PYTHON FUNDAMENTALS

PROGRAM: A program is a set of instructions that govern the processing. It forms the base for processing.

 # PYTHON CHARACTER SET: Character set is a set of valid characters that a language can recognize.

- Python supports Unicode encoding Standard.
- [letters, Digits, special symbols, whitespaces, other characters]

 # TOKENS : The smallest individual unit in a program is known as Token or a lexical unit/element.

- Python has following Tokens:

(i) Keywords (ii) Identifiers (Names) (iii) Literals
(iv) Operators (v) Punctuators

 # KEYWORDS : Keywords are the words that convey a special meaning to the language compiler/interpreter.

- These are reserved for special purpose and must not be used as normal identifier names.

 # Identifiers (Names): These are fundamental building blocks of a program and are used as the general terminology for the names given to different parts of the program.

RULES :

i) The first character must be a letter, underscore (_) counts as a letter.
ii) The digits 0 through 9 can be part of identifier except for the first character.
iii) Python is case sensitive as it treats upper and lowercase characters differently.
iv) An identifier must not be a keyword of python.
(v) An Identifier cannot contain any special character except for underscore (_).

LITERALS / VALUES :- often referred to as constant-value are data items that have a fixed value. Python allow several kind of literals.

(i) String Literals (ii) Numeric Literals (iii) Boolean literals (iv) Special literal None (v) Literal collections.

STRING LITERALS : The text enclosed in quotes form a string literal in Python.

- One can form string literals by enclosing text in both form of quotes, — single or double quotes.

- Python allows us to have certain 'non-graphic-characters' in string values.
- Nongraphic characters are those characters that cannot be typed directly from keyboard, e.g. backspace, tabs, carriage return etc.
- These non-graphic characters can be represented by using escape sequences. An escape sequence is represented by a backslash (\).

⇒ STRING TYPES IN PYTHON → (i) Single-line strings (' ', " ")
(ii) Multiline strings.

Multiline strings can be created in two ways :

(a) By adding a backslash (\) at the end of normal single quote / double quote strings.

(b) By typing the text in triple quotation marks (no (\) needed).

⇒ Size of strings : Python determines the size of a string as the count of characters in the string.

If the string literal has an escape sequence contained within it then we have to count the escape sequence as one character.

For multiline strings created with triple quotes, (EOL) character at the end of the line is also counted in the size.

- To check the size of a string, → len (<String_name>)

[24] + (b) - (ii)

NUMERIC LITERALS

→ int → often called just integers or ints, are +ve, -ve or unlimited
→ float → represent real no. and are written with a decimal point.
→ Complex → are the form a+bJ where a & b are floats J → √-1.

- Python allows 3 types of integer literals:
  (i) Decimal Integer Literals
  (ii) Octal Integer Literals (starting with 0o)
  (iii) Hexadecimal Integer Literals (starting with 0x)

- Floating Point Literals
  (i) Fractional Form (ii) Exponent Form

- Exponent Form: A real literal in Exponent form consists of two parts: mantissa and exponent. 0.58E01 — Exponent, ↳ mantissa

⇒ BOOLEAN LITERALS

A boolean literal in python is used to represent one of the two Boolean values i.e, TRUE or False.

⇒ Special Literal NONE: The None literal is used to indicate absence of value.

OPERATORS: Operators are tokens that trigger some computation when applied to variables and other objects in an expression.

→ Unary operators →	Binary operators: Arithmetic operators
unary plus +	+ - Addition
unary minus -	- - subtraction
bitwise complement ~	* - multiplication
logical negation not	/ - Division
	% - Remainder/Modulus
	** - exponent
	// - Floor Division.

- Bitwise Operators:
  - & - Bitwise AND
  - ^ - Bitwise exclusive OR (XOR)
  - | - Bitwise OR
- SHIFT OPERATORS
  - << - shift left
  - >> - shift Right
- Identify operators
  - is
  - is not
- Relational Operators
  - <, >, <=, >=, ==, !=
- Assignment Operators:
  - = assignment
  - += A Sum
  - /= A quotient
  - *= A Product
  - %= AR
  - **= Assign Exponent
  - -= AD
  - //= Assign Floor division.
- Logical operators:
  - and
  - or
- Membership operators:
  - in
  - not in

PUNCTUATORS: These are symbols that are used in programming languages to organize sentence structures and indicate rhythm.
Most common Punctuators of Python programming are:
' " # \ ( ) [ ] { } @ , : . =

BAREBONES OF A PYTHON PROGRAM:
(i) EXPRESSIONS: An expression is any legal combination of symbols that represents a value.

ii) STATEMENT: A statement is a programming instruction that does something i.e., some action takes place.

iii) Comments are the additional readable information to clarify the source code.

Multiline # Comments : ~~#~~ i)

(iii) """ . . . . . """
(docstring)

FUNCTIONS : A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.

BLOCKS AND INDENTATION : A group of statements which are part of another ~~statement~~ or a function are called block or code-block or suite in Python.

VARIABLES AND ASSIGNMENTS

- A variable in Python represents named location that refers to a value and whose values can be used and processed during program run.

- Python variables are created by assigning value of desired type to them, e.g. to create a numeric variable, assign a numeric value

- Variables are not storage containers in Python.

- Lvalues: They are the objects to which we can assign a value or expression. Lvalues can come on lhs or rhs of an assignment statement.

- Rvalues are the literals and expressions that are assigned to Lvalues. Rvalues can come on rhs of an assignment statement.

MULTIPLE ASSIGNMENTS :

One can assign different or multiple variable to same vale or multiple values to multiple variable.

Examples: a = b = c = 20

x, y = 10, 20, 30

x, y = y, x (swap)

a, b = 5, 10

b, c = a+1, b+2

print (a, b, c)

5, 6, 12

- Variable is not created until some value is assigned to it. If not assigned, it causes an error called Name Error.

Dynamic Typing: A variable pointing to a value of a certain type, can be made to point to a value/object of different type.

- To know the type of variable, we can use type(<variable name>)

Simple INPUT and OUTPUT

- The input() function always returns a value of string type.
- When we try to perform an operation on a data type not suitable for it, Python will raise error called TypeError
- For output, print( ) function is used.

7.2 DATA TYPES

Data can be of many types *e.g., character, integer, real, string* etc. Anything enclosed in quotes represents string data in Python. Numbers without fractions represent integer data. Numbers with fractions represent real data and *True* and *False* represent Boolean data. Since the data to be dealt with are of many types, a programming language must provide ways and facilities to handle all types of data.

Built-in Core Data Types

- Numbers (*int, float, complex*)
- String
- List
- Tuple
- Dictionary

Before you learn how you can process different types of data in Python, let us discuss various data-types supported in Python. In this discussion of data types, you'll be able to know Python's capabilities to handle a specific type of data, such as the memory space it allocates to hold a certain type of data and the range of values supported for a data type etc.

Python offers following built-in core data types : (*i*) *Numbers* (*ii*) *String* (*iii*) *List* (*iv*) *Tuple* (*v*) *Dictionary*.

Complex Numbers in Python

Python represents complex numbers in the form $A + B\,j$. That is, to represent imaginary number, Python uses j (or J) in place of traditional i. So in Python $j = \sqrt{-1}$. Consider the following examples where a and b are storing two complex numbers in Python :

```
a = 0 + 3.1j
b = 1.5 + 2j
```

The above complex number ***a*** has real component as **0** and imaginary component as **3.1** ; in complex number ***b***, the real part is **1.5** and imaginary part is **2**. When you display complex numbers, Python **displays complex numbers in parentheses when they have a nonzero real part** as shown in following examples.

NOTE

Python represents complex numbers as a pair of floating point numbers.

```
>>> c = 0 + 4.5j
>>> d = 1.1 + 3.4j
>>> c
4.5j
>>> d
(1.1 + 3.4j)
>>> print (c)
4.5j
>>> print (d)
(1.1 + 3.4j)
```

See, a complex number with non-zero real part is displayed with parentheses around it.

But no parentheses around complex number with real part as zero(0).

NOTE

Complex numbers are quite commonly used in *Electrical Engineering*. In the field of electricity, however, because the symbol ***i*** is used to represent current, they use the symbol *j* for the square root of – 1. Python adheres to this convention.

Unlike Python's other numeric types, complex numbers are a composite quantity made of two parts : *the real part* and the *imaginary part*, both of which are represented internally as ***float*** values (floating point numbers).

You can retrieve the two components using attribute references. For a complex number z :

- **z.real** gives the *real part*.
- **z.imag** gives the *imaginary part* as a float, not as a complex value.

For example,

```
>>> z = (1 + 2.56j) + (-4 -3.56j)
>>> a
(-3 -1j)
>>> z.real
-3.0
>>> z.imag
-1.0
```

It will display real part of complex number z

It will display imaginary part of complex number z

TIP

The real and imaginary parts of a complex number z can be retrieved through the read-only attributes ***z.real*** and ***z.imag***.

The range of numbers represented through Python's numeric data types is given below.

Table 7.1 *The Range of Python Numbers*

Data type	Range
Integers	an unlimited range, subject to available (virtual) memory only
Booleans	*two* values **True** (1), **False** (0)
Floating point numbers	an unlimited range, subject to available (virtual) memory on underlying machine architecture.
Complex numbers	Same as floating point numbers because the real and imaginary parts are represented as **floats**.

Check Point 7.2

1. What are floating point numbers ? When are they preferred over integers ?
2. What are complex numbers ? How would Python represent a complex number with real part as **3** and imaginary part as – **2.5** ?
3. What will be the output of following code ?

```
p = 3j
q = p + (1 + 1.5j)
print (p)
print (q)
```

4. What will be the output of following code ?

```
r = 2.5 + 3.9j
print (r.real)
print (r.imag)
```

5. Why does Python uses symbol *j* to represent imaginary part of a complex number instead of the conventional *i* ?

Hint. Refer note above.

7.2.2 Strings

You already know about strings (as data) in Python. In this section, we shall be talking about Python's data type string. A string data type lets you hold string data, *i.e.*, any number of valid characters into a set of quotation marks.

In Python 3.x, each character stored in a string[3] is a Unicode character. Or in other words, all strings in Python 3.x are sequences of *pure Unicode characters*. *Unicode* is a system designed to represent every character from every language. A string can hold any type of known characters *i.e.*, *letters*, *numbers*, and *special characters*, of any known scripted language.

NOTE

All Python (3.x) strings store Unicode characters.

Following are all legal strings in Python :

"abcd" , "1234", '$%^&', '????', "ŠÆËá" , "??????", '????', "????", '???', "??"

3. Python has no separate **character** datatype, which most other programming languages have – that can hold a single character. In Python, a character is a **string** type only, with single character.

String as a Sequence of Characters

A Python string is a sequence of characters and each character can be individually accessed using its **index**. Let us understand this.

Let us first study the internal structure or composition of Python strings as it will form the basis of all the learning of various string manipulation concepts. Strings in Python are stored as individual characters in contiguous location, with two-way index for each location.

The individual elements of a string are the characters contained in it (stored in contiguous memory locations) and as mentioned the characters of a string are given two-way index for each location. Let us understand this with the help of an illustration as given in Fig. 7.1.

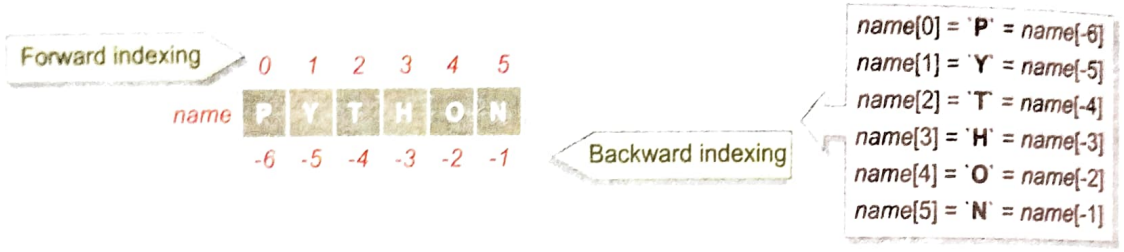


Figure 7.1 Structure of a Python String.

From Fig. 7.1 you can infer that :

- Strings in Python are stored by storing each character separately in contiguous locations.
- The characters of the strings are given two-way indices :
  - 0, 1, 2, ... in the *forward direction* and
  - – 1, – 2, – 3, ... in the *backward direction*.

Thus, you can access any character as ***<stringname>[<index]*** *e.g.*, to access the first character of string ***name*** shown in Fig. 7.1, you'll write **name[0]**, because the index of first character is 0. You may also write **name [–6]** for the above example *i.e.*, when string name is storing "PYTHON".

Let us consider another string, say **subject** = **'Computers'**. It will be stored as :

	0	1	2	3	4	5	6	7	8
subject	C	o	m	p	u	t	e	r	s
	– 9	– 8	– 7	– 6	– 5	– 4	– 3	– 2	– 1

Thus, `subject[0] = 'C'` `subject[2] = 'm'` `subject [6]= 'e'`
`subject [-1] = 's'` `subject[-7]= 'm'` `subject[-9] = 'C'`

Since **length** of string variable can be determined using function **len(<string>)**, we can say that :

- first character of the string is ***at index*** *0* or *–length*
- second character of the string is ***at index*** *1* or *–(length-1)*

  :

- second last character of the string is ***at index*** *(length -2)* or *-2*
- last character of the string is ***at index*** *(length -1)* or *-1*

In a string, say ***name***, of length ***ln***, the valid indices are 0, 1, 2, ... *ln*-1. That means, if you try to give something like :

```
>>> name[ln]
```

Python will return an error like :

```
Traceback (most recent call last):
  File "<pyshell#40>", line 1, in <module>
    name[ln]
IndexError: string index out of range
```

> **NOTE**
>
> The **index** (also called **subscript** sometimes) is the numbered position of a letter in the string. In Python, indices begin *0* onwards in the forward direction and –1, –2, ... in the backward direction.

The reason is obvious that in string there is no index equal to the length of the string, thus accessing an element like this causes an error.

Also, another thing that you must know is that you cannot change the individual letters of a string in place by assignment because **strings are immutable and** hence **item assignment is not supported**, *i.e.*,

```
name = 'hello'
name[0] = 'p'
```

individual letter assignment not allowed in Python

will cause an error like :

```
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    name[0] = 'p'
TypeError: 'str' object does not support item assignment
```

However, you can assign to a string another string or an expression that returns a string using assignment, *e.g.*, following statement is valid :

```
name = 'hello'
name = "new"
```

Strings can be assigned expressions that give strings.

> **NOTE**
>
> Valid string indices are *0, 1, 2 ...* upto *length-1* in forward direction and *–1, –2, –3...* *– length* in backward direction.

7.2.3 Lists and Tuples

The lists and tuples are Python's compound data types. We have taken them together in one section because they are basically the same types with one difference. Lists can be changed / modified (*i.e.*, mutable) but tuples cannot be changed or modified (*i.e.*, immutable). Let us talk about these two Python types one by one.

7.2.3A Lists

A List in Python represents a list of comma-separated values of any datatype between square brackets *e.g.*, following are some lists :

```
[1, 2, 3, 4, 5]
['a', 'e', 'i', 'o', 'u']
['Neha', 102, 79.5]
```

Like any other value, you can assign a list to a variable *e.g.*,

```
>>> a = [1, 2, 3, 4, 5]      # Statement 1
>>> a
[1, 2, 3, 4, 5]
>>> print (a)
[1, 2, 3, 4, 5]
```

To change first value in a list namely *a* (given above), you may write

```
>>> a [0] = 10                    # change 1st item - consider statement 1 above
>>> a
[10, 2, 3, 4, 5]
```

To change 3rd item, you may write

```
>>> a[2] = 30                     # change 3rd item
>>> a
[10, 2, 30, 4, 5]
```

You guessed it right ; the values internally are numbered from 0 (zero) onwards *i.e.*, first item of the list is internally numbered as 0, second item of the list as 1, 3rd item as 2 and so on.

We are not going further in list discussion here. Lists shall be discussed in details in a later chapter.

7.2.3B Tuples

You can think of Tuples (pronounced as tu-pp-le, rhyming with couple) as those lists which cannot be changed *i.e.*, are not modifiable. Tuples are represented as group of comma-separated values of any date type within parentheses, *e.g.*, following are some tuples :

```
p = (1, 2, 3, 4, 5)
q = (2, 4, 6, 8)
r = ('a', 'e', 'i', 'o', 'u')
h = (7, 8, 9, 'A', 'B', 'C')
```

Tuples shall be discussed in details in a later chapter.

NOTE

Values of type **list** are *mutable i.e.*, changeable – one can change / add / delete a list's elements. But the values of type **tuple** are *immutable i.e.*, non-changable ; one cannot make changes to a tuple.

7.2.4 Dictionary

Dictionary data type is another feature in Python's hat. The *dictionary* is an unordered set of comma-separated **key : value** pairs, within { }, with the requirement that within a dictionary, no two keys can be the same (*i.e.*, there are unique keys within a dictionary). For instance, following are some dictionaries :

```
{'a' : 1 , 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
```

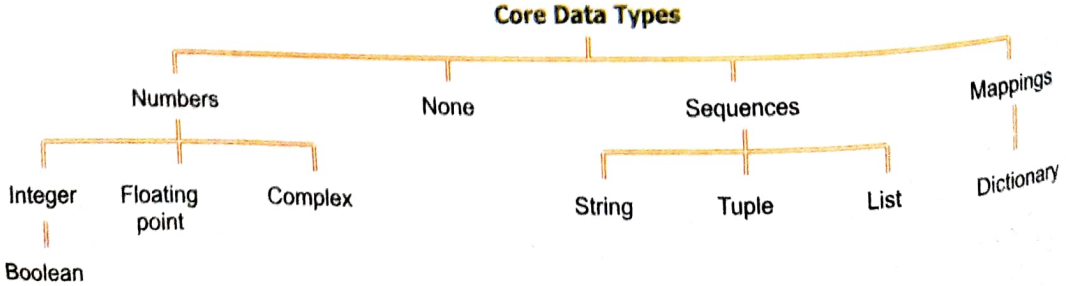
```
>>> vowels = {'a' : 1, 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
>>> vowels['a']
1
>>> vowels['u']
5
```

Here 'a', 'e', 'i', 'o' and 'u' are the keys of dictionary vowels; 1, 2, 3, 4, 5 are values for these keys respectively.

*Specifying **key** inside [ ] after dictionary name gives the corresponding **value** from the **key : value** pair inside dictionary.*

Dictionaries shall be covered in details in a later chapter.

Following figure summarizes the core data types of Python.



7.3 MUTABLE AND IMMUTABLE TYPES

The Python data objects can be broadly categorized into *two – mutable* and *immutable* types, in simple words changeable or modifiable and non-modifiable types.

1. Immutable types

The immutable types are those that can never change their value in place. In Python, the following types are immutable : *integers, floating point numbers, Booleans, strings, tuples.*

Let us understand the concept of immutable types. In order to understand this, consider the code below :

Immutable Types
- integers
- floating point numbers
- booleans
- strings
- tuples

Sample code 7.1

```
p = 5
q = p
r = 5
                    # will give 5, 5, 5
p = 10
r = 7
q = r
```

After reading the above code, you can say that values of integer variables *p, q, r* could be changed effortlessly. Since *p, q, r* are integer types, you may think that integer types can change values.

But hold : It is not the case. Let's see how.

You already know that in Python, variable-names are just the references to value-objects *i.e.*, data values. The variable-names do not store values themselves *i.e.*, they are not storage containers. Recall section 6.5.1 where we briefly talked about it.

Now consider the **Sample code 7.1** given above. Internally, how Python processes these assignments is explained in Fig. 7.2. Carefully go through figure 7.2 on the next page and then read the following lines.

So although it appears that the value of variable *p / q / r* is changing ; values are *not changing "in place"* the fact is that the variable-names are instead made to refer to new immutable integer object. (Changing **in place** means modifying the same value in same memory location).

- Initially these three statements are executed :

```
p = 5
q = p
r = 5
```

All variables having same value reference the same value object i.e., p, q, r will all reference same integer objects.

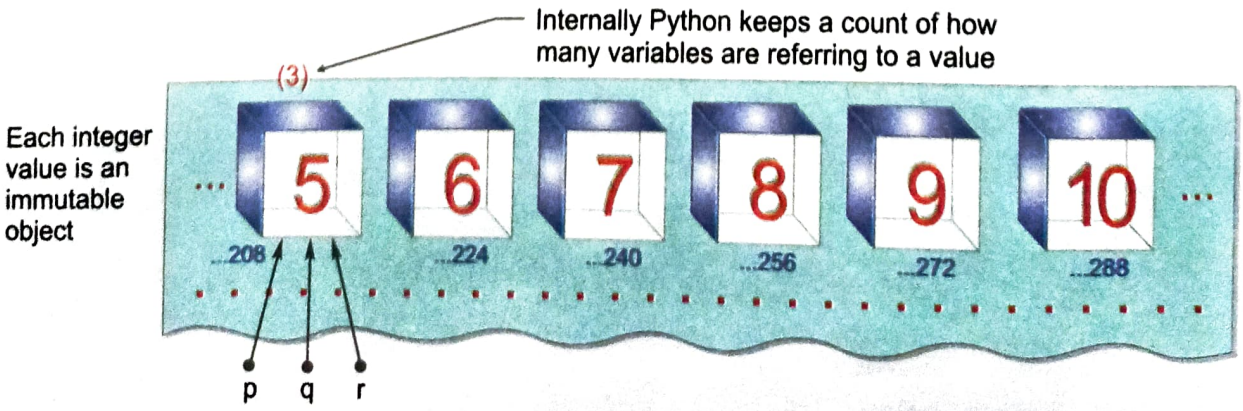


Figure 7.2

You can check/confirm it yourself using **id( )**. The **id( )** returns the memory address to which a variable is referencing.

```
In [40]: p = 5

In [41]: q = p

In [42]: r = 5

In [43]: id(5)
Out[43]: 1457662208

In [44]: id(p)
Out[44]: 1457662208

In [45]: id(q)
Out[45]: 1457662208

In [46]: id(r)
Out[46]: 1457662208
```

Notice the **id( )** is returning same memory address for *value* **5**, *p*, *q*, i - which means all these are referencing the same object.

NOTE

Please note that both **In [ ] :** and **>>>** are valid prompts of different Python shells. Where **In [ ] :** is a prompt of *IPython shell* used with Spyder IDE, **>>>** is a prompt of Python shell used by IDLE IDE, PyCcharmIDE and others. You will find **In [ ] :** are in our captured screenshots but in text component we have used **>>>**, a standard way of depicting any Python prompt.

Please note, memory addresses depend on your operating system and will vary in different sessions.

- When the next set of statements execute, *i.e.*,

```
p = 10
r = 7
q = r
```

then these variable names are made to point to different integer objects. That is, now their memory addresses that they reference will change. The original memory address of **p** that was having value **5** will be the same with the same value *i.e.*, **5** but **p** will no longer reference it. Same is for other variables.

To see Mutability/Immutability in action

Scan QR Code

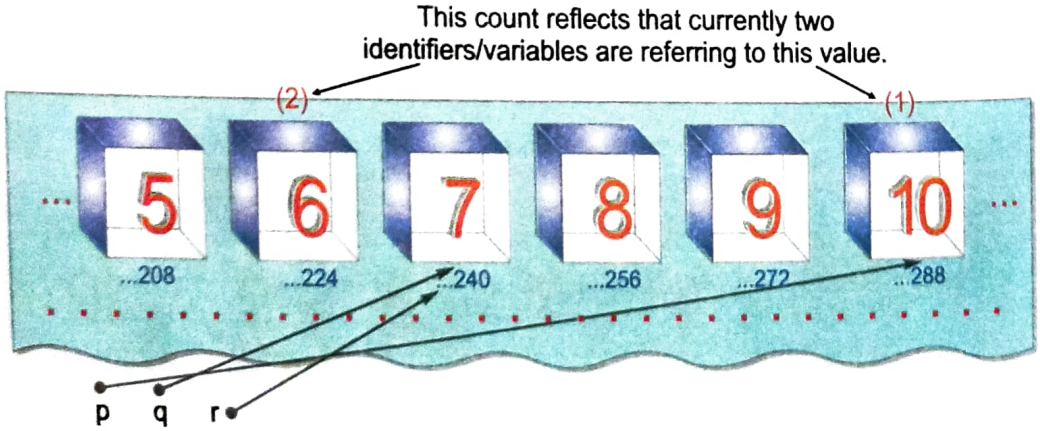


Figure 7.3

Let us check their ids

```
In [47]: p=10

In [48]: r=7

In [49]: q=r

In [50]: id(10)
Out[50]: 1457662288

In [51]: id(p)
Out[51]: 1457662288

In [52]: id(7)
Out[52]: 1457662240

In [53]: id(q)
Out[53]: 1457662240

In [54]: id(r)
Out[54]: 1457662240

In [55]: id(5)
Out[55]: 1457662208
```

Notice, this time with change in value, the reference memory address of variables ***p*** , ***q*** and *r* have changed.

The value 5 is at the same address

- Now if you assign 5 to any other variable. Let us see what happens.

```
In [56]: t = 5

In [57]: id(t)
Out[57]: 1457662208
```

Now variable ***t*** has reference memory address same as initial reference memory address of variable ***p*** when it has value 5. Compare listings given above

Thus, it is clear that **variable names** are stored as references to a value-object. Each time you change the value, the variable's reference memory address changes.

Variables (of certain types) are NOT LIKE storage containers *i.e.*, with fixed memory address where value changes every time. Hence they are IMMUTABLE.

The objects of following value types are immutable in Python :

- Integer
- floating point number
- Booleans
- strings
- tuples

NOTE

Mutablity means that in the same memory address, new value can be stored as and when you want. The types that do not support this property are **immutable types.**

2. Mutable types

The mutable types are those whose values can **be changed in place**. Only three types are mutable in Python. These are : *lists, dictionaries* and *sets.*

To change a member of a list, you may write :

```
Chk = [2, 4, 6]
Chk[1] = 40
```

Mutable Types

- Lists
- Dictionaries
- Sets

It will make the list namely **Chk** as [2, 40, 6].

```
In [60]: Chk = [2, 4, 6 ]

In [61]: id(Chk)
Out[61]: 150195536

In [62]: Chk[1] = 40

In [63]: id(Chk)
Out[63]: 150195536
```

See, even after changing a value in the list **Chk**, its reference memory address has remained same. That means the change has taken **in place - the lists are mutable**

NOTE

Mutable objects are : ***list, dictionary, set***

Immutable objects: ***int, float, complex, string, tuple***

Lists and Dictionaries shall be covered later in this book.

7.3.1 Variable Internals

Python is an object oriented language. Python calls every entity that stores any values or any type of data as an **object**.

An **object** is an entity that has certain properties and that exhibit a certain type of behavior, *e.g.*, integer values are objects – they hold whole numbers only and they have infinite precision (**properties**); they support all arithmetic operations (**behavior**).

Python object's Key Attributes

- a type
- a value
- an id

So all data or values are referred to as **object** in Python. Similarly, we can say that a variable is also an object that refers to a value.

Every Python object has *three* key attributes associated to it :

(i) The type of an object.

The type of an object determines the operations that can be performed on the object. Built-in function **type( )** returns the type of an object. Consider this :

```
>>> a = 4
>>> type (4)
<class 'int'>
>>> type (a)
<class 'int'>
```

Type of integer value 4 is returned ***int*** *i.e., integer*

Type of variable ***a*** *is also int i.e., integer because* ***a*** *is currently referring to an integer value.*

(ii) The value of an object

It is the data-item contained in the object. For a literal, the value is the literal itself and for a variable the value is the data-item it (the variable) is currently referencing. Using **print** statement you can display value of an object. For example,

```
>>> a = 4
>>> print (4)
4
>>> print (a)
4
```

value of integer literal 4 is 4

value of variable ***a*** *is 4 as it is currently referencing integer value 4.*

Check Point **7.3**

1. What is String data type in Python ?
2. What are two internal subtypes of String data in Python ?
3. How are str type strings different from Unicode strings ?
4. What are List and Tuple data types of Python ?
5. How is a list type different from tuple data type of Python ?
6. What are Dictionaries in Python ?
7. Identify the types of data from the following set of data

   'Roshan', u'Roshan', False, 'False', ['R', 'o', 's', 'h', 'a', 'n',], ('R', 'o', 's', 'h', 'a', 'n'), {'R' : 1, ' o' : 2, 's' : 3, 'h' : 4 'a' : 5, 'n' : 6}, (2.0-j), 12, 12.0, 0.0, 0, 3j, 6 + 2.3j, True, "True"
8. What do you understand by mutable and immutable objects ?

(iii) The id of an object

The **id** of an object is generally the memory location of the object. Although **id** is implementation dependent but in most implementations it returns the memory location of the object. Built-in function **id( )** returns the **id** of an object, *e.g.*,

```
>>> id(4)
30899132
>>> a = 4
>>> id (a)
30899132
```

Object 4 is internally stored at location 30899132

*Variable **a** is current referencing location 30899132 (Notice same as id(4). Recall that variable is not a storage location in Python, rather a label pointing to a value object).*

The **id( )** of a variable is same as the **id( )** of value it is storing. Now consider this :

Sample code 7.2

```
>>> id(4)
30899132
>>> a = 4
>>> id(a)
30899132
>>> b = 5
>>> id(5)
30899120
>>> id(b)
30899120
>>> b = b-1
>>> id(b)
30899132
>>>
```

*The id's of **value 4** and **variable a** are the same since the memory-location of **4** is same as the location to which **variable a** is referring to.*

***Variable b** is currently having value **5**, i.e., referring to integer value 5*

Variable b** will now refer to value **4

***Now notice that the id of variable b** is same as **id** of **integer 4**.*

Thus internal change in value of variable ***b*** (from 5 to 4) of sample code 7.2 will be represented as shown in Fig. 7.4.

Please note that while storing complex numbers, id's are created differently, so a complex literal say $2.4j$ and a complex variable say x having value $2.4j$ may have different id's.

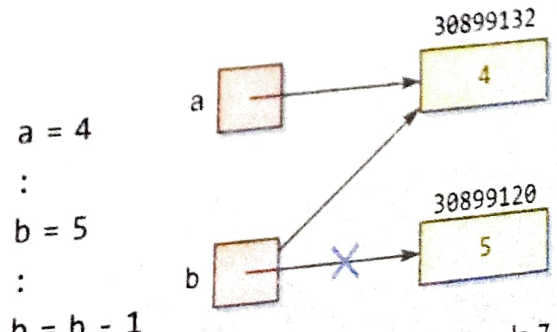


Figure 7.4 Memory representation of sample code 7.2.

Symbol	Name	Example	Result	Comment
+	addition	6 + 5 5 + 6	11 11	adds values of its two operands.
–	subtraction	6 – 5 5 – 6	1 – 1	subtracts the value of right operand from left operand.
*	multiplication	5 * 6 6 * 5	30 30	multiplies the values of its two operands.
/	division	60/5	12	divides the value of left operand with the value of right operand and returns the result as a **float** value.
%	Modulus (pronounced mo-du-lo) or Remainder	60%5 6%5	0 1	divides the two operands and gives the remainder resulting.
//	Floor division	7.2 // 2	3.0	divides and truncates the fractional part from the result.
**	Exponentiation (Power)	2.5 ** 3	15.625	returns base raised to power exponent. (2.5^3 here)

Negative Number Arithmetic in Python

Arithmetic operations are straightforward even with negative numbers, especially with non-division operators *i.e.*,

```
-5 + 3      will give you    2
-5 - 3      will give you    -8
-5 * 3      will give you    -15
-5 ** 3     will give you    -125
```

But when it comes to division and related operators (/, //, %), mostly people get confused. Let us see how Python evaluates these. To understand this, we recommend that you look at the operation shown in the adjacent screenshot and then look for its working explained below, where the result is shown shaded

(a) $-3\overline{)5}\,(-2$
$\underline{6}$
-1

(b) $3\overline{)-5}\,(-2$
$\underline{-6}$
$+1$

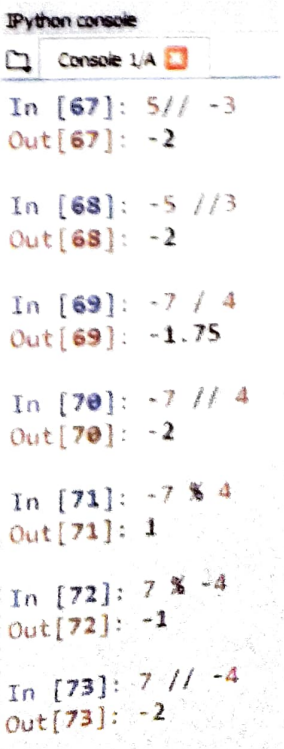
(c) $4\overline{)-7}\,(-1.75$
$\underline{-4}$
-3
$\underline{-3}$
0

(d) $4\overline{)-7}\,(-2$
$\underline{-8}$
1

(e) $4\overline{)-7}\,(-2$
$\underline{-8}$
$+1$

(f) $-4\overline{)7}\,(-2$
$\underline{8}$
-1

(d) $-4\overline{)7}\,(-2$
$\underline{8}$
-1



Operation	Description	Comment
$x \mathrel{+}= y$	$x = x + y$	Value of y added to the value of x and then result assigned to x
$x \mathrel{-}= y$	$x = x - y$	Value of y subtracted from the value of x and then result assigned to x
$x \mathrel{*}= y$	$x = x * y$	Value of y multiplied to value of x and then result assigned to x
$x \mathrel{/}= y$	$x = x / y$	Value of y divides value of x and then result assigned to x
$x \mathrel{//}= y$	$x = x // y$	Value of y does floor division to value of x and then result assigned to x
$x \mathrel{**}= y$	$x = x ** y$	x^y computed and then result assigned to x
$x \mathrel{\%}= y$	$x = x \% y$	Value of y divides value of x and then remainder assigned to x

Relational Operators in Python

p	q	p < q	p <= q	p == q	p > q	p >= q	p != q
3	3.0	False	True	True	False	True	False
6	4	True	False	False	True	True	True
'A'	'A'	False	True	True	False	True	False
'a'	'A'	False	False	False	True	True	True

7.4.3 Identity Operators

There are two identity operators in Python ***is*** and ***is not***. The identity operators are used to check if both the operands reference the same object memory *i.e.*, the identity operators compare the memory locations of two objects and return *True* or *False* accordingly.

Operator	Usage	Description
is	a **is** b	returns ***True*** if both its operands are pointing to same object (*i.e.*, both referring to same memory location), returns ***False*** otherwise.
is not	a ***is not*** b	returns ***True*** if both its operands are pointing to different objects (*i.e.*, both referring to different memory location), returns ***False*** otherwise.

Consider the following examples :

```
A = 10
B = 10
A is B
```

will return **True** because both are referencing the memory address of value 10

You can use **id( )** to confirm that both are referencing same memory address.

7.4.4B The or Operator

The **or** operator combines *two* expressions, which make its *operands*. The **or** operator works in these ways :

(*i*) relational expressions as *operands* (*ii*) numbers or strings or lists as *operands*

(*i*) Relational expressions as operands

When or operator has its operands as relational expressions (*e.g.*, $p > q$, $j != k$, etc.) then the or operator performs as per following principle :

The or operator evaluates to ***True*** *if either of its (relational) operands evaluates to True ;* ***False*** *if both operands evaluate to False.*

That is :

x	y	x or y
False	False	False
False	True	True
True	False	True
True	True	True

Following are some examples of this *or* operation :

(4 == 4) or (5 == 8) results into **True** because first expression (4 == 4) is *True*.

5 > 8 **or** 5 < 2 results into **False** because both expressions 5 > 8 and 5 < 2 are *False*.

(ii) Numbers / strings / lists as operands[7]

When or operator has its operands as numbers or strings or lists (*e.g.*, **'a' or '', 3 or 0**, etc.) then the or operator performs as per following principle :

In an expression **x or y**, *if first operand, (i.e., expression* **x***) has* ***false***$_{tval}$, *then return second operand* **y** *as result, otherwise return* **x**.

That is :

x	y	x or y
false$_{tval}$	false$_{tval}$	*y*
false$_{tval}$	true$_{tval}$	*y*
true$_{tval}$	false$_{tval}$	*x*
true$_{tval}$	true$_{tval}$	*x*

NOTE

In general, ***True*** and ***False*** represent the Boolean values and ***true*** and ***false*** represent truth values.

Examples

Operation	Results into	Reason
0 **or** 0	0	first expression (0) has false$_{tval}$, hence second expression **0** is returned.
0 **or** 8	8	first expression (0) has false$_{tval}$, hence second expression **8** is returned.
5 **or** 0.0	5	first expression (5) has true$_{tval}$, hence first expression **5** is returned.
'hello' **or** ''	'hello'	first expression ('hello') has *true*$_{tval}$, hence first expression '***hello***' is returned
'' **or** 'a'	'a'	first expression ('') has *false*$_{tval}$, hence second expression '***a***' is returned.
'' **or** ''	''	first expression ('') has *false*$_{tval}$, hence second expression '' is returned.
'a' **or** 'j'	'a'	first expression ('a') has *true*$_{tval}$, hence first expression '***a***' is returned.

How the truth value is determined ? – refer to section 7.4.4A above.

7. This type of or functioning applies to any value which is not a relational expression but whose truth-ness can be determined by Python.

The **or** operator will test the second operand **only if** the first operand is *false*, otherwise ignore it; even if the second operand is logically wrong *e.g.*,

```
20 > 10 or "a" + 1 > 1
```

will give you result as

```
True
```

without checking the second operand of **or** *i.e.*, **"a" + 1 > 1**, which is syntactically wrong – you cannot add an integer to a string.

NOTE

The **or** operator will test the second operand **only if** the first operand is *false*, otherwise ignore it.

7.4.4C The and Operator

The and operator combines *two* expressions, which make its operands. The and operator works in these ways :

(*i*) relational expressions as operands

(*ii*) numbers or strings or lists as operands

(*i*) Relational expressions as operands

When and operator has its operands as relational expressions (*e.g.*, $p > q$, $j != k$, etc.) then the and operator performs as per following principle :

The and operator evaluates to True if both of its (relational) operands evaluate to True ; False if either or both operands evaluate to False.

That is :

x	y	x and y
False	False	False
False	True	False
True	False	False
True	True	True

Following are some examples of the **and** operation :

`(4 == 4) and (5 == 8)`	results into **False** because first expression **(4 == 4) is** *True* but second expression (5 == 8) evaluates to *False*. Both operands have to result into *True* in order to have the final results as *True*
`5 > 8 and 5 < 2`	results into **False** because first expression : 5 > 8 evaluates to *False*.
`8 > 5 and 2 < 5`	results into **True** because both operands : 8 > 5 and 2 < 5 evaluate to ***True***

(*ii*) Numbers / strings / lists as operands[8]

When and operator has its operands as numbers or strings or lists (*e.g.*, **'a' or '', 3 or 0**, etc.) then the and operator performs as per following principle :

In an expression **x and y**, *if first operand, (i.e., expression x) has* **false** *tval, then return first operand* **x** *as result, otherwise return y.*

8. This type of or functioning applies to any value which is not a relational expression but whose truth-ness can be determined by Python.

x	y	x and y
$false_{tval}$	$false_{tval}$	x
$false_{tval}$	$true_{tval}$	x
$true_{tval}$	$false_{tval}$	y
$true_{tval}$	$true_{tval}$	y

Examples

Operation	Results into	Reason
0 **and** 0	0	first expression (0) has $false_{tval}$, hence *first expression* **0** is returned.
0 **and** 8	0	first expression (0) has $false_{tval}$, hence first expression **0** is returned.
5 **and** 0.0	0.0	first expression (5) has $true_{tval}$, hence second expression **0.0** is returned.
'hello' **and** ''	''	first expression ('hello') has $true_{tval}$, hence second expression '' is returned
'' **and** 'a'	''	first expression ('') has $false_{tval}$, hence first expression '' is returned.
'' **and** ''	''	first expression ('') has $false_{tval}$, hence first expression '' is returned.
'a' **and** 'j'	'j'	first expression ('a') has $true_{tval}$, hence second expression 'j' is returned.

How the truth value is determined ? – refer to section 7.4.4A above.

IMPORTANT

The **and** operator will test the second operand **only if** the first operand is *true*, otherwise ignore it ; even if the second operand is logically wrong *e.g.*,

```
10 > 20 and "a" + 10 < 5
```

will give you result as

```
False
```

ignoring the second operand completely, even if it is wrong – you cannot add an integer to a string in Python.

NOTE

The **and** operator will test the second operand **only if** the first operand is *true*, otherwise ignore it.

7.4.4D The not Operator

The Boolean/logical **not** operator, works on single expression or operand *i.e.*, it is a unary operator. The logical **not** operator negates or reverses the truth value of the expression following it *i.e.*, if the expression is ***True*** or ***true*** $_{tval}$, then ***not expression*** is ***False***, and vice versa. Unlike '**and**' and '**or**' operators that can return number or a string or a list etc. as result, the '**not**' operator returns always a Boolean value ***True*** or ***False***.

Consider some examples below :

`not 5`	results into ***False*** because 5 is non-zero (*i.e.*, $true_{tval}$)
`not 0`	results into ***True*** because 0 is zero (*i.e.*, $false_{tval}$)
`not -4`	results into ***False*** because – 4 is non zero thus $true_{tval}$.
`not (5 > 2)`	results into ***False*** because the expression 5 > 2 is *True*.
`not (5 > 9)`	results into ***True*** because the expression 5 > 9 is *False*.

NOTE

Operator **not** has a lower priority than non-Boolean operators, so **not a == b** is interpreted as **not (a == b)**, and **a == not b** is a syntax error.

Following table summarizes the logical operators.

Table 7.4 *The Logical Operators*

Operation	Result	Notes
x or y	if x is false$_{bool}$, then return y as result, else x	It **(or)** only evaluates the second argument if the first one is ***false***$_{bool}$
x and y	if x is false$_{bool}$, then x as result, else y	It **(and)** only evaluates the second argument if the first one is ***true***$_{bool}$
not x	if x is false$_{bool}$, then return *True* as result, else *False*	**not** has a lower priority than non-Boolean operators.

Chained Comparison Operators

While discussing Logical operators, Python has something interesting to offer. You can chain multiple comparisons which are like shortened version of larger Boolean expressions. Let us see how. Rather than writing 1 < 2 and 2 < 3, you can even write 1 < 2 < 3, which the chained version of earlier Boolean expression.

The above statement will check if *1 was less than* 2 **and** *if* 2 *was less than* 3.

Let's look at a few examples of using chains :

```
>>> 1 < 2 < 3          is equivalent to      >>> 1 < 2 and 2 < 3
True                                          True
```

As per the property of **and**, the expression **1 < 3** will be first evaluated and ***if only it is True***, then only the next chained expression **2 < 3** will be evaluated.

Similarly consider some more examples :

```
>>> 11 < 13 > 12
True
```

The above expression checks if *13* is larger than both the other numbers ; it is the shortened version of **11 < 13** and **13 > 12**.

7.4.5 Bitwise Operators

Python also provides another category of operators - *bitwise operators, which* are similar to the logical operators, except that they work on a smaller scale – on binary representations of data. Bitwise operators are used to change individual bits in an operand.

Python provides following Bitwise operators.

Table 7.5 *Bitwise Operators*

Operator	Operation	Use	Description
&	bitwise and	op1 & op2	The AND operator compares two bits and generates a result of 1 if both bits are 1; otherwise, it returns 0.
|	bitwise or	op1 | op2	The OR operator compares two bits and generates a result of 1 if the bits are complementary; otherwise, it returns 0.
^	bitwise xor	op1 ^ op2	The EXCLUSIVE-OR (*XOR*) operator compares two bits and returns 1 if either of the bits are 1 and it gives 0 if both bits are 0 or 1.
~	bitwise complement	~op1	The COMPLEMENT operator is used to invert all of the bits of the operand

7.4.5A The AND operator &

When its operands are numbers, the & operation performs the bitwise AND function on each parallel pair of bits in each operand. The AND function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following Table 7.6.

Table 7.6 *The Bitwise AND (&) Operation*

op1	op2	Result
0	0	0
0	1	0
1	0	0
1	1	1

For AND operations, 1 AND 1 produces 1. Any other combination produces 0.

Suppose that you were to AND the values 13 and 12, like this : 13 & 12. The result of this operation is 12 because the binary representation of 12 is 1100, and the binary representation of 13 is 1101. You can use **bin( )** to get binary representation of a number.

```
13 & 12   1101
          1100
          ----
          1100
```

```
In [76]: bin(13)
Out[76]: '0b1101'

In [77]: bin(12)
Out[77]: '0b1100'

In [78]: 13 & 12
Out[78]: 12

In [79]: bin(13 & 12)
Out[79]: '0b1100'
```

If both operand bits are 1, the AND function sets the resulting bit to 1 ; otherwise, the resulting bit is 0. So, when you line up the two operands and perform the AND function, you can see that the two high-order bits (the two bits farthest to the left of each number) of each operand are 1. Thus, the resulting bit in the result is also 1. The low-order bits evaluate to 0 because either one or both bits in the operands are 0.

7.4.5B The inclusive OR operator |

When both of its operands are numbers, the | operator performs the inclusive OR operation. Inclusive OR means that if either of the two bits is 1, the result is 1. The following Table 7.7 shows the results of inclusive OR operations.

Table 7.7 *The Inclusive OR (|) Operation*

op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	1

For OR operations, 0 OR 0 produces 0. Any other combination produces 1.

```
13 | 12   0000 1101
          0000 1100
          ---------
          0000 1101
```

```
In [80]: bin(13)
Out[80]: '0b1101'

In [81]: bin(12)
Out[81]: '0b1100'

In [82]: bin(13 | 12)
Out[82]: '0b1101'

In [83]: 13 | 12
Out[83]: 13
```

7.4.5C The eXclusive OR (XOR) operator ^

Exclusive OR means that if the two operand bits are different, the result is 1 ; otherwise the result is 0. The following Table 7.8 shows the results of an eXclusive OR operation.

Table 7.8 *The eXclusive OR (^) Operation*

op1	op2	Result
0	0	0
0	1	1
1	0	1
1	1	0

```
13 ^ 12     0000 1101
            0000 1100
            ---------
            0000 0001
```

```
In [84]: bin(13)
Out[84]: '0b1101'

In [85]: bin(12)
Out[85]: '0b1100'

In [86]: 13 ^ 12
Out[86]: 1

In [87]: bin(13 ^ 12)
Out[87]: '0b1'
```

For XOR operations, 1 XOR 0 produces 1, as does 0 XOR 1. (All these operations are commutative.) Any other combination produces 0.

7.4.5D The Complement Operator ~

The complement operator inverts the value of each bit of the operand : if the operand bit is 1 the result is 0 and if the operand bit is 0 the result is 1.

Table 7.9 *The Complement (~) Operation*

op1	Result
0	1
1	0

*This is binary code of **–13** in **2's complement form.***[9]

```
~12     0000 1100
        ---------
        1111 0011
        = - (0000 1101)
```

```
In [91]: bin(12)
Out[91]: '0b1100'

In [92]: ~12
Out[92]: -13

In [93]: bin(13)
Out[93]: '0b1101'

In [94]: bin(~12)
Out[94]: '-0b1101'
```

7.4.6 Operator Precedence

When an expression or statement involves multiple operators, Python resolves the order of execution through Operator Precedence. The chart of operator precedence from highest to lowest for the operators covered in this chapter is given below.

Operator	Description
()	Parentheses (grouping)
**	Exponentiation
~x	Bitwise nor
+x, -x	Positive, negative (unary + , –)
*, /, //, %	Multiplication, division, floor division, remainder
+, -	Addition, subtraction
&	Bitwise and
^	Bitwise XOR
|	Bitwise OR
<, <=, >, >=, <>, !=, ==, is, is not	Comparisons (Relational operators), identity operators
not x	Boolean NOT
and	Boolean AND
or	Boolean OR

Highest ↓ Lowest

9. To obtain the number whose 2's complement is given, you can calculate its 2's complement again by following this rule - starting from **right to left** , copy all the bits as it is UNTIL you find first **1**, then invert all other bits. As per this rule, we can calculate 2's complement of 11110011 as 00001101, which is 13.

- Operators are the symbols (or keywords sometimes) that represent specific operations.
- Arithmetic operators carry out arithmetic for Python. These are unary +, unary −, +, −, *, /, //, % and **.
- Unary + gives the value of its operand, unary − changes the sign of its operand's value.
- Addition operator + gives sum of is operand's value, − subtracts the value of second operand from the value of first operand, * gives the product of its operands' value. The operator / divides the first operand by second and returns a float result // performs the floor division, % gives the remainder after dividing first operand by second and ** is the exponentiation operator, i.e., it gives base raised to power.
- Relational operators compare the values of their operands. These are >, <, ==, <=, >= and != i.e., less than, greater than, equal to, less than or equal to, greater than or equal to and not equal to respectively.
- Bitwise operators are like logical operators but they work on individual bits.
- Identify operators (is, is not) compare the memory two objects are referencing.
- Logical operators perform comparisons on the basis of truth-ness of an expression or value. These are **'or'**, **'and'** and **'not'**.
- Boolean or relational expressions' truth value depends on their Boolean result True or False.
- Values' truth value depends on their emptiness or non-emptiness. Empty numbers (such as 0, 0.0, 0j etc.) and empty sequences (such as "", [ ], ( )) and None have truth-value as **false** and all others (non-empty ones) have truth-value as **true**.

LET US REVISE

- *An expression is composed of one or more operations. It is a valid combination of operators, literals and variables.*
- *In Python terms, an expression is a legal combination of atoms and operators.*
- *An atom in Python is something that has a value. Examples of atoms are variables, literals, strings, lists, tuples, sets etc.*
- *Expressions can be arithmetic, relational or logical, compound etc.*
- *Types of operators used in an expression determine its type. For instance, use of arithmetic operators makes it arithmetic expression.*

- *Arithmetic expressions can either be integer expressions or real expressions or complex number operations or mixed-mode expressions.*
- *An arithmetic expression always results in a number (integer or floating-point number or a complex number) ; a relational expression always results in a Boolean value i.e., either True or False ; and a logical expression results into a number or a string or a Boolean value, depending upon its operands.*
- *In a mixed-mode expression, different types of variables/constants are converted to one same type. This process is called type conversion.*
- *Type conversion can take place in two forms : implicit (that is performed by compiler without programmer's intervention) and explicit (that is defined by the user).*
- *In implicit conversion, all operands are converted up to the type of the largest operand, which is called type promotion or coercion.*
- *The explicit conversion of an operand to a specific type is called type casting and it is done using type conversion functions that is used as*

```
<type conversion function > ( <expression>)
```

e.g., to convert to float, one may write

```
float(<expression>)
```