3 Data Handling

In This Chapter

3.1 Introduction
3.2 Data Types
3.3 Mutable and Immutable Types
3.4 Operators
3.5 Expressions

3.1 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.

IMPORTANT : In this chapter and onwards, we have shown screenshots of Python shell used in Spyder IDE (IPython shell) which uses In[ ] : prompt to take commands. But in text section, we have given commands with >>> prompt, which is a standard way of telling that this command is given on a Python shell prompt.

All Python shells execute the code in the same way, though IPython shell is an enhanced shell, used in many universities (*e.g.*, Toronto University). You are free to use any Python IDE/shell of your choice, but we have installed Python using **Anaconda distribution** which installs maximum libraries (such as NumPy, SciPy, Panda and many more) along with Python. As **Spyder IDE** and **IPython shell** is available as part of *Anaconda Python* distribution and is very easy to use, we have given screenshots of *Spyder IDE* and *IPython shell* in this book at times.

3.2 DATA TYPES

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

Built-in Core Data Types
- Numbers (*int, float, complex*)
- String
- List
- Tuple
- Dictionary

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

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

3.2.1 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

3.2.1A 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.

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

See figure below (left side)

```
In [5]: bool(0)
Out[5]: False

In [6]: bool(1)
Out[6]: True
```

```
In [9]: str(False)
Out[9]: 'False'

In [10]: str(True)
Out[10]: '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.*

3.2.1B 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.

Recall (from Literals/Values' discussion in chapter 2) 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).

Check Point 3.1

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 ?

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".

3.2.1C 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.*

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 $i^2 = -1$.

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.

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

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

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

NOTE

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

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

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

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

For example,

```
>>> z = (1 + 2.56j) + (-4 -3.56j)
>>> a
(-3 -1j)
>>> z.real      <-- It will display real part of complex number z
-3.0
>>> z.imag      <-- It will display imaginary part of complex number z
-1.0
```

TIP

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

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

Table 3.1 *The Range of Python Numbers*

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

Check Point 3.2

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.

3.2.2 Strings

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

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

NOTE

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

Following are all legal strings in Python :

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

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

String as a Sequence of Characters

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

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

The individual elements of a string are the characters contained in it (stored in contiguous memory locations) and as mentioned the characters of a string are given two-way index for each location.

Let us understand this with the help of an illustration as given in Fig. 3.1.

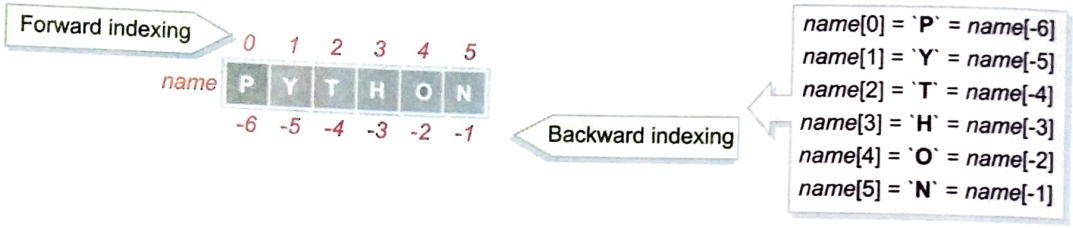


Figure 3.1 Structure of a Python String.

From Fig. 3.1 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. 3.1, you'll write **name[0]**, because the index of first character is 0. You may also write **name [–6]** for the above example *i.e.*, when string name is storing "PYTHON".

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

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

Thus,

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

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

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

  :

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

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

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

Python will return an error like :

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

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.

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.

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

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

individual letter assignment not allowed in Python

will cause an error like :

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

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

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

Strings can be assigned expressions that give strings.

NOTE

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

3.2.3 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.

3.2.3A 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.

3.2.3B 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 list of comma-separated values of any date type within parentheses, *e.g.*, following are some tuples :

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

Tuples shall be discussed in details in a later chapter.

NOTE

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

3.2.4 Dictionary

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

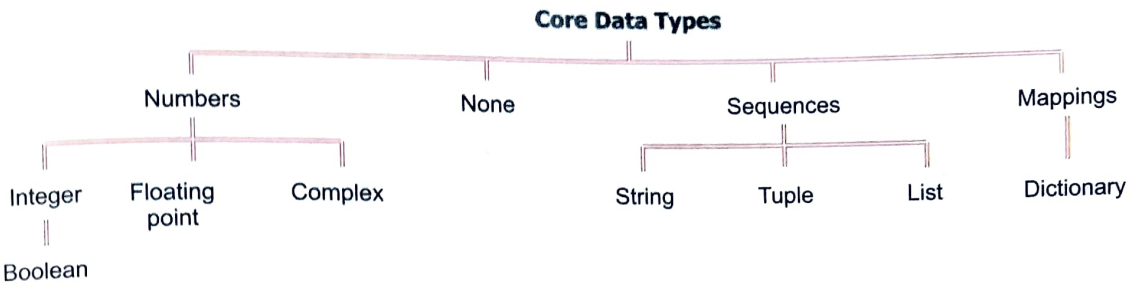
```
{'a' : 1 , 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}

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

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

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

Dictionaries shall be covered in details in a later chapter.

Following figure summarizes the core data types of Python.



3.3 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 3.1

```
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 2.5.1 where we briefly talked about it.

Now consider the **Sample code 3.1** given above. Internally, how Python processes these assignments is explained in Fig. 3.2. Carefully go through figure 3.2 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).

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.*

Mutable Types
- Lists
- Dictionaries
- 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].

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

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

In [62]: Chk[1] = 40

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

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

NOTE

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

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

Lists and Dictionaries shall be covered later in this book.

3.3.1 Variable Internals

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

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

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.

Python object's Key Attributes
- a type
- a value
- an id

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).*

Check Point 3.3

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 ?

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

Now consider this :

Sample code 3.2

```
>>> 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 3.2 will be represented as shown in Fig. 3.4.

```
a = 4
:
b = 5
:
b = b-1
```

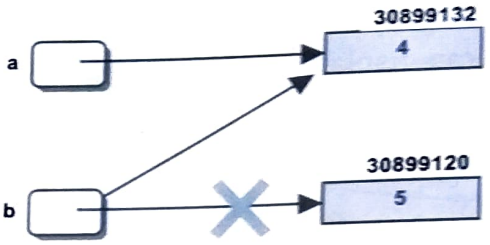


Figure 3.4 Memory representation of sample code 3.2.

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.

DATA TYPES IN PYTHON, MUTABILITY, INTERNALS — Progress In Python 3.1

In the **Python Shell IDLE** or **IPython shell of Spyder IDE**, type the statements as instructed.

1. Using value 12, create data-item (that contains 12 in it) of following types. Give at least two examples for each type of data. Check type of each of your examples using type( ) function, *e.g.*, to check the type of 3.0, you can type on Python prompt type(3.0).

(*a*)	Integer	
(*b*)	Floating point number	
(*c*)	Complex number	

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 3.1 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

3.4 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 5 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**.

3.4.1 Arithmetic Operators

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

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

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 – .

3.4.1A 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 OPERATORS

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

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.

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

3.4.1B Binary Operators

BINARY OPERATORS

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

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 5*** – String Manipulation ; for lists, it will be covered in ***Chapter 7*** – 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 5*** – 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. For Boolean values *True* and *False*, recall that Python internally takes values 1 and 0 (zero) respectively so *True + 1* will give you *2* :).

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

The operands may be of number types.

NOTE

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

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., $\sqrt{x}$, i.e., $\sqrt{49}$)
`27.009 ** 0.3`
evaluates to `2.68814413570761`. ($27.009^{0.3}$)

The operands may be of number types.

Example 3.1 *What will be the output produced by the following code ?*

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

Solution.

```
2.3
2.0
16.0
10.0
1.2
```

Example 3.2 *Print the area of a circle of radius 3.75 metres.*

Solution.

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

Table 3.2 *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-du-lo) or Remainder	60%5 6%5	0 1	divides the two operands and gives the remainder resulting
//	Floor division	7.2 // 2	3.0	divides and truncates the fractional part from the result
**	Exponentiation (Power)	2.5 ** 3	15.625	returns base raised to power exponent (2.5^3 here)

Negative Number Arithmetic in Python

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

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

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

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

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

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

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

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

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

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

IPython console

Console 1/A

```
In [67]: 5// -3
Out[67]: -2

In [68]: -5 //3
Out[68]: -2

In [69]: -7 / 4
Out[69]: -1.75

In [70]: -7 // 4
Out[70]: -2

In [71]: -7 % 4
Out[71]: 1

In [72]: 7 % -4
Out[72]: -1

In [73]: 7 // -4
Out[73]: -2
```

3.4.1C 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 $x + y$.

3.4.2 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.

```
In [14]: 'Apple' > ' Apple'
Out[14]: True

In [15]: 'Apple' == ' Apple'
Out[15]: False
```

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.

For instance, consider the following relational operations.

Given

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

Check Point 3.4

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 3.3 summarizes the action of these relational operators.

Table 3.3 *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 :

```
In [18]: 0.1+0.1+0.1 == 0.3
Out[18]: False

In [19]: print(0.1+0.1+0.1)
0.30000000000000004

In [20]: 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 3.5

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

3.4.4 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.

3.4.4A 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.

3.4.4B The or Operator

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

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

(*i*) Relational expressions as operands

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

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

That is :

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

Following are some examples of this *or* operation :

(4 == 4) or (5 == 8)	results into **True** because first expression (4 == 4) is *True*.
5 > 8 **or** 5 < 2	results into **False** because both expressions 5 > 8 and 5 < 2 are *False*.

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

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

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

That is :

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

NOTE

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

Examples

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

How the truth value is determined ? – refer to section 3.4.4A above.

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

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

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

will give you result as

```
True
```

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

NOTE

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

3.4.4C The and Operator

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

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

(*i*) Relational expressions as operands

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

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

That is :

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

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

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

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

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

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

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

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

How the truth value is determined ? – refer to section 3.4.4A above.

IMPORTANT

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

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

will give you result as

```
False
```

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

NOTE

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

3.4.4D 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.

OPERATORS IN PYTHON

PriP

Progress In Python 3.2

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 3.2 under Chapter 3 after practically doing it on the computer.

>>>❖<<<

Check Point 3.6

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

- *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**.

3.5.1 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 5***.

3.5.1A 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 3.3 *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

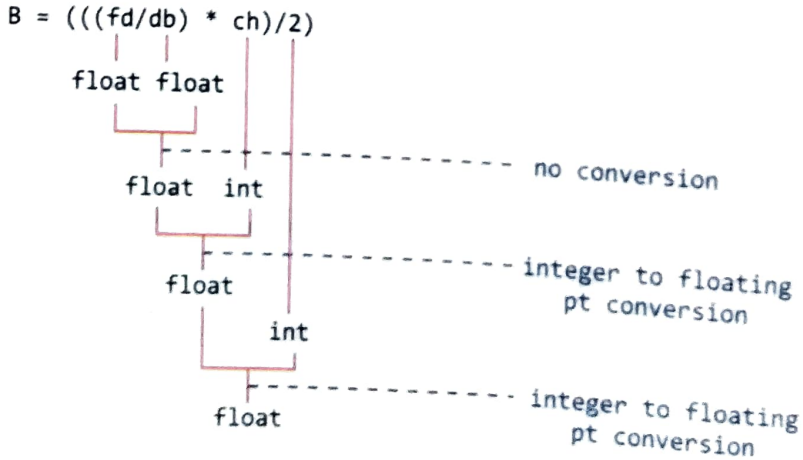
= **(7) / 5.0**

[*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 :

$$(((fd / db) * \text{ch}) / 2)$$

$$= (((\mathbf{36.0 / 5.0}) * 5L / 2) \quad [\textit{no conversion required}]$$

$$= (\mathbf{(7.2 * 5)} / 2) \quad [\textit{int to floating point conversion}]$$

$$= (\mathbf{(7.2 * 5.0)} / 2)$$

$$= (36.0 / 2) \quad [\textit{integer to floating point conversion}]$$

$$= 36.0 / 2.0$$

$$B = 18.0$$

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

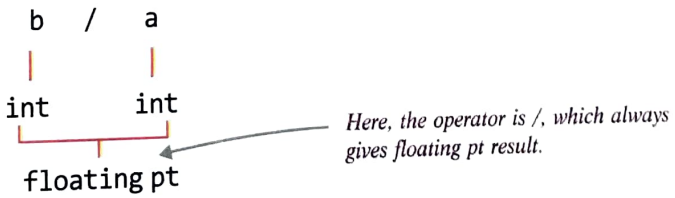
(*a*)
```
a, b = 3, 6
c = b/a
```

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

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

Ans. (*a*) In expression

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

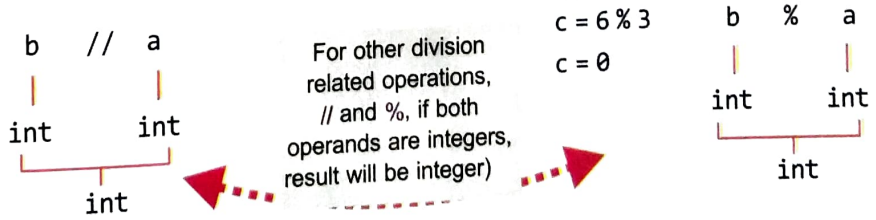


(*b*) In expression

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

(*c*) In expression

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



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

3.5.1B 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 3.5 *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 3.6 *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 $(N <= b)$

3.5.1C 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 3.4.3)

(*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))`

Check Point 3.7

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 ?

(*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.

Example 3.7 *What will be the output of following expression ?*

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

Solution. False

Example 3.8 ***'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.

Check Point 3.8

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

3.5.2 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.

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**.

Working With math Module of Python

Other than 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> .

Following table (Table 3.11) lists some useful math functions that you can use in your programs.

Table 3.11 *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.

S. No.	Function	Prototype (General Form)	Description	Example
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. *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)

- *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>)
```

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)

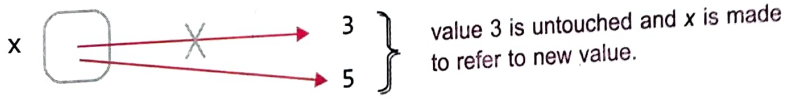
Solution.

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

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
```



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 $false_{real}$

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 $true_{int}$

Line 3

```
5 < (10 or 5)
= 5 < 10
= True
```

10 or 5 returns 10 since 10 is $true_{real}$

Line 4

```
5 < (5 or 10)
= 5 < 5
= False
```

5 or 10 returns 5 since 5 is $true_{int}$

10. *What will be 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 reason(s).*
```
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`

$\therefore$ $a = 3.625$

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 statement ? 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. *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*.

16. *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

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

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

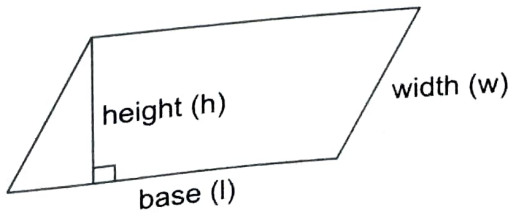
(*ii*) `len(13) or 13`

Solution.

(*i*) **13** because or evaluates first argument 13's truth value, which is *true* $_{tval}$ 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.

18. *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 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
```

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 actron or operation.
Type Casting	Explicit Type Conversion

Assignments

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`

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, 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)
```

6. What will be output produced by following code ? State reason.

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

7. What would be the output produced by following code ? Why ?

```
a, b, c = 0.1
d = 0.3
e = a + b + c - d
f = a + b + c == d
print(e)
print(f)
```

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

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

9. What will be the output of following code ?

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

10. 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)

11. 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 ?

12. 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))
```

13. 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. Uses indexes and convert to integer]

14. 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)
```

15. 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)
```

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 that reads from user – (*i*) an hour between 1 to 12 and (*ii*) number of hours ahead. The program should then print the time after those many hours, *e.g.*,

```
Enter hour between 1-12 : 9
How many hours ahead : 4
Time at that time would be : 1 o'clock
```

6. Write a program to take year as input and check if it is a leap year or not.

7. Write a program to take two numbers and print if the first number is fully divisible by second number or not.

8. 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.

9. Try writing program (similar to previous one) for three digit number *i.e.*, if you input 123, the program should print 321.

10. 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".