CHAPTER

3

DATATYPES IN PYTHON

We have already written a Python program to add two numbers and executed it. Let's now view the program once again here:

```
# First Python program to add two numbers
a = 10
b = 15
c = a + b
print("Sum= ", c)
```

When we compile this program using Python compiler, it converts the program source code into byte code instructions. This byte code is then converted into machine code by the interpreter inside the Python Virtual Machine (PVM). Finally, the machine code is executed by the processor in our computer and the result is produced as:

```
F:\PY>python first.py
Sum= 25
```

Observe the first line in the program. It starts with the # symbol. This is called the comment line. A comment is used to describe the features of a program. When we write a program, we may write some statements or create functions or classes, etc. in our program. All these things should be described using comments. When comments are written, not only our selves but also any programmer can easily understand our program. It means comments increase readability (understandability) of our programs. This is highly needed when we are working on a project. We should remember that project development is not a single man's task. It is a collective work from a team of professionals. Since a project is handled by several people, everyone in the project should be able to understand each and every program. This makes possible modifying and reusing of the programs. Hence, writing comments will make our programs clear to others.

Comments in Python

There are two types of comments in Python: single line comments and multi line comments.

Single line comments

These comments start with a hash symbol (#) and are useful to mention that the entire line till the end should be treated as comment. For example,

```
# To find sum of two numbers
a = 10  # store 10 into variable a
```

Here, the first line is starting with a # and hence the entire line is treated as a comment. In the second line, a = 10 is a statement. After this statement, # symbol starts the comment describing that the value 10 is stored into variable 'a'. The part of this line starting from # symbol to the end is treated as a comment.

Comments are non-executable statements. It means neither the Python compiler nor the PVM will execute them. Comments are for the purpose of understanding human beings and not for the compiler or PVM. Hence, they are called non-executable statements.

Multi line comments

When we want to mark several lines as comment, then writing # symbol in the beginning of every line will be a tedious job. For example,

```
# This is a program to find net salary of an employee
# based on the basic salary, provident fund, house rent allowance,
# dearness allowance and income tax.
```

Instead of starting every line with # symbol, we can write the previous block of code inside """ (triple double quotes) or ''' (triple single quotes) in the beginning and ending of the block as:

```
"""
This is a program to find net salary of an employee
based on the basic salary, provident fund, house rent allowance,
dearness allowance and income tax.
"""
```

Or,

```
'''
This is a program to find net salary of an employee
based on the basic salary, provident fund, house rent allowance,
dearness allowance and income tax.
'''
```

The triple double quotes (""") or triple single quotes (''') are called 'multi line comments' or 'block comments'. They are used to enclose a block of lines as comments.

Docstrings

In fact, Python supports only single line comments. Multi line comments are not available in Python. The triple double quotes or triple single quotes are actually not multi line comments but they are regular strings with the exception that they can span multiple lines. That means memory will be allocated to these strings internally. If these strings are not assigned to any variable, then they are removed from memory by the garbage collector and hence these can be used as comments.

So, using """ are ''' or not recommended for comments by Python people since they internally occupy memory and would waste time of the interpreter since the interpreter has to check them.

If we write strings inside """ or ''' and if these strings are written as first statements in a module, function, class or a method, then these strings are called *documentation strings* or *docstrings*. These docstrings are useful to create an API documentation file from a Python program. An API (Application Programming Interface) documentation file is a text file or html file that contains description of all the features of software, language or a product. When a new software is created, it is the duty of the developers to describe all the classes, modules, functions, etc. which are written in that software so that the user will be able to understand the software and use it in a proper way. These descriptions are provided in a separate file either as a text file or an html file. So, we can understand that the API documentation file is like a help file for the end user.

Let's take an example to understand how to create an API documentation file. The following Python program contains two simple functions. The first function is add() function that takes two numbers and displays their sum. The second one is message() function that displays a message "Welcome to Python". A function contains a group of statements and intends to perform a specific task. In Python, a function should start with the 'def' keyword. After that, we should write the function name along with parentheses as def add(). We will learn about functions fully in a later chapter.

Observe that the add(x,y) function calls a print() function that displays the sum of x and y. But it also contains documentation strings inside a pair of """. Similarly, the message() function which is the second function calls a print() function that displays the message "Welcome to Python". This function also contains documentation strings written inside a pair of '''. At the end of this program, we are calling these two functions. While calling the first function, we are passing two values 10 and 25 and calling it as add(10, 25). While calling the second function, we are not passing any value. We are calling it as message() only. Now, consider the following program:

```
# program with two functions
def add(x, y):
    """
    This function takes two numbers and finds their sum.
    It displays the sum as result.
    """
    print("Sum= ", (x+y))
```

```
def message():
    '''
    This function displays a message.
    This is a welcome message to the user.
    '''
    print("Welcome to Python")

# now call the functions
add(10, 25)
message()
```

After typing this program, save it as ex.py and then execute it as:

```
F:\PY>python ex.py
Sum= 35
Welcome to Python
```

This is the general way of executing any Python program. Now, we will execute this program once again to create API documentation file. For this purpose, we need a module *pydoc*. In the following command, we are using –m to represent a module and pydoc is the module name that is used by the python interpreter. After that –w indicates that an html file is to be created. The last word 'ex' represents the source file name.

```
F:\PY>python  -m  pydoc  -w  ex
Sum= 35
Welcome to Python
Wrote to ex.html
```

As understandable from the above lines, the output of the program is displayed and also the ex.html file is created. Open the ex.html file in any Web browser, as shown in Figure 3.1:

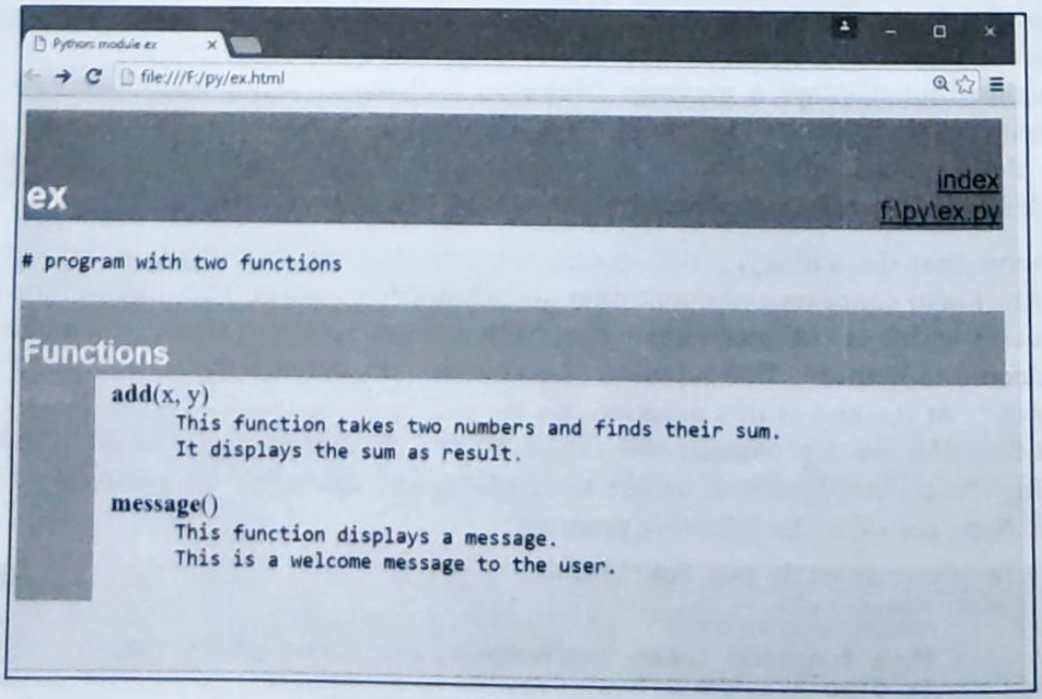

Figure 3.1: API Documentation File

After seeing Figure 3.1, we can understand that the API documentation file contains nothing but the names and the descriptions of the two functions written in our program. In this way, the API documentation file contains complete help on all the features including functions, classes, modules, etc.

After discussing about comments in Python, let's go through the following statements:

```
a = 10
b = 15
```

In the first statement, the value 10 is stored into the left side variable 'a'. Storage is done by the symbol '=' which is called *assignment operator*. In the second statement, the value 15 is stored into another variable 'b'. A variable can be imagined like a memory location where we can store data. The name given to a variable is called *identifier*. Identifiers represent names of variables, functions, objects or any such thing.

Depending upon the type of data stored into a variable, Python interpreter understands how much memory is to be allocated for that variable. Hence, we need not declare the datatype of the variable explicitly like we do in case of C or Java. For example, in C or Java we declare a variable as 'int' type as:

```
int a;    //declare a as int type variable
a = 10;  //store 10 into a
```

But, in Python we can simply write as:

```
a = 10
```

This creates a variable by the name 'a' and stores the value 10. Since we are storing 10 which is an integer, Python will understand that 'a' is of integer type variable and allocates memory required to store an integer value. On the other hand, if we store a string (a group of characters) into 'a', then Python will understand 'a' as string type variable and then allocates memory required to store the string.

How Python Sees Variables

In programming languages like C, Java and many other languages, the concept of variable is connected to memory location. In all languages, a variable is imagined as a storage box which can store some value. Suppose, we write a statement as:

```
a = 1;
```

Some memory is allocated with the name 'a' and there the value '1' is stored. Here, we can imagine the memory as a box or container that stores the value, as shown in Figure 3.2:

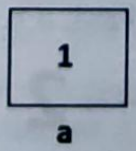

Figure 3.2: The effect of a = 1

In this way, for every variable we create, there will be a new box created with the variable name to store the value. If we change the value of the variable, then the box will be updated with the new value. For example, if we write:

```
a = 2;
```

then the new value '2' is stored into the same box, as shown in Figure 3.3:

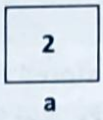

Figure 3.3: The effect of a = 2

When we assign one variable to another variable as:

```
int b = a;
```

A new memory box is created by the name 'b' and the value of the variable 'a' is copied into that box as shown in Figure 3.4:

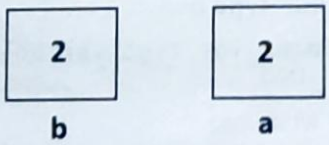

Figure 3.4: The effect of b = a;

This is how variables are visualized in other languages. However in Python, a variable is seen as a tag (or name) that is tied to some value. For example, the statement:

```
a = 1
```

means the value '1' is created first in memory and then a tag by the name 'a' is created to show that value as shown in Figure 3.5:

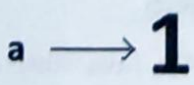

Figure 3.5: The effect of a = 1

Python considers the values (i.e. 1 or 2 etc.) as 'objects'. If we change the value of 'a' to a new value as:

```
a = 2
```

then the tag is simply changed to the new value (or object), as shown in Figure 3.6. Since the value '1' becomes unreferenced object, it is removed by garbage collector.

Figure 3.6: The effect of a = 2

Assigning one variable to another variable makes a new tag bound to the same value. For example,

```
b = a
```

Here, we are storing 'a' value into 'b'. A new tag 'b' will be created that refers to '2' as shown in Figure 3.7:

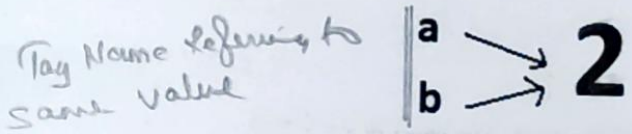

Figure 3.7: The effect of b = a

Let's understand that while other languages have variables, Python has 'tags' (or names) to represent the values. Compare Figure 3.4 and Figure 3.7. In Figure 3.4, there are two memory locations created to store two values. But in Figure 3.7, there is only one memory referenced by two names. It means Python is using memory efficiently. This is the actual way of understanding Python variables.

Datatypes in Python

A datatype represents the type of data stored into a variable or memory. The datatypes which are already available in Python language are called *Built-in* datatypes. The datatypes which can be created by the programmers are called *User-defined* datatypes.

Built-in datatypes

The built-in datatypes are of five types:

- None Type
- Numeric types
- Sequences
- Sets
- Mappings

The None Type

In Python, the 'None' datatype represents an object that does not contain any value. In languages like Java, it is called 'null' object. But in Python, it is called 'None' object.

In a Python program, maximum of only one 'None' object is provided. One of the uses of 'None' is that it is used inside a function as a default value of the arguments. When calling the function, if no value is passed, then the default value will be taken as 'None'. If

some value is passed to the function, then that value is used by the function. In Boolean expressions, 'None' datatype represents 'False'.

Numeric Types

The numeric types represent numbers. There are three sub types:

- int
- float
- complex

int Datatype

The int datatype represents an integer number. An integer number is a number without any decimal point or fraction part. For example, 200, -50, 0, 9888998700, etc. are treated as integer numbers. Now, let's store an integer number -57 into a variable 'a'.

```
a = -57
```

Here, 'a' is called int type variable since it is storing -57 which is an integer value. In Python, there is no limit for the size of an int datatype. It can store very large integer numbers conveniently.

float Datatype

The float datatype represents floating point numbers. A floating point number is a number that contains a decimal point. For example, 0.5, -3.4567, 290.08, 0.001 etc. are called floating point numbers. Let's store a float number into a variable 'num' as:

```
num = 55.67998
```

Here num is called float type variable since it is storing floating point value. Floating point numbers can also be written in scientific notation where we use 'e' or 'E' to represent the power of 10. Here 'e' or 'E' represents 'exponentiation'. For example, the number 2.5×10^4 is written as 2.5E4. Such numbers are also treated as floating point numbers. For example,

```
x = 22.55e3
```

Here, the float value 22.55×10^3 is stored into the variable 'x'. The type of the variable 'x' will be internally taken as float type. The convenience in scientific notation is that it is possible to represent very big numbers using less memory.

Complex Datatype

A complex number is a number that is written in the form of a + bj or a + bJ. Here, 'a' represents the real part of the number and 'b' represents the imaginary part of the number. The suffix 'j' or 'J' after 'b' indicates the square root value of -1. The parts 'a' and 'b' may contain integers or floats. For example, 3+5j, -1-5.5J, 0.2+10.5J are all complex numbers. See the following statement:

```
c1 = -1-5.5J
```

size → >>> from sys import getsizeof
getsizeof(x)

Here, the complex number -1-5.5J is assigned to the variable 'c1'. Hence, Python interpreter takes the datatype of the variable 'c1' as complex type.

We will write a Python program (Program 1) that adds two complex numbers and displays their sum. In this program, we store a complex number in a variable 'c1' and another complex number in another variable 'c2' and then add them. The result of addition is stored into 'c3' which is another complex type variable and then the value of 'c3' is displayed by the print() function.

Used in geometry, scientific calculation, calculus.

Program

Program 1: A Python program to display the sum of two complex numbers.

```
# python program to add two complex numbers
c1 = 2.5+2.5j
c2 = 3.0-0.5j
c3 = c1 + c2
print("Sum= ", c3)
```

Output:

```
C:\>python add.py
Sum= (5.5+2j)
```

Representing Binary, Octal and Hexadecimal Numbers

A binary number should be written by prefixing 0b (zero and b) or 0B (zero and B) before the value. For example, 0b110110, 0B101010011 are treated as binary numbers. Hexadecimal numbers are written by prefixing 0x (zero and x) or 0X (zero and big X) before the value, as 0xA180 or 0X11fb91 etc. Similarly, octal numbers are indicated by prefixing 0o (zero and small o) or 0O (zero and then O) before the actual value. For example, 0O145 or 0o773 are octal values.

Converting the Datatypes Explicitly

Depending on the type of data, Python internally assumes the datatype for the variable. But sometimes, the programmer wants to convert one datatype into another type on his own. This is called *type conversion* or *coercion*. This is possible by mentioning the datatype with parentheses. For example, to convert a number into integer type, we can write int(num).

- int(x) is used to convert the number x into int type. See the example:

```
x = 15.56
int(x) # will display 15
```

- float(x) is used to convert x into float type. For example,

```
num = 15
float(num) # will display 15.0
```

- complex(x) is used to convert x into a complex number with real part x and imaginary part zero. For example,

```
n = 10
complex(n) # will display (10+0j)
```

- complex(x, y) is used to convert x and y into a complex number such that x will be the real part and y will be the imaginary part. For example,

```
a = 10
b = -5
complex(a, b) # will display (10-5j)
```

From the previous discussion, we can understand that int(x) is converting the datatype of x into int type and producing decimal integer number. So, we can use int(x) to convert a value from other number systems into our decimal number system. This is what we are trying to do in Program 2. In this program, we are going to convert binary, octal and hexadecimal numbers into decimal integers using type conversion.

Program

Program 2: A program to convert numbers from octal, binary and hexadecimal systems into decimal number system.

```
#  python program to convert into decimal number system
n1 = 0o17
n2 = 0b1110010
n3 = 0x1c2

n = int(n1)
print('Octal 17 = ', n)
n = int(n2)
print('Binary 1110010 = ', n)
n = int(n3)
print('Hexadecimal 1c2 = ', n)
```

Output:

```
C:\>python convert.py
Octal 17 =  15
Binary 1110010 =  114
Hexadecimal 1c2 =  450
```

There is a function in the form of int (string, base) that is useful to convert a string into a decimal integer. 'string' represents the string format of the number. It should contain integer number in string format. 'base' represents the base of the number system to be used for the string. For example, base 2 indicates binary number and base 16 indicates hexadecimal number. For example,

```
str = "1c2"   #  string str contains a hexadecimal number
n = int(str, 16)   # hence base is 16. Convert str into int
print(n)  # this will display 450
```

Program

Program 3: In this program, we are going to rewrite Program 2 using int() function to convert numbers from different number systems into decimal number system.

```
# python program to convert into decimal number system - v2.0
s1 = "17"
s2 = "1110010"
s3 = "1c2"

n = int(s1, 8)
print('Octal 17 = ', n)
n = int(s2, 2)
print('Binary 1110010 = ', n)
n = int(s3, 16)
print('Hexadecimal 1c2 = ', n)
```

Output:

```
C:\>python convert.py
Octal 17 =  15
Binary 1110010 =  114
Hexadecimal 1c2 =  450
```

We can also use functions bin() for converting a number into binary number system. The function oct() will convert a number into octal number system and the function hex() will convert a number into hexadecimal number system. See the following examples:

```
a = 10
b = bin(a)
print(b) # displays 0b1010

b = oct(a)
print(b) # displays 0o12

b = hex(a)
print(b) # displays 0xa
```

bool Datatype

The bool datatype in Python represents boolean values. There are only two boolean values True or False that can be represented by this datatype. Python internally represents True as 1 and False as 0. A blank string like "" is also represented as False. Conditions will be evaluated internally to either True or False. For example,

```
a = 10
b = 20
if(a<b): print("Hello") # displays Hello.
```

In the previous code, the condition a<b which is written after if - is evaluated to True and hence it will execute print("Hello").

```
a = 10>5  # here 'a' is treated as bool type variable
print(a) # displays True

a = 5>10
print(a) # displays False
```

```
True+True will display 2  # True is 1 and false is 0
True-False will display 1
```

Sequences in Python

Generally, a sequence represents a group of elements or items. For example, a group of integer numbers will form a sequence. There are six types of sequences in Python:

- str
- bytes
- bytearray
- list
- tuple
- range

str Datatype

In Python, str represents string datatype. A string is represented by a group of characters. Strings are enclosed in single quotes or double quotes. Both are valid.

```
str = "Welcome"  # here str is name of string type variable
str = 'Welcome'  # here str is name of string type variable
```

We can also write strings inside """ (triple double quotes) or ''' (triple single quotes) to span a group of lines including spaces.

```
str1 = """This is a book on Python which
discusses all the topics of Core Python
in a very lucid manner."""

str2 = '''This is a book on Python which
discusses all the topics of Core Python
in a very lucid manner.'''
```

The triple double quotes or triple single quotes are useful to embed a string inside another string as shown below:

```
str = """This is 'Core Python' book."""
print(str) # will display: This is 'Core Python' book.

str = '''This is "Core Python" book.'''
print(str) # will display: This is "Core Python" book.
```

The slice operator represents square brackets [and] to retrieve pieces of a string. For example, the characters in a string are counted from 0 onwards. Hence, str[0] indicates the 0th character or beginning character in the string. See the examples below:

```
s = 'Welcome to Core Python'  # this is the original string
print(s)
Welcome to Core Python
```

```
print(s[0])  # display 0th character from s
W

print(s[3:7])  # display from 3rd to 6th characters
come

print(s[11:])  # display from 11th characters onwards till end
Core Python

print(s[-1])  # display first character from the end
n
```

The repetition operator is denoted by '*' symbol and useful to repeat the string for several times. For example s * n repeats the string for n times. See the example:

```
print(s*2)
Welcome to Core PythonWelcome to Core Python
```

bytes Datatype

The bytes datatype represents a group of byte numbers just like an array does. A byte number is any positive integer from 0 to 255 (inclusive). bytes array can store numbers in the range from 0 to 255 and it cannot even store negative numbers. For example,

```
elements = [10, 20, 0, 40, 15]  # this is a list of byte numbers
x = bytes(elements)  # convert the list into bytes array
print(x[0])   # display 0th element, i.e 10
```

We cannot modify or edit any element in the bytes type array. For example, x[0] = 55 gives an error. Here we are trying to replace 0th element (i.e. 10) by 55 which is not allowed.

Now, let's write a Python program (Program 4) to create a bytes type array with a group of elements and then display the elements using a for loop. In this program, we are using a for loop as:

```
for i in x: print(i)
```

It means, if 'i' is found in the array 'x' then display i value using print() function. Remember, we need not declare the datatypes of variables in Python. Hence, we need not tell which type of variable 'i' is. In the above loop, 'i' stores each element of the array 'x' and with every element, print(i) will be executed once. Hence, print(i) displays all elements of the array 'x'. We will discuss for loop in detail in the next chapter.

Program

Program 4: A Python program to create a byte type array, read and display the elements of the array.

```
# program to understand bytes type array
# create a list of byte numbers
elements = [10, 20, 0, 40, 15]

# convert the list into bytes type array
x = bytes(elements)
```

```
# retrieve elements from x using for loop and display
for i in x: print(i)
```

Output:

```
C:\>python bytes.py
10
20
0
40
15
```

bytearray Datatype

The bytearray datatype is similar to bytes datatype. The difference is that the bytes type array cannot be modified but the bytearray type array can be modified. It means any element or all the elements of the bytearray type can be modified. To create a bytearray type array, we can use the function bytearray as:

```
elements = [10, 20, 0, 40, 15]  # this is a list of byte numbers
x = bytearray(elements)  # convert the list into bytearray type array
print(x[0])   # display 0th element, i.e 10
```

We can modify or edit the elements of the bytearray. For example, we can write:

```
x[0] = 88  # replace 0th element by 88
x[1] = 99  # replace 1st element by 99
```

We will write program to create a bytearray type array and then modify the first two elements. Then we will display all the elements using a for loop. This can be seen in Program 5.

Program

Program 5: A Python program to create a bytearray type array and retrieve elements

```
# program to understand bytearray type array
# create a list of byte numbers
elements = [10, 20, 0, 40, 15]

# convert the list into bytearray type array
x = bytearray(elements)

# modify the first two elements of x
x[0] = 88
x[1] = 99

# retrieve elements from x using for loop and display
for i in x: print(i)
```

Output:

```
C:\>python bytearray.py
88
99
0
40
15
```

list Datatype

Lists in Python are similar to arrays in C or Java. A list represents a group of elements. The main difference between a list and an array is that a list can store different types of elements but an array can store only one type of elements. Also, lists can grow dynamically in memory. But the size of arrays is fixed and they cannot grow at runtime. Lists are represented using square brackets [ ] and the elements are written in [ ], separated by commas. For example,

```
list = [10, -20, 15.5, 'Vijay', "Mary"]
```

will create a list with different types of elements. The slicing operation like [0: 3] represents elements from 0th to 2nd positions, i.e. 10, 20, 15.5. Have a look at some of the operations on a list in the following IDLE window, as shown in Figure 3.8:

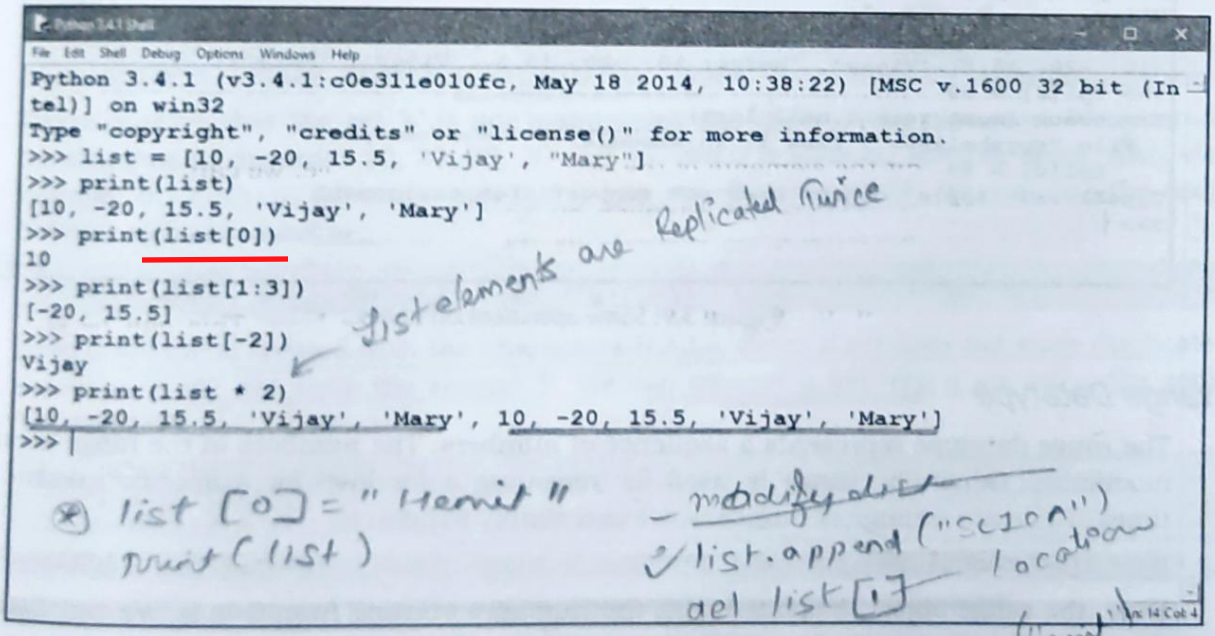

Figure 3.8: Some Operations on Lists

tuple Datatype

A tuple is similar to a list. A tuple contains a group of elements which can be of different types. The elements in the tuple are separated by commas and enclosed in parentheses ( ). Whereas the list elements can be modified, it is not possible to modify the tuple elements. That means a tuple can be treated as a read-only list. Let's create a tuple as:

```
tpl = (10, -20, 15.5, 'Vijay', "Mary")
```

The individual elements of the tuple can be referenced using square braces as tpl[0], tpl[1], tpl[2], ... Now, if we try to modify the 0th element as:

```
tpl[0] = 99
```

This will result in error. The slicing operations which can be done on lists are also valid in tuples. See the general operations on the tuple in Figure 3.9:

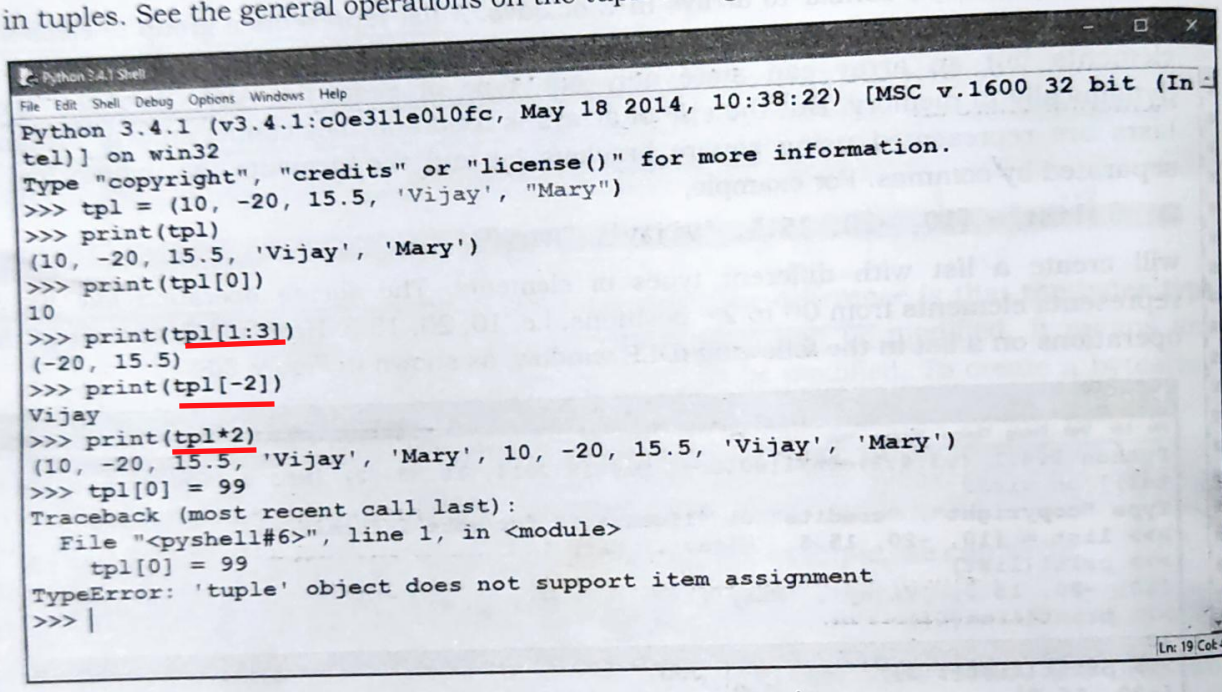

Figure 3.9: Some operations on tuples

range Datatype

The range datatype represents a sequence of numbers. The numbers in the range are not modifiable. Generally, range is used for repeating a for loop for a specific number of times. To create a range of numbers, we can simply write:

```
r = range(10)
```

Here, the range object is created with the numbers starting from 0 to 9. We can display these numbers using a for loop as:

```
for i in r: print(i)
```

The above statement will display numbers from 0 to 9. We can use a starting number, an ending number and a step value in the range object as:

```
r = range(30, 40, 2)
```

This will create a range object with a starting number 30 and an ending number 39. The step size is 2. It means the numbers in the range will increase by 2 every time. So, the for loop

```
for i in r: print(i)
```

will display numbers: 30, 32, 34, 36, 38. Let's create a list with a range of numbers from 0 to 9 as:

```
lst = list(range(10))
print(lst) # will display: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Sets { }

A set is an unordered collection of elements much like a set in Mathematics. The order of elements is not maintained in the sets. It means the elements may not appear in the same order as they are entered into the set. Moreover, a set does not accept duplicate elements. There are two sub types in sets:

- set datatype
- frozenset datatype

set Datatype (mutable)

To create a set, we should enter the elements separated by commas inside curly braces {}.

```
s = {10, 20, 30, 20, 50}
print(s) # may display {50, 10, 20, 30}
```

Please observe that the set 's' is not maintaining the order of the elements. We entered the elements in the order 10, 20, 30, 20 and 50. But it is showing another order. Also, we repeated the element 20 in the set, but it has stored only one 20. We can use the set() function to create a set as:

```
ch = set("Hello")
print(ch) # may display {'H', 'e', 'l', 'o'}
```

Here, a set 'ch' is created with the characters H,e,l,o. Since a set does not store duplicate elements, it will not store the second 'l'. We can convert a list into a set using the set() function as:

```
lst = [1,2,5,4,3]
s = set(lst)
print(s) # may display {1, 2, 3, 4, 5}
```

Since sets are unordered, we cannot retrieve the elements using indexing or slicing operations. For example, the following statements will give error messages:

```
print(s[0])   # indexing, display 0th element
print(s[0:2])  # slicing, display from 0 to 1st elements
```

The update() method is used to add elements to a set as:

```
s.update([50,60])
print(s) # may display {1, 2, 3, 4, 5, 50, 60}
```

On the other hand, the remove() method is used to remove any particular element from a set as:

```
s.remove(50)
print(s) # may display {1, 2, 3, 4, 5, 60}
```

frozenset Datatype

The frozenset datatype is same as the set datatype. The main difference is that the elements in the set datatype can be modified; whereas, the elements of frozenset cannot be modified. We can create a frozenset by passing a set to frozenset() function as:

```
s = {50,60,70,80,90}
print(s) # may display {80, 90, 50, 60, 70}

fs = frozenset(s)  # create frozenset fs
print(fs) # may display frozenset({80, 90, 50, 60, 70})
```

Another way of creating a frozenset is by passing a string (a group of characters) to the frozenset() function as:

```
fs = frozenset("abcdefg")
print(fs) # may display frozenset({'e', 'g', 'f', 'd', 'a', 'c', 'b'})
```

However, update() and remove() methods will not work on frozensets since they cannot be modified or updated.

Mapping Types

A map represents a group of elements in the form of key value pairs so that when the key is given, we can retrieve the value associated with it. The *dict* datatype is an example for a map. The 'dict' represents a 'dictionary' that contains pairs of elements such that the first element represents the key and the next one becomes its value. The key and its value should be separated by a colon (:) and every pair should be separated by a comma. All the elements should be enclosed inside curly brackets { }.

We can create a dictionary by typing the roll numbers and names of students. Here, roll numbers are keys and names will become values. We write these key value pairs inside curly braces as:

```
d = {10: 'Kamal', 11: 'Pranav', 12:  'Hasini', 13: 'Anup', 14:
'Reethu'}
```

Here, d is the name of the dictionary. 10 is the key and its associated value is 'Kamal'. The next key is 11 and its value is 'Pranav'. Similarly 12 is the key and 'Hasini' is it value. 13 is the key and 'Anup' is the value and 14 is the key and 'Reethu' is the value.

We can create an empty dictionary without any elements as:

```
d = {}
```

Later, we can store the key and values into d as:

```
d[10] = 'Kamal'
d[11] = 'Pranav'
```

In the preceding statements, 10 represents the key and 'Kamal' is its value. Similarly, 11 represents the key and its value is 'Pranav'. Now, if we write

```
print(d)
```

It will display the dictionary as:

```
{10: 'Kamal', 11: 'Pranav'}
```

We can perform various operations on dictionaries. To retrieve value upon giving the key, we can simply mention d[key]. To retrieve only keys from the dictionary, we can use the method keys() and to get only values, we can use the method values().

We can update the value of a key, as: d[key] = newvalue. We can delete a key and corresponding value, using del module. For example del d[11] will delete a key with 11 and its value. Figure 3.10 demonstrates these operations:

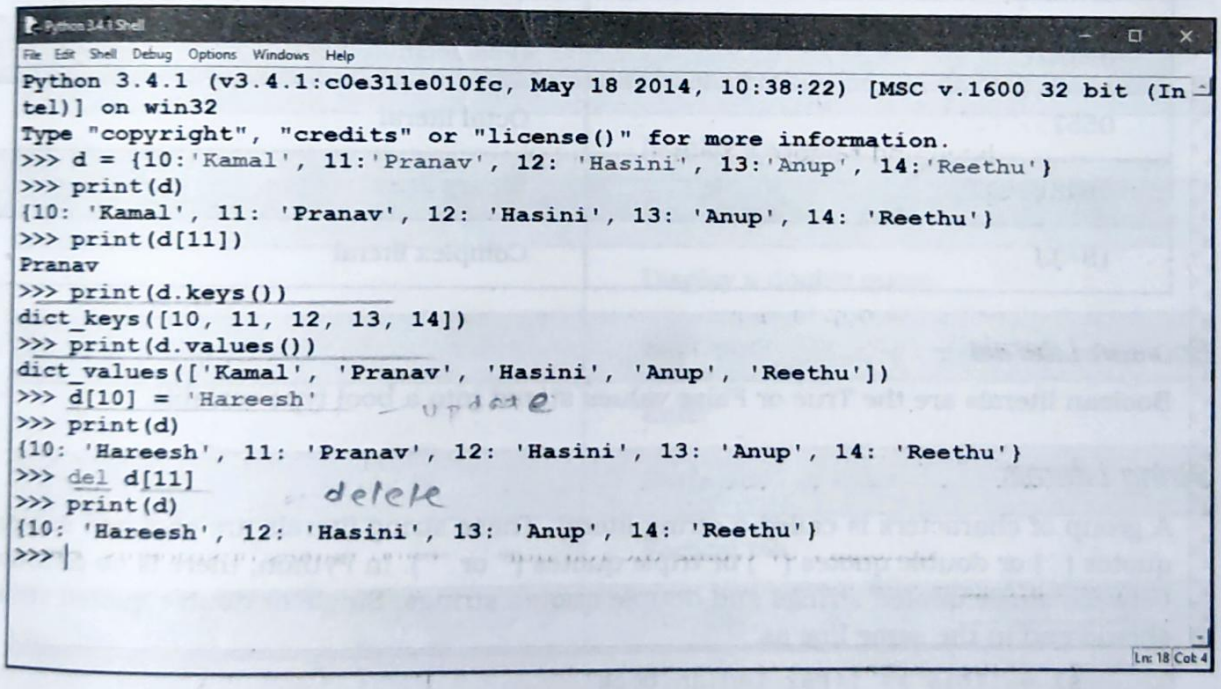

Figure 3.10: Some Operations on Dictionaries

Literals in Python

A *literal* is a constant value that is stored into a variable in a program. Observe the following statement:

```
a = 15
```

Here, 'a' is the variable into which the constant value '15' is stored. Hence, the value 15 is called 'literal'. Since 15 indicates integer value, it is called 'integer literal'. The following are different types of literals in Python:

- Numeric literals
- Boolean literals
- String literals

Numeric Literals

These literals represent numbers. Please observe the different types of numeric literals available in Python, as shown in Table 3.1:

Table 3.1: Numeric Literals

Examples	Literal name
450, -15	Integer literal
3.14286, -10.6, 1.25E4	Float literal
0x5A1C	Hexadecimal literal
0557	Octal literal
0B110101	Binary literal
18+3J	Complex literal

Boolean Literals

Boolean literals are the True or False values stored into a bool type variable.

String Literals

A group of characters is called a string literal. These string literals are enclosed in single quotes (') or double quotes (") or triple quotes (''' or """). In Python, there is no difference between single quoted strings and double quoted strings. Single or double quoted strings should end in the same line as:

```
s1 = 'This is first Indian book'
s2 = "Core Python"
```

When a string literal extends beyond a single line, we should go for triple quotes as:

```
s1 = '''This is first Indian book on
     Core Python exploring all important
     and useful features''',

s2 = """This is first Indian book on
     Core Python exploring all important
     and useful features"""
```

In the preceding examples, the strings have taken up to 3 lines. Hence, we inserted them inside triple single quotes (''') or triple double quotes (""""). When a string spans more than one line adding backslash (\ ) will join the next string to it. For example,

```
str = "This is first line and \
this will be continued with the first line"
print(str)
This is first line and this will be continued with the first line
```

We can use escape characters like \n inside a string literal. For example,

```
str = "This is \nPython"
print(str)
This is
Python
```

Escape characters escape the normal meaning and are useful to perform a different task. When a \n is written, it will throw the cursor to the new line. Table 3.2 gives some important escape characters:

Table 3.2: Important Escape Characters in Strings

Escape character	Meaning
\	New line continuation
\\	Display a single \
\'	Display a single quote
\"	Display a double quote
\b	backspace
\r	Enter
\t	Horizontal tab space
\v	Vertical tab
\n	New line

Determining the Datatype of a Variable

To know the datatype of a variable or object, we can use the type() function. For example, type(a) displays the datatype of the variable 'a'.

```
a = 15
print(type(a))
<class 'int'>
```

Since we are storing an integer number 15 into variable 'a', the type of 'a' is assumed by the Python interpreter as 'int'. Observe the last line. It shows class 'int'. It means the variable 'a' is an object of the class 'int'. It means 'int' is treated as a class. Every datatype is treated as an object internally by Python. In fact, every datatype, function, method, class, module, lists, sets, etc. are all objects in Python, as shown in Figure 3.11:

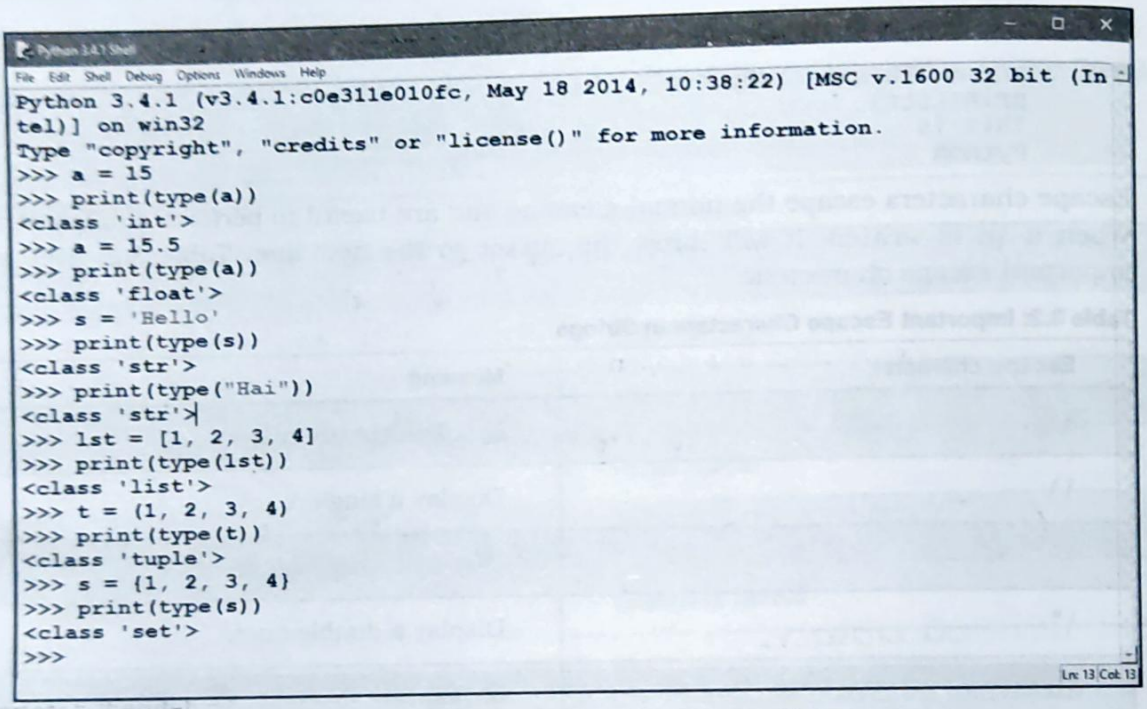

Figure 3.11: All datatypes are class objects in Python

What about Characters

Languages like C and Java contain char datatype that represents a single character. Python does not have a char datatype to represent individual characters. It has str datatype which represents strings. We know a string is composed of several characters. Hence, when we think of a character, we should think of it as an element in a string. In the following statements, we are assigning a single character 'A' to a variable 'ch'. Then we find out the type of the 'ch' variable using the type() function.

```
ch = 'A'
print(type(ch))
<class 'str'>
```

See that the type of 'ch' is displayed as 'str' type. So, there is no concept like char datatype in Python. To work with strings, we have to access the individual characters of a string using index or position number. Suppose there is a string str where we stored a string literal 'Bharat' as:

```
str = 'Bharat'
print(str[0])
B
```

In the preceding example, the str[0] represents the 0th character, i.e. 'B'. Similarly, str[1] represents 'h' and str[2] represents 'a' and so on. To retrieve all the characters using a for loop, we can write:

```
for i in str: print(i)
```

and it will display the characters as:

```
B
h
a
r
a
t
```

However, some of the useful methods in case of strings can also be used for characters. For example, isupper() is a method that tests whether a string is uppercase or lowercase. It returns True if the string is uppercase otherwise False. This method can be used on a character to know whether it is uppercase letter or not. Let's check the case of first two letters in 'Bharat' as:

```
str[0].isupper()   # checking if 'B' is capital letter or not
True

str[1].isupper()  # checking if 'h' is capital letter or not
False
```

User-defined Datatypes

The datatypes which are created by the programmers are called 'user-defined' datatypes. For example, an array, a class, or a module is user-defined datatypes. We will discuss about these datatypes in the later chapters.

Constants in Python

A constant is similar to a variable but its value cannot be modified or changed in the course of the program execution. We know that the variable value can be changed whenever required. But that is not possible for a constant. Once defined, a constant cannot allow changing its value. For example, in Mathematics, 'pi' value is 22/7 which never changes and hence it is a constant. In languages like C and Java, defining constants is possible. But in Python, that is not possible. A programmer can indicate a variable as constant by writing its name in all capital letters. For example, MAX_VALUE is a constant. But its value can be changed.

Identifiers and Reserved words

An identifier is a *name* that is given to a variable or function or class etc. Identifiers can include letters, numbers, and the underscore character (_). They should always start with a nonnumeric character. Special symbols such as ?, #, $, %, and @ are not allowed in identifiers. Some examples for identifiers are salary, name11, gross_income, etc.

We should also remember that Python is a case sensitive programming language. It means capital letters and small letters are identified separately by Python. For example,

the names 'num' and 'Num' are treated as different names and hence represent different variables. Figure 3.12 shows examples of a variable, an operator and a literal:

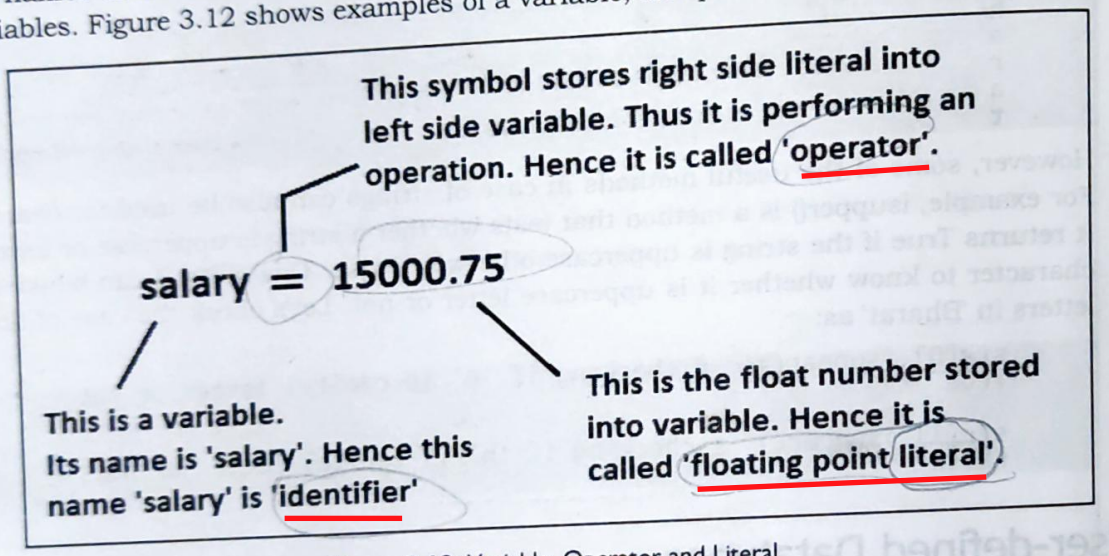

Figure 3.12: Variable, Operator and Literal

Reserved words are the words that are already reserved for some particular purpose in the Python language. The names of these reserved words should not be used as identifiers. The following are the reserved words available in Python:

and	del	from	nonlocal	try
as	elif	global	not	while
assert	else	if	or	with
break	except	import	pass	yield
class	exec	in	print	False
continue	finally	is	raise	True
def	for	lambda	return	

Naming Conventions in Python

Python developers made some suggestions to the programmers regarding how to write names in the programs. The rules related to writing names of packages, modules, classes, variables, etc. are called *naming conventions*. The following naming conventions should be followed:

- **Packages:** Package names should be written in all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).

- **Modules:** Modules names should be written in all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).
- **Classes:** Each word of a class name should start with a capital letter. This rule is applicable for the classes created by us. Python's built-in class names use all lowercase words. When a class represents exception, then its name should end with a word 'Error'.
- **Global variables or Module-level variables:** Global variables names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).
- **Instance variables:** Instance variables names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_). Non-public instance variable name should begin with an underscore.
- **Functions:** Function names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).
- **Methods:** Method names should be all lower case letters. When multiple words are used for a name, we should separate them using an underscore (_).
- **Method arguments:** In case of instance methods, their first argument name should be 'self'. In case of class methods, their first argument name should be 'cls'.
- **Constants:** Constants names should be written in all capital letters. If a constant has several words, then each word should be separated by an underscore (_).
- **Non accessible entities**: Some variables, functions and methods are not accessible outside and they should be used as they are in the program. Such entities names are written with two double quotes before and two double quotes after. For example, __init__(self) is a function used in a class to initialize variables.

Points to Remember

- A variable represents a memory location where some data can be stored.
- Python has only single line comments starting with a hash symbol (#).
- Triple single quotes (''') or triple doubles quotes (""") can be used to create multi line comments.
- Docstrings are the strings written inside triple single quotes or triple double quotes and as first statements in a function, class or a method. These docstrings are useful to create API documentation file that contains the description of all features of a program or software.
- A datatype represents the type of data stored into a variable. The data stored into the variable is called literal.

- There are four built-in datatypes in Python. They are numeric types, sequences, sets and mappings.
- A binary literal is represented using 0b or 0B in the beginning of a number.
- An octal literal is represented using 0o or 0O in the beginning of a number.
- If a number starts with 0x or 0X, then it is considered as a hexadecimal number.
- The 'bool' datatype represents either True or False. True is represented by 1 and False is represented by 0.
- A string literal can be enclosed in single quotes or double quotes. When a string spans more than single line, then we need to enclose it inside triple single quotes (''') or triple double quotes (""").
- A list is a dynamically growing array that can store different types of elements. Lists are written using square brackets [ ].
- A tuple is similar to a list but its elements cannot be modified. A tuple uses parentheses ( ) to enclose its elements.
- The 'range' datatype represents sequence of numbers and is generally used to repeat a for loop.
- A set is an unordered collection of elements. A set uses curly braces { } to enclose its elements.
- A frozenset is similar to a set but its elements cannot be modified.
- A 'dict' datatype represents a group of elements written in the form of several key-value pairs such that when the key is given, its corresponding value can be obtained.
- We can use the type() function to determine the datatype of a variable.
- Python does not have a datatype to represent single character.
- A constant represents a fixed value that cannot be changed. Defining constants is not possible in Python.
- An identifier is a name that is given to a variable, function, class, etc.
- The rules related to writing names for packages, modules, classes, functions, variables, etc., are called naming conventions.