "available"
else:
```

Output:

```
print " not available"
```

Identity Operators

Identity operators compare the memory locations of two objects.

Operator	Description	Example
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).

Example:

```
a = 20
b = 20
if ( a is b ):
        print "Line 1 - a and b have same identity"
else:
        print "Line 1 - a and b do not have same identity"
if ( id(a) == id(b) ):
        print "Line 2 - a and b have same identity"
    else:
```

Output:

```
print "Line 2 - a and b do not have same identity"
        Line 1 - a and b have same
        identity Line 2 - a and b have
        same identity
```

Python Operators Precedence

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

Operator	Description
()	Parenthesis
**	Exponentiation (raise to the power)
~ x, +x, -x	Complement, unary plus and minus
* / % //	Multiply, divide, modulo and floor division
+ -	Addition and subtraction
>> <<	Right and left bitwise shift
&	Bitwise 'AND'
^ |	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

Expression:

An expression is a combination of variables constants and operators written according to the syntax of Python language. In Python every expression evaluates to a value i.e., every expression results in some value of a certain type that can be assigned to a variable. Some examples of Python expressions are shown in the table given below.

Algebraic Expression	**Python Expression**
a x b – c	a * b – c
(m + n) (x + y)	(m + n) * (x + y)
(ab / c)	a * b / c
$3x^2$ +2x + 1	3*x*x+2*x+1
(x / y) + c	x / y + c

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression

Variable is any valid C variable name. When the statement is encountered, the expression is evaluated first and then replaces the previous value of the variable on the left hand side. All variables used in the expression must be assigned values before evaluation is attempted.

Example:

```
a=10 b=22 c=34
x=a*b+c y=a-b*c z=a+b+c*c-a print "x=",x print "y=",y
print "z=",z
```

Output:

```
x= 254
y= -738
z= 1178
```

Python Type Conversion

In programming, type conversion is the process of converting data of one type to another. For example: converting int data to str.

There are two types of type conversion in Python.

- Implicit Conversion - automatic type conversion
- Explicit Conversion - manual type conversion

Python Implicit Type Conversion

Implicit character data conversion is used in Python when a data type conversion occurs, whether during compilation or runtime. We do not need to manually change the file format into some other data type because Python performs the implicit character data conversion. Throughout the tutorial, we have seen numerous examples of this type of data type conversion. Without user input, the Programming language automatically changes one data type to another in an implicit shift of data **types.**

Example 1: Converting integer to float

Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

```
integer_number = 123
float_number = 1.23
new_number = integer_number + float_number
# display new value and resulting data type
print("Value:",new_number)
print("Data Type:",type(new_number))
```

Run Code

Output

```
Value: 124.23
Data Type: <class 'float'>
```

In the above example, we have created two
variables: integer_number and float_number of int and float type respectively.
Then we added these two variables and stored the result in new_number.
As we can see new_number has value **124.23** and is of the float data type.
It is because Python always converts smaller data types to larger data types to avoid the loss of data.

Explicit Type Conversion

In Explicit Type Conversion, users convert the data type of an object to required data type. Explicit type conversion, commonly referred to as type casting, now steps in to save the day. Using built-in Language functions like str() to convert to string form and int() to convert to integer type, a programmer can explicitly change the data form of an object by changing it manually. We use the built-in functions like int(), float(), str(), etc to perform explicit type conversion.

This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Example 2: Addition of string and integer Using Explicit Conversion

```
num_string = '12'
num_integer = 23
print("Data type of num_string before Type Casting:",type(num_string))
# explicit type conversion
num_string = int(num_string)
print("Data type of num_string after Type Casting:",type(num_string))
num_sum = num_integer + num_string
print("Sum:",num_sum)
print("Data type of num_sum:",type(num_sum))
```

Run Code

Output

```
Data type of num_string before Type Casting: <class 'str'>
Data type of num_string after Type Casting: <class 'int'>
Sum: 35
Data type of num_sum: <class 'int'>
```

In the above example, we have created two
variables: num_string and num_integer with str and int type values respectively. Notice the code,
num_string = int(num_string)
Here, we have used int() to perform explicit type conversion of num_string to integer type. After converting num_string to an integer value, Python is able to add these two variables. Finally, we got the num_sum value i.e **35** and data type to be int.

Program code:

Here we give another example of explicit type conversion in Python. In the below code, we initialize the values and find their data type in Python. So, the code is given below -

```
# Python code to show type conversion
# Initializing string with int() and float()
a = "10010"
# # outputting string to int base 2 conversion.
b = int(a,2)
print ("following the conversion to integer base 2: ", end="")
print (r)
# printing a float after converting a string
d = float(a)
print ("After converting to float : ", end="")
print (d)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
following the conversion to integer base 2: 18
After converting to float : 1010.0
```

c. The ord() method turns a character into an integer.

d. The hex() method turns an integer into a hexadecimal string.

e. The oct() method turns an integer into an octal string.

Program code:

Here we gave an example of finding the integer, hexadecimal, and octal values of given values. The code is given below -

```
# Python code to show type conversion
 # initialising integer
 # using ord(), hex(), and oct()
a = '4'
# printing and converting a character to an integer
b = ord(a)
print ("After converting character into integer : ",end="")
print (b)
# printing integer converting to hexadecimal string
b = hex(56)
print ("After converting 56 to hexadecimal string : ",end="")
print (b)
# printing the integer converting into octal string
b = oct(56)
print ("After converting 56 into octal string : ",end="")
```

```
print (b)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
After converting the character into integer : 52
After converting the 56 to hexadecimal string : 0x38
After converting the 56 into octal string : 0o70
```

f. The tuple() method is used to transform data into a tuple.

g. The set() function, which converts a type to a set, returns the set.

h. The list() function transforms any data type into a list type.

Program code:

Here we give an example of how to convert a given string into a list(), tuple(), and set(). The code is given below -

```
# Python code to show type conversion
 # Initializing string with tuple(), set(), and list()
x = 'javaTpoint'
# printing string into converting to tuple
y = tuple(x)
print ("After converting the string to a tuple: ",end="")
print (y)
# printing string the converting to set
y = set(x)
print ("After converting the string to a set: ",end="")
print (y)
# printing the string converting to list
y = list(x)
print ("After converting the string to the list: ",end="")
print (y)
```

Output:

Now we compile the above code in Python, and after successful compilation, we run it. Then the output is given below -

```
After converting the string to a tuple: ('j,' 'a', 'v,' 'a', 'T,' 'p,' 'o', 'i', 'n,' 't')
After converting the string to a set: {'n,' 'o', 'p,' 'j,' 'v,' 'T,' 'i', 'a', 't'}
After converting the string to the list: ['j,' 'a', 'v,' 'a', 'T,' 'p,' 'o', 'i', 'n,' 't']
```

Explanation:

In this above code, we convert a given string, "javaTpoint" into the tuple(), set(), and list(). The set is not ordered, so the elements are not order-wise in the output.

Python is both a strongly typed and a dynamically typed language

Strong typing means that variables do have a type and that the type matters when performing operations on a variable.

Dynamic typing means that the type of the variable is determined only during run time.

other hand, types in Python are determined during run-time as opposed to compile-time and thus programmers are not required to declare variables before using them in the code.Technically, a variable is created when it is assigned a value for the first time in the code.

This means that a variable must first be assigned a value before it is referenced in the code otherwise an error will be reported. And as mentioned earlier, the variable assignment never comes with a type definition since the type is stored with objects and not with variables/names.

Every time a variable is identified, it automatically gets replaced with the object in memory it references.