

Unit – 1

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 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.

Apart from the above-mentioned features, Python has a big list of good features, few are listed below –

- It supports functional and structured programming methods as well as OOP.
- It can be used as a scripting language or can be compiled to byte-code for building large applications.
- It provides very high-level dynamic data types and supports dynamic type checking.
- It supports automatic garbage collection.
- It can be easily integrated with C, C++, COM, ActiveX, CORBA, and Java.

Python Integrated Development Environment (IDE)

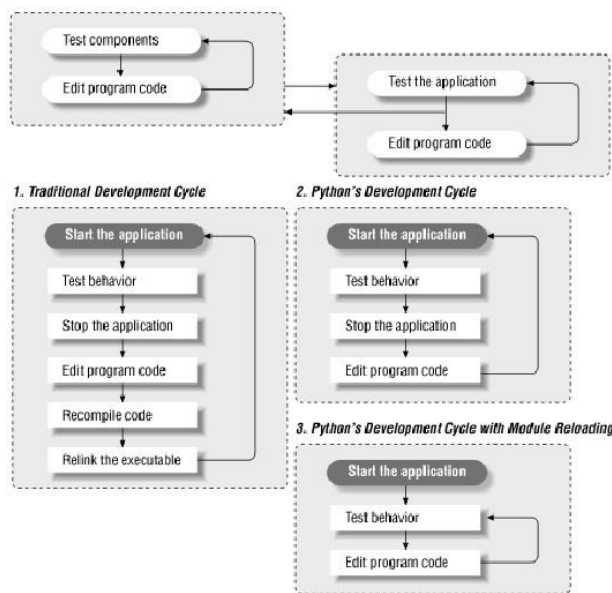
You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- **Unix** – IDLE is the very first Unix IDE for Python.
- **Windows** – PythonWin is the first Windows interface for Python and is an IDE with a GUI.

- **Macintosh** – The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

Python 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.



First Python Program

Let us execute programs in different modes of programming.

Interactive Mode Programming

Invoking the interpreter without passing a script file as a parameter brings up the following prompt –

```
$ python
Python 2.4.3 (#1, Nov 11 2010, 13:34:43)
[GCC 4.1.2 20080704 (Red Hat 4.1.2-48)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

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

If you are running new version of Python, then you would need to use print statement with parenthesis as in **print ("Hello, Python!")**; However in Python version 2.4.3, this produces the following result –

```
Hello, Python!
```

Script Mode Programming

Invoking the interpreter with a script parameter begins execution of the script and continues until the script is finished. When the script is finished, the interpreter is no longer active.

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

[Live Demo](#)

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

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

```
$ python test.py
```

This produces the following result –

```
Hello, Python!
```

Python Identifiers

A Python identifier is a name used to identify a variable, function, class, module or other object. An identifier starts with a letter A to Z or a to z or an underscore (_) followed by zero or more letters, underscores and digits (0 to 9).

Python does not allow punctuation characters such as @, \$, and % within identifiers. Python is a case sensitive programming language. Thus, **Manpower** and **manpower** are two different identifiers in Python.

Here are naming conventions for Python identifiers –

- Class names start with an uppercase letter. All other identifiers start with a lowercase letter.
- Starting an identifier with a single leading underscore indicates that the identifier is private.
- Starting an identifier with two leading underscores indicates a strongly private identifier.
- If the identifier also ends with two trailing underscores, the identifier is a language-defined special name.

Reserved Words

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

and	exec	not
assert	finally	or
break	for	pass
class	from	print
continue	global	raise
def	if	return

del	import	try
elif	in	while
else	is	with
except	lambda	yield

Quotation in Python

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

The triple quotes are used to span the string across multiple lines. For example, all the following are legal –

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

Comments in Python

A hash sign (#) that is not inside a string literal begins a comment. All characters after the # and up to the end of the physical line are part of the comment and the Python interpreter ignores them.

[Live Demo](#)

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

```
# First comment
print "Hello, Python!" # second comment
This produces the following result –
Hello, Python!
```

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

```
name = "Madisetti" # This is again comment
```

You can comment multiple lines as follows –

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

Following triple-quoted string is also ignored by Python interpreter and can be used as a multiline comments:

```
"""
This is a multiline
comment.
"""
```

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 –

[Live Demo](#)

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

```
counter = 100      # An integer assignment
miles   = 1000.0    # A floating point
name    = "John"    # A string
```

```
print counter
print miles
print name
```

Here, 100, 1000.0 and "John" are the values assigned to *counter*, *miles*, and *name* variables, respectively. This produces the following result –

```
100
1000.0
John
```

Multiple Assignment

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,"john"
```

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.

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
- List
- Tuple
- Dictionary

Python Numbers

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

```
var1 = 1
var2 = 10
```

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is –

```
del var1[,var2[,var3[....,varN]]]
```

You can delete a single object or multiple objects by using the del statement. For example –

```
del var
del var_a, var_b
```

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)

Examples

Here are some examples of numbers –

int	long	float	complex
10	51924361L	0.0	3.14j
100	-0x19323L	15.20	45.j
-786	0122L	-21.9	9.322e-36j
080	0xDEFABCECBDAECBFBAEI	32.3+e18	.876j
-0490	535633629843L	-90.	-.6545+0J
-0x260	-052318172735L	-32.54e100	3e+26J
0x69	-4721885298529L	70.2-E12	4.53e-7j

- 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 and y are the real numbers and j is the imaginary unit.

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 –

[Live Demo](#)

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

```
str = 'Hello World!'
```

```
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 + "TEST" # Prints concatenated string
```

This will produce the following result –

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

Python Lists

Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]). To some extent, lists are similar to arrays in C. One difference between them is that all the items belonging to a list can be of different data type. The values stored in a list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1. The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator. For example –

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

```
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tinylist = [123, 'john']
```

```
print list      # Prints complete list
print list[0]   # Prints first element of the list
print list[1:3] # Prints elements starting from 2nd till 3rd
print list[2:]  # Prints elements starting from 3rd element
print tinylist * 2 # Prints list two times
print list + tinylist # Prints concatenated lists
```

This produce the following result –

```
['abcd', 786, 2.23, 'john', 70.2]
abcd
[786, 2.23]
[2.23, 'john', 70.2]
[123, 'john', 123, 'john']
['abcd', 786, 2.23, 'john', 70.2, 123, 'john']
```

Python Tuples

A tuple is another sequence data type that is similar to the list. A tuple consists of a number of values separated by commas. Unlike lists, however, tuples are enclosed within parentheses. The main differences between lists and tuples are: Lists are enclosed in brackets ([]) and their elements and size can be changed, while tuples are enclosed in parentheses (()) and cannot be updated. Tuples can be thought of as **read-only** lists. For example –

[Live Demo](#)

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

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
tinytuple = (123, 'john')
```

```
print tuple      # Prints complete list
print tuple[0]   # Prints first element of the list
print tuple[1:3] # Prints elements starting from 2nd till 3rd
print tuple[2:]  # Prints elements starting from 3rd element
print tinytuple * 2 # Prints list two times
print tuple + tinytuple # Prints concatenated lists
```

This produce the following result –

```
('abcd', 786, 2.23, 'john', 70.2)
abcd
(786, 2.23)
(2.23, 'john', 70.2)
(123, 'john', 123, 'john')
('abcd', 786, 2.23, 'john', 70.2, 123, 'john')
```

The following code is invalid with tuple, because we attempted to update a tuple, which is not allowed. Similar case is possible with lists –

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

```
tuple = ( 'abcd', 786 , 2.23, 'john', 70.2 )
list = [ 'abcd', 786 , 2.23, 'john', 70.2 ]
tuple[2] = 1000    # Invalid syntax with tuple
list[2] = 1000     # Valid syntax with list
```

Python Dictionary

Python's dictionaries are kind of hash table type. They work like associative arrays or hashes found in Perl and consist of key-value pairs. A dictionary key can be almost any Python type, but are usually numbers or strings. Values, on the other hand, can be any arbitrary Python object. Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]). For example –

[Live Demo](#)

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

```
dict = {}
dict['one'] = "This is one"
dict[2]    = "This is two"
```

```
tinydict = {'name': 'john','code':6734, 'dept': 'sales'}
```

```
print dict['one']    # Prints value for 'one' key
print dict[2]        # Prints value for 2 key
print tinydict       # Prints complete dictionary
print tinydict.keys() # Prints all the keys
print tinydict.values() # Prints all the values
```

This produce the following result –

```
This is one
This is two
{'dept': 'sales', 'code': 6734, 'name': 'john'}
['dept', 'code', 'name']
['sales', 6734, 'john']
```

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

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.

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.

Sr.No.	Function & Description
--------	------------------------

- | | |
|---|--|
| 1 | int(x [,base])
Converts x to an integer. base specifies the base if x is a string. |
| 2 | long(x [,base])
Converts x to a long integer. base specifies the base if x is a string. |
| 3 | float(x)
Converts x to a floating-point number. |

- 4 **complex(real [,imag])**
 Creates a complex number.
- 5 **str(x)**
 Converts object x to a string representation.
- 6 **repr(x)**
 Converts object x to an expression string.
- 7 **eval(str)**
 Evaluates a string and returns an object.
- 8 **tuple(s)**
 Converts s to a tuple.
- 9 **list(s)**
 Converts s to a list.
- 10 **set(s)**
 Converts s to a set.
- 11 **dict(d)**
 Creates a dictionary. d must be a sequence of (key,value) tuples.
- 12 **frozenset(s)**
 Converts s to a frozen set.
- 13 **chr(x)**
 Converts an integer to a character.
- 14 **unichr(x)**
 Converts an integer to a Unicode character.
- 15 **ord(x)**
 Converts a single character to its integer value.
- 16 **hex(x)**
 Converts an integer to a hexadecimal string.
- 17 **oct(x)**
 Converts an integer to an octal string.

Operators are the constructs which can manipulate the value of operands.

Consider the expression $4 + 5 = 9$. Here, 4 and 5 are called operands and + is called operator.

Types of Operator

Python language supports the following types of operators.

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

Let us have a look on all operators one by one.

Python Arithmetic Operators

Assume variable a holds 10 and variable b holds 20, then –

[[Show Example](#)]

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	$a ** b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity) –	$9 // 2 = 4$ and $9.0 // 2.0 = 4.0$, $-11 // 3 = -4$, $-11.0 // 3 = -4.0$

Python Comparison Operators

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

Assume variable a holds 10 and variable b holds 20, then –

[[Show Example](#)]

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 `(a <= b)` is true. condition becomes true.

Python Assignment Operators

Assume variable a holds 10 and variable b holds 20, then –

[[Show Example](#)]

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

Python Bitwise Operators

Bitwise operator works on bits and performs bit by bit operation. Assume if `a = 60`; and `b = 13`; Now in binary format they will be as follows –

`a = 0011 1100`

`b = 0000 1101`

`a&b = 0000 1100`

`a|b = 0011 1101`

`a^b = 0011 0001`

`~a = 1100 0011`

There are following Bitwise operators supported by Python language

[[Show Example](#)]

Operator	Description	Example
<code>&</code> Binary AND	Operator copies a bit to the result if it exists in both operands	<code>(a & b)</code> (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.	$(a \wedge b) = 49$ (means 0011 0001)
~ Binary Ones Complement	It is unary and has the effect of 'flipping' bits.	$(\sim a) = -61$ (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.	$a \ll 2 = 240$ (means 1111 0000)
>> Binary Right Shift	The left operands value is moved right by the number of bits specified by the right operand.	$a \gg 2 = 15$ (means 0000 1111)

Python Logical Operators

There are following logical operators supported by Python language. Assume variable a holds 10 and variable b holds 20 then

[[Show Example](#)]

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

Used to reverse the logical state of its operand.

Python Membership Operators

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

[[Show Example](#)]

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	$x \text{ in } y$, 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.	$x \text{ not in } y$, here not in results in a 1 if x is not a member of sequence y.

Python Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators explained below –

[[Show Example](#)]

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).

Python Operators Precedence

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

[[Show Example](#)]

Sr.No.	Operator & Description
1	** Exponentiation (raise to the power)
2	~ + - Complement, unary plus and minus (method names for the last two are +@ and -@)
3	* / % // Multiply, divide, modulo and floor division
4	+ - Addition and subtraction
5	>> << Right and left bitwise shift
6	& Bitwise 'AND'
7	^ Bitwise exclusive 'OR' and regular 'OR'
8	<= < > >= Comparison operators
9	<> == != Equality operators
10	= %= /= //= -= += *= **= Assignment operators
11	is is not Identity operators
12	in not in Membership operators
13	not or and Logical operators