8 Data Handling

In This Chapter

8.1 INTRODUCTION

In any language, there are some fundamentals you need to know before you can write even the most elementary programs. This chapter introduces some such fundamentals : *data types, variables, operators* and *expressions* in Python.

Python provides a predefined set of data types for handling the data it uses. Data can be stored in any of these data types. This chapter is going to discuss various types of data that you can store in Python. Of course, a program also needs a means to identify stored data.

So, this chapter shall also talk about mutable and immutable variables in Python.

8.2 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.

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.

8.2.1 Numbers

As it is clear by the name the Number data types are used to store numeric values in Python. The Numbers in Python have following core data types :

(i) Integers
- Integers (signed)
- Booleans

(ii) Floating-Point Numbers

(iii) Complex Numbers

8.2.1A Integers

Integers are whole numbers such as 5, 39, 1917, 0 etc. They have no fractional parts. Integers are represented in Python by numeric values with no decimal point. Integers can be positive or negative, *e.g.*, +12, –15, 3000 (missing + or – symbol means it is positive number).

Types of Integers in Python
- Integers (signed)
- Booleans

There are *two* types of integers in Python :

(i) **Integers (signed).** It is the normal integer[1] representation of whole numbers. Integers in *Python 3.x* can be of any length, it is only limited by the memory available. Unlike other languages, *Python 3.x* provides single data type (***int***) to store any integer, whether *big* or *small*.

It is signed representation, *i.e.*, the integers can be positive as well as negative.

(ii) **Booleans.** These represent the truth values ***False*** and ***True***. The Boolean type is a subtype of plain integers, and Boolean values *False* and *True* behave like the values 0 and 1, respectively. To get the Boolean equivalent of 0 or 1, you can type ***bool(0)*** or ***bool (1)***, Python will return ***False*** or ***True*** respectively. See figure below (left side).

```
>>> bool(0)
False

>>> bool(1)
True
```

```
>>> str(False)
'False'

>>> str(True)
'True'
```

NOTE

The str( ) function converts a value to string.

However, when you convert Boolean values ***False*** and ***True*** to a string, the strings **'False'** or **'True'** are returned, respectively. The **str( )** function converts a value to string. See figure above (right side).

NOTE

The two objects representing the values ***False*** and ***True*** (not **false** or **true**) are the only Boolean objects in Python.

8.2.1B Floating Point Numbers

A number having fractional part is a floating-point number. *For example,* 3.14159 is a floating-point number. The decimal point signals that it is a floating-point number, not an integer. The number 12 is an integer, but 12.0 is a floating-point number.

1. In some cases, the exception *OverflowError* is raised instead if the given number cannot be represented through available number of bytes.

Recall (from Literals/Values' discussion in chapter 6) that fractional numbers can be written in *two* forms :

(*i*) **Fractional Form** (*Normal Decimal Notation*) *e.g.*, 3500.75, 0.00005, 147.9101 etc.

(*ii*) **Exponent Notation** *e.g.*, 3.50075E03, 0.5E-04, 1.479101E02 etc.

Floating point variables represent real numbers, which are used for measurable quantities like distance, area, temperature etc. and typically have a fractional part.

Floating-point numbers have *two* **advantages** over integers :

- They can represent values between the integers.
- They can represent a much greater range of values.

But floating-point numbers suffer from one disadvantage also :

- Floating-point operations are usually slower than integer operations.

In Python, floating point numbers represent machine-level **double precision floating point numbers**[2] **(15 digit precision)**. The range of these numbers is limited by underlying machine architecture subject to available (virtual) memory.

NOTE

In Python, the Floating point numbers have precision of 15 digits (double-precision).

8.2.1C Complex Numbers

Python is a versatile language that offers you a numeric type to represent *Complex Numbers* also. Complex Numbers ? Hey, don't you know about *Complex numbers* ? Uhh, I see. You are going to study about *Complex numbers* in class XI Mathematics book. Well, if you don't know anything about complex numbers, then for you to get started, I am giving below brief introduction of *Complex numbers* and then we shall talk about *Python's representation of Complex numbers.*

Check Point 8.1

1. What are the built-in core data types of Python ?
2. What do you mean by Numeric types ? How many numeric data types does Python provide ?
3. What will be the data types of following two variables ?

```
A = 2147483647
B = A + 1
```

(**Hint.** Carefully look the values they are storing. You can refer to range of Python number table.)

4. What are Boolean numbers ? Why are they considered as a type of integers in Python ?

Mathematically, a complex number is a number of the form $A + Bi$ where i is the imaginary number, ***equal to the square root of*** -1 *i.e.*, $\sqrt{-1}$.

A complex number is made up of both **real and imaginary components**. In complex number $A + Bi$, A and B are real numbers and i is imaginary. If we have a complex number z, where $z = a + bi$ then a would be the *real component* and b would represent the *imaginary component* of z, *e.g.*, real component of $z = 4 + 3i$ is 4 and the imaginary component would be 3.

NOTE

A complex number is in the form $A + Bi$ where i is the imaginary number, **equal to the square root of** -1 *i.e.*, $\sqrt{-1}$, that is $i^2 = -1$.

2. As per Python documentation, "Python does not support single-precision floating point numbers ; the savings in processor and memory usage that are usually the reason for using these is dwarfed by the overhead of using objects in Python, so there is no reason to complicate the language with two kinds of floating point numbers".

Complex Numbers in Python

Python represents complex numbers in the form $A + Bj$. 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.

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.

```
>>> 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).

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***.

Check Point 8.2

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.

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

Table 8.1 *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**.

8.2.2 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" , "??????", '????', "????", '???', "??"

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. 8.1.

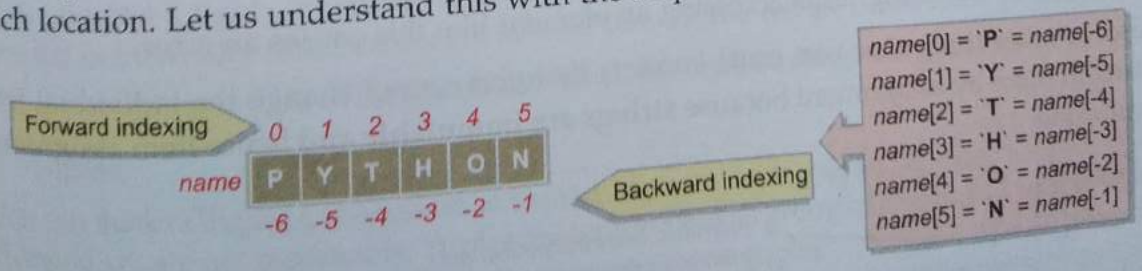


Figure 8.1 Structure of a Python String.

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.

From Fig. 8.1 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. 8.1, 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 :

```
    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 :

```
    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.

8.2.3 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.

8.2.3A 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.

8.2.3B 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-changeable ; one cannot make changes to a tuple.

8.2.4 Sets

Sets are also similar to lists that can store multiple values same as in mathematical sets. But Python sets are little different from lists.

These differences are :

- Sets are created by specifying comma separated values enclosed in curly brackets { }.
- Sets' elements are unordered and unindexed unlike lists.
- Sets do not allow duplicate entries unlike lists.
- Sets cannot contain mutable elements.

Consider below given example which is creating a set of values.

```
>>> myset = { 1, 2, 3, 4}
>>> print(myset)
{1, 2, 3, 4}
```

Set created by specifying comma separated values enclosed in curly brackets { }.

```
>>> myset1 = { 1, 2, 3, 4, 4}
>>> print(myset1)
{1, 2, 3, 4}
```

Sets do not allow duplicate entries. See, it rejected the duplicate entry 4. As you can see that 4 appears only once in the set even though you gave 4 twice while creating.

```
>>> type(myset)
<class 'set'>
```

```
>>> set1 = {1, 2, [3, 4]}
Traceback(most recent call last) :
    File "<pyshell#68>", line 1, in <module>
        set1 = {1, 2, [3, 4]}
TypeError : unhashable type : 'list'
>>> set1 = {1, 2, (3, 4)}
```

Sets do not allow mutable elements like lists. See it gave error when we tried to add a list as element, but it gave no error when we gave a tuple as an element, because a tuple is not mutable.

A set cannot contain a mutable element in it. However, a set itself is mutable, *i.e.*, we can add or remove items from it.

NOTE

Python data types *strings, lists, tuples, dictionary* etc. are all **iterables**.

8.2.5 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.

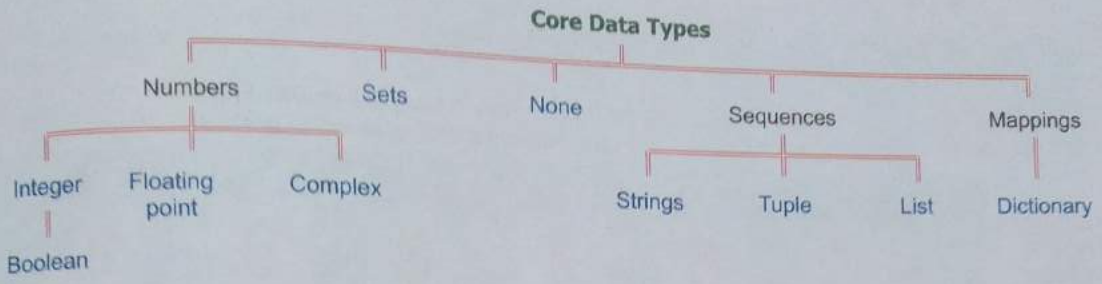
*Specifying **key** inside [ ] after dictionary name gives the corresponding **value** from the **key : value** pair inside dictionary.*

ITERABLE

An iterable is any Python object that can return its members, **one at a time**. Since, you can access elements of strings, lists, tuples and dictionary one at a time, these are all iterables.

Dictionaries shall be covered in details in a later chapter.

Following figure summarizes the core data types of Python.



8.3 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 8.1

```
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 7.5.1 where we briefly talked about it.

Now consider the **Sample code 8.1** given above. Internally, how Python processes these assignments is explained in Fig. 8.2. Carefully go through figure 8.2 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.

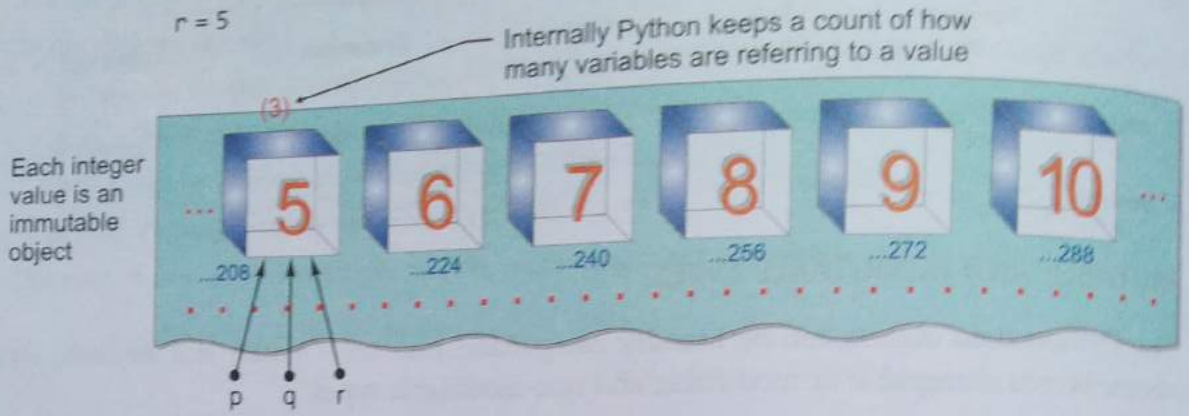


Figure 8.2

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

```
>>> p = 5
>>> q = p
>>> r = 5
>>> id(5)
1457662208
>>> id(p)
1457662208
>>> id(q)
1457662208
>>> id(r)
1457662208
```

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

NOTE

Please note that **everything is an object in Python**, be it data items, any literal value, strings, numbers, data types, variables and so forth. This is because Python is an object oriented language and it creates and treats everything as an object. The **id( )** function when you use on any data value or variable, it returns the unique identity of the object, which is its assigned memory address.

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

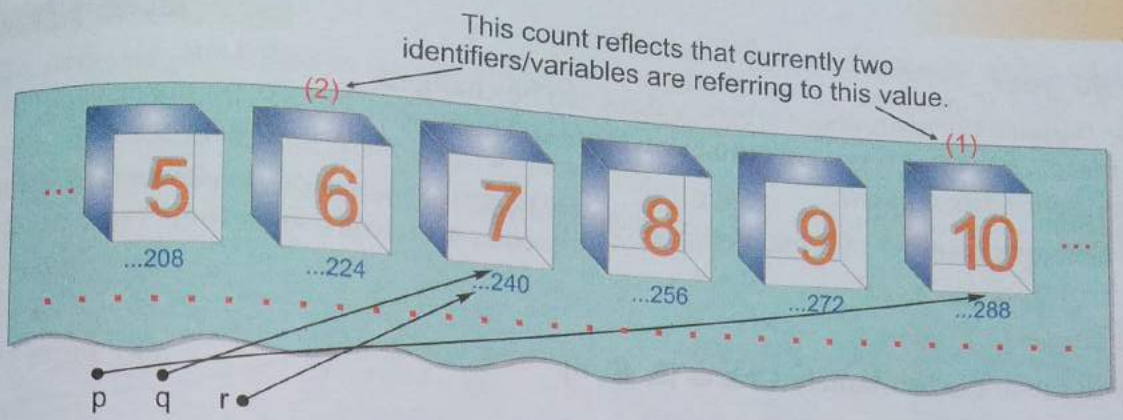


Figure 8.3

Let us check their ids

```
>>> p = 10
>>> r = 7
>>> q = r
>>> id(10)
1457662288
>>> id(p)
1457662288
>>> id(7)
1457662240
>>> id(q)
1457662240
>>> id(r)
1457662240
>>> id(5)
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

(Compare the result of id(5) on the previous page)

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**.

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

```
>>> t = 5
>>> id(t)
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.

Mutable Types
- Lists
- Dictionaries
- Sets

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
```

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

```
>>> Chk = [2, 4, 6]
>>> id(Chk)
150195536
>>> Chk[1] = 40
>>> id(Chk)
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**

Lists and Dictionaries shall be covered later in this book.

8.3.1 Variable Internals

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

Check Point 8.3

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 ?

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**).

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.

(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 8.2

```
>>> 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 8.2 will be represented as shown in Fig. 8.4.

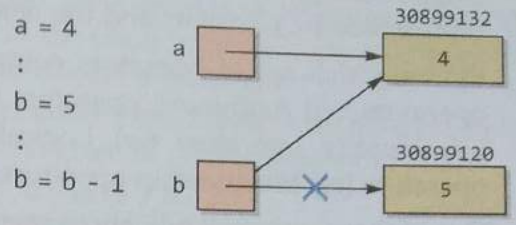


Figure 8.4 Memory representation of sample code 8.2.

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

EXAMPLE 1 *What is the output of the following code ?*

```
nset1 = { 11, 12, 13, 13}
print(nset1)
```

SOLUTION `{11, 12, 13}`

EXAMPLE 2 *In the previous example, Why did the code not print the output exactly as the input ?*

SOLUTION This is because the sets ignore the duplicate values. Since the input contained a duplicate value 13, the set **nset1** ignored it and hence the output is different from the input.

EXAMPLE 3 *How are following two variables created using { } different from one another ?*

`V1 = { 11, 12, 13, 14}, V2 = {11:12, 13:14}`

SOLUTION Variable **V1** is a set and **V2** is a dictionary as it contains *key:value pairs*. We can confirm it by checking the types of **V1** and **V2** as :

```
>>> V1 = { 11, 12, 13, 14}
>>> V2 = {11:12, 13:14}
>>> type(V1)
<class 'set'>
>>> type(V2)
<class 'dict'>
```

DATA TYPES, MUTABILITY INTERNALS — Progress In Python 8.1

In the Python IDE of your choice

:

>>>❖<<<

8.4 OPERATORS

The operations being carried out on data, are represented by operators. The symbols that trigger the operation / action on data, are called operators. The operations(specific tasks) are represented by *Operators* and the objects of the operation(s) are referred to as *Operands*.

Python's rich set of operators comprises of these types of operators : (*i*) Arithmetic operators (*ii*) Relational operators (*iii*) Identity operators (*iv*) Logical operators (*v*) Bitwise operators (*vi*) Membership operators.

Out of these, we shall talk about membership operators later when we talk about strings, lists, tuples and dictionaries. (Chapter 10 onwards).

Let us discuss these operators in detail.

OPERATORS

The symbols[4] that trigger the operation / action on data, are called **Operators**, and the data on which operation is being carried out, *i.e.*, the objects of the operation(s) are referred to as **Operands**.

8.4.1 Arithmetic Operators

To do arithmetic, Python uses arithmetic operators. Python provides operators for basic calculations, as given below :

+	addition	*	multiplication	//	floor division	**	exponentiation
–	subtraction	/	division	%	remainder		

Each of these operators is a binary operator *i.e.*, it requires two values (operands) to calculate a final answer. Apart from these binary operators, Python provides two unary arithmetic operators (that require one operand) also, which are unary +, and unary – .

UNARY OPERATORS

The operators that act on one operand are referred to as *Unary Operators*.

8.4.1A Unary Operators

Unary +

The operators unary '+' precedes an operand. The operand (the value on which the operator operates) of the unary + operator must have arithmetic type and the result is the value of the argument.

For example,

if a = 5 then + *a* means 5.
if a = 0 then + *a* means 0.
if a = – 4 then + *a* means – 4.

Unary –

The operator unary – precedes an operand. The operand of the unary – operator must have arithmetic type and the result is the negation of its operand's value. *For example,*

if a = 5 then – *a* means – 5.
if a = 0 then – *a* means 0
(*there is no quantity known as* – 0)
if a = – 4 then – *a* means 4.

This operator reverses the sign of the operand's value.

8.4.1B Binary Operators

Operators that act upon two operands are referred to as **Binary Operators**. The operands of a binary operator are distinguished as the left or right operand. Together, the operator and its operands constitute an expression.

BINARY OPERATORS

Operators that act upon two operands are referred to as *Binary Operators*.

4. Sometimes these can be words too, *e.g.*, in Python **is** and **is not** are also operators.

1. Addition operator +

The arithmetic binary operator + adds values of its operands and the result is the sum of the values of its two operands. *For example,*

```
4 + 20                    results in  24
a + 5 (where a = 2)       results in  7
a + b (where a = 4, b = 6) results in 10
```

For Addition operator + operands may be of *number* types[5].

Python also offers + as a ***concatenation operator*** *when used with strings, lists and tuples. This functionality for strings will be covered in* ***Chapter 10*** – String Manipulation ; for lists, it will be covered in ***Chapter 12*** – List Manipulation.

2. Subtraction operator –

The – operator subtracts the second operand from the first. *For example,*

```
14 – 3                    evaluates to 11
a – b (where a = 7, b = 5) evaluates to 2
x – 3 (where x = –1)      evaluates to –4
```

The operands may be of number types.

3. Multiplication operator *

The * operator multiplies the values of its operands. *For example,*

```
3 * 4                     evaluates to 12
b * 4 (where b = 6)       evaluates to 24
p * 2 (where p = –5)      evaluates to –10
a * c (where a = 3, c = 5) evaluates to 15
```

The operands may be of *integer* or *floating point number* types.

*Python also offers * as a* ***replication operator*** *when used with strings. This functionality will be covered in* ***Chapter 10*** – String Manipulation.

4. Division operator /

The / operator in *Python 3.x* divides its *first operand* by the *second operand* and always returns the result as a **float** value, *e.g.,*

```
4/2        evaluates to  2.0
100/10     evaluates to  10.0
7/2.5      evaluates to  2.8
100/32     evaluates to  3.125
13.5/1.5   evaluates to  9.0
```

Please note that in older version of Python (2.x), the / operator worked differently.

5. Floor Division operator //

Python also offers another division operator //, which performs the floor division. The floor division is the division in which only the whole part of the result is given in the output and the fractional part is truncated.

To understand this, consider the third example of division given in division operator / , *i.e.,*

```
a = 15.9 , b = 3,
a/b evaluates to 5.3.
```

Now if you change the division operator /, with floor division operator // in above expression, *i.e.,*

```
If a = 15.9 , b = 3,
a//b will evaluate to 5.0
```

See, the Fractional part 0.3 is discarded from the actual result 5.3

Consider some more examples :

```
100//32   evaluates to  3
7//3      evaluates to  2
6.5//2    evaluates to  3.0
```

The operands may be of number types.

6. Modulus operator %

The % operator finds the modulus (*i.e.,* remainder but pronounced as *mo-du-lo*) of its first operand relative to the second. That is, it produces the remainder of dividing the first operand by the second operand. *For example,*

```
19 % 6 evaluates to 1, since 6 goes into 19
three times with a remainder 1.
```

Similarly, `7.2 % 3 will yield 1.2`

`6 % 2.5 will yield 1.0`

The operands may be of *number* types.

7. Exponentiation operator **

The exponentiation operator ** performs exponentiation (power) calculation, *i.e.,* it returns the result of a number raised to a power (exponent). *For example,*

```
4 ** 3 evaluates to 64 (4^3)
a ** b (a = 7, b = 4)
   evaluates to 2401 (a^b i.e., 7^4).
x ** 0.5 ( x = 49.0)
   evaluates to 7.0. (x^0.5 , i.e., √x, i.e., √49)
27.009 ** 0.3
   evaluates to 2.68814413570761. (27.009^0.3)
```

The operands may be of number types.

5. For Boolean values *True* and *False*, recall that Python internally takes values 1 and 0 (zero) respectively so *True + 1* will give you 2 :).

EXAMPLE 4 *What will be the output produced by the following code ?*

```
A, B, C, D = 9.2, 2.0, 4, 21
print ( (A/4), (A // 4) )
print (B ** C)
print (D // B)
print (A % C)
```

SOLUTION

```
2.3 2.0
16.0
10.0
1.2
```

EXAMPLE 5 *Print the area of a circle of radius 3.75 metres.*

SOLUTION

```
Radius = 3.75
Area = 3.14159 * Radius ** 2
print (Area, 'sq. metre')
```

NOTE

Floor division (//) truncates fractional remainders and gives only the whole part as the result.

Table 8.2 *Binary Arithmetic Operators*

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-dú-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.*,

$-5 + 3 = 2$; $-5 - 3 = -8$; $-5 * 3 = -15$; $-5 ** 3 = -125$

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) $-3 \overline{)\,5\,(-2}$ with $\frac{6}{-1}$

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

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

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

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

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

(g) $-4 \overline{)\,7\,(-2}$ with $\frac{8}{-1}$

```
>>> 5// -3
-2
>>> -5 // 3
-2
>>> -7 / 4
-1.75
>>> -7 // 4
-2
>>> -7 % 4
1
>>> 7 % -4
-1
>>> 7 // -4
-2
```

8.4.1C Augmented Assignment Operators[6]

You have learnt that Python has an assignment operator = which assigns the value specified on RHS to the variable/object on the LHS of = . Python also offers augmented assignment arithmetic operators, which combine the impact of an arithmetic operator with an assignment operator, *e.g.*, if you want to add value of *b* to value of *a* and assign the result to *a*, then instead of writing

```
a = a + b
```

you may write

```
a += b
```

To add value of *a* to value of *b* add assign the result to *b*, you may write

```
b += a                    # instead of b = b + a
```

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

These operators can be used anywhere that ordinary assignment is used. Augmented assignment doesn't violate *mutability*. Therefore, writing $x+=y$ creates an entirely new object x with the value $x+y$.

Please note that this expansion such as '$x+=y$ is same as $x=x+y$', *holds only for immutable types* such as *numbers*. For iterable mutables such as lists this is not the case (*i.e.*, for lists $x+=y$ is not equal to $x=x+y$). We shall explain this in the chapter of lists.

8.4.2 Relational Operators

In the term *relational operator*, relational refers to the relationships that values (or operands) can have with one another. Thus, the relational operators determine the relation among different operands. Python provides six relational operators for comparing values (thus also called *comparison operators*). If the comparison is true, the relational expression results into the Boolean value ***True*** and to Boolean value ***False***, if the comparison is false.

The *six* relational operators are :

<	less than,	<=	less than or equal to,	==	equal to
>	greater than,	>=	greater than or equal to,	!=	not equal to,

Relational operators work with nearly all types of data in Python, such as numbers, strings, lists, tuples etc. Relational operators work on following principles :

- For *numeric types*, the values are compared after removing trailing zeros after decimal point from a floating point number. *For example*, 4 and 4.0 will be treated as equal (after removing trailing zeros from 4.0, it becomes equal to 4 only).

6. Please note this style of combining assignment with other operator also works with bitwise operators.

- Strings are compared on the basis of lexicographical ordering (ordering in dictionary).
  - Capital letters are considered lesser than small letters, *e.g.*, 'A' is less than 'a' ; 'Python' is not equal to 'python' ; 'book' is not equal to 'books'.

  Lexicographical ordering is implemented via the corresponding codes or **ordinal values** (*e.g.*, ASCII code or Unicode code) of the characters being compared. That is the reason 'A' is less than 'a' because ASCII value of letter 'A' (65) is less than 'a'(97). You can check for ordinal code of a character yourself using **ord(<character>)** function (*e.g.*, **ord('A')**).
  - For the same reason, you need to be careful about nonprinting characters like spaces. Spaces are real characters and they have a specific code (ASCII code 32) assigned to them. If you are comparing two strings that appear same to you but they might produce a different result – if they have some spaces in the beginning or end of the string.
    See screenshot on the right.

```
>>> 'Apple' > ' Apple'
True

>>> 'Apple' == ' Apple'
True
```

Results different from anticipated results because the second string contains a space in the beginning and space s ordinal value (32) is less than A s ordinal value (65)

- Two lists and similarly two tuples are equal if they have same elements in the same order.
- Boolean *True* is equivalent to 1 (numeric one) and Boolean *False* to 0 (numeric zero) for comparison purposes.

IMPORTANT

You should not use == to compare two floating point values, the reason being that floating-point numbers are stored with some precision limit and which may result in some rounding off. All this will not yield the correct result. Thus, it is not recommended to use == operator to compare two floating point values.

For instance, consider the following relational operations.

Given $a = 3, b = 13, p = 3.0$
$c = 'n', d = 'g', e = 'N',$
$f = 'god', g = 'God', h = 'god', j = 'God', k = \text{"Godhouse"},$
$L = [1, 2, 3], M = [2, 4, 6], N = [1, 2, 3]$
$O = (1, 2, 3), P = (2, 4, 6), Q = (1, 2, 3)$

Check Point 8.4

1. What is the function of operators ? What are arithmetic operators ? Give their examples.
2. How is 'unary +' operator different from '+' operator ? How is 'unary –' operator different from '–' operator ?
3. What are binary operators ? Give examples of arithmetic binary operators.
4. What does the modulus operator % do ? What will be the result of 7.2 % 2.1 and 8 % 3 ?
5. What will be the result of 5.0/3 and 5.0//3 ?
6. How will you calculate 4.5^5 in Python ?
7. Write an expression that uses a relational operator to return *True* if the variable *total* is greater than or equal to variable *final*.

```
a < b          will return True
c < d          will return False
f < h          will return False
f == h         will return True
c == e         will return False
g == j         will return True
"God" < "Godhouse"   will return True
"god" < "Godhouse"   Will return False
a == p         will return True
L == M         will return False
L == N         will return True
O == P         will return False
O == Q         will return True
a == True      will return False
0 == False     will return True
1 == True      will return True
```

Both match up to the letter 'd' but 'God' is shorter than 'Godhouse' so it comes first in the dictionary.

The length of the strings does not matter here. Lower case g is greater than upper case G.

Table 8.3 summarizes the action of these relational operators.

Table 8.3 *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

IMPORTANT

While using floating-point numbers with relational operators, you should keep in mind that floating point numbers are **approximately** presented in memory in binary form up to the allowed precision (15 digit precision in case of Python). This approximation may yield unexpected results if you are comparing floating-point numbers especially for equality (==). Numbers such as 1/3 etc., cannot be fully represented as binary as it yields 0.3333... etc. and to represent it in binary some approximation is done internally.

Consider the following code for to understand it :

```
>>> 0.1+0.1+0.1==0.3
False
>>> print(0.1+0.1+0.1)
0.30000000000000004
>>> print(0.3)
0.3
```

Notice , Python returns False when you compare 0.1+ 0.1 +0.1 with 0.3.

See, it does not give you 0.3 when you print the result of expression 0.1+ 0.1 +0.1 ***(because of floating pt approximation)*** and this is the reason the result is False for quality comparison of 0.1+ 0.1 +0.1 and 0.3

Thus, you should avoid floating point equality comparisons as much as you can.

Relational Operators with Arithmetic Operators

The relational operators have a lower precedence than the arithmetic operators.

That means the expression

```
a+5 > c-2        ...    expression 1
```

corresponds to

```
(a+5) > (c-2)    ...    expression 2
```

and not the following

```
a+(5 > c)-2      ...    expression 3
```

Expression 1 means the *expression 2* and not the *expression 3*.

Though relational operators are easy to work with, yet while working with them, sometimes you get unexpected results and behaviour from your program. To avoid so, I would like you to know certain tips regarding relational operators.

Check Point

8.5

1. Given that $i=4$, $j=5$, $k=4$, what will be the result of following expressions ?
   (i) $i<k$ (ii) $i<j$ (iii) $i<=k$ (iv) $i==j$
   (v) $i==k$ (vi) $j>k$ (vii) $j>=i$
   (viii) $j!=i$ (ix) $j<=k$
2. What will be the order of evaluation for following expressions ?
   (i) $i+5>=j-6$ (ii) $s*10<p**2$
   (iii) $i<j+k>1-n$
3. How are following two expressions different ? (i) ans = 8 (ii) ans == 8

A very common mistake is to use the assignment operator = in place of the relational operator == . Do not confuse testing the operator == with the assignment operator (=). For instance, the expression :

```
value1 == 3
```

tests whether *value1* is equal to 3 ? The expression has the value ***True*** if the comparison is true otherwise it is ***False***. But the expression

```
value1 = 3
```

assigns 3 to *value1* ; no comparison takes place.

NOTE

In Python, arithmetic operators have higher precedence over relational operators *i.e.*, $y + x > y * 2$ would be evaluated internally as $(y + x) > (y * 2)$.

TIP

Do not confuse the = and the == operators.

8.4.3 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.

```
>>> a = 235
>>> b = 240
>>> c = 235
>>> a is b
False
>>> a is c
True
>>> print(id(a), id(b), id(c))
492124000 492124080 492124000
```

a is b returns ***False*** because **a** and **b** are referring to different objects (235 and 240)

a is c returns ***True*** because both **a** and **c** are referring to same object (235)

The **ids** (***id( )***) of **a**, **b** and **c** tell that **a** and **c** are referring to same object (their memory addresses are same) but **b** is referring to a different object as its memory address is different from the other two

Now if you change the value of *b* so that it is not referring to same integer object, then expression a is b will return *True* :

```
>>> b = b - 5
>>> a is b
True
>>> print(id(a), id(b), id(c))
492124000 492124000 492124000
```

Now b is also pointing to same integer object(235) thus a is b is giving True this time.

Their *ids* also reflect the same i.e., all **a** , **b** and **c** are referring to same memory location

The **is not** operator is opposite of the **is** operator. It returns *True* when both its operands are not referring to same memory address.

8.4.3A Equality (==) and Identity (is) – Important Relation

You have seen in above given examples that when two variables are referring to same value, the **is** operator returns **True**. When the **is** operator returns **True** for two variables, it implicitly means that the equality operator will also return **True**. That is, expression a is b as **True** means that a == b will also be **True**, always. See below :

```
>>> print(a, b)
235 235
>>> a is b
True
>>> a == b
True
```

But it is not always true other way round. That means there are some cases where you will find that the two objects are having just the same value *i.e.*, == operator returns **True** for them but the **is** operator returns **False**.

See in the screenshots shown here.

```
>>> s1 = 'abc'
>>> s2 = input("Enter a string:")
Enter a string : abc
>>> s1 == s2
True
>>> s1 is s2
False
>>> s3 = 'abc'
>>> s1 is s3
True
```

- The strings **s1** and **s2** although have the same value '**abc**' in them
- The **==** operator also returns ***True*** for s1 == s2
- But the **is** operator returns ***False*** for s1 is s2

Similarly,

```
>>> i = 2 + 3.5j
>>> j = 2 + 3.5j
>>> i is j
False
```

- Objects *i* and *j* store the same complex number value **2+3.5 j** in them
- But **is** operator returns ***False*** for **i is j**

Also,

```
>>> k = 3.5
>>> l = float(input("Enter a value:"))
Enter a value : 3.5
>>> k == l
True
>>> k is l
False
```

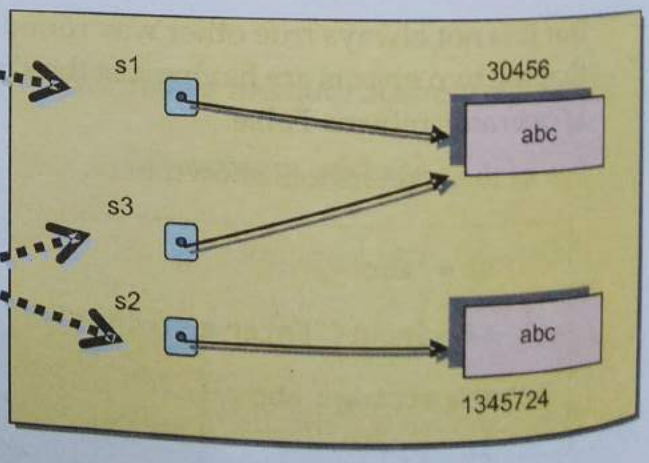
- The variables ***k*** and ***l*** both store float value **3.5** (k has been assigned 3.5 and l has taken this value through input() and float())
- But **k == l** returns **True** and **k is l** returns **False**

The reason behind this behaviour is that there are **a few cases** where **Python creates two different objects even though both store the same value**. These are :

- input of strings from the console;
- writing integers literals with many digits (very big integers);
- writing floating-point and complex literals.

Following figure illustrates one of the above given screenshots.

```
>>> s1 = 'abc'
>>> s2 = input("Enter a string:")
Enter a string : abc
>>> s1 == s2
True
>>> s1 is s2
False
>>> s3 = 'abc'
>>> s1 is s3
True
```



Most of the times we just need to check whether the two objects refer to the same value or not – in this case the equality operator (==) is sufficient for this test. However, in advanced programs or in your projects, you may need to check whether they refer to same memory address or not – in this case, you can use the **is** operator.

NOTE

If the **is** operator returns **True** for two objects then the == operator must return **True** for the same objects, *i.e.*, if two references refer to the same memory address, then their values are always same.

8.4.4 Logical Operators

An earlier section discussed about relational operators that establish relationships among the values. This section talks about logical operators, the Boolean logical operators (**or, and, not**) that refer to the ways these relationships (among values) can be connected. Python provides *three* logical operators to combine existing expressions. These are **or, and**, and **not**.

Before we proceed to the discussion of logical operators, it is important for you to know about **Truth Value Testing**, because in some cases logical operators base their results on truth value testing.

8.4.4A Truth Value Testing

Python associates with every value type, some truth value (the *truthiness*), *i.e.*, Python internally categorizes them as *true* or *false*. Any object can be tested for truth value. Python considers following values *false*, (*i.e.*, with truth-value as *false*) and *true* :

Values with truth value as false	Values with truth value as true
None	All other values are considered ***true***.
False (Boolean value False)	
zero of any numeric type, for example, 0, 0.0, 0j	
any empty sequence, for example, '', ( ), [ ] (please note, '' is empty string ; ( ) is empty tuple ; and [ ] is empty list)]	
any empty mapping, for example, { }	
The result of a relational expression can be ***True*** or ***False*** depending upon the values of its operands and the comparison taking place.	

Do not confuse between Boolean values ***True, False*** and truth values (truthiness values) *true, false*. Simply put truth-value tests for zero-ness or emptiness of a value. Boolean values belong to just one data type, *i.e.*, Boolean type, whereas we can test truthiness for every value object in Python. But to avoid any confusion, we shall be giving truth values *true* and *false* in small letters and with a subscript *tval*, *i.e.*, now on in this chapter ***true*** $_{tval}$ and ***false*** $_{tval}$ will be referring to truth-values of an object.

The utility of Truth Value testing will be clear to you as we are discussing the functioning of logical operators.

8.4.4B 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	1st expression (0) has false$_{tval}$, thus 2nd expression **0** is returned.
0 **or** 8	8	1st expression (0) has false$_{tval}$, thus 2nd expression **8** is returned.
5 **or** 0.0	5	1st expression (5) has true$_{tval}$, thus 1st expression **5** is returned.
'hello' **or** ''	'hello'	1st expression ('hello') has *true*$_{tval}$, thus 1st expression '***hello***' is returned
'' **or** 'a'	'a'	1st expression ('') has *false*$_{tval}$, thus 2nd expression '***a***' is returned.
'' **or** ''	''	1st expression ('') has *false*$_{tval}$, thus 2nd expression '' is returned.
'a' **or** 'j'	'a'	1st expression ('a') has *true*$_{tval}$, thus 1st expression '***a***' is returned.

How the truth value is determined ? – refer to section 8.4.4A above.

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.

8.4.4C 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*

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

`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*.

(iii) 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.*

That is :

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	1st expression (0) has *false* $_{tval}$, *thus first expression* **0** is returned.
0 and 8	0	1st expression (0) has *false* $_{tval}$, thus 1st expression **0** is returned.
5 and 0.0	0.0	1st expression (5) has *true* $_{tval}$, thus 2nd expression **0.0** is returned.
'hello' and ''	''	1st expression ('hello') has *true* $_{tval}$, thus 2nd expression '' is returned
'' and 'a'	''	1st expression ('') has *false* $_{tval}$, thus 1st expression '' is returned.
'' and ''	''	1st expression ('') has *false* $_{tval}$, thus 1st expression'' is returned.
'a' and 'j'	'j'	1st expression ('a') has *true* $_{tval}$, thus 2nd expression 'j' is returned.

How the truth value is determined ? – refer to section 8.4.4A.

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.

8.4.4D 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.

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

Following table summarizes the logical operators.

Table 8.4 *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**.

8.4.5 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 8.5 *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.

Let us examine them one by one.

8.4.5A 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 8.6.

Table 8.6 *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
```

```
>>> bin(13)
'0b1101'
>>> bin(12)
'0b1100'
>>> 13 & 12
12
>>> bin(13 & 12)
'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.

8.4.5B 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 8.7 shows the results of inclusive OR operations.

Table 8.7 *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
```

```
>>> bin(13)
'0b1101'
>>> bin(12)
'0b1100'
>>> bin(13 | 12)
'0b1101'
>>> 13 | 12
13
```

8.4.5C 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 8.8 shows the results of an eXclusive OR operation.

Table 8.8 *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
```

```
>>> bin(13)
'0b1101'
>>> bin(12)
'0b1100'
>>> 13 ^ 12
1
>>> bin(13 ^ 12)
'0b1'
```

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

8.4.5D 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 8.9 *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)
```

```
>>> bin(12)
'0b1100'
>>> ~12
-13
>>> bin(13)
'0b1101'
>>> bin(~12)
'-0b1101'
```

8.4.6 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.

Operator Associativity

Python allows multiple operators in a single expression as you have learnt above, *e.g.*, `a<b+2<c` or `p<q>r` etc. If the operators used in an expression have different precedence, there is not any problem as Python will evaluate the operator with higher precedence first. BUT what if the expression contains two operators that have the same precedence ?

In that case, **associativity** helps determine the order of operations.

Associativity is the order in which an expression (having multiple operators of same precedence) is evaluated. Almost all the operators have **left-to-right associativity except exponentiation (`**`), which has right-to left associativity**. That means, in case of multiple operators with same precedence, other than `**`, in same expression – the operator on the left is evaluated first and then the operator on its right and so on.

For example, multiplication operator (`*`), division operator (`/`) and floor division operator (`//`) have the same precedence. So, if we have an expression having these operators simultaneously, then the same-precedence-operators will be evaluated in left-to-right order.

For example,

```
>>> 7 * 8 / 5 // 2
5.0

>>> (((7 * 8) / 5 ) // 2 )
5.0

>>> 7 * ( (8 / 5) // 2)
0.0

>>> 7 * (8 / (5 // 2) )
28.0
```

This first expression is evaluated in left to right order of operators as evident from the 2nd expression's evaluation that clearly marks this order of evaluation

An expression having multiple `**` operators is evaluated from right to left, *i.e.*,

`2 ** 3 ** 4` will be evaluated as `2 ** (3 **4)` and NOT AS `(2**3)**4`

Consider following example :

```
>>> 3 ** 3 ** 2
19683

>>> 3 ** (3 ** 2)
19683

>>> (3 ** 3) ** 2
729
```

*See the default order of evaluation (**first expression**) **MATCHES** with the **second expression** where parentheses are added as per right-to-left associativity order and **NOT LIKE third expression** that has parentheses from left-to-right order, because exponentiation (`**`) has right-to-left associativity*

NOTE

Associativity is the order in which an expression having multiple operators of same precedence, is evaluated.

All operators are left associative except ***exponentiation*** *(right associative)*.

EXAMPLE 6 *What will be the output produced by the following code ?*

```
x, z = 5, 10
y = x + 3
x = x - 1
x = x + z
print('x:', x, 'y:', y, 'z:', z)
```

SOLUTION

```
x: 14 y: 8 z: 10
```

EXAMPLE 7 *What will be the output produced by the following code ?*

```
print(14//4, 14%4, 14/4)
```

SOLUTION

```
3  2  3.5
```

EXAMPLE 8 *What will be the output produced by the following code ?*

```
print(2*'No' + 3*'!')
print(2 * ('No' + 3*'!'))
```

SOLUTION

```
NoNo!!!
No!!!No!!!
```

EXAMPLE 9 *What will be the output produced by the following code ?*

```
print(type(1 + 3))
print(type(1 + 3.0))
```

SOLUTION

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

EXAMPLE 10 *What will be the output produced by the following code ?*

```
print (11 + 3)
print ('11' + '3')
```

SOLUTION

```
14
113
```

EXAMPLE 11 *Why is the following code giving error ?*

```
print (11 + 3)
print ('11' + 3)
```

SOLUTION

The problem is with line 2 of given code because an integer cannot be added to a string value, thus Python will give :

```
TypeError: can only concatenate str (not "int") to str
```

EXAMPLE 12 *Which of the following expressions will yield an* ***integer*** *type value as its output ?*

(*i*) 5 * 2 (*ii*) 5 ** 2
(*iii*) '5' + '2' (*iv*) '5' * 2
(*v*) 5 / 2 (*vi*) 5 // 2
(*vii*) 5 % 2 (*viii*) 5 + 2.0
(*ix*) 5.0 * 2.0 (*x*) '5' – 2

SOLUTION

Expressions (*i*), (*ii*), (*vi*), (*vii*) will yield **integer** type result.

EXAMPLE 13 *Considering the expressions given in the previous example, which expressions will yield a floating point result ?*

SOLUTION

Expressions (*v*), (*viii*), (*ix*) will yield a **floating point** result.

EXAMPLE 14 *Considering the expressions given in the previous example 12, which expressions will yield a* ***String*** *result ?*

SOLUTION

Expressions (*iii*), (*iv*) will yield a **String** result.

EXAMPLE 15 *Considering the expressions given in the previous example 10, which expressions will result into an Error and why ?*

SOLUTION

Expression (*x*) will result into error because the minus operator is not a valid operator for string and integer values.

OPERATORS IN PYTHON

PriP Progress In Python 8.2

This practical session aims at strengthening operators' concepts. It involves both interactive mode and script mode. For better understanding of the concepts, it would be better if you first perform the *interactive mode practice questions* followed by *script mode practice questions*.

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 8.2 under Chapter 8 after practically doing it on the computer.

>>>❖<<<

Check Point 8.6

1. What is the function of logical operators ? Write an expression involving a logical operator to test if *marks* are 55 and *grade* is '*B*'.
2. What is the order of evaluation in the following expressions :
   (*i*) a > b or b <= d ?
   (*ii*) x == y and y >= m ?
   (*iii*) a > b < c > d
3. What is the result of following expression : $a >= b$ and $(a + b) > a$ if
   (*i*) $a = 3$, $b = 0$ (*ii*) $a = 7$, $b = 7$?
4. What is the result of following expressions (*a* to *e*) if
   (*i*) check = 3, mate = 0.0
   (*ii*) check = 0, mate = −5
   (*iii*) check = ' ', mate = 'e'
   (*iv*) check = 'meera', mate = ' '
   (*a*) check or mate (*b*) mate or check
   (*c*) check and not mate
   (*d*) check and mate (*e*) mate and check
   Evaluate each of the above expressions for all four sets of values.
5. Identify the order of evaluation in the following expression :
   4 * 5 + 7 * 2 – 8 % 3 + 4 and
   5.7 // 2 – 1 + 4 or not 2 == 4 and not 2 ** 4 > 6 * 2
6. a = 15.5, b = 15.5 then why is **a is b** is False while **a == b** is True ?

LET US REVISE

- *Immutable Types in Python mean that their values cannot be changed in place.*
- *Mutable Types Python mean that their values can be changed in place.*
- *Python has only 3 mutable type : **lists**, **dictionaries** and **sets**.*
- *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**.*

8.5 EXPRESSIONS

An expression in Python is any valid combination of *operators*, *literals* and *variables*. An expression is composed of one or more *operations*, with *operators*, *literals*, and *variables* as the constituents of expressions.

Python puts it in this way : a valid combination of **atoms** and **operators** forms a Python expression. In simplest words, **an *atom* is something that has a value**. So all of these are atoms in Python : *identifiers*, *literals* and *values-in-enclosures* such as quotes (''), parentheses, brackets, etc. *i.e.*, *strings*, *tuples*, *lists*, *dictionaries*, *sets* etc.

The expressions in Python can be of any type : *arithmetic expressions*, *string expressions*, *relational expressions*, *logical expressions*, *compound expressions* etc.

ATOM

An atom is something that has a value. Identifiers, literals, strings, lists, tuples, sets, dictionaries etc. are all atoms.

EXPRESSION

An *expression* in Python is any valid combination of operators and atoms. An expression is composed of one or more *operations*.

- The types of operators and operands used in an expression determine the expression type.
- An expression can be **compound expression** too if it involves multiple types of operators, *e.g.*, $a + b > c ** d$ or $a * b < c * d$ is a *compound expression* as it involves *arithmetic* as well as *relational* as well as *logical* operators.

Let us talk about these one by one.

1. Arithmetic Expressions

Arithmetic expressions involve numbers (integers, floating-point numbers, complex numbers) and arithmetic operators.

2. Relational Expressions

An expression having literals and/or variables of any valid type and relational operators is a relational expression. *For example*, these are valid relational expressions :

```
x > y,  y <= z,  z <> x,  z == q,  x < y > z,  x == y <> z
```

3. Logical Expressions

An expression having literals and/or variables of any valid type and logical operators is a logical expression. *For example*, these are valid logical expressions :

```
a or b ,  b and c,  a and not b ,  not c or not b
```

4. String expressions

Python also provides two string operators + and *, when combined with string operands and integers, form string expressions.

- With operator +, *the concatenation operator*, the operands should be of string type only.
- With * operator, *the replication operator*, the operands should be one string and one integer.

For instance, following are some legal string expressions :

```
"and" + "then"      # would result into 'andthen' - concatenation
"and" * 2           # would result into 'andand' - replication
```

String manipulation is being covered in a separate chapter – ***chapter 10***.

8.5.1 Evaluating Expressions

In this section, we shall be discussing how Python evaluates different types of expressions : arithmetic, relational and logical expressions. String expressions, as mentioned earlier will be discussed in a separate chapter – ***chapter 10***.

8.5.1A Evaluating Arithmetic Expressions

You all are familiar with arithmetic expressions and their basic evaluation rules, right from your primary and middle-school years. Likewise, Python also has certain set of rules that help it evaluate an expression. Let's see how Python evaluates them.

Evaluating Arithmetic Expressions

To evaluate an arithmetic expression (with operator and operands), Python follows these rules :

- Determines the order of evaluation in an expression considering the operator precedence.
- As per the evaluation order, for each of the sub-expression (generally in the form of **<value><operator><value>** *e.g.*, 13% 3)
  - Evaluate each of its operands or arguments.
  - Performs any implicit conversions (*e.g.*, promoting **int** to **float** or **bool** to **int** for arithmetic on mixed types). For implicit conversion rules of Python, read the text given after the rules.
  - Compute its result based on the operator.
  - Replace the subexpression with the computed result and carry on the expression evaluation.
  - Repeat till the final result is obtained.

Implicit type conversion (Coercion). An implicit type conversion is a conversion performed by the compiler without programmer's intervention. An implicit conversion is applied generally whenever differing data types are intermixed in an expression (mixed mode expression), so as not to lose information.

In a mixed arithmetic expression, Python converts all operands up to the type of the largest operand (***type promotion***). In simplest form, an expression is like ***op1 operator op2*** (*e.g.*, *x*/*y* or *p* ** *a*). Here, if both arguments are standard numeric types, the following coercions are applied :

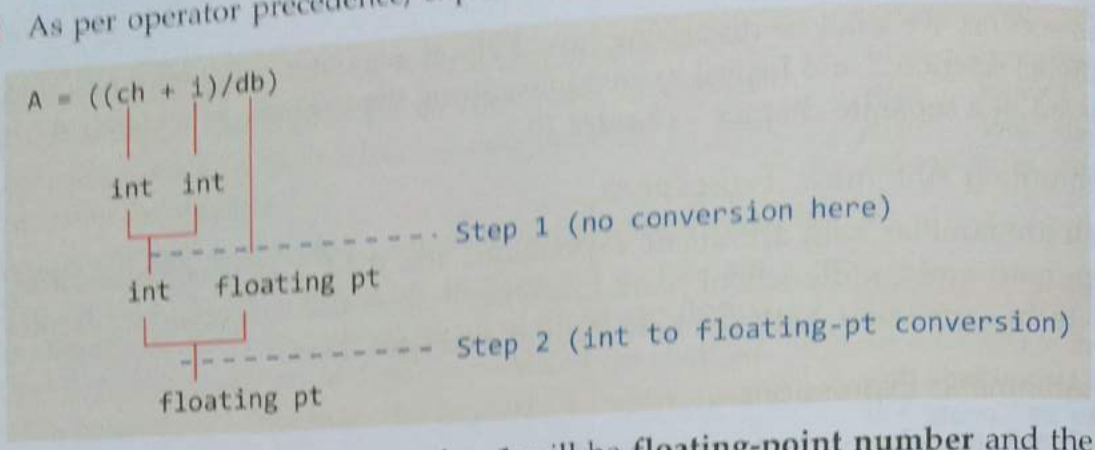
- If either argument is a complex number, the other is converted to complex ;
- Otherwise, if either argument is a floating point number, the other is converted to floating point ;
- No conversion if both operands are integers.

To understand this, consider the following example, which will make it clear how Python internally coerces (*i.e.*, promotes) data types in a mixed type arithmetic expression and then evaluates it.

EXAMPLE 16 *Consider the following code containing mixed arithmetic expression. What will be the final result and the final data type ?*

```
ch = 5                  # integer
i = 2                   # integer
fl = 4                  # integer
db = 5.0                # floating point number
fd = 36.0               # floating point number
A = (ch + i) / db       # expression 1
B = fd /db * ch /2      # expression 2
print (A)
print (B)
```

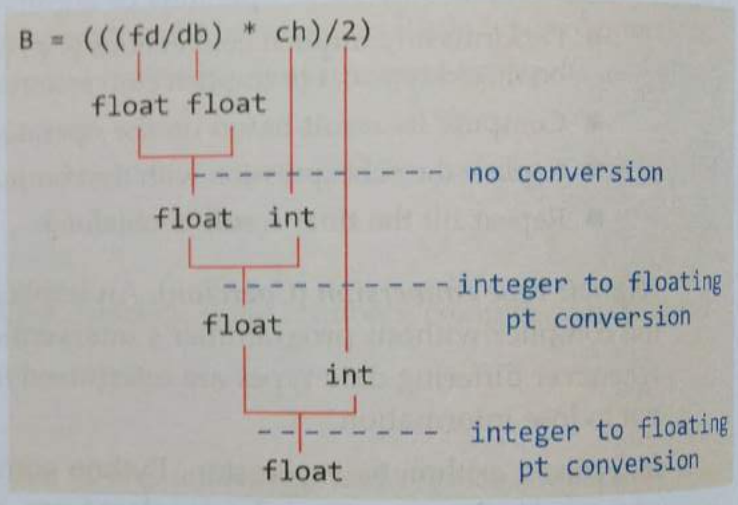
SOLUTION As per operator precedence, expression 1 will be internally evaluated as :



So overall, final datatype for **expression 1** will be **floating-point number** and the expression will be evaluated as :

((ch + i) / db)
((5 + 2)) / 5.0
((5 + 2)) / 5.0
= **(7)** / 5.0
[*int to floating point conversion*]
= 7.0 / 5.0
A = **1.4**

As per operator precedence, expression 2 will be internally evaluated as :



So, final datatype for expression 2 will be **floating point number**.

The expression, expression 2 will be evaluated as :

(((*fd* / *db*) * ch) / 2)
= (((**36.0 / 5.0**) * 5*L* / 2)
[*no conversion required*]
= (**(7.2 * 5)** / 2)
[*int to floating point conversion*]
= (**(7.2 * 5.0)** / 2)
= (36.0 / 2)
[*integer to floating point conversion*]
= 36.0 / 2.0
B = 18.0

The output will be

```
1.4
18.0
```

The final datatype of expression 1 will be **floating point number** and of expression 2, it will be **floating-point number**.

IMPORTANT In Python, if the operator is the division operator (/), the result will always be a floating point number, even if both the operands are of integer types (an *exception* to the rule). Consider following example that illustrates it.

EXAMPLE 17 *Consider below given expressions what. Will be the final result and final data type ?*

(a)
```
a, b = 3, 6
c = b/a
```

(b)
```
a, b = 3, 6
c = b // a
```

(c)
```
a, b = 3, 6
c = b % a
```

SOLUTION (a) In expression

```
c = 6/3
c = 2.0
```

```
b   /   a
|       |
int     int
   floating pt
```

Here, the operator is /, which always gives floating pt result.

(b) In expression

```
c = 6 // 3
c = 2
```

```
b   //  a
|       |
int     int
   int
```

For other division related operations, // and %, if both operands are integers, result will be integer.

(c) In expression

```
c = 6 % 3
c = 0
```

```
b   %   a
|       |
int     int
   int
```

You, yourself, can run these expressions in Python shell and then check the type of C using type (C) function.

8.5.1B Evaluating Relational Expressions (Comparisons)

All comparison operations in Python have the same priority, which is lower than that of any arithmetic operations. All relational expressions (comparisons) yield Boolean values only *i.e.*, *True* or *False*.

Further, chained expressions like $a < b < c$ have the interpretation that is conventional in mathematics *i.e.*, comparisons in Python are chained arbitrarily, *e.g.*, $a < b < c$ is internally treated as $a < b$ and $b < c$.

For chained comparisons like $x < y <= z$ (which is internally equivalent to $x < y$ and $y <= z$), the common expression (the middle one, y here) is evaluated only once and the third expression (z here) is not evaluated at all when first comparison ($x < y$ here) is found to be *False*.

EXAMPLE 18 *What will be the output of following statement when the inputs are :*

(i) $a = 10$, $b = 23$, $c = 23$ (ii) $a = 23$, $b = 10$, $c = 10$

```
print (a < b)
print (b <= c)
print (a < b <= c)
```

SOLUTION For input combination (i), the output would be :

```
True
True
True
```

For input combination (ii), the output would be :

```
False
True
False
```

EXAMPLE 19 *How would following relational expressions be internally interpreted by Python ?*

(i) $p > q < y$ (ii) $a <= N <= b$

SOLUTION (i) $(p > q)$ and $(q < y)$ (ii) $(a <= N)$ and $(N <= b)$

8.5.1C Evaluating Logical Expressions

Recall that the use of logical operators **and, or** and **not** makes a logical expression. While evaluating logical expressions, Python follows these rules :

(*i*) The precedence of logical operators is lower than the arithmetic operators, so constituent arithmetic sub-expression (if any) is evaluated first and then logical operators are applied, *e.g.*,

`25/5 or 2.0 + 20/10` will be first evaluated as : `5 or 4.0`

So, the overall result will be 5. (For logical operators' functioning, refer to section 8.4.3)

(*ii*) The precedence of logical operators among themselves is **not, and, or**. So, the expression *a* or *b* and not *c* will be evaluated as :

`(a or (b and (not c) ))`

Similarly, following expression *p* and *q* or not *r* will be evaluated as : `((p and q) or (not r))`

(*iii*) **Important.** While evaluating, Python minimizes internal work by following these rules :

(*a*) In **or** evaluation, Python only evaluates the second argument if the first one is *false* $_{tval}$.

(*b*) In **and** evaluation, Python only evaluates the second argument if the first one is *true* $_{tval}$.

For instance, consider the following examples :

- In expression **(3 < 5) or (5 < 2)**, since first argument (3 < 5) is ***True***, simply its (first argument's) result is returned as overall result ; the **second argument (5 < 2) will not be evaluated at all.**
- In expression **(5 < 3) or (5 < 2)**, since first argument (5 < 3) is ***False***, it will now evaluate the second argument (5 < 2) and its (second argument's) result is returned as overall result.
- In expression **(3 < 5) and (5 < 2)**, since first argument (3 < 5) is ***True***, it will now evaluate the second argument (5 < 2) and its (second argument's) result is returned as overall result.
- In expression **(5 < 3) and (5 < 2)**, since first argument (5 < 3) is ***False***, simply its (first argument's) result is returned as overall result ; the second argument (5 < 2) will not be evaluated at all.

Check Point

8.7

Evaluation of expression and type conversion

1. What is an expression ? How many different types of expressions can you have in Python ?
2. What is atom in context of expression ?
3. From the following expression, identify atoms and operators.

   `str(a + b > c + d > e + f or not g-h)`
4. What is type conversion (coercion) ? How does Python perform it ?

EXAMPLE 20 *What will be the output of following expression ?*

`(5 < 10) and (10 < 5) or (3 < 18) and not 8 < 18`

SOLUTION False

EXAMPLE 21 ***'Divide by zero'*** *is an undefined term. Dividing by zero causes an error in any programming language, but when following expression is evaluated in Python, Python reported no error and returned the result as* ***True****. Could you tell, why ?*

`(5 < 10 ) or (50 < 100/0)`

SOLUTION

In or evaluation, firstly Python tests the first argument, *i.e.*, 5 < 10 here, which is ***True***. In or evaluation, Python does not evaluate the second argument if the first argument is ***True*** and returns the result of first argument as the result of overall expression.

So, for the given expression, the second argument expression (50 < 100/0) is NOT EVALUATED AT ALL. That is why, Python reported no error, and simply returned ***True***, the result of first argument.

8.5.2 Type Casting

You have learnt in earlier section that in expression with mixed types, Python internally changes the data type of some operands so that all operands have same data type. This type of conversion is automatic, *i.e.*, implicit and hence known as implicit type conversion. Python, however, also supports explicit type conversion.

TYPE CASTING

The explicit conversion of an operand to a specific type is called type casting.

Explicit Type Conversion

An explicit type conversion is user-defined conversion that forces an expression to be of specific type. The explicit type conversion is also known as **Type Casting**.

Type casting in Python is performed by <type>( ) function of appropriate data type, in the following manner :

```
<datatype> (expression)
```

where <datatype> is the data type to which you want to type-cast your expression.

For example, if we have (*a* = 3 and *b* = 5.0), then

```
int(b)
```

will cast the data-type of the expression as **int**.

Similarly,

```
d = float(a)
```

will assign value 3.0 to *d* because ***float(a)*** cast the expression's value to ***float*** type and then assigned it to *d*.

Python offers some conversion functions that you can use to type cast a value in Python. These are being listed in following Table 8.10.

Check Point 8.8

1. Are the following expressions equal ? Why/why not ?
   x or y and not z
   (x or y) and (not z))
   (x or (y and (not z)))
2. State when would only first argument be evaluated and when both first and second arguments are evaluated in following expressions if *a* = 5, *b* = 10, *c* – 5, *d* = 0.
   (*i*) *b* > *c* and *c* > *d*
   (*ii*) *a* <= *b* or *c* <= *d*
   (*iii*) (*b* + *c*) <= *a* and not (*c* < *a*)
   (*iv*) *b* < *d* and *d* < *a*
3. What is Type casting ?
4. How is implicit type conversion different from explicit type conversion ?
5. Write conversion function you would use for following type of conversions.
   (*i*) Boolean to string
   (*ii*) integer to float
   (*iii*) float to integer
   (*iv*) string to integer
   (*v*) string to float
   (*vi*) string to Boolean
   (*vii*) integer to complex

EXPRESSION EVALUATION — Progress In Python 8.3

This is an important practical session to reinforce the concepts of expression evaluation in Python.

>>>❖<<<

Table 8.10 *Python Data Conversion Functions*

S. No.	Conversion From	Conversion To	Conversion Function	Examples
1.	any number-convertible type *e.g.*, a float, a string having digits	integer	int( )	**int(7.8)** will give 7 (floating point number to integer conversion) **int('34')**[10] will give 34 (string to integer conversion)
2.	any number-convertible type *e.g.*, a float, a string having digits	floating point number	float( )	**float(7)** will give **7.0** (integer to floating point number conversion) **float('34')** will give **34.0** (string to floating point number conversion)
3.	numbers	complex number	complex( )	**complex(7)** will give $7+0j$ (ONE ARGUMENT- integer to complex number conversion) **complex(3, 2)** will give $3+2j$ (TWO ARGUMENTS – integer to complex number conversion)
4.	number Booleans	string	str( )	**str(3)** will give '3' (integer to string conversion) **str(5.78)** will give '5.78' (floating-point number to string conversion) **str(0o17)** will give '15' (octal number to string conversion ; string converts the equivalent decimal number to string : 0o17 = 15) **str(1 + 2j)** will give '(1+2j)' (complex number to string conversion) **str(True)** will give 'True' (Boolean to string conversion)
5.	any type	Boolean	bool( )	**bool(0)** will give **False** ; **bool(0.0)** will give **False** **bool(1)** will give **True** ; **bool(3)** will give **True** **bool('')** will give **False** ; **bool('a')** will give **True** **bool('hello')** will give **True** With **bool( )**, non-zero, non-empty values of any type will give **True** and rest (zero, empty values) will give **False**.

10. If a number (in string form) is given in any other base *e.g.*, octal or hexadecimal or binary, it can also be converted to integers using **int( )** as int (<number-in-string-form>, base). *For example*, to convert a string **'0o11'** (octal bases equivalent of) into integer, you can write **int('0o10', 8)** and it will give 8.
If you want to convert an integer to octal or hexadecimal or binary form then oct( ), hex( ) or bin( ) functions respectively are there but they produce the equivalent number in string form *i.e.*, hex(10) will give you '0x A'. This value can be displayed or printed but cannot be used in calculations as it is not number. However, by combining hex( ), oct( ), bin( ) with int (<number string>, base) you can convert to appropriate type.

Type Casting Issues

Assigning a value to a type with a greater range (*e.g.*, from *short* to *long*) poses no problem, however, assigning a value of larger data type to a smaller data type (*e.g.*, from *floating-point* to *integer*) may result in losing some precision.

Floating-point type to integer type conversion results in loss of fractional part. Original value may be out of range for target type, in which case result is undefined.

EXAMPLE 22 *Which of the following expressions will yield a* **Boolean** *type value as its output ?*

(*i*) 10 > 2 (*ii*) 2 != 4 (*iii*) (2+5, 6+6) (*iv*) 5 > 2 and not (10 > 11)
(*v*) [2, 4, [3, 4]] (*vi*) 4 == 2**2 (*vii*) 4, 2**2 (*viii*) 'cat' < 'dog'
(*ix*) 'Cat' < 'dog' (*x*) 5 > 2 and 10 > 11 (*xi*) 5 > 2 or 10 > 11

SOLUTION Expressions (*i*), (*ii*), (*iv*), (*vi*), (*viii*) to (*xi*) will yield a **Boolean** result.

EXAMPLE 23 *Write Boolean expressions in Python for these conditions :*

(*i*) *x* is a factor of *y* (that is, *x* divides evenly into *y*)
(*ii*) ***age*** is at least 18 and ***state*** equals 'Goa'
(*iii*) the string **name** is not 'Nimra'

SOLUTION

(*i*) x % y == 0
(*ii*) age >= 18 and state == 'Goa'
(*iii*) name != 'Nimra'

8.6 INTRODUCTION TO PYTHON STANDARD LIBRARY MODULES

Other than built-in functions, standard library also provides some modules having functionality for specialized actions. A Python module is a file which contains some variables and constants, some functions, objects etc. defined in it, which can be used in other Python programs. In order to use a module, you need to **first import the module** in a program and then you can use the module functions, variables, constants and other objects in your program file.

Let us learn to use some such modules. In the following lines we shall talk about how to use *random* and *statics* modules of Python's standard library.

8.6.1 Working with math Module of Python

Other than the built-in functions, Python makes available many more functions through modules in its standard library. Python's standard library is a collection of many modules for different functionalities, *e.g.*, module *time* offers time related functions ; module *string* offers functions for string manipulation and so on.

Python's standard library provides a module namely **math** for math related functions that work with all number types except for complex numbers.

In order to work with functions of **math** module, you need to first *import* it to your program by giving statement as follows as the top line of your Python script :

```
import math
```

Then you can use **math** library's functions as math.<function-name>. *Conventionally (not a syntactical requirement), you should give import statements at the top of the program code.*

Following table (Table 8.11) lists some useful math functions that you can use in your programs.

Table 8.11 *Some Mathematical Functions in math Module*

S. No.	Function	Prototype (General Form)	Description	Example
1.	ceil	math.ceil(num)	The **ceil( )** function returns the smallest integer not less than *num*.	math.ceil(1.03) gives 2.0 math.ceil(–1.03) gives –1.0.
2.	sqrt	math.sqrt (num)	The **sqrt( )** function returns the square root of *num*. If num <0, domain error occurs.	math.sqrt(81.0) gives 9.0.
3.	exp	math.exp(arg)	The **exp( )** function returns the natural logarithm e raised to the *arg* power.	math.exp(2.0) gives the value of e^2.
4.	fabs	math.fabs (num)	The **fabs( )** function returns the absolute value of *num*.	math.fabs(1.0) gives 1.0 math.fabs(–1.0) gives 1.0.
5.	floor	math.floor (num)	The **floor( )** function returns the largest integer not greater than *num*.	math.floor(1.03) gives 1.0 math.floor(–1.03) gives –2.0.
6.	log	math.log (num, [base])	The **log( )** function returns the natural logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument *num* is zero.	math.log(1.0) gives the natural logarithm for 1.0. math.log(1024, 2) will give logarithm of 1024 to the base 2.
7.	log10	math.log10 (num)	The **log10( )** function returns the base 10 logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument is zero.	math.log10(1.0) gives base 10 logarithm for 1.0.
8.	pow	math.pow (base, exp)	The **pow( )** function returns *base* raised to *exp* power *i.e,. base exp*. A domain error occurs if *base* =0 and *exp* <=0 ; also if *base* <0 and *exp* is not integer.	math.pow(3.0, 0) gives value of 3^0. math.pow(4.0, 2.0) gives value of 4^2.
9.	sin	math.sin(arg)	The **sin()** function returns the sine of *arg*. The value of *arg* must be in radians.	math.sin(val) (*val* is a number).
10.	cos	math.cos(arg)	The **cos( )** function returns the cosine of *arg*. The value of *arg* must be in radians.	math.cos(val) (*val* is a number).
11.	tan	math.tan(arg)	The **tan( )** function returns the tangent of *arg*. The value of *arg* must be in radians.	math.tan(val) (val is a number)
12.	degrees	math.degrees(*x*)	The **degrees( )** converts angle *x* from radians to degrees.	math.degrees(3.14) would give 179.91
13.	radians	math.radians(*x*)	The **radians( )** converts angle *x* from degrees to radians.	math.radians(179.91) would give 3.14

The **math** module of Python also makes available two useful constants namely **pi** and **e**, which you can use as :

math.pi gives the mathematical constant $\pi = 3.141592\ldots$, to available precision.

math.e gives the mathematical constant $e = 2.718281\ldots$, to available precision.

Following are examples of **valid** arithmetic expressions (after **import math** statement) :

Given : $a = 3,\ b = 4,\ c = 5,\ p = 7.0,\ q = 9.3,\ r = 10.51,\ x = 25.519,\ y = 10\text{-}24.113,\ z = 231.05$

(*i*) math.pow(a / b, 3.5)

(*ii*) math.sin(p / q) + math.cos($a - c$)

(*iii*) x / y + math.floor($p * a$ / b)

(*iv*) (math.sqrt (b) * a) − c

(*v*) (math.ceil (p) + a) * c

Following are examples of **invalid** arithmetic expressions :

(*i*) $x + * r$ — two operators in continuation.

(*ii*) $q(a + b - z / 4)$ — operator missing between q and a.

(*iii*) math.pow(0, −1) — Domain error because if base =0 then exp should not be <=0.

(*iv*) math.log(−3)+ p / q — Domain error because logarithm of a negative number is not possible.

EXAMPLE 24 *Write the corresponding Python expressions for the following mathematical expressions :*

(*i*) $\sqrt{a^2 + b^2 + c^2}$

(*ii*) $2 - ye^{2y} + 4y$

(*iii*) $p + \dfrac{q}{(r+s)^4}$

(*iv*) $(\cos x / \tan x) + x$

(*v*) $\left| e^2 - x \right|$

SOLUTION

(*i*) math.sqrt ($a * a + b * b + c * c$)

(*ii*) $2 - y *$ math.exp($2 * y$) + $4 * y$

(*iii*) $p + q$ / math.pow(($r + s$), 4)

(*iv*) (math.cos (x) / math.tan (x)) +x

(*v*) math.fabs (math.exp(2) −x)

EXAMPLE 25 *The radius of a sphere is 7.5 metres. Write Python script to calculate its area and volume. (Area of a sphere = πr^2 Volume of a sphere = $4\pi r^3$)*

SOLUTION

```
import math
r = 7.5
area = math.pi * r * r
volume = 4 * math.pi * math.pow(r, 3)
print("Radius of the sphere : ", r , "metres")
print("Area of the sphere : ", area, "units square")
print("Volume of the sphere : ", volume, "units cube")
```

Output :

```
Radius of the sphere :  7.5 metres
Area of the sphere :  176.71458676442586 units square
Volume of the sphere :  5301.437602932776 units cube
```

8.6.2 Using random Module

Python has a module namely **random** that provides random-number[11] generators. A random number in simple words means – *a number generated by chance, i.e.,* randomly.

To use random number generators in your Python program, you first need to import module ***random*** using any import command, *e.g.,*

```
import random
```

Three most common random number generator functions in ***random module*** are :

random( )	it returns a random floating point number N in the range [0.0, 1.0), *i.e.,* $0.0 \le N < 1.0$. Notice that the number generated with *random( )* will always be less than 1.0. (only lower range-limit is inclusive). Remember, it generates a floating point number.
randint(*a, b*)	it returns a random integer N in the range (a, b), *i.e.,* $a \le N \le b$ (both range-limits are inclusive). Remember, it generates an integer.
randrange (<start>, <stop>, <step>)	it returns random numbers from range start...stop with step value.

Let us consider some examples. In the following lines we are giving some sample codes along with their output.

1. To generate a random floating-point number between 0.0 to 1.0, simply use **random( )** :

```
>>> import random
>>> print(random.random())
0.022353193431
```

The output generated is between range [0.0, 1.0)

2. To generate a random floating-point number between range ***lower to upper*** using **random( )** :
   (*a*) multiply **random( )** with difference of upper limit with lower limit, *i.e.,* (upper – lower)
   (*b*) add to it lower limit

   For example, to generate between 15 to 35 , you may write :

```
>>> import random      # need not re-write this command, if random
                       # module already imported
>>> print(random.random()* (35 -15 ) + 15)
28.3071872734
```

The output generated is floating point number between range 15 to 35

3. To generate a random integer number in range 15 to 35 using *randint( )*, write :

```
>>> print(random.randint(15, 35))
16
```

The output generated is integer between range 15 to 35

Using randrange( ) function

The function randrange( ) can be used in following *three* ways :

(*i*) random.randrange(<stopvalue>) to generate a random number in the range 0 to <stopvalue>, *e.g.,* following code will generate a random number from 0 to 45

```
>>> random.randrange(45)
13
```

A random number generated in the range 0..45

11. In fact, pseudo-random numbers are generated because it is generated via some algorithm or procedure and hence deterministic somewhere.

(ii) random.randrange(<start>, <stop>) to generate a random number in the range <start> to <stop>, *e.g.*, following code will generate a random number in the range 11 to 45

```
>>> random.randrange(11, 45)
25
```
← *A random number generated in the range 11..45*

(iii) random.randrange(<start>, <stop>, <step>) to generate a random number in the range <start> to <stop>, but here, the difference between two such generated random numbers will be a multiple of <step> value. For example, following code will generate a random number in the range 11 to 45, with a step value 4. That means, the possible random numbers that may be generated will be one of the values 11, 15, 19, 23, 27, 31, 35, 39, 43

```
>>> random.randrange(11, 45, 4)
15
>>> random.randrange(11, 45, 4)
35
>>> random.randrange(11, 45, 4)
39
```
← *See each generated random number is one of the above given values*

So, internally, randarange with a <step> value creates a series from <start> to <stop> with each value at <step> values apart and randomly picks one from this.

EXAMPLE 26 *What could be the minimum possible and maximum possible numbers by following code ?*

```
import random
print(random.randint(3, 10) - 3)
```

SOLUTION. minimum possible number = 0

maximum possible number = 7

Because,

- randint(3, 10) would generate a random integer in the range 3 to 10
- subtracting 3 from it would change the range to 0 to 7 (because if ***randint(3,10)*** generates 10 then –3 would make it 7 ; similarly, for lowest generated value 3, it will make it 0)

EXAMPLE 27 *What will the following code produce ? Discuss.*

```
import random
print(random.random() * 100)
print(random.random())
```

SOLUTION

The given code will firstly print a random number generated in the range 0.0 to 100.0.

And in the next line, it will print a random number generated in the range 0.0 to 1.0.

Output can be somewhat like

```
77.41547442568118
0.03735925715458066
```

Every time the code is run, a different output will be printed as different random numbers will be generated.

EXAMPLE 28 *Write a code fragment to generate a random floating number between 45.0 and 95.0. Print this number along with its nearest integer greater than it.*

SOLUTION

```
import random
import math
fnum = random.random( ) * (95 - 45) + 45
inum = math.ceil( fnum )
print("Random numbers between 45..95:")
print(fnum)
print("Nearest higher integer :", inum )
```

Output :

```
Random numbers between 45..95 :
48.24212504903489
Nearest higher integer : 49
```

EXAMPLE 29 *Write a code fragment to generate two random integers between 450 and 950. Print these numbers along with their average.*

SOLUTION

```
import random
num1 = random.randint(450, 950) - 450
num2 = random.randint(450, 950) - 450
avg = (num1 + num2)/2
print("Random integers in the\
        range 450 to 950 :", num1, num2)
print("Their average :", avg )
```

Output :

```
Random integers in the range 450 to 950 : 472 145
Their average : 308.5
```

EXAMPLE 30 *Write a code fragment to generate three random integers in the range 10, 70 with a step of 13. Create a set with these numbers.*

SOLUTION

```
import random
num1 = random.randrange(10, 70, 13)
num2 = random.randrange(10, 70, 13)
num3 = random.randrange(10, 70, 13)
set1 = {num1, num2, num3}
print("Random integers in the range 10..70,\
        step-value 13 :", num1, num2, num3)
print("Set created :", set1)
```

Output :

```
Random integers in the range 10..70,
step-value 13 : 10   10   62
Set created : {10, 62}
```

8.6.3 Using the statistics Module

The *statistics* module of the Python Standard Library provides many statistics functions such as *mean( ), median( ), mode( )* etc. In order to use these in your program, you need to first import the Python *statistics* module by giving one of the following statements :

```
import statistics
```

Or ← *Import full statistics module*

```
from statistics import mean, median, mode
```

← *Import only three functions from statistics module*

You can then use these functions as given below :

(*i*) statistics.mean(<seq>) it returns the average value of the set/sequence of values passed.

(*ii*) statistics.median(<seq>) it return the middle value of the set/sequence of values passed.

(*iii*) statistics.mode (<seq>) it returns the most often repeated value of the set/sequence of values passed.

Consider following examples :

```
>>> import statistics
>>> seq = [5, 6, 7, 5, 6, 5, 5, 9, 11, 12, 23, 5]
>>> statistics.mean(seq)
8.25
>>> statistics.median(seq)
6.0
>>> statistics.mode(seq)
5
```

← *See, mean, median and mode of above sequence calculated using statistics module*

EXAMPLE 31 *Given a list containing these values [22, 13, 28, 13, 22, 25, 7, 13, 25]. Write code to calculate mean, median and mode of this list.*

SOLUTION

```
import statistics as stat
list1 = [ 22, 13, 28, 13, 22, 25, 7, 13, 25]
list_mean = stat.mean(list1)
list_median = stat.median(list1)
list_mode = stat.mode(list1)
print("Given list :", list1)
print("Mean :", list_mean)
print("Median :", list_median)
print("Mode :", list_mode)
```

Output :

```
Given list : [22, 13, 28, 13, 22, 25, 7, 13, 25]
Mean : 18.666666666666668
Median : 22
Mode : 13
```

EXAMPLE 32 *Have a look at the below given Heron's formula to calculate the area of a triangle through its three sides a, b, and c.*

$$s = \frac{a+b+c}{2} \; ; \; Area = \sqrt{s(s-a)(s-b)(s-c)}$$

Which Python module would you need to import to calculate area using this formula ? Can you use statistics module's ***mean( )*** *for calculating s in above given formula ?*

SOLUTION

Python's **math module** should be imported in order to calculate the square root using **sqrt( )**.

No, we cannot use **mean( )** function of **statistics** module because *s* is not the **mean** of ***a, b*** and ***c***; it is the half of the sides' sum.

8.7 DEBUGGING

DEBUGGING

Debugging refers to the process of locating the place of error, cause of error, and correcting the code accordingly.

Debugging in simple English means to remove 'bugs' from a program. An error causing disruption in program's running or in producing right output, is a '**program bug**'. Debugging involves rectifying the code so that the reason behind the bug gets resolved and thus bug is also removed.

In this section, we shall talk about general debugging techniques and how you can debug code in Python.

Before we talk about debugging techniques, it is important for you to know the errors, error types, what causes them etc. So, let us first talk about errors and exceptions.

8.7.1 Errors in a Program

An error, sometimes called 'a bug', is anything in the code that prevents a program from compiling and running correctly. Some program bugs are catastrophic in their effects, while others are comparatively less harmful and still others are so unclear that you will ever discover them. There are broadly *three* types of errors : *Compile-time errors*, *run-time errors* and *logical errors*.

8.7.1A Compile-Time Errors

Errors that occur during compile-time, are compile-time errors. When a program compiles, its source code is checked for whether it follows the programming language's rules or not.

Two types of errors fall into category of compile-time errors.

1. Syntax Errors

Syntax errors occur when rules of a programming language are misused *i.e.*, when a grammatical rule of Python is violated. *For example*, observe the following two statements :

```
X <- Y * Z
if X = (X * Y)
```

These two statements will result in syntax errors as '<-' is not an assignment operator in Python and '=' is the assignment operator, not a relational operator. Also, **if** is a block statement, it requires a colon (:) at the end of it, which is missing above.

Therefore, the correct statements will be as follows :

```
X = Y * Z
if X == (X * Y) :
```

One should always try to get the syntax right the first time, as a syntax error wastes computing time and money, as well as programmer's time and it is preventable.

SYNTAX

Syntax refers to formal rules governing the construction of valid statements in a language.

2. Semantics Errors

Semantics Errors occur when statements are not meaningful. For instance, the statement 'Sita plays Guitar' is syntactically and semantically correct as it has some meaning but the statement 'Guitar plays Sita' is syntactically correct (as the grammar is correct) but semantically incorrect. Similarly, there are semantics rules of a programming language, violation of which results in semantical errors. For instance, the statement

```
X * Y = Z
```

will result in a semantical error as an expression cannot come on the left side of an assignment statement.

SEMANTICS

Semantics refers to the set of rules which give the meaning of a statement.

8.7.1B Logical Errors

Sometimes, even if you don't encounter any error during compile-time and run-time, your program does not provide the correct result. This is because of the programmer's mistaken analysis of the problem he or she is trying to solve. Such errors are logical errors. For instance, an incorrectly implemented algorithm, or use of a variable before its initialization, or unmarked end for a loop, or wrong parameters passed are often the hardest to prevent and to locate. These must be handled carefully. Sometimes logical errors are treated as a subcategory of run-time errors.

8.7.1C Run-Time Errors

Errors that occur during the execution of a program are run-time errors. These are harder to detect errors. Some run-time errors stop the execution of the program which is then called program "crashed" or "abnormally terminated".

Most run-time errors are easy to identify because program halts when it encounters them *e.g.*, an infinite loop or wrong value (of different data type other than required) is input.

Normally, programming languages incorporate checks for run-time errors, so does Python. However, Python usually takes care of such errors by terminating the program, but a program that crashes whenever it detects an error condition is not desirable. Therefore, a program should be robust so as to recover and continue following an error.

Exceptions

Errors and exceptions are similar but different terms. While ***error*** represents, **any bug in the code** that disrupts running of the program or causes improper output, an **Exception** refers to any irregular situation occurring during execution/run-time, which you have no control on. Errors in a program can be fixed by making corrections in the code, fixing exceptions is not that simple.

Let us try to understand the difference between an error and exception with the help of real life example. For instance, if you operate an ATM then

- entering wrong account number or wrong pin number is an ERROR.
- 'not that much amount in account' is an EXCEPTION.
- 'ATM machine struck' is also an EXCEPTION.

So you can think of **Exceptions** in a program as a situation that occurs during runtime and you have no control on it *e.g.*, you write a code that opens a data file and displays its contents on screen. This program's code is syntactically correct and logically correct too. But when you run this program, the file that you are opening, does not exist on disk – this will cause an EXCEPTION. So the program has no errors but an exception occurred.

NOTE

While Error is a bug in the code that causes irregular output or stops a program from executing, an **Exception** is an irregular unexpected situation occurring during execution on which programmer has no control.

Table 8.12 *Some Built-in Exceptions*

Exception Name	Description
EOFError	Raised when one of the built-in functions (input( )) hits an end-of-file condition (EOF) without reading any data.
IOError	Raised when an I/O operation (such as a **print**( ), the built-in open( ) function or a method of a file object) fails for an I/O-related reason, *e.g.*, *"file not found"* or *"disk full"*.
NameError	Raised when an identifier name is not found.
IndexError	Raised when a sequence subscript or index is *out of range, e.g.*, from a string of length 4 if you try to read a value of index like 4 or more *i.e.*, string[4], string[5], string[–5] etc. will raise exception as legal indexes for a string of length 4 are 0, 1, 2, 3 and –1, –2, –3, –4 only.
ImportError	Raised when an **import** statement fails to find the module definition or when a **from ... import** fails to find a name that is to be imported.
TypeError	Raised when an operation or function is applied to an object of inappropriate type, *e.g.*, if you try to compute a square-root of a string value.
ValueError	Raised when a built-in operation or function receives an argument with an inappropriate value *e.g.*, *int("z10")* will raise *ValueError*.
ZeroDivisionError	Raised when the second argument of a division or modulo operation is zero.
OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
KeyError	Raised when a mapping (dictionary) key is not found in the set of existing keys.

For example, following screenshots are showing two different exceptions.

```
>>> 100 / 0
Traceback (most recent call last):
  File "<pyshell#130>", line 1, in <module>
    100 / 0
ZeroDivisionError: division by zero
```

```
>>> int('11ab')
Traceback (most recent call last):
  File "<pyshell#131>", line 1, in <module>
    int('11ab')
ValueError: invalid literal for int() with base 10: '11ab'
```

8.7.2 Debugging using Code Tracing

Most common technique to debug an error is to find the point of error and origin of error. This is often done by printing the values of every intermediate result and of all values. For this purpose, code tracing is one most commonly used technique.

Common useful technique for debugging is code tracing. Code tracing means executing code one line at a time and watching its impact on variables. One way of code tracing is using **Dry run**, which you have already learnt in an earlier chapter. Code tracing can also be done by debugging tools or debugg available in software form.

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>)`
- *Debugging refers to the process of locating the place of error, cause of error, and correcting the code accordingly.*
- *Compile-time errors (syntax error and semantics errors) refer to the errors that violate the grammatical rules and regulations of a programming language.*
- *Runtime errors occur during the execution of a program.*
- *Logical errors occur due to mistaken analysis of the problem.*
- *Irregular unexpected situations occurring during runtime are called Exceptions.*
- *Exceptions may occur even if the program is free from all types of errors.*

Objective Type Questions — OTQs

Multiple Choice Questions

1. Which of the following are valid Python data types ?

   (*a*) Numeric (*b*) None (*c*) Mappings
   (*d*) list (*e*) Sequence (*f*) set
   (*g*) tuple (*h*) dictionary

2. Which of the following are datatypes considered as Numbers in Python.

   (*a*) Integer (*b*) Boolean
   (*c*) complex (*d*) floating point
   (*e*) list (*f*) None

3. Consider the following sequence of statements:

   ```
   a = 35
   m = a
   ```

   Following the execution of these statements, Python has created how many objects and how many references ?

   (*a*) Two integer objects, two references
   (*b*) One integer object, two references
   (*c*) One integer object, one reference
   (*d*) Two integer objects, one reference

4. Which Python built-in function returns the unique number assigned to an object?
(a) identity( ) (b). id( )
(c) refnum( ) (d) ref( )

5. The operator used to check if both the operands reference the same object memory, is the _____ operator.
(a) in (b) is (c) id (d) ==

6. For two objects x and y, the expression x is y will yield True, if and only if
(a) id(x) == id(y) (b) len(x) == len(y)
(c) x == y (d) all of these

7. Which of the following is not an immutable type in Python ?
(a) String (b) Tuples (c) Set (d) dictionary

8. Python operator always yields the result of _____ datatype.
(a) integer (b) floating point
(c) complex (d) all of these

9. What is the value of the expression 100 / 25 ?
(a) 4 (b) 4.0
(c) 2.5 (d) none of these

10. What is the value of the expression 100 // 25 ?
(a) 4 (b) 4.0
(c) 2.5 (d) none of these

11. In Python, a variable must be declared before it is assigned a value.
(a) True (b) False
(c) Only in Functions (d) Only in modules

12. In Python, a variable is assigned a value of one type, and then later assigned a value of a different type. This will yield _____ .
(a) Warning (b) Error
(c) None (d) No Error

13. In Python, a variable may be assigned a value of one type, and then later assigned a value of a different type. This concept is known as _____ .
(a) mutability (b) static typing
(c) dynamic typing (d) immutability

14. Is it safe to directly use the == operator to determine whether objects of type float are equal ?
(a) Yes (b) No
(c) Yes, if the values are <100
(d) Yes, if the values are >100

15. What will the following code produce ?
```
a = 8.6
b = 2
print ( a//b )
```
(a) 4.3 (b) 4.0
(c) 4 (d) compilation error

16. In the Python statement $x = a + 5 - b$: a and b are _____ .
(a) Operands (b) Expression
(c) operators (d) Equation

17. In the Python statement $x = a + 5 - b$: $a + 5 - b$ is _____
(a) Operands (b) Expression
(c) operators (d) Equation

18. What will be the value of y after following code fragment is executed ?
```
x = 10.0
y = (x < 100.0) and x >= 10
```
(a) 110 (b) True
(c) False (d) Error.

19. Which of the following literals has True truth-value ?
(a) 0.000001 (b) 'None' (c) 0
(d) [ ] (e) False (f) True
(g) 1 (h) 33 (i) None (j) 0.0

20. What will the following code result as ?
```
import math
x = 100
print ( x > 0 and math.sqrt( x ) )
```
(a) True (b) 1
(c) 10 (d) 10.0

21. Which of the following operators has the lowest precedence ?
(a) not (b) %
(c) and (d) + (e) **

22. What is the value of the expression 10 + 3 ** 3 * 2?
(a) 28 (b) 739
(c) 829 (d) 64

23. To increase the value of x five times using an augmented assignment operator, the correct expression will be
(a) x += 5 (b) x *= 5
(c) x = x ** 5 (d) none of these

24. What will be the result of the expression **10 or 0** ?
(a) 0 (b) 1 (c) 10 (d) 1.0

25. What will be the result of the expression **5 or 10** ?
(a) 5 (b) 1 (c) 10 (d) 0

26. What will be the result of the expression **5 and 10** ?
(a) 5 (b) 1 (c) 10 (d) 0

27. What will be the result of the expression **15 and 10** ?
(a) 5 (b) 1 (c) 10 (d) 0

28. What will be the result of the expression **10 or 0** ?
(a) 0 (b) 1 (c) 10 (d) 1.0

29. What will be the result of the expression **'a' or "** (" is an empty string) ?
(a) 'a' (b) '' (c) 1 (d) 0

30. What will be the result of the expression **'a' and "** (" is an empty string) ?
(a) 'a' (b) '' (c) 1 (d) 0

31. What will be the result of the expression **'x' and 'a'** ?
(a) 'a' (b) '' (c) 'x' (d) 1

32. What will be the result of the expression **'a' and 'x'** ?
(a) 'a' (b) ' ' (c) 'x' (d) 1

33. What will be the result of the expression **'a' and 'None'** ?
(a) 'a' (b) ' '
(c) 'None' (d) 1

34. What will be the result of the expression **'None' and 'a'** ?
(a) 'a' (b) ' '
(c) 'None' (d) 1

35. What will be the result of the expression **'false' and False** ?
(a) false (b) False (c) 'false' (d) 'False'

36. What will be the result of the expression **'false' or False** ?
(a) false (b) False (c) 'false' (d) 'False'

Fill in the Blanks

1. Boolean data type is internally treated as _____ data type.
2. Two forms of floating-point numbers are _____ form and _____ notation.
3. Python's floating point numbers have precision of _____ digits.
4. Three mutable types of Python are _____, _____ and _____ .
5. The floor division of two integers yields a result of _____ type.
6. The division of two integers yields a result of _____ type.
7. The _____ sequence type cannot store duplicate values.
8. The _____ datatype is like lists but is not mutable.
9. The _____ of an object gives the memory location of the object.
10. To check if two objects reference the same memory address, _____ operator is used.
11. To use function fabs( ), _____ module should be imported.
12. To generate a random floating number in the range 0 to 100, _____ function is used.
13. To generate a random integer in a range, _____ function is used.
14. To generate a random number in a sequence of values where two values have a difference a step value, _____ function is used.
15. To use mean( ) function, _____ module is to be imported.

True/False Questions

1. List is an immutable data type.
2. Set is a mutable data type.
3. A set can contain duplicate values in it.
4. A Boolean value is internally treated as an integer value.
5. " (an empty string) has truth value as False.
6. ' ' (a space) has truth value as False.
7. Value false is a legal literal value in Python.
8. Value False is a legal literal value in Python.
9. Value 'False' is a legal literal value in Python.

10. Value 'false' is a legal literal value in Python.
11. None and 'None' are the same.
12. None has the truth value as False.
13. 'None' has the truth value as False.
14. The result of bool(0) is False.
15. The result of bool('None') is False.
16. Dividing two integers results in an integer.
17. Floor division of two integers results in an integer.
18. Two floating point numbers should not be compared for equality using == .
19. In implicit conversion, all operands' data types are converted to the datatype of the largest operand.
20. Explicit type conversion involves the use of a function to convert datatype of a value.

NOTE : Answers for OTQs are given at the end of the book.

Solved Problems

1. *What are data types ? What are Python's built-in core data types ?*

Solution. The real life data is of many types. So to represent various types of real-life data, programming languages provide ways and facilities to handle these, which are known as *data types*.

Python's built-in core data types belong to :

- Numbers (integer, floating-point, complex numbers, Booleans)
- String
- List
- Tuple
- Dictionary

2. *Which data types of Python handle Numbers ?*

Solution. Python provides following data types to handle numbers

(*i*) Integers (*ii*) Boolean
(*iii*) Floating-point numbers (*iv*) Complex numbers

3. *Why is Boolean considered a subtype of integers ?*

Solution. Boolean values *True* and *False* internally map to integers *1* and *0*. That is, internally *True* is considered equal to 1 and *False* equal to 0 (zero). When 1 and 0 are converted to Boolean through bool( ) function, they return *True* and *False*. That is why Booleans are treated as a subtype of integers.

4. *Identify the data types of the values given below :*

3, 3j, 13.0, '13', "13", 2+0j, 13, [3, 13, 2], (3, 13, 2)

Solution.

3	integer
13.0	Floating-point number
"13"	String
13	integer
(3, 13, 2)	Tuple
3j	complex number
'13'	string
2+0j	complex number
[3, 13, 2]	List

5. *What do you understand by term 'immutable' ?*

Solution. Immutable means unchangeable. In Python, immutable types are those whose values cannot be changed in place. Whenever one assigns a new value to a variable referring to immutable type, variable's reference is changed and the previous value is left unchanged. *e.g.*,

```
x = 3
x = 5
```

value 3 is untouched and x is made to refer to new value.

6. *What will be the output of the following ?*

```
print (len(str(17//4)) )
print (len(str(17/4)) )
```

Solution.
```
1
3
```

because

```
len(str(17//4))
    = len(str(4))
    = len('4')
    = 1
```

and

```
len(str(17/4))
    = len(str(4.0))
    = len('4.0')
    = 3
```

7. *What will be the output produced by these ?*

(*a*) 12/4 (*b*) 14//14 (*c*) 14%4 (*d*) 14.0/4 (*e*) 14.0//4 (*f*) 14.0%4

Solution. (*a*) 3.0 (*b*) 1 (*c*) 2 (*d*) 3.5 (*e*) 3.0 (*f*) 2.0

8. *Given that variable CK is bound to string "Raman" (i.e., CK = "Raman"). What will be the output produced by following two statements if the input given in "Raman" ? Why ?*

```
DK = input("Enter name:")
Enter name : Raman
```

(*a*) DK == CK (*b*) DK is CK

Solution. The output produced will be as : (*a*) True (*b*) False

The reason being that both **DK** and **CK** variable are bound to identical strings 'Raman'. But input strings are always bound to fresh memory even if they have value identical to some other existing string in memory.

Thus DK == CK produces True as strings are identical.

But **DK is CK** produces False as they are bound to different memory addresses.

9. *What will be the output of following code ? Explain reason behind output of every line :*

```
5 < 5 or 10
5 < 10 or 5
5 < (10 or 5)
5 < (5 or 10)
```

Solution.
```
10
True
True
False
```

Explanation

Line 1	`5 < 5 or 10`
	`= False or 10`
	`= 10`
		precedence of < is higher than or
		because or would evaluate the second argument if first argument is False or false$_{tval}$

Line 2	`5 < 10 or 5`
	`= True or 5`
	`= True`
		precedence of < is higher than or
		or would return first argument if it is True or true$_{tval}$

Line 3	`5 < (10 or 5)`
	`= 5 < 10`
	`= True`
		10 or 5 returns 10 since 10 is true$_{tval}$

Line 4	`5 < (5 or 10)`
	`= 5 < 5`
	`= False`
		5 or 10 returns 5 since 5 is true$_{tval}$

10. *What will be the output produced by the three expressions of the following code ?*

```
a = 5
b = -3
c = 25
d = -10
a + b + c > a + c - b * d
str(a + b + c > a + c - b * d) == 'true'
len(str(a + b + c > a + c - b * d)) == len(str(bool(1)))
```

Solution.
```
True
False
True
```

11. *What would Python produce if for the following code, the input given is*

(*i*) 11 (*ii*) hello (*iii*) just return key pressed, no input given (*iv*) 0 (*v*) 5-5

Code

```
bool(input("Input:")) and 10 < 13 - 2
```

Solution.

(*i*) Input : 11 would yield

```
bool ('11') and 10 < 11
True and 10 < 11
= True
```

(*ii*) **True** [For the same reason as in (*i*)]

(*iii*) **False** because when just return key is pressed, input is '' *i.e.*, empty string, hence expression becomes

```
bool( '' ) and 10 < 11
False and True
= False
```

(*iv*)
```
bool ( '0' ) and 10 < 11
True and True
= True
```
(Please note '0' is a non-empty string and hence has truth value as true$_{tval}$)

(*v*)
```
bool ( '5-5' ) and 10 < 11
= True and 10 < 11
= True
```

12. *What would be the output of the following code ? Explain your answer.*

```
a = 3 + 5/8
b = int(3 + 5/8)
c = 3 + float(5/8)
d = 3 + float(5)/8
e = 3 + 5.0/8
f = int(3 + 5/8.0)
print (a, b, c, d, e, f)
```

Solution. The output would be

```
3.625  3  3.625  3.625  3.625  3
```

Explanation

Line 1 `a = 3 + 5/8`
= 3 + 0.625
∴ $a = 3.625$

Line 2 `b = int (3 + 5/8)`
= int (3 + 0.625) (int( ) will drop the fractional part)
= int (3.625)
$b = 3$

Line 3 `c = 3 + float(5/8)`
c = 3 + float(0.625)
= 3 + 0.625
$c = 3.625$

Line 4 `d = 3 + float(5)/8`
= 3 + 5.0/8 5.0/8 = 0.625 because one operand is floating point, the integer operand will be internally converted to floating-pt
= 3 + 0.625
$d = 3.625$

Line 5 `e = 3 + 5.0/8`
= 3 + 0.625 (same reason as above)
$e = 3.625$

Line 6 `f = int(3 + 5/8.0)` (same reason as above)
f = int (3 + 0.625)
f = int (3.0) (int( ) will drop the fractional part)
$f = 3$

13. *What will be the output produced by following code statements ?*

(a) 87 // 5
(b) 87 // 5.0
(c) (87 // 5.0) == (87 // 5)
(d) (87 // 5.0) == int (87 / 5.0)
(e) (87 // int (5.0)) == (87 // 5.0)

Solution. (a) 17 (b) 17.0 (c) True (d) True (e) True

14. *What will be the output produced by following code statements ? State reasons(s).*

(a) 17 % 5
(b) 17 % 5.0
(c) (17 % 5) == (17 % 5)
(d) (17 % 5) is (17 % 5)
(e) (17 % 5.0) == (17 % 5.0)
(f) (17 % 5.0) is (17 % 5.0)

Solution. (a) 2 (b) 2.0 (c) True (d) True (e) True (f) False

Both (c) and (e) evaluate to **True** because both the operands of == operator are same values (2 == 2 in (c) and 2.0 == 2.0 in (e).

(d) evaluates to **True** as both the operands of **is** operator are same integer objects (2).

Since both operand expressions evaluate to same integer value 2 and 2 is a small integer value, both are bound to same memory address, hence **is** operator returns **True**.

In (f), even though both operands evaluate to same floating point value 2.0, the **is** operator returns **False** because Python assigns different memory address to floating point values even if their exists a same value in the memory.

15. *In Python, you can write -5^2 in following ways : –5**2 and math.pow(–5, 2) and –5 * –5*

*But you execute these, the results vary, i.e., – 5**2 gives result as –25*

*But both math.pow(–5, 2) and –5 * –5 gives result as +25.*

Why do results vary for these expressions? What changes will you incorporate to get the same result in all these expressions ?

Solution. The result of – 5**2 is –25 because :

Exponential operator ** has higher precedence that unary – operator and it (exponentiation operator **) has ***right to left associativity***, *i.e.*, the right side operator ** attaches with value 5 before unary operator – .

Hence it is internally interpreted as : – (5**2)

And hence we get the result as : –25.

In other two expressions there is only single operator with value 5, hence no need of using associativity and thus the result as +25.

We can changes the first expression as (– 5)**2 to get the correct result.

16. *What will be the output produced by the following code statements ? State reasons.*

(a) bool(0) (b) bool (1) (c) bool('0') (d) bool('1') (e) bool('')

(f) bool(0.0) (g) bool('0.0') (h) bool(0j) (i) bool('0j')

Solution.

(a) **False.** Integer value 0 has false truth value hence bool( ) converts it to **False**.

(b) **True.** Integer value 1 has true truth value hence bool( ) converts it to **True**.

(c) **True.** '0' is string value, which is a non-empty string and has a true truth value, hence bool( ) converts it to *True*.

(d) **True.** Same reason as above.

(e) **False.** '' is an empty string, thus has false truth value, hence bool( ) converts it to *False*.

(f) **False.** 0.0 is zero floating point number and has false truth value, hence bool( ) converts it to *False*.

(g) **True.** '0.0' is a non-empty string hence it has true truth value and thus bool( ) converted it to *True*.

(h) **False.** 0j is zero complex number and has false truth value and thus bool( ) converted it to *False*.

(i) **True.** '0j' is non-empty string, hence it has true truth value and thus boo( ) converted it to *True*.

17. *What will be the output produced by these code statement ?*

(a) bool(int('0')) (b) bool(str(0)) (c) bool(float('0.0')) (d) bool(str(0.0))

Solution. (a) False (b) True (c) False (d) True

18. *What will be the output of following code ? Why ?*

(*i*) `13 or len(13)` (*ii*) `len(13) or 13`

Solution.

(*i*) 13 because or evaluates first argument 13's truth value, which is *true* and hence returns the result as 13 without evaluating second argument.

(*ii*) error because when or evaluates fist argument **len(13)**, Python gives error as **len( )** works on strings only.

19. *Given the following Python code, which is repeated four times. What could be the possible set of outputs out of given four sets (dddd represent any combination of digits) ?*

```
import random
print(15 + random.random() * 5)
```

(*i*) 17.dddd, 19.dddd, 20.dddd, 15.dddd

(*ii*) 15.dddd, 17.dddd. 19.dddd, 18.dddd

(*iii*) 14.dddd, 16.dddd, 18.dddd, 20.dddd

(*iv*) 15.dddd, 15.dddd, 15.dddd, 15.dddd

Solution. Option (*ii*) and (*iv*) are the correct possible outputs because :

(*a*) *random( )* generates number *N* between range 0.0 <= *N* < 1.0.

(*b*) when it is multiplied with 5, the range becomes 0.0 to < 5

(*c*) when 15 is added to it, the range becomes 15 to < 20

Only option (*ii*) and (*iv*) fulfill the condition of range 15 to < 20.

20. *What do you mean by Syntax errors and Semantics errors ?*

Solution. Syntax errors are the errors that occur when rules of a programming language are violated. Semantics errors occur when statements are not meaningful.

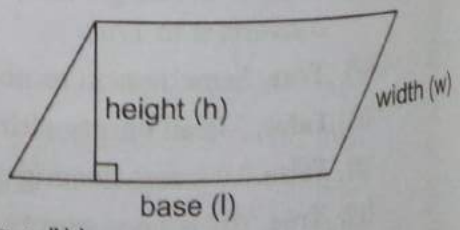
21. *Why are logical errors harder to locate ?*

Solution. In spite of logical errors' presence, the program executes without any problems but the output produced is not correct. Therefore, each and every statement of the program needs to be scanned and interpreted. Thus the logical errors are harder to locate.

22. *What is an Exception ?*

Solution. Exception in general refers to some contradictory or unusual situation which can be encountered unexpectedly while executing a program.

23. *Write a program to read base/length (l), width(w) and height(h) of a parallelogram and calculate its area and perimeter.*



Solution.

```
l = float(input("Enter base/length of the parallelogram : "))
w = float(input("Enter width of the parallelogram : "))
h = float(input("Enter height of the parallelogram : "))
area = l * h
perimeter = 2*l + 2*w
print("The area of given parallelogram is :", area)
print("The perimeter of given parallelogram is :", perimeter)
```

The sample run of the above program is as shown below :

```
Enter base/length of the parallelogram : 13.5
Enter width of the parallelogram : 7
Enter height of the parallelogram : 5
The area of given parallelogram is : 67.5
The perimeter of given parallelogram is : 41.0
```

24. *In a school fest, three randomly chosen students out of 100 students (having roll numbers 1-100) have to present bouquets to the guests. Help the school authorities choose three students randomly.*

Solution.

```
import random
student1 = random.randint(1, 100)
student2 = random.randint(1, 100)
student3 = random.randint(1, 100)
print("3 chosen students are",)
print(student1, student2, student3)
```

25. *A triangle has three sides a, b, c as 17, 23, 30. Calculate and display its area using Heron's formula as*

$$s = \frac{a+b+c}{2} \; ; \quad \text{Area} = \sqrt{s(s-a)(s-b)(s-c)}$$

Solution.

```
import math
a, b, c = 17, 23, 30
s = (a + b + c)/2
area = math.sqrt(s * (s-a) * (s-b) * (s-c))
print("Sides of triangle:", a, b, c)
print("Area :", area, "units square")
```

Output :

```
Sides of triangle: 17  23  30
Area : 194.42222095223582 units square
```

Guidelines to NCERT Questions [NCERT Chapter 5]

3. *Write logical expressions corresponding to the following statements in Python and evaluate the expressions (assuming variables num1, num2, num3, first, middle, last are already having meaningful values) :*

(a) *The sum of 20 and –10 is less than 12.*

(b) *num3 is not more than 24.*

(c) *6.75 is between the values of integers num1 and num2.*

(d) *The string 'middle' is larger than the string 'first' and smaller than the string 'last'.*

Ans. (a) (20 + –10) < 12

(b) num3 <= 24 or not (n3 > 24)

(c) num1 <= 6.75 <= num2

(d) 'first' < 'middle' < 'last'.

4. *Add a pair of parentheses to each expression so that it evaluates to True.*

(a) `0 == 1 == 2`

(b) `2 + 3 == 4 + 5 == 7`

(c) `1 < -1 == 3 > 4`

Ans. (*a*) `0 == (1 == 2)`
(*b*) `(2 + (3 == 4) + 5) == 7`
(*c*) `(1 < -1) == (3 > 4)`

6. *Which data type will be used to represent the following data values and why ?*
(*a*) *Number of months in a year* (*b*) *Resident of Delhi or not*
(*c*) *Mobile number* (*d*) *Pocket money*
(*e*) *Volume of a sphere* (*f*) *Perimeter of a square*
(*g*) *Name of the student* (*h*) *Address of the student*

Ans. (*a*) Integer (*b*) Boolean (*c*) Integer (*d*) Integer or Floating point (*e*) Floating point
(*f*) Floating point (*g*) String (*h*) String

7. *Give the output of the following when num1 = 4, num2 = 3, num3 = 2*

(*a*)
```
num1 += num2 + num3
```
(*b*)
```
num1 = num1 ** (num2 + num3)
print(num1)
```
(*c*)
```
num1 **= num2 + num3
```
(*d*)
```
num1 = '5' + '5'
print(num1)
```
(*e*)
```
print(4.00/(2.0 + 2.0))
```
(*f*)
```
num1 = 2 + 9 * ((3 * 12) - 8)/10
print(num1)
```
(*g*)
```
num1 = 24//4//2
print(num1)
```
(*h*)
```
num1 = float(10)
print(num1)
```
(*i*)
```
num1 = int('3.14')
print(num1)
```
(*j*)
```
print('Bye' == 'BYE')
```
(*k*)
```
print(10 != 9 and 20 >= 20)
```
(*l*)
```
print(10 + 6 * 2 ** 2 != 9/4 - 3 and 29 >= 29/9)
```
(*m*)
```
print(5 % 10 + 10 < 50 and 29 <= 29)
```
(*n*)
```
print((0 < 6) or (not(10 == 6) and (10 < 0)))
```

Ans. (*a*) 9 (*b*) 1024 (*c*) 1024 (*d*) 55 (*e*) 1.0 (*f*) 27.2 (*g*) 3 (*h*) 10.0
(*i*) ValueError (*j*) False (*k*) True (*l*) False (*m*) True (*n*) True

8. *Categorise the following as syntax error, logical error or runtime :*
(*a*) 25/0 (*b*) num1 = 25 ; num2 = 0 ; num1/num2

Ans. (*a*) Runtime error (exception) (*b*) runtime error (exception)

9. *A dartboard of radius 10 units and the wall it is hanging on are represented using a two-dimensional coordinate system, with the board's center at coordinate (0, 0). Variables x and y store the x-coordinate and the y-coordinate of a dart that hits the dartboard. Write a Python expression using variables x and y that evaluates to True if the dart hits (is within) the dartboard, and then evaluate the expression for these dart coordinates.*
(*a*) (0, 0) (*b*) (10, 10) (*c*) (6, 6) (*d*) (7, 8)

Ans. Expression to test if the dart hits within the dartboard = $(x^2 + y^2) < 100$
(*a*) (0, 0) lies within the dartboard as $(0^2 + 0^2) < 100$ is *True*.
(*b*) (10, 10) does not lies within the dartboard as $(10^2 + 10^2) < 100$ is *False*.
(*c*) (6, 6) lies within the dartboard as $(6^2 + 6^2) < 100$ is *True*.
(*d*) (7, 8) does not lie within the dartboard as $(7^2 + 8^2) < 100$ is *False*.

10. *Write a Python program to convert temperature in degree Celsius to degree Fahrenheit. If water boils at 100 degree C and freezes as 0 degree C, use the program to find out what is the boiling point and freezing point of water on the Fahrenheit scale. [Hint. $T(°F) = T(°C) \times \frac{9}{5} + 32$)]*

Ans.

```
bp_c = 100          # boiling point in celcius
fp_c = 0            # freezing point in celcius
degree = u'\N{DEGREE SIGN}'   # it is the literal for degree sign
# Fahrenheit = ( Celsius * 9/5 ) + 32
bp_f = ( bp_c * 9/5 ) + 32    # bp_f : boiling point in fahrenheit
fp_f = ( fp_c * 9/5 ) + 32    # fp_f : freezing point in fahrenheit
print("Boiling point in", degree + "C :", bp_c, "and in", degree + "F :", bp_f)
print("Freezing point in", degree + "C :", fp_c, "and in", degree + "F :", fp_f)
```

Output :

```
Boiling point in °C : 100 and in °F : 212.0
Freezing point in °C : 0 and in °F : 32.0
```

11. *Write a Python program to calculate the amount payable if money has been lent on simple interest. Principal or money lent = P, Rate of interest = R% per annum and Time = T years. Then simple interest (SI) = (P× R×T)/100. Amount payable = Principal + SI. P, R and T are given as input to the program.*

Ans.

```
P = float( input("Enter money lent:"))
R = float( input("Enter rate of interest:"))
T = float( input("Enter time in years:"))
SI = ( P * R * T ) / 100
amount_payable = P + SI
print("Amount payable is : Rs.", amount_payable)
```

Sample Run :

```
Enter money lent: 7000
Enter rate of interest: 8.5
Enter time in years: 3.5
Amount payable is : Rs. 9082.5
```

16. *Write a program to repeat the string "GOOD MORNING" n time. Here 'n' times. Here 'n' is an integer entered by the user.*

Ans.

```
n = int( input("How many times ?"))
print ("GOOD MORNING" * n)
```

Output :

```
How many times ? 3
GOOD MORNINGGOOD MORNINGGOOD MORNING
```

20. *The formula $E = mc^2$ states that the equivalent energy (E) can be calculated as the mass (m) multiplied by the speed of light (c = about 3×10^8 m/s) squared. Write a program that accepts the mass of an object and determines its energy.*

Ans.

```
import math
m = float(input("Enter mass : "))
c = 3 * pow(10, 8)
E = m * c * c
print ("Equivalent energy :", E, "Joule")    # unit of energy is Joule
```

Output :

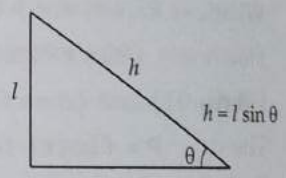
```
Enter mass : 8.95
Equivalent energy : 8.055e+17 Joule
```

21. *Presume that a ladder is put upright against a wall. Let variables length and angle store the length of the ladder and the angle that it forms with the ground as it leans against the wall. Write a Python program to compute the height reached by the ladder on the wall for the following values or length and angle :*

(*a*) 16 feet and 75 degrees (*b*) 20 feet and 0 degrees

(*c*) 24 feet and 45 degrees (*d*) 24 feet and 80 degrees

Solution. Since the ladder leaning against the wall will form a right-angled triangle, the height up to which a ladder reaches can be calculated as :



Thus, our Python program is based on the same.

```
import math
length = float(input("Enter length of the ladder :"))
angle = float( input("Enter angle of leaning (in degrees) :"))
ang_radian = math.radians(angle)              # convert degrees to radians
height = length * math.sin(ang_radian)
print ("Ladder's height on the wall :", height)
```

Sample run :

```
Enter length of the ladder : 16
Enter angle of leaning (in degrees) : 75
Ladder's height on the wall :  15.454813220625093
-----------------------------------------------
Enter length of the ladder : 20
Enter angle of leaning (in degrees) : 0
Ladder's height on the wall :  0.0
-----------------------------------------------
Enter length of the ladder : 24
Enter angle of leaning (in degrees) : 45
Ladder's height on the wall :  16.970562748477143
-----------------------------------------------
Enter length of the ladder : 24
Enter angle of leaning (in degrees) : 80
Ladder's height on the wall :  23.63538607229299
```

GLOSSARY

Atom	Something that has a value.
Coercion	Implicit Type Conversion
Expression	Valid combination of operators and atoms.
Explicit Type Conversion	Forced data type conversion by the user.
Immutable Type	A type whose value is not changeable in place.
Implicit Type Conversion	Automatic Internal Conversion of data type (lower to higher type) by Python.
Mutable Type	A type whose value is changeable in place.
Operator	Symbol/word that triggers an action or operation.
Type Casting	Explicit Type Conversion.

Assignments

For Solutions for Selected Questions

Scan QR Code

Type A : Short Answer Questions/Conceptual Questions

1. What are data types ? How are they important ?
2. How many integer types are supported by Python ? Name them.
3. How are these numbers different from one another ? 33, 33.0, 33j, 33 + j
4. The complex numbers have two parts : real and imaginary. In which data type are real and imaginary parts represented ?
5. How many string types does Python support ? How are they different from one another ?
6. What will following code print ?

```
str1 = '''Hell ↵
          o'''
str2 = '''Hell\
          o'''
print(len(str1) > len(str2))
```

7. What are immutable an mutable types ? List immutable and mutable types of Python.
8. What are three internal key-attributes of a value-variable in Python ? Explain with example.
9. Is it true that if two objects return True for **is** operator, they will also return True for == operator ?
10. Are these values equal ? Why/why not ?
    (*i*) 20 and 20.0 (*ii*) 20 and int(20) (*iii*) str(20) and str(20.0) (*iv*) 'a' and "a"
11. What is an atom ? What is an expression ?
12. What is the difference between implicit type conversion and explicit type conversion ?
13. Two objects (say *a* and *b*) when compared using == , return True. But Python gives False when compared using **is** operator. Why ? (*i.e.*, *a* == b is True but why is *a is b* False ?)
14. Given str1 = "Hello", what will be the values of ?
    (*a*) str1 [0] (*b*) str1 [1] (*c*) str [–5] (*d*) str [–4] (*e*) str [5]
15. If you give the following for str1 = "Hello", why does Python report error ?

```
str1[2] = 'p'
```

16. What will the result given by the following ?
(a) type (6 + 3) (b) type (6 – 3) (c) type (6 * 3) (d) type (6/3) (e) type (6//3) (f) type (6 % 3)
17. What are augmented assignment operators ? How are they useful ?
18. Differentiate between (555/222)**2 and (555.0/222)**2.
19. Given three Boolean variables *a*, *b*, *c* as : *a* = False, *b* = True, *c* = False.
Evaluate the following Boolean expressions :
(a) *b* and *c* (b) *b* or *c* (c) not *a* and *b* (d) (*a* and *b*) or not *c*
(e) not *b* and not (*a* or *c*) (f) not ((not *b* or not *a*) and *c*) or *a*
20. What would following code fragments result in ? Given $x = 3$.
(a) $1 < x$ (b) $x >= 4$ (c) $x == 3$ (d) $x == 3.0$ (e) "Hello" == "Hello"
(f) "Hello" > "hello" (g) 4/2 == 2.0 (h) 4/2 == 2 (i) $x < 7$ and $4 > 5$.
21. Write following expressions in Python
(a) $\frac{1}{3}b^2h$ (b) $\pi r^2 h$ (c) $\frac{1}{3}\pi r^2 h$ (d) $d = \sqrt{(x2-x1)^2+(y2-y1)^2}$ **[Hint.** for sqrt( ) use **math.sqrt( )]**
(e) $(x-h)^2+(y-k)^2 = r^2$ (f) $x = \frac{-b+\sqrt{b^2-4ac}}{2a}$ (g) $a^n \times a^m = a^{n+m}$
(h) $(a^n)^m = a^{nm}$ (i) $\frac{a^n}{a^m} = a^{n-m}$ (j) $a^{-n} = \frac{1}{a^n}$
22. int('*a*') produces error. Why ?
23. **int('*a*')** produces error but following expression having **int('*a*')** in it, does not return error. Why ?
len('*a*') + 2 or int('*a*')
24. Write expression to convert the values 17, len('ab') to (*i*) integer (*ii*) str (*iii*) Boolean values
25. Evaluate and Justify : (*i*) 22 / 17 = 37 / 47 + 88 / 83 (*ii*) len('375')**2.
26. Evaluate : (*i*) 22.0/7.0 – 22/7 (*ii*) 22.0/7.0 – int(22.0/7.0) (*iii*) 22/7 – int (22.0)/7 and justify.
27. Evaluate and justify : (*i*) false and None (*ii*) 0 and None (*iii*) True and None (*iv*) None and None.
28. Evaluate and justify :
(a) `0 or None and "or"` (b) `1 or None and 'a' or 'b'` (c) `False and 23`
(d) `23 and False` (e) `not (1 == 1 and 0 != 1)`
(f) `"abc" == "Abc" and not (2 == 3 or 3 == 4)`
(g) `False and 1 == 1 or not True or 1 == 1 and False or 0 == 0`
29. Evaluate the following for each expression that is successfully evaluated, determine its value and type for unsuccessful expression, state the reason.
(a) `len("hello") == 25/5 or 20/10` (b) `3 < 5 or 50/(5 - (3 + 2))`
(c) `50/(5 - (3 + 2)) or 3 < 5` (d) `2 * (2 * (len("01")))`
30. Write an expression that uses exactly 3 arithmetic operators with integer literals and produces result as 99.
31. Add parentheses to the following expression to make the order of evaluation more clear.
`y % 4 == 0` and `y % 100 != 0` or `y % 400 == 0`
32. A program runs to completion but gives an incorrect results. What type of error would have caused it ?
33. In Python, strings are immutable while lists are mutable. What is the difference ?
34. How does the // operator differ from the / operator ? Give an example of where // would be needed.
35. MidAir Airlines will only allow carry-on bags that are no more than 22 inches long, 14 inches wide, and 9 inches deep. Assuming that variables named length, width, and depth have already been assigned values, write an expression combining the three that evaluates to *True* if bag fits within those limits, and *False* otherwise.

36. What are main error types ? Which types are most dangerous and why ?
37. Correct any false statements :
    (*a*) Compile-time errors are usually easier to detect and to correct than run-time errors.
    (*b*) Logically errors can usually be detected by the compiler.
38. Differentiate between a syntax error and a semantics error.
39. Differentiate between a syntax error and a logical error in a program. When is each type of error likely to be found ?
40. What is the difference between an error and exception ?

Type B : Application Based Questions

1. What is the result produced by (*i*) bool (0) (*ii*) bool (str(0)) ? Justify the outcome.
2. What will be the output, if input for both the statements is 5+4/2.

```
6 == input ("Value 1:")
6 == int(input ("value 2:"))
```

3. Following code has an expression with all integer values. Why is the result in floating point form ?

```
a, b, c = 2, 3, 6
d = a + b * c/b
print(d)
```

4. What will following code print ?

(*a*)
```
a = va = 3
b = va = 3
print (a, b)
```

(*b*)
```
a = 3
b = 3.0
print (a == b)
print (a is b)
```

5. What will be output produced by following code ? State reason for this output.

(*a*)
```
a, b, c = 1, 1, 2
d = a + b
e = 1.0
f = 1.0
g = 2.0
h = e + f
print(c == d)
print(c is d)
print(g == h)
print(g is h)
```

(*b*)
```
a = 5 - 4 - 3
b = 3**2**3
print(a)
print(b)
```

(*c*)
```
a, b, c = 1, 1, 1
d = 0.3
e = a + b + c - d
f = a + b + c == d
print(e)
print(f)
```

6. What will be the output of following Python code ?

(*a*)
```
a = 12
b = 7.4
c = 1
a - = b
print(a, b)
a *= 2 + c
print(a)
b += a * c
print(b)
```

(*b*)
```
x, y = 4, 8
z = x/y*y
print(z)
```

7. Make change in the expression for z of previous question so that the output produced is zero. You cannot change the operators and order of variables. (Hint. Use a function around a sub-expression)

8. Consider the following expression.

```
x = "and" * (3 + 2) > "or" + "4"
```

What is the data type of value that is computed by this expression ?

9. Consider the following code segment :

```
a = input()
b = int(input())
c = a + b
print(c)
```

When the program is run, the user first enters 10 and then 5, it gives an error. Find the error, its reason and correct it.

10. Consider the following code segment :

```
a = input("Enter the value of a:")
b = input("Enter the value of b:")
print(a + b)
```

If the user runs the program and enters 11 for a and 9 for b then what will the above code display ?

11. Find out the error and the reason for the error in the following code. Also, give the corrected code.

```
a, b = "5.0", "10.0"
x = float(a/b)
print(x)
```

Consider the following program for the next two questions. It is supposed to compute the hypotenuse of a right triangle after the user enters the lengths of the other two sides.

```
a = float(input("Enter the length of the first side:"))
b = float(input("Enter the length of the second side:"))
h = sqrt(a * a + b * b)
print("The length of the hypotenuse is", h)
```

12. When this program is run, the following output is generated (note that input entered by the user is shown in bold) :

```
Enter the length of the first side: 3
Enter the length of the second side: 4
Traceback (most recent call last):
h = sqrt(a * a + b * b)
NameError: name 'sqrt' is not defined
```

Why is this error occurring? How would you resolve it ?

13. After adding *import math* to the code given above (before question 12), what other change(s) are required in the code to make it fully work ?

14. Which of the following expressions will result in an error message being displayed when a program containing it is run ?

(*a*) `2.0/4` (*b*) `"3" + "Hello"` (*c*) `4 % 15`

(*d*) `int("5")/float("3")` (*e*) `float("6"/"2")`

15. Following expression does not report an error even if it has a sub-expression with 'divide by zero' problem :

```
3 or 10/0
```

What changes can you make to above expression so that Python reports this error ?

15. What is the output produced by following code ?

```
a, b = bool(0), bool(0.0)
c, d = str(0), str(0.0)
print (len(a), len(b))
print (len(c), len(d))
```

16. Given a string s = "12345". Can you write an expression that gives sum of all the digits shown inside the string s *i.e.*, the program should be able to produce the result as **15** (1+2+3+4+5).
[Hint. Use indexes and convert to integer]

17. Predict the output if **e** is given input as 'True'

```
a = True
b = 0 < 5
print (a == b)
print (a is b)
c = str (a)
d = str (b)
print (c == d)
print (c is d)
e = input ("Enter :")
print (c == e)
print (c is e)
```

18. Find the errors(s).

(*a*)
```
name = "HariT"
print (name)
name[2] = 'R'
print (name)
```

(*b*)
```
a = bool (0)
b = bool (1)
print (a == false)
print (b == true)
```

(*c*)
```
print (type (int("123")))
print (type (int("Hello")))
print (type (str("123.0")))
```

(*d*)
```
pi = 3.14
print (type (pi))
print (type ("3.14"))
print (type (float ("3.14")))
print (type (float("three point fourteen")))
```

(*e*)
```
print ("Hello" + 2)
print ("Hello" + "2")
print ("Hello" * 2)
```

(*f*)
```
print ("Hello"/2)
print ("Hello" / 2)
```

19. What will be the output produced ?

(*a*)
```
x, y, z = True, False, False
a = x or (y and z)
b = (x or  y) and z
print(a, b)
```

(*b*)
```
x, y  = '5', 2
z = x + y
print(z)
```

(*c*)
```
s = 'Sipo'
s1 = s + '2'
s2 = s * 2
print(s1)
print(s2)
```

(*d*)
```
w,x, y, z = True , 4, -6, 2
result = -(x + z) < y or x ** z < 10
print(result)
```

20. Program is giving a weird result of "0.50.50.50.50.50.....". Correct it so that it produces the correct result which is the probability value (input as 0.5) times 150.

```
probability = input("Type a number between 0 and 1: ")
print("Out of 150 tries, the odds are that only", (probability * 150), "will succeed.")
```

[**Hint.** Consider its datatype.]

21. Consider the code given below :

```
import random
r = random.randrange(100, 999, 5)
print(r, end = ' ')
r = random.randrange(100, 999, 5)
print(r, end = ' ')
r = random.randrange(100, 999, 5)
print(r)
```

Which of the following are the possible outcomes of the above code ? Also, what can be the maximum and minimum number generated by line 2 ?

(*a*) 655, 705, 220 (*b*) 380, 382, 505 (*c*) 100, 500, 999 (*d*) 345, 650, 110

22. Consider the code given below :

```
import random
r = random.randint(10, 100) - 10
print(r, end = ' ')
r = random.randint(10, 100) - 10
print(r, end = ' ')
r = random.randint(10, 100) - 10
print(r)
```

Which of the following are the possible outcomes of the above code ? Also, what can be the maximum and minimum number generated by line 2 ?

(*a*) 12 45 22 (*b*) 100 80 84 (*c*) 101 12 43 (*d*) 100 12 10

23. Consider the code given below :

```
import random
r = random.random() * 10
print(r, end = ' ')
r = random.random() * 10
print(r, end = ' ')
r = random.random() * 10
print(r)
```

Which of the following are the possible outcomes of the above code? Also, what can be the maximum and minimum number generated by line 2 ?

(*a*) 0.5 1.6 9.8 (*b*) 10.0 1.0 0.0 (*c*) 0.0 5.6 8.7 (*d*) 0.0 7.9 10.0

24. Consider the code given below :

```
import statistics as st
v = [7, 8, 8, 11, 7, 7]
m1 = st.mean(v)
m2 = st.mode(v)
m3 = st.median(v)
print(m1, m2, m3)
```

Which of the following is the correct output of the above code ?

(*a*) 7 8 7.5 (*b*) 8 7 7 (*c*) 8 7 7.5 (*c*) 8.5 7 7.5

Type C : Programming Practice/Knowledge based Questions

1. Write a program to obtain principal amount, rate of interest and time from user and compute simple interest.
2. Write a program to obtain temperatures of 7 days (Monday, Tuesday ... Sunday) and then display average temperature of the week.
3. Write a program to obtain x, y, z from user and calculate expression : $4x^4 + 3y^3 + 9z + 6\pi$.
4. Write a program that reads a number of seconds and prints it in form : mins and seconds, *e.g.*, 200 seconds are printed as 3 mins and 20 seconds.
   [Hint. use // and % to get minutes and seconds]
5. Write a program to take year as input and check if it is a leap year or not.
6. Write a program to take two numbers and print if the first number is fully divisible by second number or not.
7. Write a program to take a 2-digit number and then print the reversed number. That is, if the input given is **25**, the program should print 52.
8. Try writing program (similar to previous one) for three digit number *i.e.*, if you input 123, the program should print 321.
9. Write a program to take two inputs for day, month and then calculate which day of the year, the given date is. For simplicity, take 30 days for all months. For example, if you give input as : **Day3, Month2** then it should print "Day of the year : 33".
10. Write a program that asks a user for a number of years, and then prints out the number of days, hours, minutes, and seconds in that number of years.

```
How many years ? 10
10.0 years is :
3650.0 days
87600.0 hours
5256000.0 minutes
315360000.0 seconds
```

11. Write a program that inputs an age and print age after 10 years as shown below :

```
What is your age ? 17
In ten years, you will be 27 years old!
```

12. Write a program whose three sample runs are shown below :

Sample Run 1 :

```
Random number between 0 and 5 (A) : 2
Random number between 0 and 5 (B) :5.
A to the power B = 32
```

Sample Run 2 :

```
Random number between 0 and 5 (A) : 4
Random number between 0 and 5 (B) :3.
A to the power B = 64
```

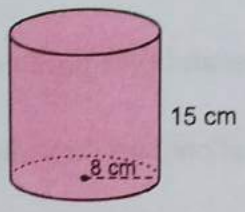
Sample Run 3 :

```
Random number between 0 and 5 (A) : 1
Random number between 0 and 5 (B) :1.
A to the power B = 1
```

13. Write a program that generates six random numbers in a sequence created with (start, stop, step). Then print the mean, median and mode of the generated numbers.
14. Write a program to generate 3 random integers between 100 and 999 which is divisible by 5.
15. Write a program to generate 6 digit random secure OTP between 100000 to 999999.
16. Write a program to generate 6 random numbers and then print their mean, median and mode.
17. Write a program to find a side of a right angled triangle whose two sides and an angle is given.
18. Write a program to calculate the radius of a sphere whose area ($4\pi r^2$) is given.
19. Write a program that inputs a string and then prints it equal to number of times its length, *e.g.*,

```
Enter string : "eka"
Result ekaekaeka
```

20. Find the volume of the cylinder ($\pi r^2 h$) as shown :



21. Write a program to calculate the area of an equilateral triangle. (area $= \frac{\sqrt{3}}{2}$ * side * side).
22. Write a program to input the radius of a sphere and calculate its volume $\left(V = \frac{4}{3}\pi r^3\right)$.
23. Write a program to calculate amount payable after simple interest.
24. Write a program to calculate amount payable after compound interest.
25. Write a program to computer $(a+b)^3$ using the formula $a^3 + b^3 + 3a^2b + 3ab^2$.