CLASS-XI

Study Notes

Data Handling

Period-1

Introduction:

Learning Outcomes

- An in-depth understanding Data Types.
- Introduce the fundamentals of Mutable & Immutable Types.
- Working with different types of Operators.
- Concepts of Expression
- Concept of Random Module
- Debugging Concepts

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.

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.

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

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) Numbers

(iii) Complex Numbers

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). There are two types of integers in Python.

- Integers (signed)
- Booleans

Integers (signed).

It is the normal integer1 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 O 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.

```
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'
```

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).

The str() function converts a value to string. The two objects representing the values False and True (not false or true) are the only Boolean objects in Python.

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.

As we already know 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., 3.5007E03, O.5E-04, I.479101E02 etc

Floating-point numbers have two advantages over integers:

- They can represent values between the integers.
- Operations are usually slower than integer operations 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 (15 digit precision), The range of these numbers is limited by underlying machine architecture subject to available (virtual Memory).

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

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.

Complex Numbers in Python:

Python represents complex numbers in the form A + B j. That is to represent imaginary number, Python uses f (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 number 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 exmples.

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

See, a complex number with non-zero real part is displayed with parentheses around it. But noparentheses around complex number with real part as zero(0).

```
4.5j
>>>print(d)
(1.1+3.4j)
```

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 complex value.

For example,

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

it will display real part of complex number z

it will display imaginary part of complex number z

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

1000 + 1000 + 800 + 400 + 5000 + 1000 + 19000 + 13500 + 11000 + 13000 + 800 + 1500 + 12000

Table 3.1: The Range of Python Numbers.

Datatype	**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**

Strings

You already know about strings (as data) in Python. In this section, we shall be talking about Python's datatype string. A string datatype 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.

Following are all legal strings in Python:

"abce", "1234", "$%^&", "????", "ŠÆÈÉá", "??????", "????", "????", "???", "??"

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.

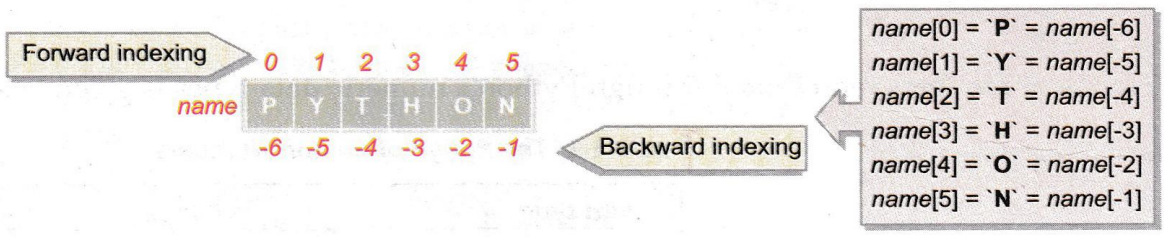


Figure 3.1 Structure of a Python String.

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:

subject	0	1	2	3	4	5	6	7	8
	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>", line1, 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.

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>", line1, 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 expressiosn that give strings.

Lists and Tuples

The lists and tuples are Python's compound datatypes. 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.

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]          #Statement1
```

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

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

```
>>>a[0]=10                    #change 3rd item
>>>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.

Tuples

You can think of Tuples (pronounced as tu-pp-le, thyming 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.

Dictionary

Dictionary datatype 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}
```

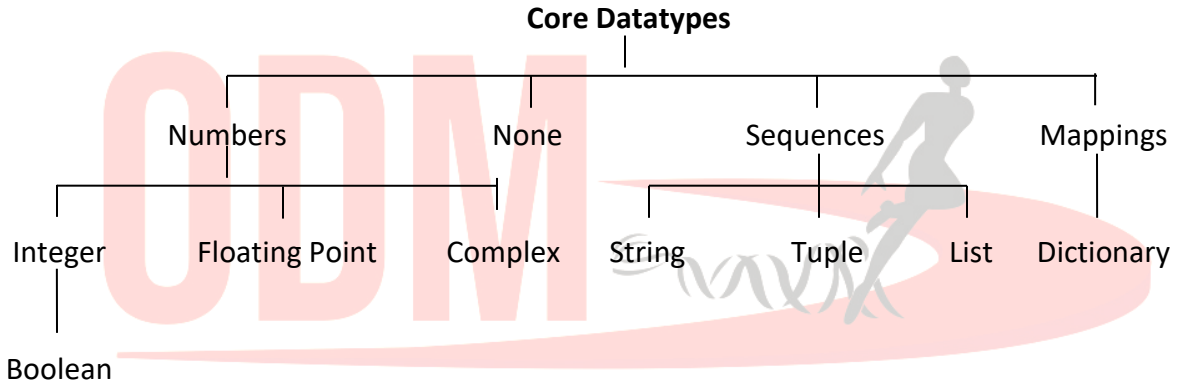
```
>>>vowels={'a' : 1, 'e' : 2, 'i' : 3, 'o' : 4, 'u' : 5}
>>>vowels['a']
1
>>>vowels['u']
5
```

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

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

Dictionaries shall be covered in details in a later chapter.

Following figure summarizes the core datatypes of Python.



MUTABLE AND IMMUTABLE TYPES

The Python data objects can be broadly categorized into *two-mutuable* 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, tupes.*

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).

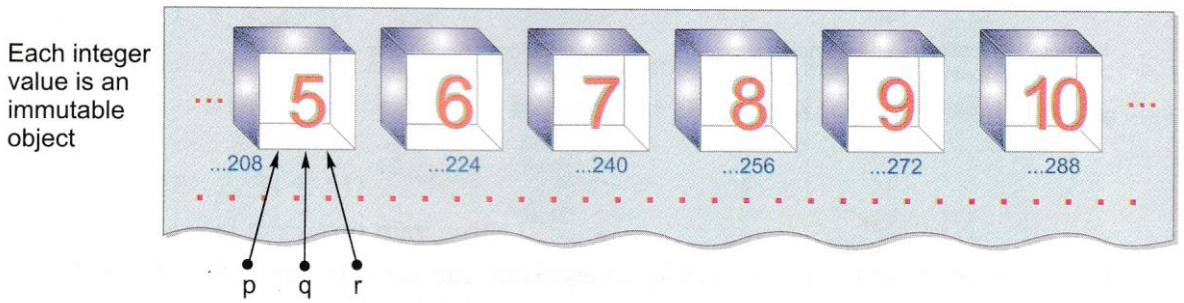
➢ Initially these three statements are executed:

p=5

q=p

r=5

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



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

```
In [40]: p = 5

In [41]: q = p

In [42]: r = 5

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

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

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

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

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

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

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

p=10

r=7

q=r

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

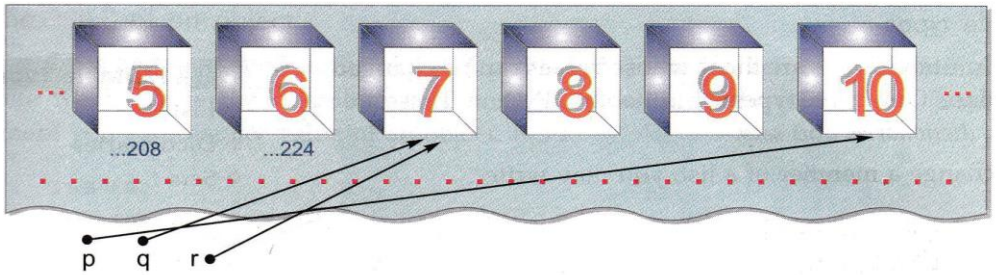


Figure 3.3

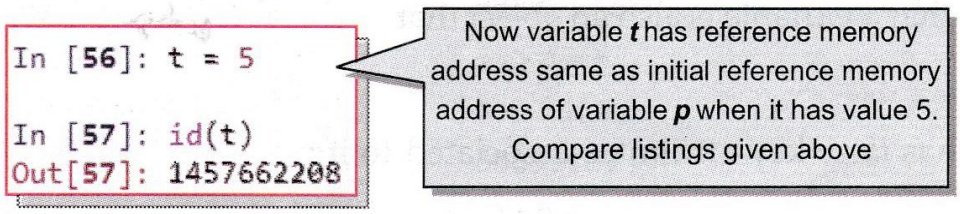
Let us check their ids

```
In [47]: p=10

In [48]: r=7

In [49]: q=r

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

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

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

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

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

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

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

The value 5 is at the same address

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



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

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

The objects of following value types are immutable in Python:

- Integer
- Booleans
- Tuples
- Floating point number
- Strings

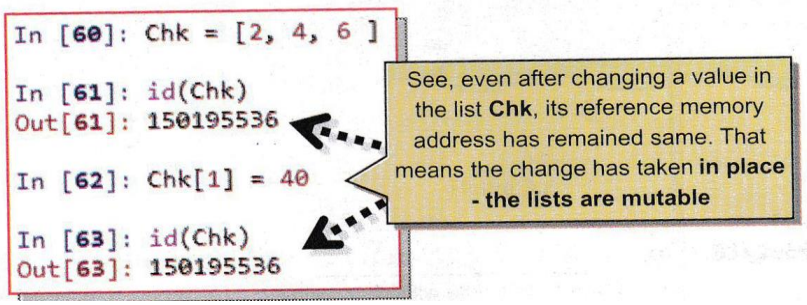
Mutable types

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

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

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

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



Lists and Dictionaries shall be covered later in this book.

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.

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
>>>tye(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*

(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 vaoue of an object. For example,

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

value of integer literal 4 is 4

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

(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. Build-in function **id()** returns the **id** of an object, *e.g.,*

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

Object 4 is internally stored at location 30899132

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

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

Now consider this:

Sample code 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.

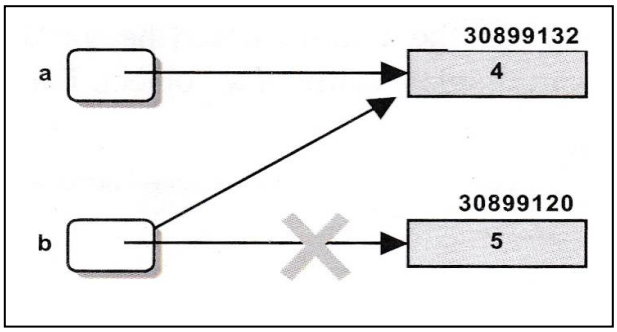
Thus internal change in value of variable *b* (from 5 to 4) of sample code 3.2 will be represented as shown in Fig. 3.4.

a=4

:

b=5

b=b-1



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

DATA TYPES IN PYTHON, MUTABILITY, INTERNALS

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 atleast 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	

Period-02

OPERATORS

The operations being carried out on data, are represented by operatiors. 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) Arithmetric 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.

Arithmetic Operators

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

+	addition
-	substraction
*	multiplication
/	division
//	floor division
%	remainder
**	exponentiation

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

Unary Operators

Unary +

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

if a = 5 then + a means 5.

if a = 0 then + a means 0.

if a = -4 then + a means -4.

Unary –

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

if a = 5 then –a means -5.

if a = 0 then –a means 0 (there is no quantity known as -0)

if a = -4 then –a means 4.

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

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 srings. This functionality will be covered in Chapter 5 – String Manipulation.

4. Division Opeator (/)

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

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

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

5. Floor Division Operator (//)

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

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

a=15.9, b=3

a/b evaluates to 5.3.

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

If a=15.9, b=3,

a/b will evaluate to 5.0

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

Consider some more examples:

100/32	evaluates to 3.0
7//3	evaluates to 2
6.5/22	evaluates to 3.0

The operands may be of number types.

6. Modulus operator (%)

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

For example,

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

Similarly,

7.2 % 3 will yield 1.2

6 % 2.5 will yield 1.0

The operands may be of number types.

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

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 $\left(x^{0.5}, \text{i.e.} \sqrt{x}, \text{i.e.,} \sqrt{49}\right)$

27.009 ** 0.3

evaluates to 2.68814413570761. ($27.009^{0.3}$)

The operands may be of number types.

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: 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)5(-2
 6
-1
```

b)
```
3)-5(-2
-6
+1
```

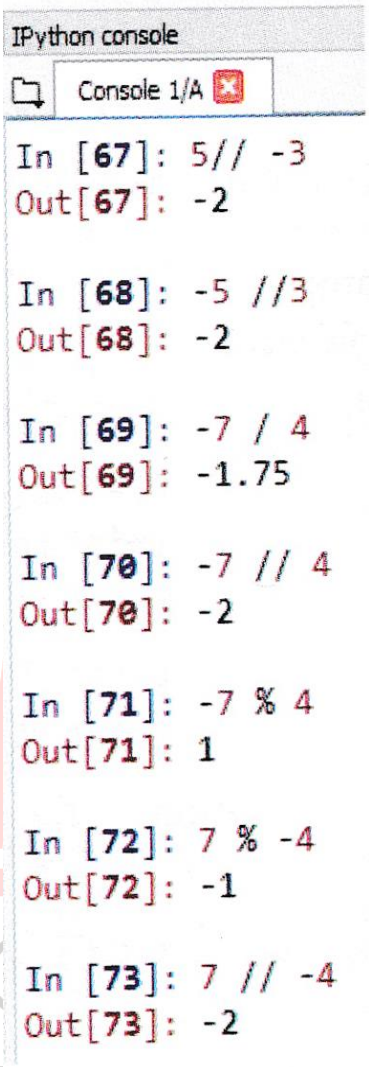
c)
```
4)-7(-1.75
-4
-3
-3
 0
```

d)
```
4)-7(-2
-8
 1
```

e)
```
4)-7(-2
-8
+1
```

f)
```
4)7(-2
 8
-1
```

g)
```
-4)7(-2
 8
-1
```



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

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 oerators). 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 :

< 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).
- 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' == 'Apple
Out [15] : False
```

- 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='Godhouse',

L=[1, 2, 3], M=[2, 4, 6], N=[1, 2, 3]

O=(1, 2, 3), P=(2, 4, 6), Q=(1, 2, 3)

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	wil return False
0==False	will return True
1==True	will return True

Both match upto the Letter 'd' but 'God' is shorter than 'Godhouse' so it comes first in the dictionary.

Both match upto the Letter 'd' but 'God' is shorter than 'Godhouse' so it comes first in the dictionary.

Table: 3.3 summarizes the action of these relational operators.

Table: 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 approsximation 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 … expression1

corresponds to

(a+5) > (c-2) … expression2

and not the following

a + (5>c) -2 … expression3

Expression 1 means the expression 2 and not the expression3.

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.

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

value==3

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

value = 3

assigns 3 to value1; no comparison takes place.

Period-03

Identity Operators

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

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

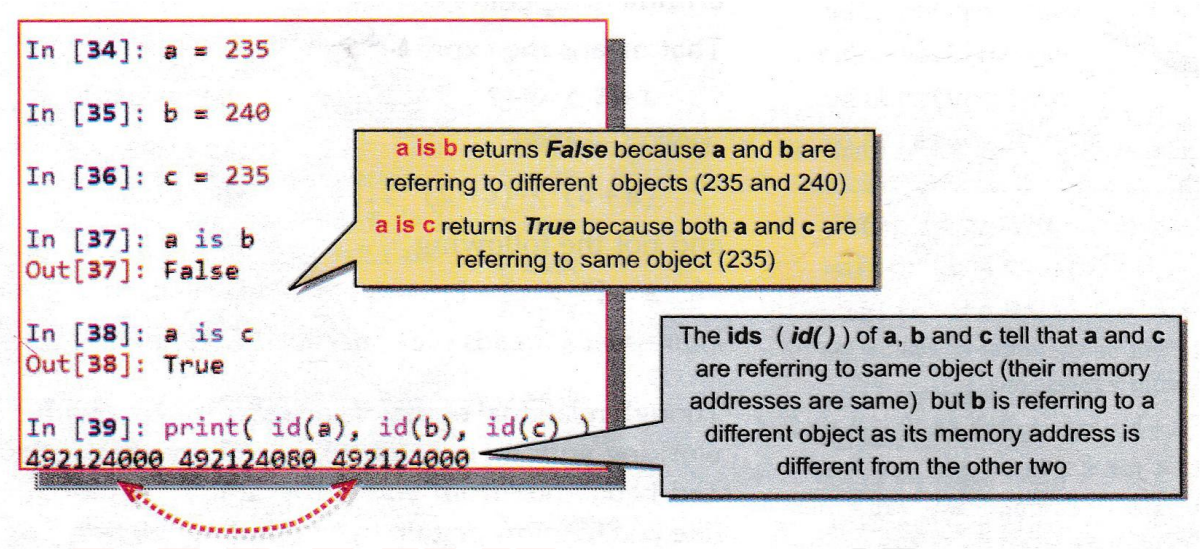
Consider the following examples:

A=10

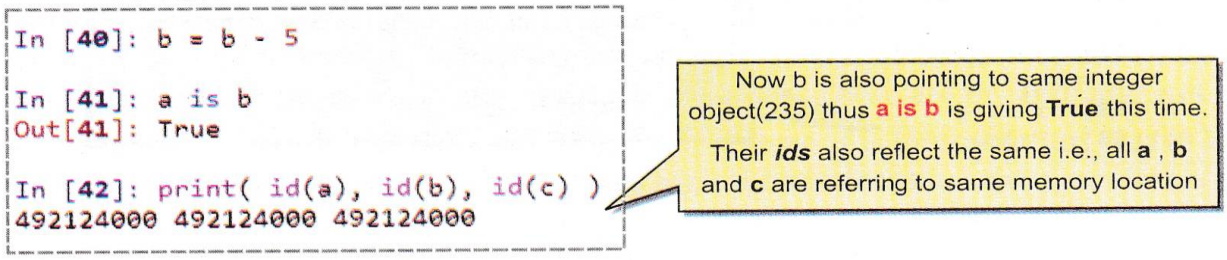
B=10

A is B will return True because both are referencing the memory address of value 10.

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



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



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

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

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

```
In [58]:  print (a, b )
235 235

In [59]: a is b
Out[59]: True

In [60]: a == b
Out[60]: True
```

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

See in the screenshots shown here.

```
In [45]: s1 = 'abc'

In [46]: s2 = input("Enter a string:")

Enter a string:abc

In [47]: s1 == s2
Out[47]: True

In [48]: s1 is s2
Out[48]: False

In [49]: s3 = 'abc'

In [50]: s1 is s3
Out[50]: True
```

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

Similarly,

```
In [51]: i = 2+3.5j

In [52]: j = 2+3.5j

In [53]: i is j
Out[53]: False
```

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

Also,

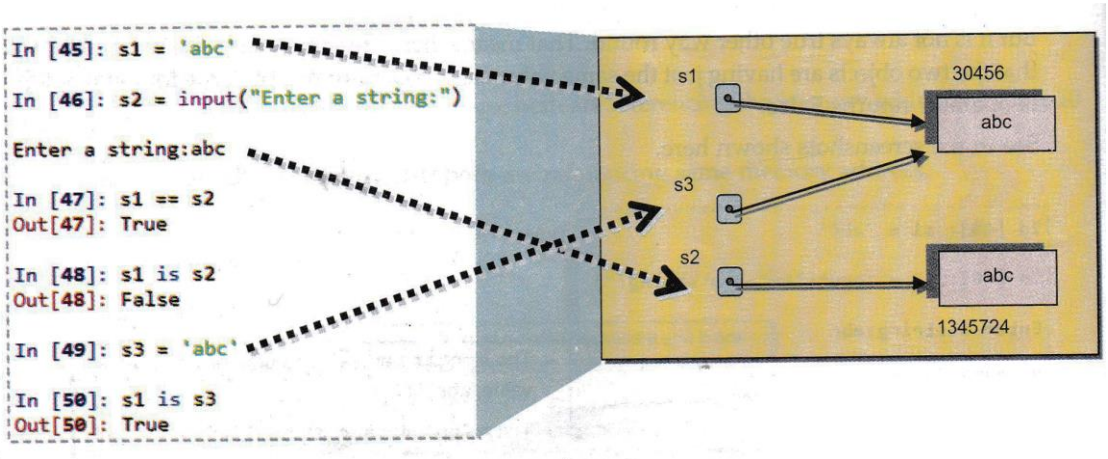
```
In [54]: k = 3.5

In [55]: l = float(input("Enter a value:"))

Enter a value:3.5

In [56]: k == l
Out[56]: True

In [57]: k is l
Out[57]: False
```

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

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

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

Following figure illustrates one of the above given screenshots.



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

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.

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:

<table>
<tr><th>Values with truth value as false</th><th>Values with truth value as true</th></tr>
<tr><td>None</td><td rowspan="5">All other values are considered true.</td></tr>
<tr><td>False (Boolean value False)</td></tr>
<tr><td>Zero of any numeric type, for example, 0, 0.0, 0j</td></tr>
<tr><td>any empty sequence, for example, '', (), []
(please note, "is empty string; () is empty tuple; and [] is empty list)</td></tr>
<tr><td>any empty mapping, for example, {}</td></tr>
<tr><td colspan="2">The result of a rational expression can be True or False depending upon the values of its operands and the comparison taking place.</td></tr>
</table>

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 datatype, 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.

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

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 3.4.4 A above.

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

20>10 or "a" + 1 >1

will give you result as

True

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

Period -04

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	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[8]

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

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

That is:

x	y	x and y
False $_{tval}$	False $_{tval}$	x
False $_{tval}$	True $_{tval}$	x
True $_{tval}$	False $_{tval}$	y
True $_{tval}$	True $_{tval}$	y

Examples

Operation	Results into	Reason
0 and 0	0	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 first 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' or 'j'	'j'	first expression ('a') has true$_{tval}$, hence second expression 'j' is returned.

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

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.

Following table summarizes the logical operators.

Table 3.4 The Logical Operators

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

Chained Comparison Operators

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

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

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

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

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

Similarly consider some more examples:

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

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

Bitwise Operators

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

Python provides following Bitwise operators.

Table 3.5 Bitwise Operators

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

Let us examine them one by one.

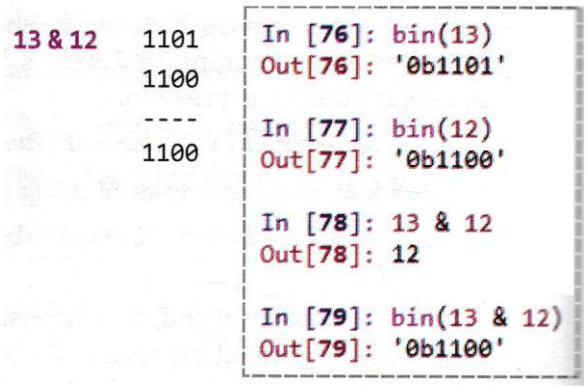
The AND operator &

When its operands are numbers, the & operation performs the bitwise AND function on each parallel pair of bits in each operand. The AND function sets the resulting bit to 1 if the corresponding bit in both operands is 1, as shown in the following Table 3.6.

Table 3.6 The Bitwise AND (&) Operation

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

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



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

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

The inclusive OR operatir |

When both of its operands are numbers, the | operator performs the inclusive OR operation. Inclusive OR means that if either of the two bits is 1, the result is 1. The following Table 3.7 shows the results of inclusive OR operations.

Table 3.7 The inclusive OR (|) Operation

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

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

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

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

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

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

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

The eXclusive OR (XOR) Operator ^

Exclusive OR means that if the two operand bits are different, the result is 1; otherwise the result is 0. The following Table 3.8 shows the results of an eXclusive OR operation.

Table 3.8 The eXclusive OR (^) Operation

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

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

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

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

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

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

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

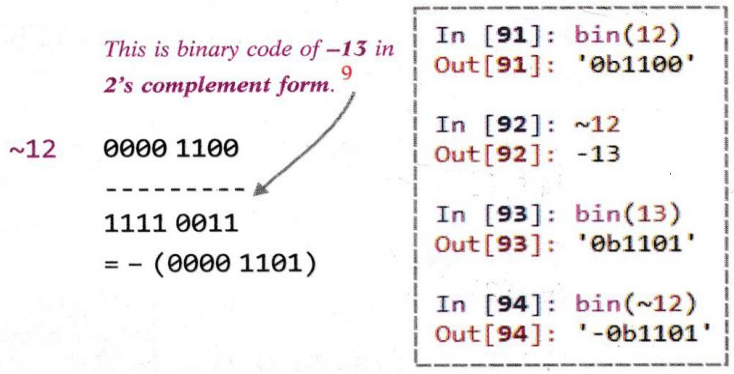
The Complement Operators ~

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

Table 3.9 The Complement (~) Operation

op1	Result
0	1

1	0



Operator Precedence

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

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

Operator Associativity:

Python allows multiple operators in a single expression as you have learnt above, e.g., a<b+2<c or p<q>r etc. If the operators used in an expression have different preceedence, there is not any

problem as Python will evaluate the operator with higher precedence first. BUT what if the expression contains two operators that have the same precedence?

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

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

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

For example,

In[1]: 7*8/5//2

out[1]: 5,0

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

In[2]: (((7*8)/5)//2)

Out[2]: 5,0

In[3]: 7*((8/5)//2)

Out[3]: 0,0

In[4]: 7*(8/5//2))

Out[4]: 28,0

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

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

Consider following example:

In[11]: 3**3**2

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

Out[11]: 19683

In[12]: 3**(3**2)

Out[12]: 19683

In[13]: (3**3)**2

Out[13]: 729

Operators in Python:

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.

Period-05

EXPRESSIONS

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

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

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

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

Let us talk about these one by one.

1. Arithmetic Expressions

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

2. Relational Expressions

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

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

3. Logical Expressions

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

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

4. String Expressions

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

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

For instance, following are some legal string expressions:

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

String manipulation is being covered in a separate chapter-**chapter 5**.

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**.

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 is 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 sub-expression 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 upto 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;

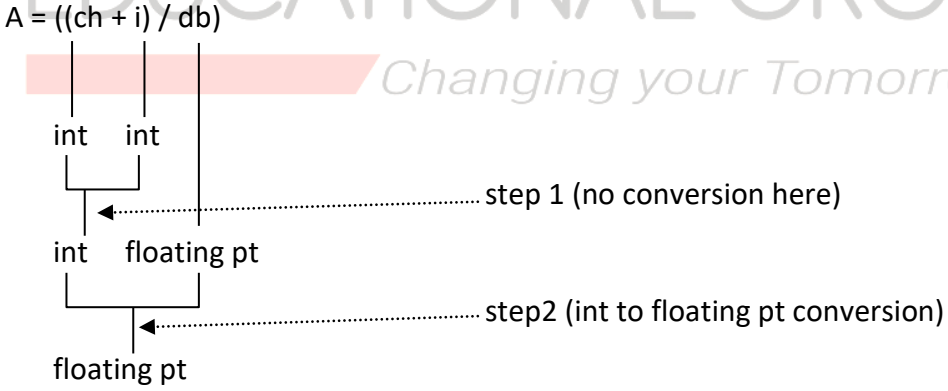
- Otherwise, if either argument is a floating point number, the other is converted to floating point;
- No conversion if both operands are integers.

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+1)/db       #expression1
B=fd/db*ch/2        #expression2
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 expression2 will be floating point number.

The expression, expression 2 will be evaluated as:

(((fd/db)*ch)/2)

= (((36.0/5.0)*5L/2) [*no conversion required*]

=((7.2*5)/2) [*int to floating point conversion*]

=((7.2*5.0)/2)

=(36.0/2) [*integer to floating point conversion*]

=36.0/2.0

B=18.0

The output will be 1.4
18.0

The final data type 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 datatype?

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

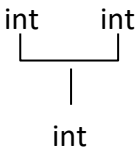
b / a
int int
floating pt

Here, the operator is /, which always gives floating

b) In expression

c =6//3 b // a

c =2

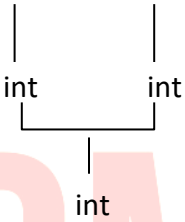


For other division related operations, // and % , if both operands are integers, result will be integer.

c) In expression

c = 6%3 b % a

c=0



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

Period-06

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* of *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 the *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: 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:	For int combination (ii), the output would be:
True	False
True	True
True	False

Example: 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)

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))

iii) **Important:** While evaluating, Python minimize 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: What will be the output of following expression?

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

Solution: False

Example: '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.

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 datatype. This type of conversion is automobile, 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**.

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

<datatype>(expression)

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

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

int(b)

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

Similarly,

d=float(a)

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

Python offers some conversion functions that you can use to type cast a value in Python. These are being listed in following Table 3.10.

Table Python Data Conversion Functions

S. N.	Conversion		Conversion Function	Examples
	From	To		
1	any number-convertible type e.g., a float, a string having digits	integer	int()	**int(7,8)** will give 7 (floating point number to integer conversion) **int('34')**^{10} will give 34 (string to integer conversion)
2	any number-convertible type e.g., a float, a string having digits	floating point number	float()	**float(7)** will give **7.0** (integer to floating point number conversion)

				float('34') will give **34.0** (string to floating point number conversion)
3	numbers	complex number	complex()	**complex(7)** will give 7+0j (ONE ARGUMENT-integer to complex number conversion) **complex(3,2)** will give 3+2j (TWO ARGUMENTS- integer to complex number conversion)
4	number Booleans	string	str()	**str(3)** will give 'e' (integer to string conversion) **string(5.78)** will give '5.78' (floating-point number to string conversion) **str(0o17)** will give '15' (octal number to string conversion; string converts the equivalent decimal number to string: 0o17-15) **str(1+2j)** will give '(1+2j)' (complex number to string conversion) **str(True)** will give 'True' (Boolean to string conversion)
5	any type	Boolean	bool()	**bool(0)** will give **False: bool(0.0)** will give **False bool(1)** will give **True; bool(3)** will give **True; bool('')** will give **False; bool('a')** will give **True; bool('hello')** will give **True** with **bool(),** non-zero, non-empty values of any type will give **True**

				and rest (zero, empty values) will give **False**.

10. If a number (in string form) is given in any other base e.g., octal or hexadecimal or binary, it can also be converted to integers using **int()** as int (<number-in-string-form>, bae). For example to convert a string '0o11' (octal bases equivalent of) into integer, you can write **int('0o11', 8)** and it will give 8.

    If you want to convert an integer to octal or hexadecimal or binary form then oct(), hex() or bin() function respectively are there but they produce the equivalent number in string they produce the equivalent number in string form i.e., hex(10) will give you 'oxA'. This value can be displayed or printed but cannot be used in calculations as it is not number. However, by combining hex(), oct(), bin() with int (<number string>, base) you can convert to appropriate type.

Type Casting Issues

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

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

With this, we have come to end of our chapter. Let us quickly recap what we have learnt so far.

Period-07

Expression Evaluation

This is an important practical session to reinforce the concepts of expression evaluation in Python. Proper practice of this PriP Session would lay a strong foundation, which would help you solve application based questions like outputs, errors etc. This practical session would involve both interactive mode and script mode.

Question1: (Solved for your reference)

Evaluate the following expression (stepwise) on paper first. Then execute it in Python shell. Compare your result with Python's. State reasons

```
len(str(10<20))
```

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 **marh** module, you need to first *import* it to your program by giving statement as follows as the top line of your Python script:

import math

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

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

Table 3.11: Some Mathematical Functions in math Module

S. N.	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(-103) gives -1.0.
2	sqrt	math.sqrt(num)	The **sqrt()** function returns the square root of *num*. If sum<0, domain error occurs.	math.sqrt(81.0) gives 9.0
3	exp	math.exp(arg)	The **exp()** function returns the natural logarithm e railsed 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
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 base2.
7	log10	math.log10(num)	The **log10()** function returns the base 10 logarithm for *num*. A domainerror 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 *base<0* and *exp* is not integer.	math.pow(3.0,0) gives value of 3º. math.pow(4.0, 2.0) gives value of 4^2.
9	sin	math.sin(arg)	The **sin()** function returns the sine of *arg*. The value of *arg* must be in radians.	math.sin(val) (*val* is a number)
10	cos	math.cos(arg)	The **cos()** function returns the cosine of *arg.* The value of *arg* must be in radians.	math.cos(val) (*val* is a number)
11	tan	math.tan(arg)	The **tan()** function returns the tangent of *arg.* The value of *arg* must be in radians.	math.tan(val) (*val* is a number)
12	degrees	math.degrees(x)	The **degrees()** converts angle *x* from radians to degrees.	math.degrees(3.14) would give 179.91

13	radians	math.radians(x)	The **radians()** converts angle *x* from degrees to radians.	math.radians(179.91) would give 3.14

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

math.pi gives the mathematical constant $\pi = 3.141592\ldots$ to availale 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=10-24.113, 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) $p+\frac{q}{(r+s)^4}$

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

v) $|e^2-x|$

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)

Period-08

Discussion of Output Questions

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

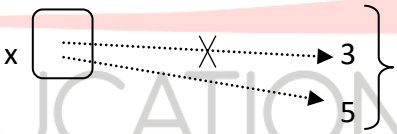
Solution:

3	integer	3j	complex number
13.0	Floating-point number	'13'	string
"13"	String	2+0j	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 variable referring to immutable type, variable's reference is changed and the previous value is left unchanged. e.g.,

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



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

6. What will be the output of the following?

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

Solution: 1
3

because	and
len(str(17//4))	len(str(17/4))
=len(str(4))	=len(str(4.0))
=len('4')	=len('4.0')
=1	=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 produced 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
```

Explain:

Line1 5<5 or 10 precedence of < is higher than or

	= False or 10	
	= **10**	because or would evaluate the second argument if first argument is False or false$_{tval}$
Line2	5<10 or 5	precedence of < is higher than or
	=True or 5	
	=**True**	or would return first argument if it is True of true$_{tval}$
Line3	5<(10 or 5)	
	=5<10	10 or 5 returns 10 since 10 is true$_{tval}$
	=**True**	
Line4	5<(5 or 10)	
	=5<5	5 or 10 returns 5 since 5 is true$_{tval}$
	=**False**	

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 (Please note '0' is a non-empty string and

True and True hence has truth value as $true_{tval}$)

=**True**

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

Explain:

Line1	a=3+5/8 =3+0.625 ∴a=3.625	
Line2	b=int(3+5/8) =int(3+0.625) =int(3.625) **b=3**	(int() will drop the fractional part)
Line3	c=3+float(5/8) c=3+float(0.625) =3+0.625 **c=3.625**	
Line4	d=3+float(5)/8 =3+5.0/8 =3+5.0/8 d=**3.625**	5.0/8=0.625 because one operand is floating point, the integer operand will be internally converted to floating-pt.
Line5	e=3+5.0/8 =3+0.625 **e=3.625**	(same reason as above)
Line6	f=int(3+5/8.0) f=int(3+0.625) f=int(3.0) f=3	(same reason as above) (int()will drop the fractional part)

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 reason(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==2in (c) and 2.0==2.0 in (e).

(d) evaluates to **True** as both 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 bool() 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 first argument **len(13)**, Python gives error as **len()** works on strings only.

***