

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, 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?

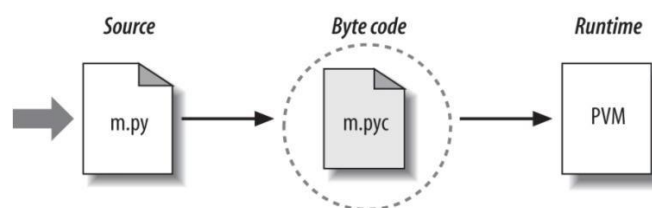
1. *Google* makes extensive use of Python in its web search systems.
2. The popular *YouTube* video sharing service is largely written in Python.
3. The *Dropbox* storage service codes both its server and desktop client software primarily in Python.
4. The *Raspberry Pi* single-board computer promotes 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:

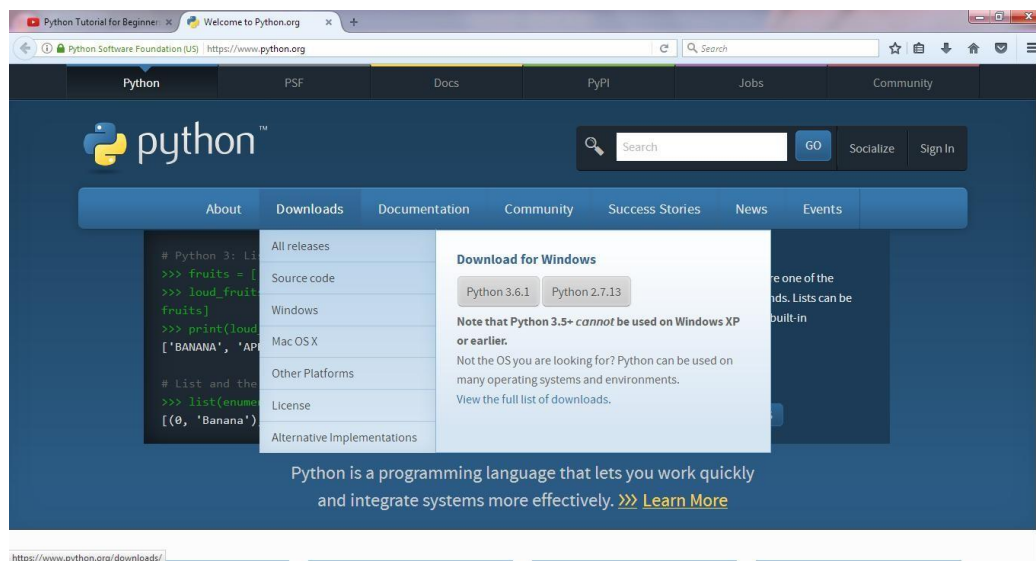
1. Systems Programming
2. GUIs
3. Internet Scripting
4. Component Integration
5. Database Programming
6. Rapid Prototyping
7. Numeric and Scientific Programming

What Are Python's Technical Strengths?

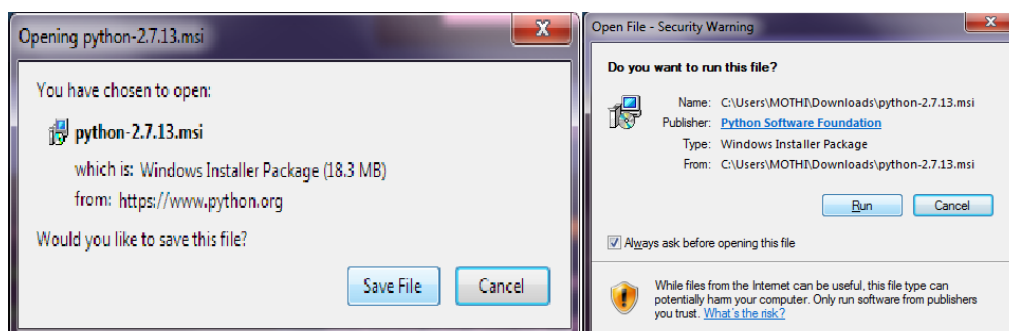
1. It's Object-Oriented and Functional
2. It's Free
3. It's Portable
4. It's Powerful
5. It's Mixable
6. It's Relatively Easy to Use
7. It's Relatively Easy to Learn

Download and installation Python software:

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



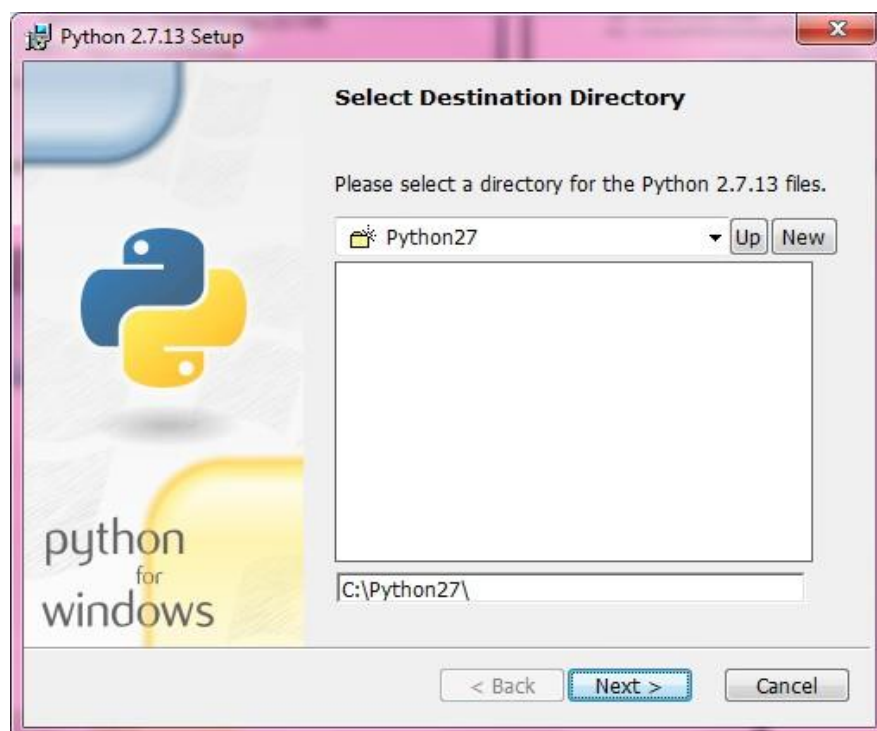
Step 2: Click on **Python 2.7.13** and download. After download open the file.



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



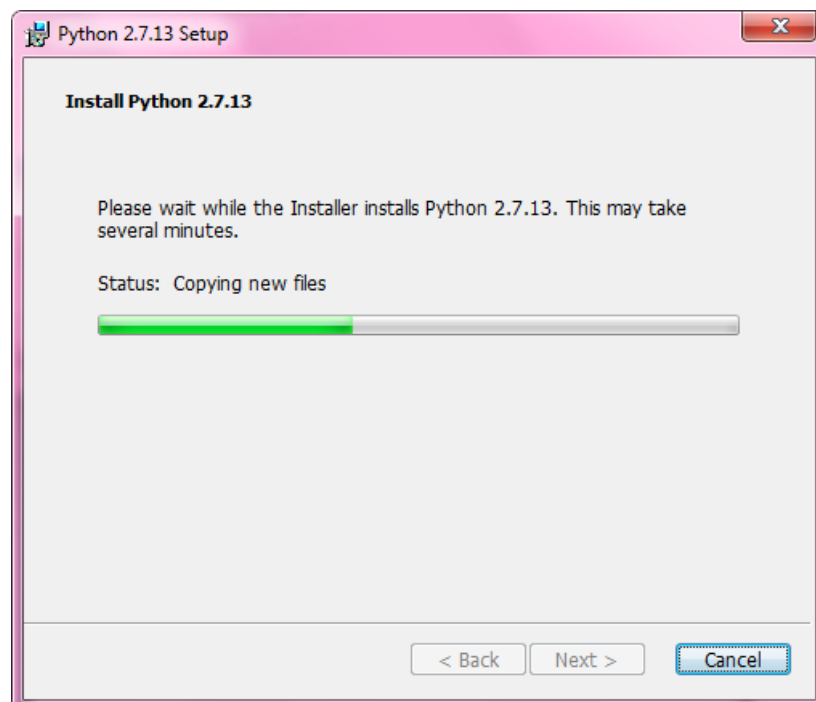
Step 4: After installation location will be displayed. The Default location is **C:\Python27**. Click on next to continue.



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.

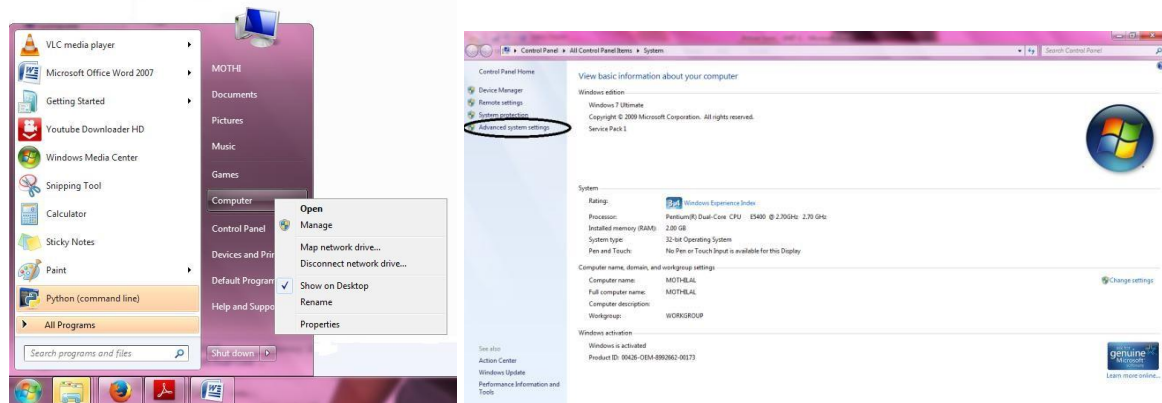


Setting up PATH to python:

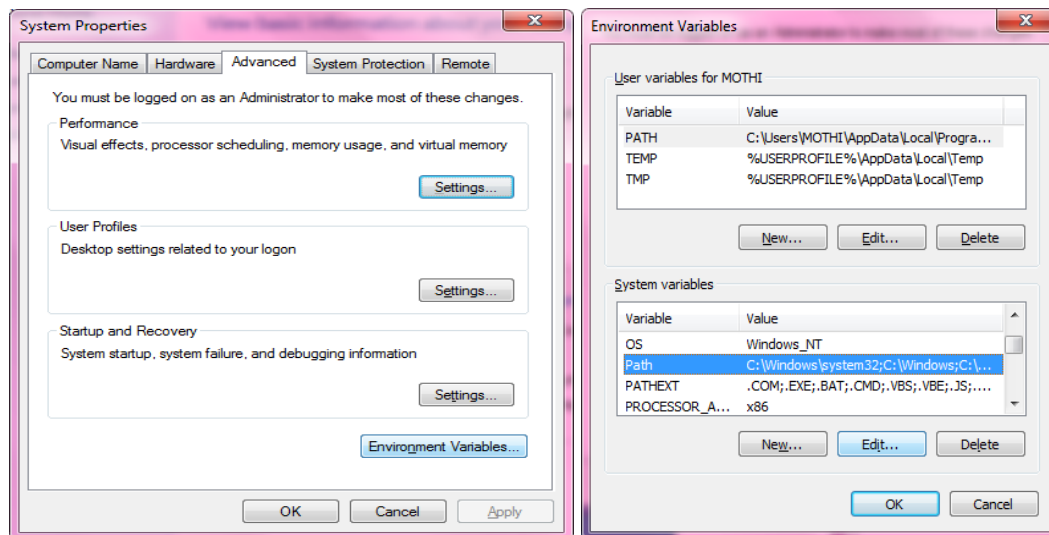
- Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.
- The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs.
- Copy the Python installation location `C:\Python27`



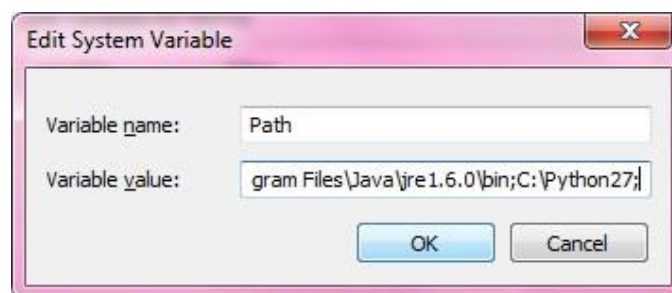
- Right-click the My Computer icon on your desktop and choose **Properties**. And then select **Advanced System properties**.



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



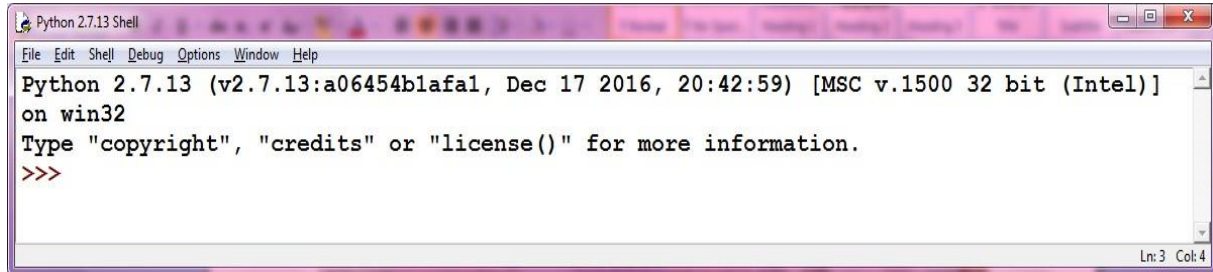
- Add semicolon (;) at end and copy the location **C:\Python27** and give semicolon (;) and click OK.



Running Python:

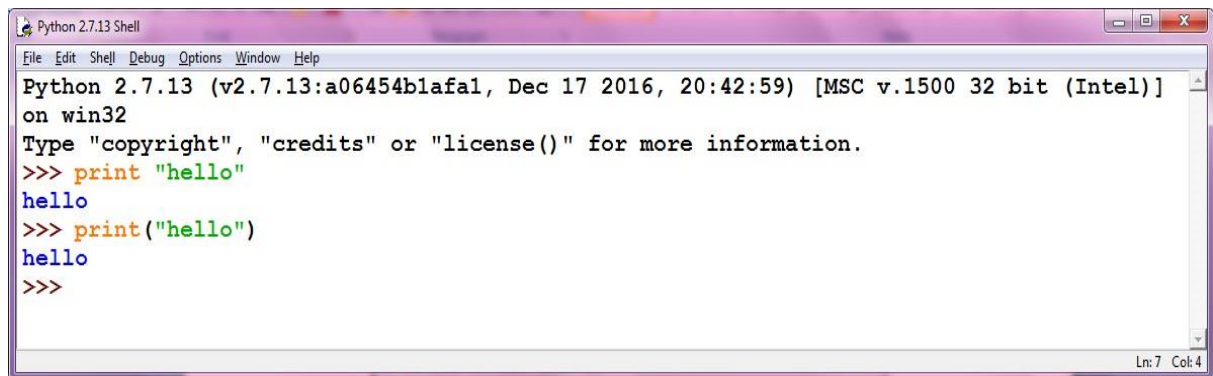
a. Running Python Interpreter:

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.



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>>
```

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

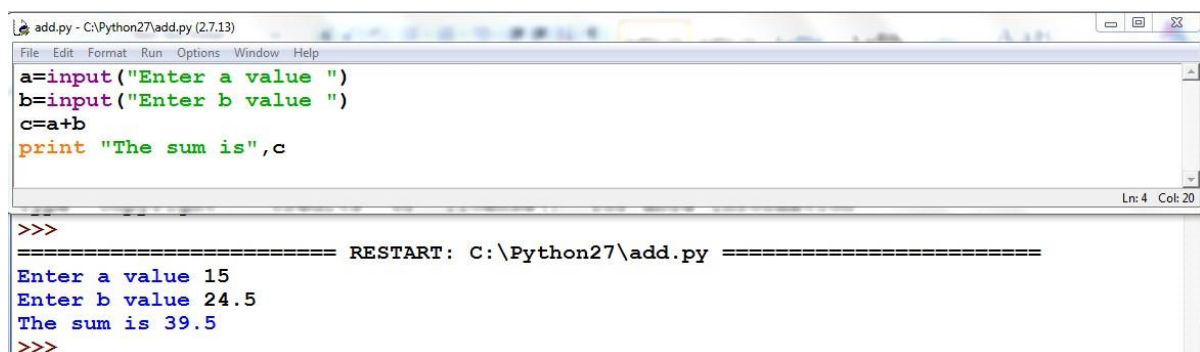


```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello"
hello
>>> print("hello")
hello
>>>
```

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.



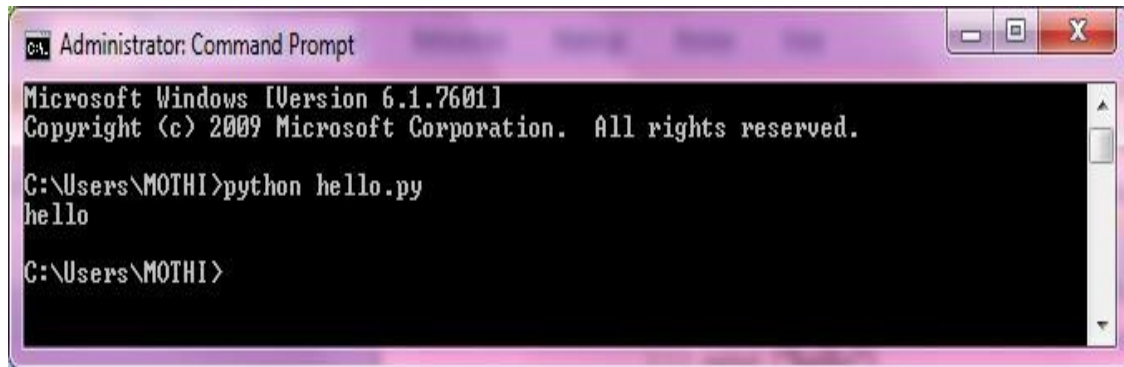
```
add.py - C:\Python27\add.py (2.7.13)
File Edit Format Run Options Window Help
a=input("Enter a value ")
b=input("Enter b value ")
c=a+b
print "The sum is",c

>>>
===== RESTART: C:\Python27\add.py =====
Enter a value 15
Enter b value 24.5
The sum is 39.5
>>>
```

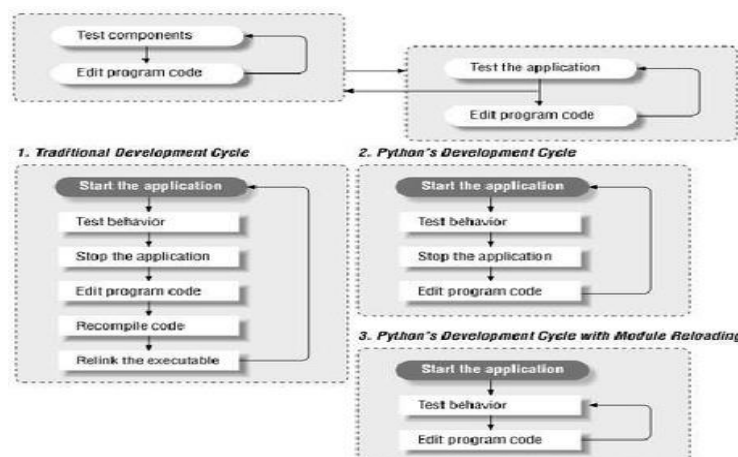
c. Running python scripts in Command Prompt:

- 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.

**Program Development Cycle:**

Python's development cycle is dramatically shorter than that of traditional tools. In Python, there are no compile or link steps -- Python programs simply import modules at runtime and use the objects they contain. Because of this, Python programs run immediately after changes are made. And in cases where dynamic module reloading can be used, it's even possible to change and reload parts of a running program without stopping it at all. Figure shows Python's impact on the development cycle.



Because Python is interpreted, there's a rapid turnaround after program changes. And because Python's parser is embedded in Python-based systems, it's easy to modify programs at runtime. For example, we saw how GUI programs developed with Python allow developers to change the code that handles a button press while the GUI remains active; the effect of the code change may be observed immediately when the button is pressed again. There's no need to stop and rebuild.

More generally, the entire development process in Python is an exercise in rapid prototyping. Python lends itself to experimental, interactive program development, and encourages developing systems incrementally by testing components in isolation and putting them together later. In fact, we've seen that we can switch from testing components (unit tests) to testing whole systems (integration tests) arbitrarily.

Variables:

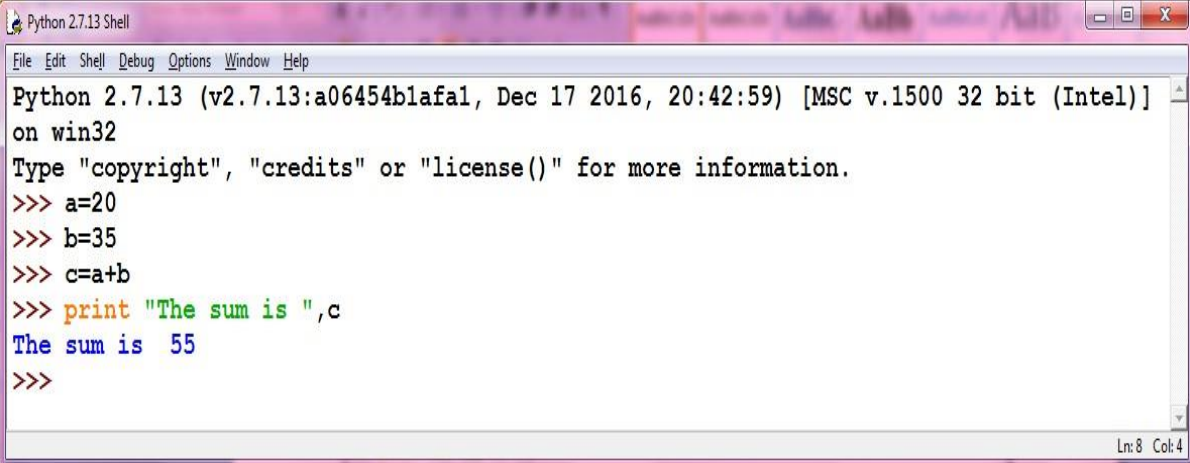
Variables are nothing but reserved memory locations to store values. This means that when you create a variable you reserve some space in memory.

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 –

A screenshot of a Python 2.7.13 Shell window. The window has a title bar that says "Python 2.7.13 Shell" and a menu bar with "File", "Edit", "Shell", "Debug", "Options", "Window", and "Help". The main text area shows the following code and output:

```
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> a=20
>>> b=35
>>> c=a+b
>>> print "The sum is ",c
The sum is  55
>>>
```

The status bar at the bottom right indicates "Ln: 8 Col: 4".

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.

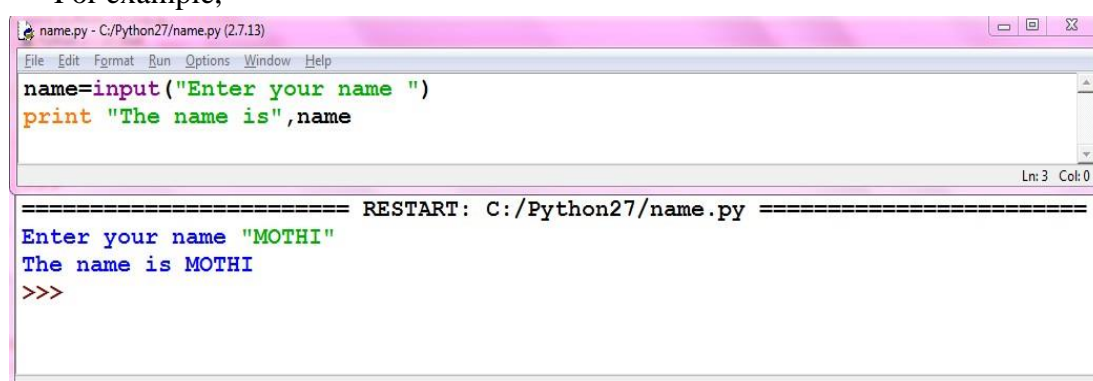
and	elif	if	print
as	else	import	raise
assert	except	in	return
break	exec	is	try
class	finally	lambda	while
continue	for	not	with
def	from	or	yield
del	global	pass	

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,



The screenshot shows a Python IDE window titled 'name.py - C:/Python27/name.py (2.7.13)'. The code in the editor is:

```
name=input("Enter your name ")
print "The name is",name
```

The output window shows the execution results:

```
===== RESTART: C:/Python27/name.py =====
Enter your name "MOTHI"
The name is MOTHI
>>>
```


OUTPUT function:

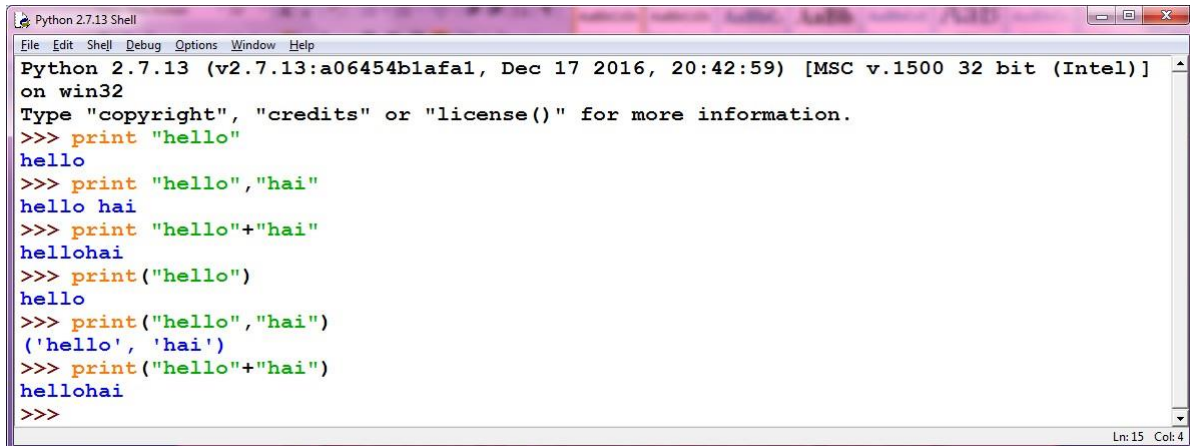
We use the `print()` function or `print` keyword to output data to the standard output device (screen). This function prints the object/string written in function.

The actual syntax of the `print()` function is

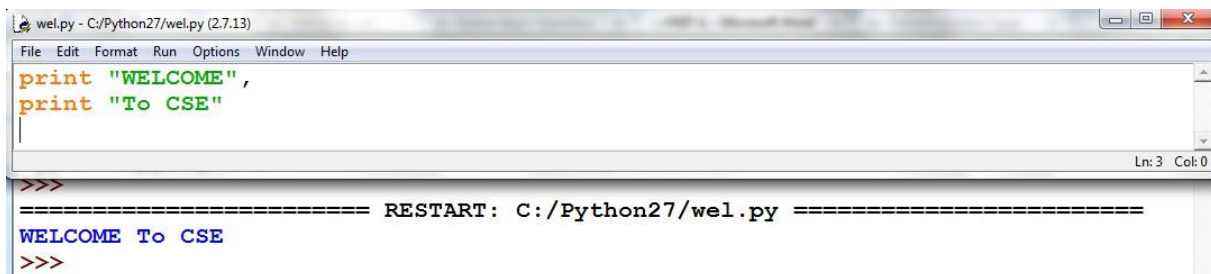
`print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)`

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`).



```
Python 2.7.13 Shell
File Edit Shell Debug Options Window Help
Python 2.7.13 (v2.7.13:a06454b1afaf1, Dec 17 2016, 20:42:59) [MSC v.1500 32 bit (Intel)]
on win32
Type "copyright", "credits" or "license()" for more information.
>>> print "hello"
hello
>>> print "hello", "hai"
hello hai
>>> print "hello"+"hai"
hellohai
>>> print("hello")
hello
>>> print("hello", "hai")
('hello', 'hai')
>>> print("hello"+"hai")
hellohai
>>>
```



```
wel.py - C:/Python27/wel.py (2.7.13)
File Edit Format Run Options Window Help
print "WELCOME",
print "To CSE"

>>>
===== RESTART: C:/Python27/wel.py =====
WELCOME To CSE
>>>
```

Formatting Output:

There are several ways to present the output of a program, data can be printed in a human-readable form, or written to a file for future use. Sometimes user often wants more control the formatting of output than simply printing space-separated values. There are several ways to format output.

- To use formatted string literals, begin a string with `f` or `F` before the opening quotation mark or triple quotation mark.
- The `str.format()` method of strings help a user to get a fancier Output
- User can do all the string handling by using string slicing and concatenation operations to create any layout that user wants. The string type has some methods that perform useful operations for padding strings to a given column width.

Formatting output using String modulo operator(`%`)

The `%` operator can also be used for string formatting. It interprets the left argument much like a `printf()`-style format string to be applied to the right argument.

Example

```
# string modulo operator(%) to print
# print integer and float value
print("Vishesh : % 2d, Portal : % 5.2f" %(1, 05.333))
# print integer value
print("Total students : % 3d, Boys : % 2d" %(240, 120))
# print octal value
print("% 7.3o"% (25))
# print exponential value
print("% 10.3E"% (356.08977))
```

Output

Vishesh : 1, Portal : 5.33

Total students : 240, Boys : 120

031

3.561E+02

Formatting output using format method:

The format() method was added in Python(2.6). Format method of strings requires more manual effort. User use {} to mark where a variable will be substituted and can provide detailed formatting directives, but user also needs to provide the information to be formatted.

Example

```
# show format () is used in dictionary
tab = {'Vishesh': 4127, 'for': 4098, 'python': 8637678}
# using format() in dictionary
print('Vishesh: {0[vishesh]:d}; For: {0[for]:d}; '
      'python: {0[python]:d}'.format(tab))
data = dict(fun = "VisheshforPython", adj = "Python")
# using format() in dictionary
print("I love {fun} computer {adj}".format(**data))
```

Formatting output using String method:

In this output is formatted by using string slicing and concatenation operations.

Example

```
# format a output using string() method
cstr = "I love python"
# Printing the center aligned
# string with fillchr
print ("Center aligned string with fillchr: ")
print (cstr.center(40, '$'))
# Printing the left aligned string with "-" padding
print ("The left aligned string is : ")
print (cstr.ljust(40, '-'))
# Printing the right aligned string with "-" padding
print ("The right aligned string is : ")
print (cstr.rjust(40, '-'))
```

Output

Center aligned string with fillchr:

\$\$\$\$\$\$\$\$\$\$\$\$I love python\$\$\$\$\$\$\$\$\$\$\$\$

The left aligned string is :

I love python-----

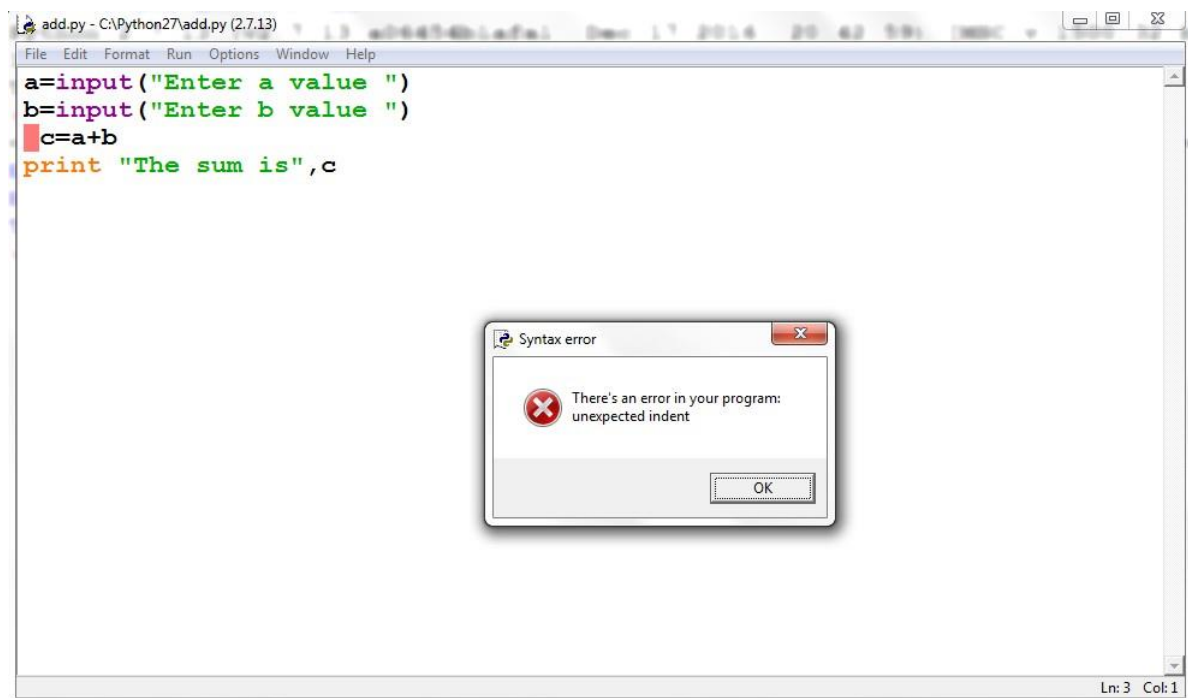
The right aligned string is :

-----I love python

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.



Comments:

Comments are lines that exist in computer programs that are ignored by compilers and interpreters. Including comments in programs makes code more readable for humans as it provides some information or explanation about what each part of a program is doing.

Depending on the purpose of your program, comments can serve as notes to yourself or reminders, or they can be written with the intention of other programmers being able to understand what your code is doing.

In general, it is a good idea to write comments while you are writing or updating a program as it is easy to forget your thought process later on, and comments written later may be less useful in the long term.

Comment Syntax

Comments in Python begin with a hash mark (#) and whitespace character and continue to the end of the

line.

Generally, comments will look something like this:

```
# This is a comment
```

Because comments do not execute, when you run a program you will not see any indication of the comment there. Comments are in the source code for humans to read, not for computers to execute.

In a “Hello, World!” program, a comment may look like this:

```
hello.py
# Print “Hello, World!” to console
print("Hello, World!")
```

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 big

Assignment 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=10
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 find 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:
    print " not available"
```

Output:

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

Output:

```
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
$a \times b - c$	<code>a * b - c</code>
$(m + n)(x + y)$	<code>(m + n) * (x + y)</code>
(ab / c)	<code>a * b / c</code>
$3x^2 + 2x + 1$	<code>3*x*x+2*x+1</code> or <code>3*x**2+2*x+1</code>
$(x / y) + c$	<code>x / y + c</code>

Evaluation of Expressions

Expressions are evaluated using an assignment statement of the form

Variable = expression

Variable is any valid 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
```

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 y 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:] * 2 # Prints string two times
print str + "CSE" # Prints concatenated string
print str[:]

```

Output:

```

WELCOME
W
LCO
LCOME
WELCOMEWELCOME
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	<code>translate(table, deletechars="")</code>	Translates string according to translation table str(256 chars), removing those in the del string.
38	<code>upper()</code>	Converts lowercase letters in string to uppercase.
39	<code>zfill (width)</code>	Returns original string leftpadded with zeros to a total of width characters; intended for numbers, <code>zfill()</code> retains any sign given (less one zero).
40	<code>isdecimal()</code>	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="77777777cse7777777";
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
rstrip function--- cse7777777
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.

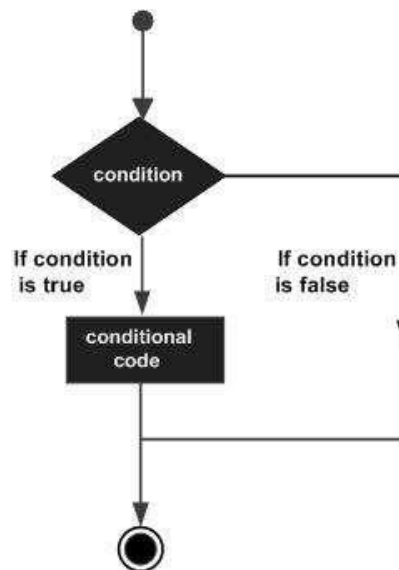
Function	Description
int(x [,base])	Converts x to an integer.
long(x [,base])	Converts x to a long integer.
float(x)	Converts x to a floating-point number.
complex(real [,imag])	Creates a complex number.
str(x)	Converts object x to a string representation.
repr(x)	Converts object x to an expression string.
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.

Decision Making:

Decision making is anticipation of conditions occurring while execution of the program and specifying actions taken according to the conditions.

Decision structures evaluate multiple expressions which produce True or False as outcome. You need to determine which action to take and which statements to execute if outcome is True or False otherwise.

Following is the general form of a typical decision making structure found in most of the programming languages:

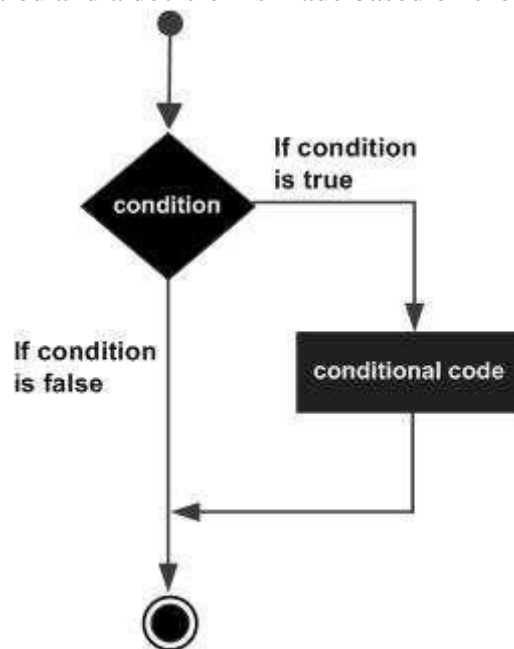


Python programming language assumes any non-zero and non-null values as True, and if it is either zero or null, then it is assumed as False value.

Statement	Description
if statements	if statement consists of a boolean expression followed by one or more statements.
if...else statements	if statement can be followed by an optional else statement , which executes when the boolean expression is FALSE.
nested if statements	You can use one if or else if statement inside another if or else if statement(s).

The *if* Statement

It is similar to that of other languages. The **if** statement contains a logical expression using which data is compared and a decision is made based on the result of the comparison.



Syntax:

```
if condition:
    statements
```

First, the condition is tested. If the condition is True, then the statements given after colon (:) are executed. We can write one or more statements after colon (:).

Example:

```
a=10
b=15
if a < b:
    print "B is big"
    print "B value is",b
```

Output:

```
B is big
B value is 15
```

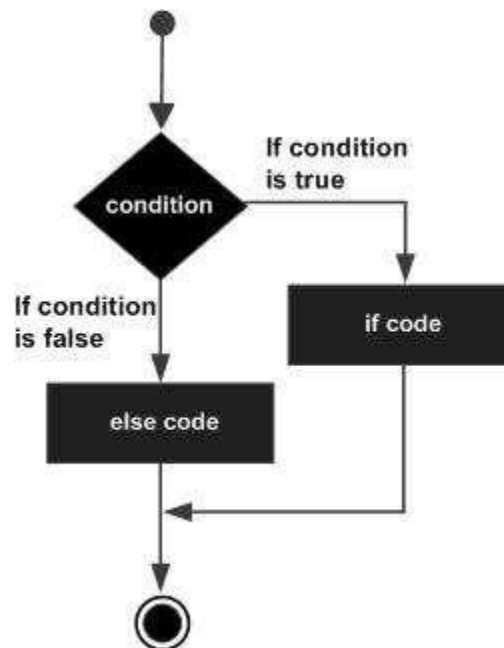
The *if ... else* statement

An **else** statement can be combined with an **if** statement. An **else** statement contains the block of code that executes if the conditional expression in the if statement resolves to 0 or a FALSE value.

The *else* statement is an optional statement and there could be at most only one **else** statement following **if**.

Syntax:

```
if condition:
    statement(s)
else:
    statement(s)
```

**Example:**

```

a=48
b=34
if a < b:
    print "B is big"
    print "B value is", b
else:
    print "A is big"
    print "A value is", a
print "END"
  
```

Output:

```

A is big
A value is 48
END
  
```

Q) Write a program for checking whether the given number is even or not. Program:

```

a=int(input("Enter a value: "))
if a%2==0:
    print "a is EVEN number"
else:
    print "a is NOT EVEN Number"
  
```

Output-1:

```

Enter a value: 56
a is EVEN Number
  
```

Output-2:

```

Enter a value: 27
a is NOT EVEN Number
  
```


The *elif* Statement

The **elif** statement allows you to check multiple expressions for True and execute a block of code as soon as one of the conditions evaluates to True.

Similar to the **else**, the **elif** statement is optional. However, unlike **else**, for which there can be at most one statement, there can be an arbitrary number of **elif** statements following an **if**.

Syntax:

```
if condition1:
    statement(s)
elif condition2:
    statement(s)
else:
    statement(s)
```

Example:

```
a=20
b=10
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

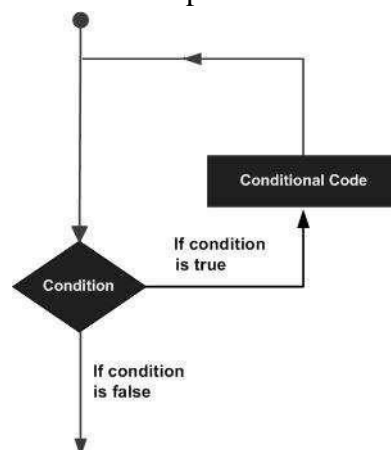
```
#example for nested if
a=20
b=10
c=30
if a >= b:
    if a >= c:
        print("a is big")
    else:
        print("c is big")
else:
    if b >= c:
        print("b is big")
    else:
        print("c is big")
```

Decision Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths.

A loop statement allows us to execute a statement or group of statements for multiple times. The following diagram illustrates a loop statement:



Python programming language provides following types of loops to handle looping requirements.

Loop Type	Description
while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
for loop	Executes a sequence of statements multiple times and abbreviates the code that manages the loop variable.
nested loops	You can use one or more loop inside any another while, for loop.

The *while* Loop

A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is True.

Syntax

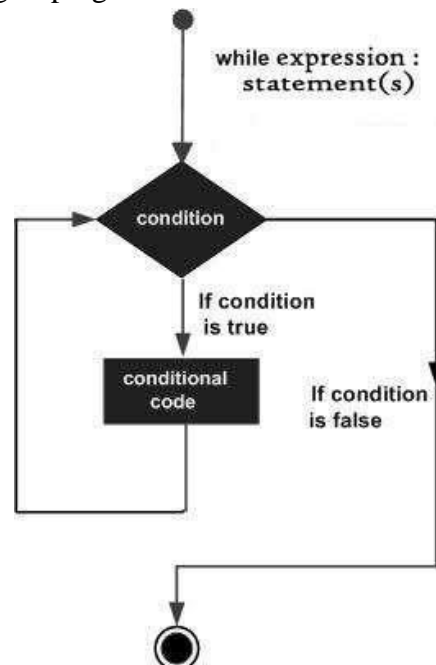
The syntax of a **while** loop in Python programming language is:

```
while expression:  
    statement(s)
```

Here, **statement(s)** may be a single statement or a block of statements.

The **condition** may be any expression, and true is any non-zero value. The loop iterates while the condition is true. When the condition becomes false, program control passes to the line immediately following the loop.

In Python, all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements.



Example-1:

```
i=1
while i < 4:
    print i
    i+=1
print "END"
```

Example-2:

```
i=1
while i < 4:
    print i
    i+=1
print "END"
```

Output-1:

```
1
END
2
END
3
END
```

Output-2:

```
1
2
3
END
```

Q) Write a program to display factorial of a given number.

Program:

```
n=int(input("Enter the number: "))
f=1
while n>0:
    f=f*n
    n=n-1
print "Factorial is",f
```

Output:

```
Enter the number: 5
Factorial is 120
```

The for loop:

The *for* loop is useful to iterate over the elements of a sequence. It means, the *for* loop can be used to execute a group of statements repeatedly depending upon the number of elements in the sequence. The *for* loop can work with sequence like string, list, tuple, range etc.

The syntax of the for loop is given below:

```
for var in sequence:
    statement (s)
```

The first element of the sequence is assigned to the variable written after „for“ and then the statements are executed. Next, the second element of the sequence is assigned to the variable and then the statements are executed second time. In this way, for each element of the sequence, the statements are executed once. So, the *for* loop is executed as many times as there are number of elements in the sequence.

Example-1:

```
for i in range(1,5):
    print i
    print "END"
```

Output-1:

```
1
END
2
END
3
END
```

Example-2:

```
for i in range(1,5):
    print i
    print "END"
```

Output-2:

```
1
2
3
END
```

Example-3:

```
name= "python"
for letter in name:
    print letter
```

Output-3:

```
p
y
t
h
o
n
```

Example-4:

```
for x in range(10,0,-1):
    print x,
```

Output-4:

```
10 9 8 7 6 5 4 3 2 1
```

Q) Write a program to display the factorial of given number.

Program:

```
n=input("Enter the number: ")
f=1
for i in range(1,n+1):
    f=f*i
print "Factorial is",f
```

Output:

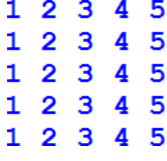
```
Enter the number: 5
Factorial is 120
```

Nested Loop:

It is possible to write one loop inside another loop. For example, we can write a for loop inside a while loop or a for loop inside another for loop. Such loops are called “nested loops”.


Example-1:

```
for i in range(1,6):
    for j in range(1,6):
        print j,
    print ""
```




Example-2:

```
for i in range(1,6):
    for j in range(1,6):
        print "*",
    print ""
```




Example-3:

```
for i in range(1,6):
    for j in range(1,6):
        if i==1 or j==1 or i==5 or j==5:
            print "*",
        else:
            print " ",
    print ""
```



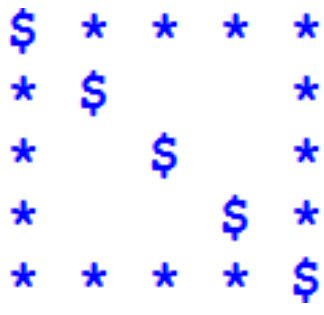
Example-4:

```
for i in range(1,6):
    for j in range(1,6):
        if i==j:
            print "*",
        elif i==1 or j==1 or i==5 or j==5:
            print "*",
        else:
            print " ",
    print ""
```



Example-5:

```
for i in range(1,6):
    for j in range(1,6):
        if i==j:
            print "$",
        elif i==1 or j==1 or i==5 or j==5:
            print "*",
        else:
            print " ",
    print ""
```



Example-6:

```

for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or i==5:
            print "*",
        else:
            print " ",
    print ""

```

```

* * *
*
*
*
* * *

```

Example-7:

```

for i in range(1,6):
    for j in range(1,4):
        if i==2 and j==1:
            print "*",
        elif i==4 and j==3:
            print "*",
        elif i==1 or i==3 or i==5:
            print "*",
        else:
            print " ",
    print ""

```

```

* * *
*
* * *
      *
* * *

```

Example-8:

```

for i in range(1,6):
    for j in range(1,4):
        if i==1 or j==1 or i==3 or i==5:
            print "*",
        else:
            print " ",
    print ""

```

```

* * *
*
* * *
*
* * *

```

Example-9:

```

for i in range(1,6):
    for c in range(i,6):
        print "",
    for j in range(1,i+1):
        print "*",
    print ""

```

```

      *
    * *
  * * *
* * * *

```

Example-10:

```

for i in range(1,6):
    for j in range(1,i+1):
        print j,
    print ""

```

```

1
1 2
1 2 3
1 2 3 4
1 2 3 4 5

```

Example-11:

```

a=1
for i in range(1,5):
    for j in range(1,i+1):
        print a,
        a=a+1
    print ""

```

1
2 3
4 5 6
7 8 9 10

1) Write a program for print given number is prime number or not using for loop.

Program:

```

n=input("Enter the n value")
count=0
for i in range(2,n):
    if n%i==0:
        count=count+1
        break
if count==0:
    print "Prime Number"
else:
    print "Not Prime Number"

```

Output:

```

Enter n value: 17
Prime Number

```

1) Write a program print Fibonacci series and sum the even numbers. Fibonacci series is 1,2,3,5,8,13,21,34,55

```

n=input("Enter n value ")
f0=1
f1=2
sum=f1
print f0,f1,
for i in range(1,n-1):
    f2=f0+f1
    print f2,
    f0=f1
    f1=f2
    if f2%2==0:
        sum+=f2
print "\nThe sum of even Fibonacci numbers is", sum

```

Output:

```

Enter n value 10
1 2 3 5 8 13 21 34 55 89
The sum of even fibonacci numbers is 44

```


2) Write a program to print n prime numbers and display the sum of prime numbers.

Program:

```
n=input("Enter the range: ")
sum=0
for num in range(1,n+1):
    for i in range(2,num):
        if (num % i) == 0:
            break
    else:
        print num,
        sum += num
print "\nSum of prime numbers is",sum
```

Output:

```
Enter the range: 21
1 2 3 5 7 11 13 17 19
Sum of prime numbers is 78
```

3) Using a for loop, write a program that prints out the decimal equivalents of $1/2$, $1/3$, $1/4$, ..., $1/10$

Program:

```
for i in range(1,11):
    print "Decimal Equivalent of 1/" ,i,"is",1/float(i)
```

Output:

```
Decimal Equivalent of 1/ 1 is 1.0
Decimal Equivalent of 1/ 2 is 0.5
Decimal Equivalent of 1/ 3 is 0.333333333333
Decimal Equivalent of 1/ 4 is 0.25
Decimal Equivalent of 1/ 5 is 0.2
Decimal Equivalent of 1/ 6 is 0.166666666667
Decimal Equivalent of 1/ 7 is 0.142857142857
Decimal Equivalent of 1/ 8 is 0.125
Decimal Equivalent of 1/ 9 is 0.111111111111
Decimal Equivalent of 1/ 10 is 0.1
```

4) Write a program that takes input from the user until the user enters -1. After display the sum of numbers.

Program:

```
sum=0
while True:
    n=input("Enter the number: ")
    if n== -1:
        break
    else:
        sum+=n
print "The sum is",sum
```

Output:

```
Enter the number: 1
Enter the number: 5
Enter the number: 6
Enter the number: 7
Enter the number: 8
Enter the number: 1
Enter the number: 5
Enter the number: -1
The sum is 33
```

5) Write a program to display the following sequence.

A B C D E F G H I J K L M N O P Q R S T U V W X Y Z

Program:

```
ch='A'
for j in range(1,27):
    print ch,
    ch=chr(ord(ch)+1)
```

6) Write a program to display the following sequence.

```
A
A B
A B C
A B C D
```

Program:

```
for i in range(1,6):
    ch='A'
    for j in range(1,i+1):
        print ch,
        ch=chr(ord(ch)+1)
    print ""
```

7) Write a program to display the following sequence.

A
B C
D E F
G H I J
K L M N O

Program:

```
ch='A'
for i in range(1,6):
    for j in range(1,i+1):
        print ch,
        ch=chr(ord(ch)+1)
    print ""
```

8) Write a program that takes input string user and display that string if string contains at least one Uppercase character, one Lowercase character and one digit.

Program:

```
pwd=input("Enter the password:")
u=False
l=False
d=False
for i in range(0,len(pwd)):
    if pwd[i].isupper():
        u=True
    elif pwd[i].islower():
        l=True
    elif pwd[i].isdigit():
        d=True
if u==True and l==True and d==True:
    print pwd.center(20,"*")
else:
    print "Invalid Password"
```

Output-1:

```
Enter the password:"Mothi556"
*****Mothi556*****
```

Output-2:

```
Enter the password:"mothilal"
Invalid Password
```

9) Write a program to print sum of digits.

Program:

```
n=input("Enter the number: ")
sum=0
while n>0:
    r=n%10
    sum+=r
    n=n/10
print "sum is",sum
```

Output:

```
Enter the number:
123456789sum is 45
```

10) Write a program to print given number is Armstrong or not.

Program:

```
n=input("Enter the number: ")
sum=0
t=n
while n>0:
    r=n%10
    sum+=r*r*r
    n=n/10
if sum==t:
    print "ARMSTRONG"
else:
    print "NOT ARMSTRONG"
```

Output:

```
Enter the number:
153ARMSTRONG
```

11) Write a program to take input string from the user and print that string after removing ovals.

Program:

```
st=input("Enter the string:")
st2=""
for i in st:
    if i not in "aeiouAEIOU":
        st2=st2+i
print st2
```

Output:

```
Enter the string:"Welcome to
you"Wlcm t y
```

Module:

A module allows you to logically organize your Python code. Grouping related code into a module makes the code easier to understand and use. A module is a Python object with arbitrarily named attributes that you can bind and reference.

Simply, a module is a file consisting of Python code. A module can define functions, classes and variables. A module can also include runnable code.

Example

The Python code for a module named aname normally resides in a file named aname.py. Here's an example of a simple module, support.py

```
def print_func( par ):
    print "Hello : ", par
    return
```

The import Statement

You can use any Python source file as a module by executing an import statement in some other Python source file. The import has the following syntax –

```
import module1[, module2[,... moduleN]
```

When the interpreter encounters an import statement, it imports the module if the module is present in the search path. A search path is a list of directories that the interpreter searches before importing a module. For example, to import the module support.py, you need to put the following command at the top of the script –

```
#!/usr/bin/python
# Import module support
import support
# Now you can call defined function that module as follows
support.print_func("Zara")
```

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

Hello : Zara

Functions:

In Python, a function is a group of related statements that performs a specific task.

Functions help break our program into smaller and modular chunks. As our program grows larger and larger, functions make it more organized and manageable.

Furthermore, it avoids repetition and makes the code reusable.

Syntax of Function

```
def function_name(parameters):
    """docstring"""
    statement(s)
```

Above shown is a function definition that consists of the following components.

1. Keyword **def** that marks the start of the function header.
2. A function name to uniquely identify the function. Function naming follows the same rules of writing identifiers in Python.
3. Parameters (arguments) through which we pass values to a function. They are optional.
4. A colon (:) to mark the end of the function header.

5. Optional documentation string (docstring) to describe what the function does.
6. One or more valid python statements that make up the function body. Statements must have the same indentation level (usually 4 spaces).
7. An optional **return** statement to return a value from the function.

Example

```
def greet(name):  
    """  
    This function greets to  
    the person passed in as  
    a parameter  
    """  
    print("Hello, " + name + ". Good morning!")
```

How to call a function in python?

Once we have defined a function, we can call it from another function, program or even the Python prompt. To call a function we simply type the function name with appropriate parameters.

```
>>> greet('Paul')  
Hello, Paul. Good morning!
```

The return statement

The return statement is used to exit a function and go back to the place from where it was called.

Syntax of return

```
return [expression_list]
```

This statement can contain an expression that gets evaluated and the value is returned. If there is no expression in the statement or the return statement itself is not present inside a function, then the function will return the None object.

For example:

```
>>> print(greet("May"))  
Hello, May. Good morning!  
None
```

Here, None is the returned value since greet() directly prints the name and no return statement is used.

Example of return

```
def absolute_value(num):  
    """This function returns the absolute  
    value of the entered number"""
```

```
if num >= 0:
    return num
else:
    return -num

print(absolute_value(2))

print(absolute_value(-4))
```

Output

```
2
4
```