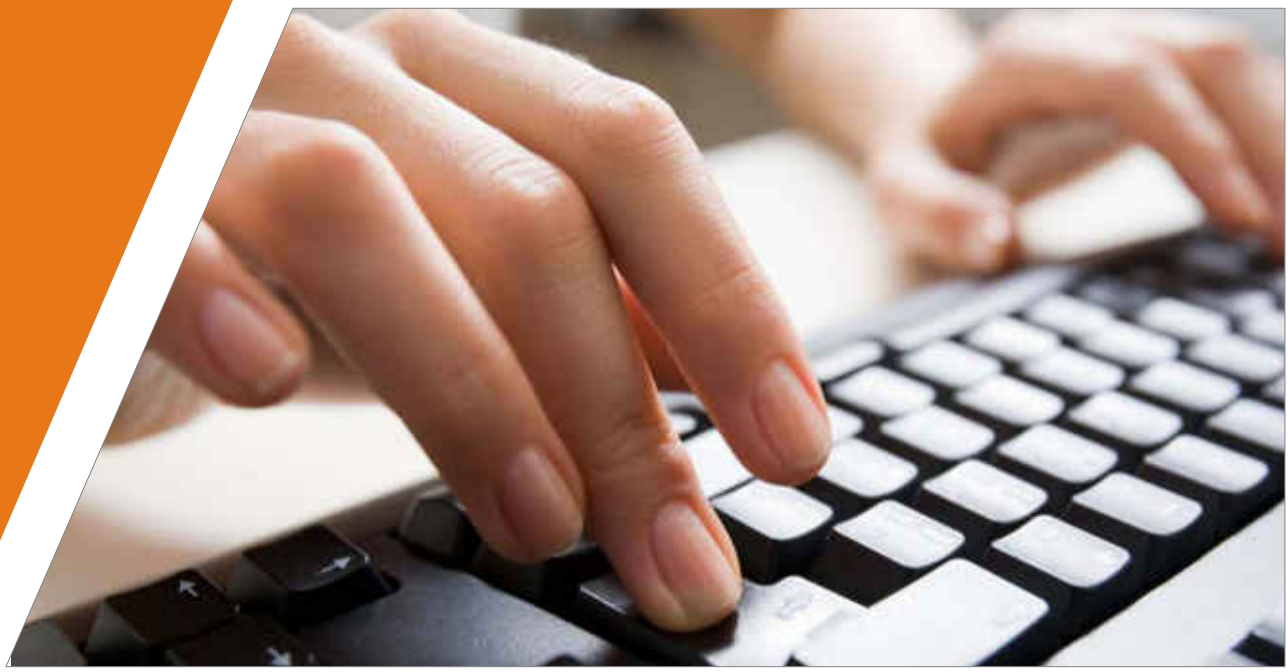
Viva Technologies



PART 1

# PYTHON PROGRAMMING

THATIPUR ,CHAUHAN PIYAU, DARPANCOLONY,
VINAY NAGAR, HAZIRA, DD NAGAR Ph: 94257-01888, 02888, 03888


# DISCLAIMER

This book has been designed & developed by Viva Technologies for institute purpose only. It is not for sale in any means.


# 1 Python Revision Tour

## In This Chapter

## 1.2 TOKENS IN PYTHON

The smallest individual unit in a program is known as a *Token* or a *lexical unit*.

Python has following tokens :

(*i*) Keywords (*ii*) Identifiers (Names) (*iii*) Literals

(*iv*) Operators (*v*) Punctuators

**TOKENS**

The smallest individual unit in a program is known as a ***Token*** or a ***lexical unit***.

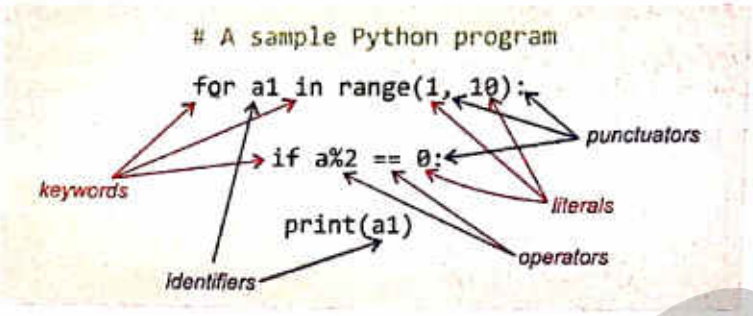



Figure 1.1 Some tokens in a Python program.

Let us revise our understanding of tokens.

### 1.2.1 Keywords

*Keywords* are predefined words with special meaning to the language compiler or interpreter. These are reserved for special purpose and must not be used as normal identifier names.

**KEYWORD**

A ***keyword*** is a word having special meaning reserved by programming language.

Python programming language contains the following keywords :

| | | | | | | |
|---|---|---|---|---|---|---|
| False | assert | del | for | in | or | while |
| None | break | elif | from | is | pass | with |
| True | class | else | global | lambda | raise | yield |
| and | continue | except | if | nonlocal | return | |
| as | def | finally | import | not | try | |

### .2.2 Identifiers (Names)

Identifiers are the names given to different parts of the program *viz. variables, objects, classes, functions, lists, dictionaries* and so forth.

The naming rules for Python identifiers can be summarized as follows :

- Variable names must only be a *non-keyword word* with no spaces in between.
- Variable names must be made up of only letters, numbers, and underscore (_).
- Variable names cannot begin with a number, although they can contain numbers.

**NOTE**

*Python is case sensitive* as it treats upper and lower-case characters differently.

The following are some *valid* identifiers :

| | |
|---|---|
| Myfile | DATE9_7_77 |
| MYFILE | _DS |
| _CHK | FILE13 |
| Z2T0Z9 | _HJI3_JK |

The following are some *invalid* identifiers :

| | |
|---|---|
| DATA-REC | contains special character - (hyphen) (other than A - Z, a - z and _ (underscore) ) |
| 29CLCT | Starting with a digit |
| break | reserved keyword |
| My.file | contains special character dot ( . ) |

### 1.2.3 Literals/Values

Literals are data items that have a fixed/constant value.

Python allows several kinds of literals, which are being given below.

#### (i) String Literals

A *string literal* is a sequence of characters surrounded by quotes (single or double or triple quotes). String literals can either be ***single line strings*** or ***multi-line strings***.

- **Single line strings** must terminate in one line *i.e.*, the closing quotes should be on the same line as that of the opening quotes. (See below)
- **Multiline strings** are strings spread across multiple lines. With single and double quotes, each line other that the concluding line has an end character as \ (backslash) but with triple quotes, no backslash is needed at the end of intermediate lines. (see below) :

```
>>> Text1 = "Hello World"          <-- Single line string

>>> Text2 = "Hello\                <-- Multi-line string
    World "

Text3 = '''Hello                   <-- No backslash needed
    World'''
```

In strings, you can include non-graphic characters through escape sequences. Escape sequences are given in following table :

| Escape sequence | What it does [Non-graphic character] | Escape sequence | What it does [Non-graphic character] |
|---|---|---|---|
| \\ | Backslash (\) | \r | Carriage Return (CR) |
| \' | Single quote (') | \t | Horizontal Tab (TAB) |
| \" | Double quote (") | \uxxxx | Character with 16-bit hex value xxxx (Unicode only) |
| \a | ASCII Bell (BEL) | \Uxxxxxxxx | Character with 32-bit hex value xxxxxxxx (Unicode only) |
| \b | ASCII Backspace (BS) | \v | ASCII Vertical Tab (VT) |
| \f | ASCII Formfeed (FF) | \ooo | Character with octal value ooo |
| \n | New line character | \xhh | Character with hex value hh |
| \N{name} | Character named name in the Unicode database (Unicode only) | | |

### (ii) Numeric Literals

Numeric literals are numeric values and these can be one of the following types :

(a) int **(signed integers)** often called just **integers** or **ints,** are positive or negative whole numbers with no decimal point.

The integer literals can be written in :

- **Decimal form** : an integer beginning with digits 1-9. *e.g.,* 1234, 4100 etc.
- **Octal form** : an integer beginning with **0o** (zero followed by letter o) *e.g.,* 0o35, 0o77 etc. Here do remember that for Octal, **8** and **9** are invalid digits.
- **Hexadecimal form** : an integer beginning with **0x** (zero followed by letter X) *e.g.,* 0x73, 0xAF etc. Here remember that valid digits/letters for hexadecimal numbers are *0-9* and *A-F*.

(b) **Floating Point Literals. Floating point literals** or **real literals floats** represent real numbers and are written with a decimal point dividing the integer and fractional parts are numbers having fractional parts. These can be written in fractional form *e.g.,* –13.0, .75, 7. etc. or in **Exponent form** e.g., *0.17E5, 3.E2, .6E4* etc.

(c) **Complex number literals** are of the form $a + bJ$, where $a$ and $b$ are *floats* and $J$ (or $j$) represents $\sqrt{-1}$, which *is* an imaginary number). $a$ is the real part of the number, and $b$ is the imaginary part.

### (iii) Boolean Literals

A Boolean literal in Python is used to represent one of the two Boolean values *i.e.,* **True** (Boolean true) or **False** (Boolean false). A Boolean literal can either have value as *True* or as *False*.

### (iv) Special Literal None

Python has one special literal, which is **None**. The **None** literal is used to indicate absence of value.

Python can also store literal collections, in the form of **tuples** and **lists** etc.

## 1.2.4 Operators

*Operators* are tokens that trigger some computation / action when applied to variables and other objects in an expression.

The operators can be **arithmetic operators** (+, –, *, /, %, **, //), **bitwise operators** (&, ^, |), **shift operators** (<<, >>), **identity operators** (is, is not), **relational operators** (>, <, >=, <=, ==, !=), **logical operators** (and, or), **assignment operator** (=), **membership operators** (in, not in), and **arithmetic-assignment operators** (/=, +=, –=, */, %=, **=, //=).

## 1.2.5 Punctuators

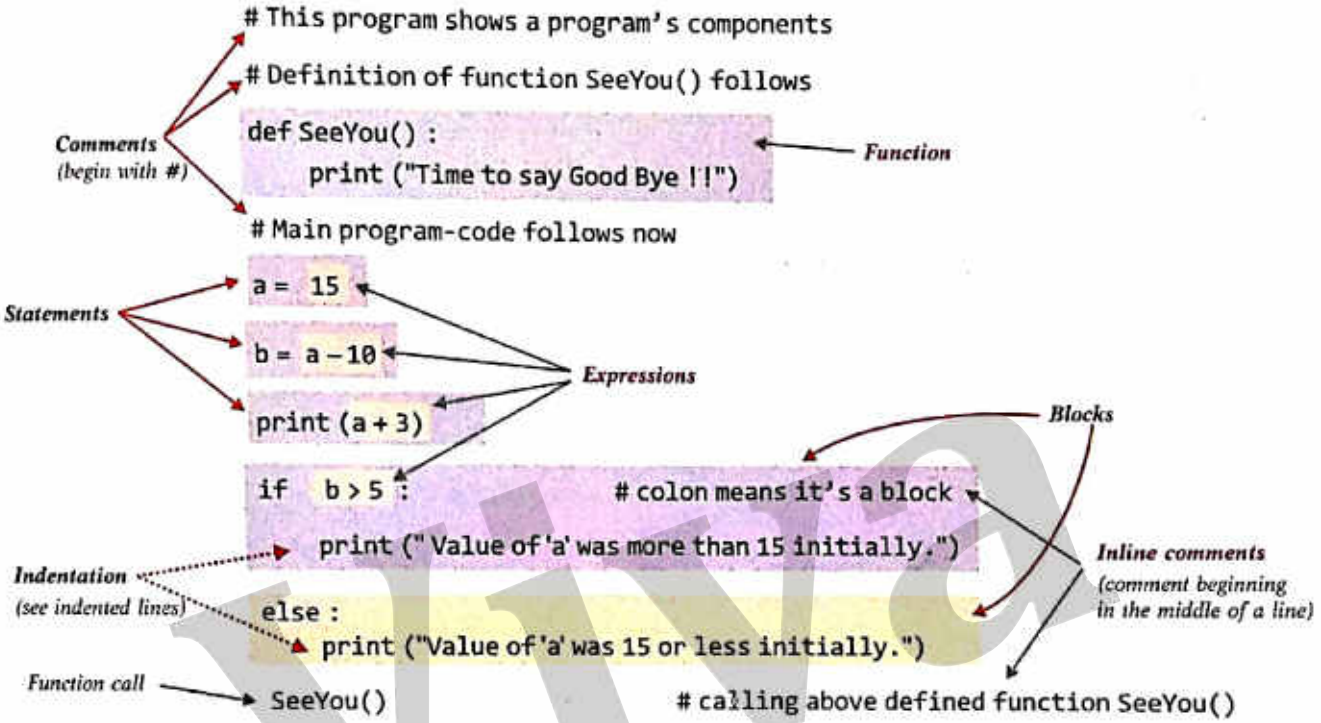
Punctuators are symbols that are used in programming languages to organize sentence structures, and indicate the rhythm and emphasis of expressions, statements, and program structure.

Most common punctuators of Python programming language are :

```
' " # \ ( ) [ ] { } @ , : . ` =
```

## 1.3 BAREBONES OF A PYTHON PROGRAM

A Python program may contain various elements such as *comments, statements, expressions* etc. Let us talk about the basic structure of a Python program.




As you can see that the above sample program contains various components like :

- **Expressions**, which are any legal combination of symbols that ***represents a value.***
- **Statements**, which are programming instructions.
- **Comments**, which are the additional readable information to clarify the source code. Comments can be **single line comments**, that start with # and **multi-line comments** that can be either triple-quoted strings or multiple # style comments.
- **Functions**, which are named code-sections and can be reused by specifying their names (*function calls*).
- **Block(s) or suite(s)**, which is a group of statements which are part of another statement or a function. All statements inside a block or suite are indented at the same level.

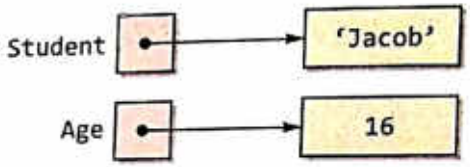
## 1.4 VARIABLES AND ASSIGNMENTS

Variables represent labelled storage locations, whose values can be manipulated during program run.

In Python, to create a variable, just assign to its name the value of appropriate type. For example, to create a variable namely **Student** to hold student's name and variable **age** to hold student's age, you just need to write somewhat similar to what is shown below :

```
Student = 'Jacob'
Age = 16
```

Python will internally create labels referring to these values as shown below.




### 1.4.1 Dynamic Typing

In Python, as you have learnt, a variable is defined by assigning to it some value (of a particular type such as numeric, string etc.). For instance, after the statement :

```
X = 10
```

We can say that variable *x* is referring to a value of integer type.

Later in your program, if you reassign a value of some other type to variable *x*, Python will not complain (no error will be raised), *e.g.*,

```
X = 10
print(X)
X = "Hello World"
print(X)
```

**DYNAMIC TYPING**

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

Above code will yield the output as :

```
10
Hello World
```

So, you can think of Python variables as labels associated with objects (literal values in our case here) ; with dynamic typing, Python makes a label refer to new value with new assignment (Fig. 1.2). Following figure illustrates it.

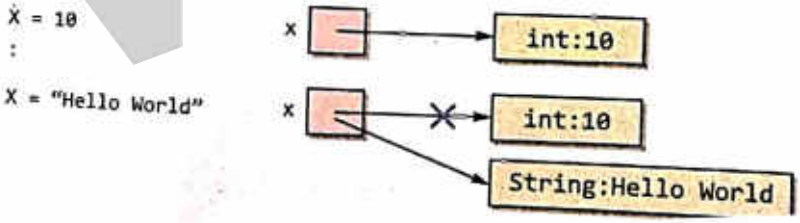



Figure 1.2 Dynamic typing in Python variables.

#### Dynamic Typing vs. Static Typing

**Dynamic typing** is different from **Static Typing**. In **Static typing**, a data type is attached with a variable when it is defined first and it is fixed. That is, data type of a variable cannot be changed in *static typing* whereas there is no such restriction in *dynamic typing*, which is supported by Python.

### 1.4.2 Multiple Assignments

Python is very versatile with assignments. Let's see how.

1. ***Assigning same value to multiple variables.*** You can assign same value to multiple variables in a single statement, *e.g.*,

```
a = b = c = 10
```

It will assign value 10 to all three variables *a, b, c.*

2. ***Assigning multiple values to multiple variables.*** You can even assign multiple values to multiple variables in single statement, *e.g.*,

```
x, y, z = 10, 20, 30
```

It will assign the values *order wise, i.e.,* first variable is given first value, second variable the second value and so on. That means, above statement will assign value 10 to *x*, 20 to y and 30 to z.

If you want to swap values of *x* and *y*, you just need to write as follows :

```
x, y = y, x
```

In Python, assigning a value to a variable means, variable's label is referring to that value.

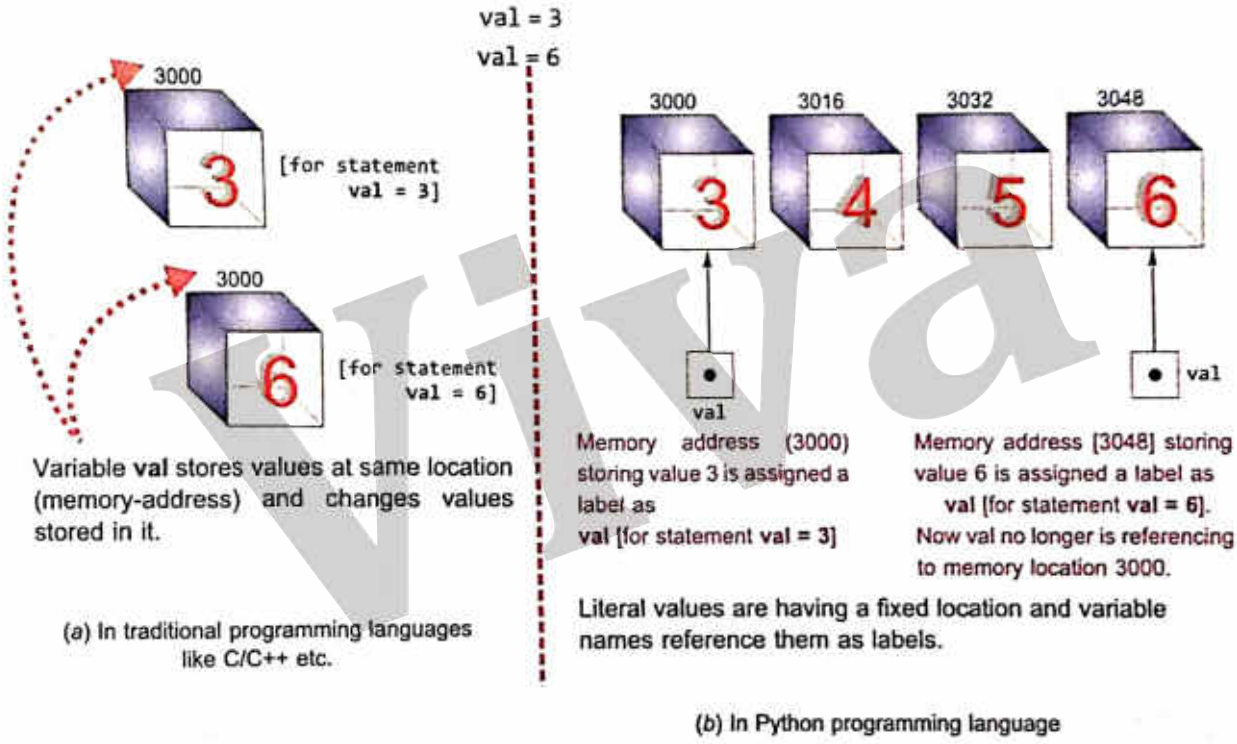



Figure 1.3 How variables are stored in traditional programming languages and in Python.

## 1.5 SIMPLE INPUT AND OUTPUT

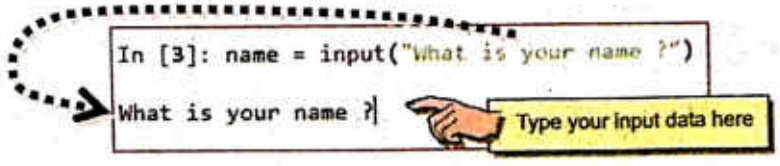
In Python 3.x, to get input from user interactively, you can use built-in function **input( )**. The function **input( )** is used in the following manner :

```
variable_to_hold_the_value = input (<prompt to be displayed>)
```

For example,

```
name = input ('What is your name ?')
```

The above statement will display the prompt as :



The **input( )** function always **returns a value of *String* type**. Python offers two functions **int( )** and **float( )** to be used with **input( )** to convert the values received through **input( )** into **int** and **float** types. You can :

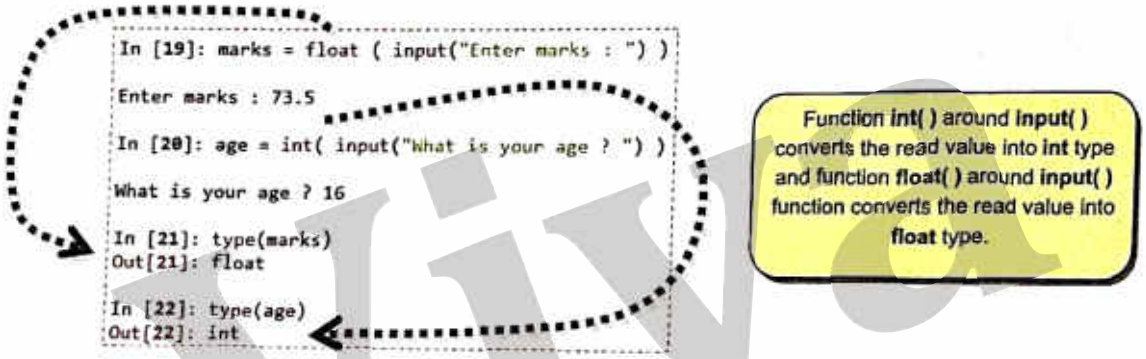
- Read in the value using **input( )** function.
- And then use **int( )** or **float( )** function with the *read value* to change the type of input value to **int** or **float** respectively.

You can also combine these two steps in a single step too, *i.e.*, as :

```
<variable_name> = int( input( <prompt string>) )
```

Or

```
<variable_name> = float( input( <prompt string>) )
```




While inputting integer values using **int( )** with **input( )**, make sure that the value being entered must be **int** type compatible. Similarly, while inputting floating point values using **float( )** with **input( )**, make sure that the value being entered must be **float** type compatible (*e.g.*, *'abc'* cannot be converted to *int* or *float*, hence it is not compatible).

## Output Through print( ) Statement

The print( ) function of Python 3.x is a way to send output to standard output device, which is normally a monitor.

The simplified syntax to use print( ) function is as follows :

```
print(*objects, [ sep = ' ' or <separator-string> end = '\n' or <end-string> ])¶
```

***objects** means it can be one or multiple comma separated *objects* to be printed.

Let us consider some simple examples first :

```
print ("hello")                          # a string
print (17.5)                             # a number
print (3.14159*(r*r))                    # the result of a calculation, which will
                                         # be performed by Python and then printed
                                         # out (assuming that some number has been
                                         # assigned to the variable r)
print ("I\'m", 12 + 5, "years old.")     # multiple comma separated expressions
```

The print statement has a number of features :

- it auto-converts the items to strings *i.e.*, if you are printing a numeric value, it will automatically convert it into equivalent string and print it ; for numeric expressions, it first evaluates them and then converts the result to string, before printing.
- it inserts spaces between items automatically because the default value of *sep* argument (separator character) is space(' ').

Consider this code :

```
print ("My", "name", "is", "Amit.")
```

*Four different string objects with no space in them are being printed.*

will print

```
My name is Amit.
```

*But the output line has automatically spaces inserted in between them because default **sep** character is a space.*

You can change the value of separator character with **sep** argument of **print( )** as per this :

The code :

```
print ("My", "name", "is", "Amit.", sep = '...' )
```

will print

```
My...name...is...Amit.
```

*This time the **print( )** separated the items with given **sep** character, which is '...'*

- it appends a newline character at the end of the line unless you give your own **end** argument. Consider the code given below :

```
print ("My name is Amit.")
print("I am 16 years old")
```

It will produce output as :

```
My name is Amit.
I am 16 years old
```

**NOTE**

A print( ) function without any value or name or expression prints a blank line.

If you explicitly give an **end** argument with a **print( )** function then the **print( )** will print the line and end it with the string specified with the **end** argument, and not the newline character, *e.g.*, the code

```
print("My name is Amit. ", end = '$')
print("I am 16 years old. ")
```

will print output as :

```
My name is Amit. $I am 16 years old.
```

*This time the **print( )** ended the line with given **end** character, which is'$' here. And because it was not **newline**, next line was printed from here itself.*

**Program 1.1** *Write a program to input a number and print its cube.*

```
num = float(input('Enter a number: '))
num_cube = num * num * num
print('The cube of', num, 'is', num_cube)
```

**Program 1.2** *Write a program to input a number and print its square root.*

```
num = float(input('Enter a number: '))
num_sqrt = num ** 0.5
print('The square root of', num, 'is', num_sqrt)
```

## 1.6 DATA TYPES

Data types are means to identify type of data and set of valid operations for it. Python offers following built-in core data types : (*i*) *Numbers* (*ii*) *String* (*iii*) *List* (*iv*) *Tuple* (*v*) *Dictionary*.

### (*i*) Data types for Numbers

Python offers following data types to store and process different types of numeric data :

(*a*) Integers
- Integers (signed)
- Booleans

(*b*) Floating-Point Numbers

(*c*) Complex Numbers

(*a*) **Integers.** There are *two* types of integers in Python :

(*i*) **Integers (signed).** It is the normal integer representation of whole numbers. *Python 3.x* provides single data type (***int***) to store any integer, whether *big* or *small*.
It is signed representation, *i.e.*, the integers can be positive as well as negative.

(*ii*) **Booleans.** These represent the truth values ***False*** and ***True***. The Boolean type is a subtype of plain integers, and Boolean values *False* and *True* behave like the values 0 and 1, respectively.

(*b*) **Floating Point Numbers.** 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.

(*c*) **Complex Numbers.** Python represents complex numbers in the form $A + B\,j$. Complex numbers are a composite quantity made of two parts : *the real part* and the *imaginary part*, both of which are represented internally as ***float*** values (floating point numbers).

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

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

**NOTE**

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

Table 1.1 *The Range of Python Numbers*

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

### (ii) Data Type for Strings

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 :

"abcd" , "1234", '$%^&', '????', "ŠÆËá" , "αβγ", 'इक', "बस", "ت ج د ن ث"

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

**NOTE**

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

### (iii) Lists

A List in Python represents a group 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]
```

In list too, 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.

### (iv) Tuples

Tuples are represented as group of comma-separated values of any date type within parentheses, *e.g.,* following are some tuples :

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

**NOTE**

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

### (v) Dictionaries

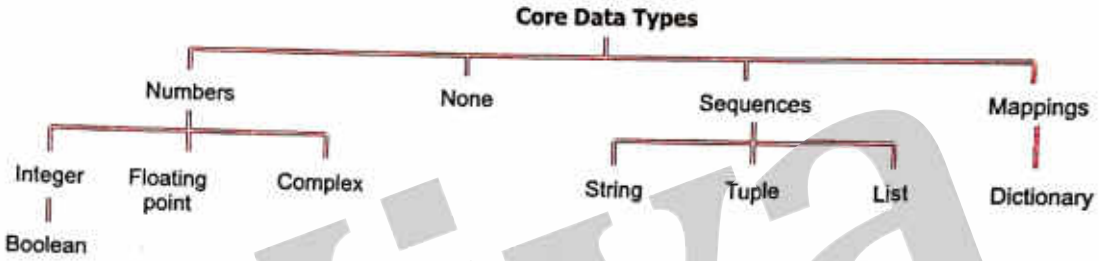
The *dictionary* is an unordered set of comma-separated **key : value** pairs, within { }, with the requirement that within a dictionary, no two keys can be the same (*i.e.*, there are unique keys within a dictionary). For instance, following are some dictionaries :

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

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

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

Following figure summarizes the core data types of Python.




## 1.7 MUTABLE AND IMMUTABLE TYPES

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

### Immutable Types

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

In immutable types, the **variable names** are stored as references to a value-object. Each time you change the value, the variable's reference memory address changes. See following explanation for sample code given below :

```
P = 5
q = P
r = 5
      :
P = 10
r = 7
q = r
```

- Initially these three statements are executed :

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

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

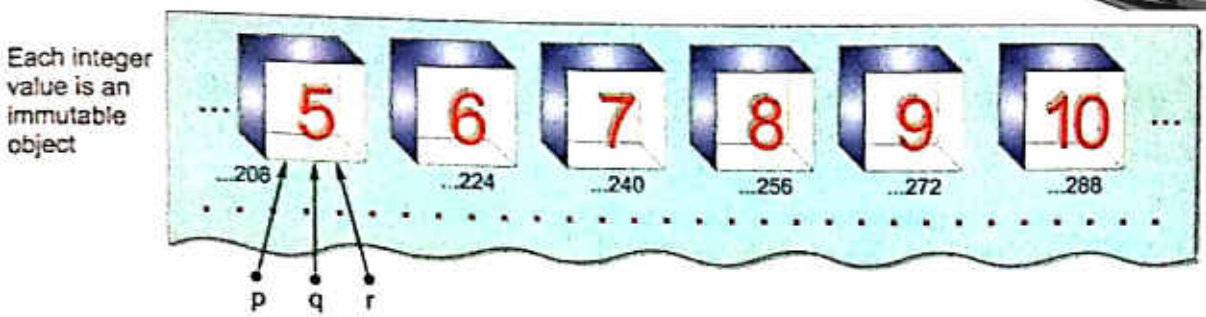



Figure 1.4

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

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

then these variable names are made to point to different integer objects.

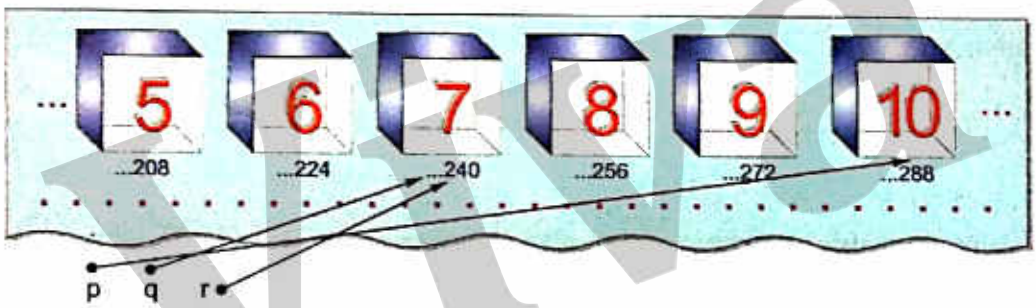



Figure 1.5

## Mutable Types

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

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].

**NOTE**

Python frontloads some commonly used values in memory. Each variable referring to that value actually stores that memory address of the value. Multiple variables/identifiers can refer to a value. Internally Python keeps count of how many identifiers/ variables are referring to a value.

**NOTE**

Mutable objects are :
***list, dictionary, set***
Immutable objects:
***int, float, complex, string, tuple***

## 1.8 EXPRESSIONS

An expression in Python is any valid combination of *operators, literals* and *variables*. The expressions in Python can be of any type : *arithmetic expressions, string expressions, relational expressions, logical expressions, compound expressions* etc.

**Arithmetic expressions** involve numbers (integers, floating-point numbers, complex numbers) and arithmetic operators, *e.g.*, 2 + 5 ** 3, – 8 * 6/5

**EXPRESSION**

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

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
```

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
```

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

Following are some legal string expressions :

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

### 1.8.1 Evaluating Arithmetic Operations

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

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

In a mixed arithmetic expression, Python converts all operands up to the type of the largest operand (***type promotion***). In the 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.

**NOTE**

An Implicit type conversion is a conversion performed by the compiler without programmer's intervention.

Table operator precedence

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

Highest

Lowest

Example 1.1 *Consider below given expressions what. What will be the final result and final data type ?*

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

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

(*c*)
```
a, b = 3, 6.0
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 pt result.*

(*b*) In expression

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

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

(*c*) In expression

```
c = 6.0 % 3
c = 0.0
```

```
b   %   a
|       |
float   int
  \     /
   float
```

## 1.8.2 Evaluating Relational Expressions

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

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

Example 1.2 *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 $(N <= b)$

### 1.8.3 Evaluating Logical Expressions

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.

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

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

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

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

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

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

Example 1.3 *What will be the output of following expression ?*

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

**Solution.** False

### 1.8.4 Type Casting (Explicit Type Conversion)

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

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

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

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

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

```
int(b)
```

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

**TYPE CASTING**

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

### 1.8.5 Math Library Functions

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

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

```
import math
```

Then you can use **math** library's functions as math.<function-name> .

Table 1.2 *Some Mathematical Functions in math Module*

| S. No. | Function | Prototype (General Form) | Description | Example |
|---|---|---|---|---|
| 1. | ceil | math.ceil(num) | The **ceil( )** function returns the smallest integer not less than *num*. | math.ceil(1.03) gives 2.0<br>math.ceil(–1.03) gives –1.0. |
| 2. | sqrt | math.sqrt (num) | The **sqrt( )** function returns the square root of *num*. If num <0, domain error occurs. | math.sqrt(81.0) gives 9.0. |
| 3. | exp | math.exp(arg) | The **exp( )** function returns the natural logarithm **e** raised to the *arg* power. | math.exp(2.0) gives the value of $e^2$. |
| 4. | fabs | math.fabs (num) | The **fabs( )** function returns the absolute value of *num*. | math.fabs(1.0) gives 1.0<br>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<br>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.<br>math.log(1024, 2) will give logarithm of 1024 to the base 2. |
| 7. | log10 | math.log10 (num) | The **log10( )** function returns the base 10 logarithm for *num*. A domain error occurs if *num* is negative and a range error occurs if the argument is zero. | math.log10(1.0) gives base 10 logarithm for 1.0. |
| 8. | pow | math.pow (base, exp) | The **pow( )** function returns *base* raised to *exp* power *i.e,. base exp*. A domain error occurs if *base* =0 and *exp* <=0 ; also if *base* <0 and *exp* is not integer. | math.pow(3.0, 0) gives value of $3^0$.<br>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)<br>(*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)<br>(*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)<br>(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 |

## LET US REVISE

- *A Python program can contain various components like expressions, statements, comments, functions, blocks and indentation.*
- *Blocks are represented through indentation.*
- *Python supports dynamic typing i.e., a variable can hold values of different types at different times.*
- *The input( ) is used to obtain input from user ; it always returns a string type of value.*
- *Output is generated through print( ) (by calling print function) statement.*
- *Operators are the symbols (or keywords sometimes) that represent specific operations.*
- *An expression is composed of one or more operations. It is a valid combination of operators, literals and variables.*
- *Types of operators used in an expression determine its type. For instance, use of arithmetic operators makes it arithmetic expression.*
- *Expressions can be arithmetic, relational or logical, compound etc.*
- *In implicit conversion, all operands are converted up to the type of the largest operand, which is called type promotion or coercion.*
- *The explicit conversion of an operand to a specific type is called type casting and it is done using type conversion functions that is used as*

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

## 1.9 STATEMENT FLOW CONTROL

In a program, statements may be executed sequentially, selectively or iteratively. Every programming language provides constructs to support *sequence, selection* or *iteration*.

A **conditional** is a statement set which is executed, on the basis of result of a condition. A **loop** is a statement set which is executed repeatedly, until the end condition is satisfied.

### 1. Compound Statement

A compound statement represents a group of statements executed as a unit. The compound statements of Python are written in a specific pattern as shown below :

```
<compound statement header > :
    <indented body containing multiple simple and/or compound statements>
```

The **conditionals** and the **loops** are **compound statements**, *i.e.*, they contain other statements. For all compound statements, following points hold :

- The contained statements are not written in the same column as the control statement, rather they are indented to the right and together they are called a block.
- The first line of compound statement, *i.e.*, its header contains a colon (:) at the end of it.

*For example :*

```
num1 = int(input("Enter number1"))
num2 = int(input("Enter number1"))

if num2 < num1 :
    t = num2 * num2
    t = t + 10

print(num2, num1, t)
```

*The colon ( : ) at the end of header line means it is a compound statement*

*The contained statements within if are indented to the right*

### 2. Simple Statement

Compound statements are made of simple statements. Any single executable statement is a simple statement in Python.

### 3. Empty Statement

The simplest statement is the empty statement *i.e.*, a statement which does nothing. In Python an empty statement is the *pass* statement.

It takes the following form :

```
pass
```

Wherever Python encounters a **pass** statement, Python does nothing and moves to next statement in the flow of control.

## 1.10 THE IF CONDITIONALS

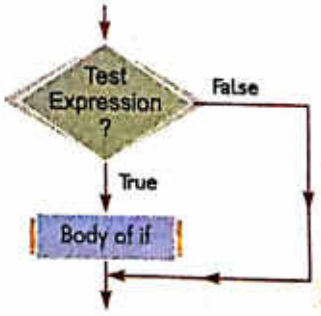
The if conditionals of Python come in multiple forms : plain **if** conditional, **if-else** conditional and **if-elif** conditionals.

### 1.10.1 Plain if Conditional Statement

An **if** statement tests a particular condition; if the condition evaluates to *true*, a course-of-action is followed *i.e.*, a statement or set-of-statements is executed. If the condition is *false*, it does nothing :

The syntax (general form) of the if statement is as shown below :

```
if <conditional expression> :
    statement
    [statements]
```



For example, consider the following code fragments using **if** conditionals :

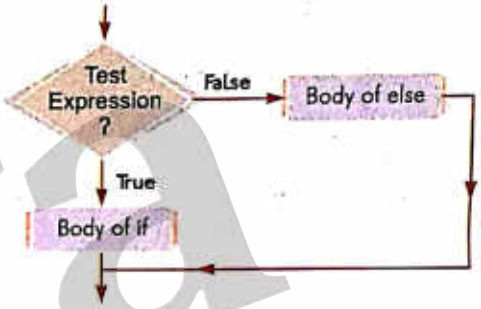
```
if grade == 'A' :
    print("Congratulations! You did well")
:
if a > b :
    print("A has more than B has")
    print("Their difference is", (a-b))
```

### 1.10.2 The if-else Conditional Statement

This form of **if** statement tests a condition and if the condition evaluates to *true*, it carries out statements indented below **if** and in case condition evaluates to *false*, it carries out statements indented below **else**.

The syntax (general form) of the **if-else** statement is as shown below :

```
if <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```




For example, consider the following code fragments using **if-else** conditionals :

```
if a >= 0 :
    print(a, "is zero or a positive number")
else :
    print(a, "is a negative number")
```

*The colon ( : ) is in both : the if header as well as else line*

*The statements in if-block and else are indented*

### 1.10.3 The if-elif Conditional Statement

Sometimes, you want to check a condition when control reaches else, *i.e.*, condition test in the form of *else if*. To serve such conditions, Python provides **if-elif** and **if-elif-else** statements.

The general form of these statements is :

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
```

and

```
if <conditional expression> :
    statement
    [statements]
elif <conditional expression> :
    statement
    [statements]
else :
    statement
    [statements]
```


Consider following two example code fragments :

```
if runs >= 100 :
    print("Batsman scored a century")
elif runs >= 50 :
    print("Batsman scored a fifty")
else :
    print("Batsman has neither scored a century nor fifty")
:
if num < 0 :
    print(num, "is a negative number.")
elif num == 0 :
    print(num, "is equal to zero.")
else :
    print(num, "is a positive number.")
```

**Program 1.3** Write a program that inputs an integer in range 0 - 999 and then prints if the integer entered is a 1/2/3 digit number.

```
num = int(input("Enter a number (0..999) : "))
if num < 0:
    print("Invalid entry. Valid range is 0 to 999.")
elif num < 10:
    print("Single digit number is entered")
elif num < 100:
    print("Two digit number is entered")
elif num <= 999:
    print("Three digit number is entered")
else:
    print("Invalid entry. Valid range is 0 to 999.")
```

Sample runs of this program are given below :

```
Enter a number (0..999) : -3
Invalid entry. Valid range is 0 to 999.
=======================================
Enter a number (0..999) : 4
Single digit number is entered
=======================================
Enter a number (0..999) : 100
Three digit number is entered
=======================================
Enter a number (0..999) : 10
Two digit number is entered
=======================================
Enter a number (0..999) : 3000
Invalid entry. Valid range is 0 to 999.
```

### 1.10.4 Nested if Statements

Sometimes you may need to test additional conditions. For such situations, Python also supports **nested-if** form of **if**. A nested if is an if that has another **if** in its **if's body** or in **elif's body** or in its **else's body**.

Consider the following example code using **nested-if** statements :

```
x = int(input("Enter first number :"))
y = int(input("Enter second number :"))
z = int(input("Enter third number :"))

min = mid = max = None
if x < y and x < z :
    if y < z :
        min, mid, max = x, y, z
    else :
        min, mid, max = x, z, y
elif y < x and y < z :
    if x < z :
        min, mid, max = y, x, z
    else :
        min, mid, max = y, z, x
else :
    if x < y :
        min, mid, max = z, x, y
    else:
        min, mid, max = z, y, x
print("Numbers in ascending order :", min, mid, max)
```

*if statements inside another if (Nested ifs)*

### 1.10.5 Storing Conditions

Sometimes the conditions being used in code are complex and repetitive. In such cases to make your program more readable, you can use **named conditions** *i.e.*, you can store conditions in a name and then use that named conditional in the **if** statements.

Consider the following example code :

```
b, c = 2, 3

#Store condition in a name called all.
all = a == 1 and b == 2 and c == 3
#Test variable.
if all:
    print("Condition fulfilled")
#Use it again.
if all:
    print("Condition fulfilled again.")
```

*this condition is given name as all.*

**TIP**

Using Named or Stored conditionals reduces repetition. It may also improve processing speed: fewer comparisons are done

Using nested if statements, we can rewrite the previous program as follows :

**Program 1.4** Write a program that inputs an integer in range 0 - 999 and then prints if the integer entered is a 1/2/3 digit number. Use Nested if statements.

```
num = int(input("Enter a number (0..999) : "))
if num < 0 or num > 999:
    print("Invalid entry. Valid range is 0 to 999.")
else:
    if num < 10:
        print("Single digit number is entered")
    else :
        if num < 100:
            print("Two digit number is entered")
        else :
            print("Three digit number is entered")
```

The sample run of this program is just the same as previous program.

## 1.11 LOOPING STATEMENTS

Python provides *two* kinds of loops : *for loop* and *while loop* to represent counting loop and conditional loop respectively.

### 1.11.1 The for Loop

The **for loop** of Python is designed to process the items of any sequence, such as a list or a string, one by one.

#### The General Form of for loop

The general form of **for loop** is as given below :

```
for <variable> in <sequence> :
    statements_to_repeat
```

For example, consider the following loop :

```
for element in [10, 15, 20, 25] :
    print(element + 2, end = ' ')
```

The above loop would give output as :

```
12  17  22  27
```

- The above-given **for loop** executes a sequence of statements for each of the elements of given sequence [10, 15, 20, 25].
- To keep the count, it uses a control variable (*element* in above case) in that takes a different value on each iteration. Firstly value 10, then the next value of sequence, 15, then 20 and lastly 25.


## The *range( )* based for loop

For number based lists, you can specify **range( ) function** to represent a list as in :

```
for val in range(3, 18) :
    print(val)
```

In the above loop, **range(3, 18)** will first generate a list [3, 4, 5, ..., 16, 17] with which for loop will work. You need not define loop variable (**val** above) beforehand in a for loop.

As mentioned, a **range( )** produces an integer number sequence. There are *three* ways to define a range :

```
range(stop)                 #elements from 0    to stop-1, incrementing by 1
range(start, stop)          #elements from start to stop-1, incrementing by 1
range(start, stop, step)    #elements from start to stop-1, incrementing by step
```

- The ***start value*** is always part of the range. The ***stop value*** is never part of the range. The ***step*** shows the difference between two consecutive values in the range.
- If ***start value*** is omitted, it supposed to be 0. If the ***step*** is omitted, it is supposed to be 1.

Consider some examples of **range( )** as given below :

```
range(7)          0, 1, 2, 3, 4, 5, 6
range(5, 12)      5, 6, 7, 8, 9, 10, 11
range(5, 13, 2)   5, 7, 9, 11
range(10, 4)      no value
range(10, 4, -1)  10, 9, 8, 7, 6, 5
```

For example, the next program shows the cube of the numbers from 15 to 20 :

**Program 1.5** *Write a program to print cubes of numbers in the range 15 to 20.*

```
for i in range(15, 21):
    print("Cube of number", i, end = ' ')
    print("is", i ** 3)
```

```
Cube of number 15 is  3375
Cube of number 16 is  4096
Cube of number 17 is  4913
Cube of number 18 is  5832
Cube of number 19 is  6859
Cube of number 20 is  8000
```

**Program 1.7** Write a program to print square root of every alternate number in the range 1 to 10.

```
for i in range(1, 10, 2) :
    print("square root of", i, "is", (i ** 0.5))
```

### 1.11.2 The while Loop

A while loop is a conditional loop that will repeat the instructions within itself as long as a conditional remains true (Boolean *True* or truth value *true*).

The general form of Python while loop is :

```
while <logicalExpression> :
    loop-body
```

where the loop-body may contain a single statement or multiple statements or an *empty statement* (i.e., **pass statement**). The loop iterates while the *logical Expression* evaluates to *true*. When the expression becomes *false*, the program control passes to the line after the loop-body.

**Program 1.8** Write a program that multiplies two integer numbers without using the * operator, using repeated addition.

```
n1 = int(input("Enter first number :"))
n2 = int(input("Enter second number :"))
product = 0
count = n1
while count > 0 :
    count = count - 1
    product = product + n2
print("The product of", n1, "and", n2, "is", product)
```

```
Enter first number : 4
Enter second number : 5
The product of 4 and 5 is 20
```

In general, the while loop is used when it is not possible to know in advance how many times the loop will be executed, but the termination condition is known. The while loop is an entry-controlled loop as it has a control over entry in the loop in the form of test condition.

## 1.12 JUMP STATEMENTS – break AND continue

Python offers two jump statements – **break** and **continue** – to be used within loops to jump out of loop-iterations.

### The *break* Statement

A **break** statement terminates the very loop it lies within. Execution resumes at the statement immediately following the body of the terminated statement.

**NOTE**

A ***break*** statement skips the rest of the loop and jumps over to the statement following the loop.

The following figure (Fig. 1.6) explains the working of a *break* statement.

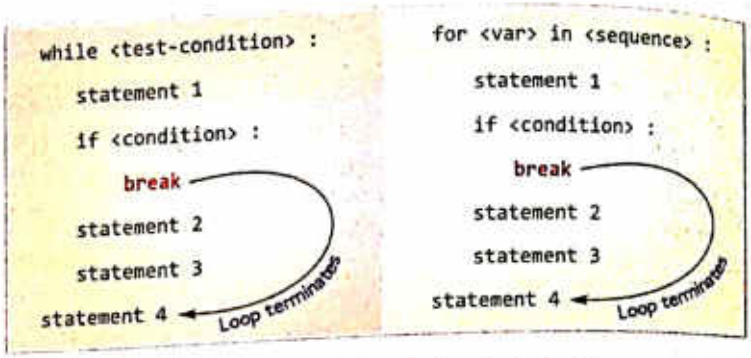



Figure 1.6 The working of a **break** statement.

The following code fragment gives you an example of a break statement :

```
a = b = c = 0
for i in range(1, 11) :          # range (1, 11) will generate a sequence of numbers from 1 .. 10.
    a = int(input ("Enter number 1 :"))
    b = int(input ("Enter number 2 :"))
    if b == 0 :
        print("Division by zero error! Aborting!")
        break
    else :
        c = a / b
        print("Quotient = ", c)
print("Program over !")
```

The above code fragment intends to divide ten pairs of numbers by inputting two numbers *a* and *b* in each iteration. If the number *b* is zero, the loop is immediately terminated displaying message '**Division by zero error! Aborting!**' otherwise the numbers are repeatedly input and their quotients are displayed.

## The continue Statement

Unlike **break** statement, the continue statement forces the next iteration of the loop to take place, skipping any code in between.

**NOTE**

The **continue** statement skips the rest of the loop statements and causes the next iteration of the loop to take place.

The following figure (Fig. 1.7) explains the working of *continue* statement :

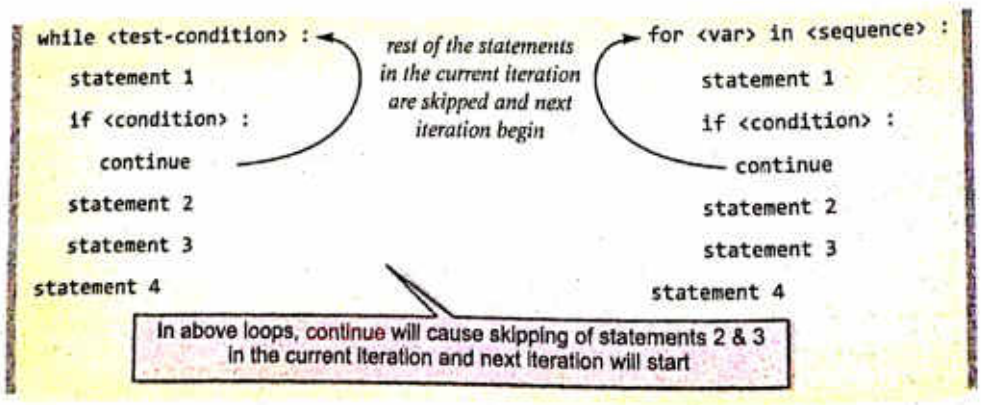



Figure 1.7 The working of a **continue** statement.

## 1.13 MORE ON LOOPS

There are two more things you need to know about loops – the *loop else clause* and nested loops.

### 1.13.1 Loop else Statement

Python loops have an optional else clause. Complete syntax of Python loops along with **else** clause is as given below :

```
for <variable> in <sequence> :
    statement1
    statement2
    :
else :
    statement(s)
```

```
while <test condition> :
    statement1
    statement2
    :
else :
    statement(s)
```

The **else clause of a Python loop** executes when the loop terminates normally, *i.e.*, when test-condition results into *false* for a *while loop* or *for loop* has executed for the last value in the sequence; **not when the break statement terminates the loop.**

Following figure (1.8) shows you control flow in Python loops.

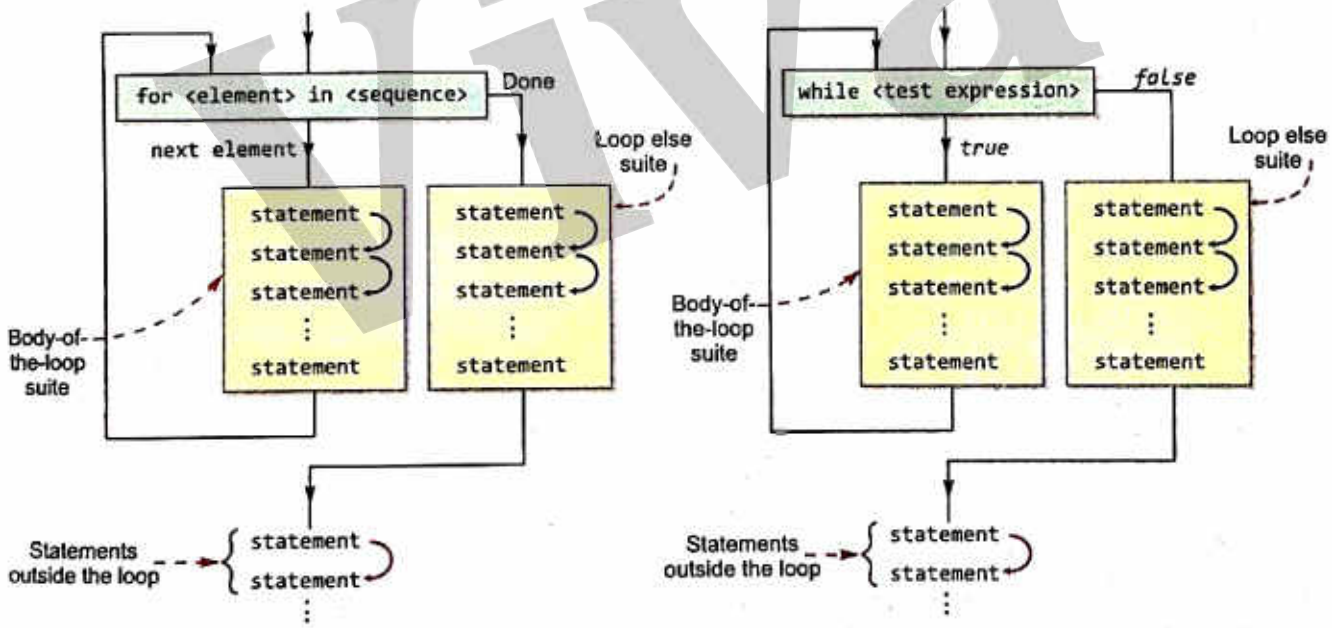



Figure 1.8 Control flow in Python loops.

Consider following example :

```
for a in range (1, 4) :
    if a % 8 == 0 :
        break
    print("Element is", end = ' ')
    print(a)
else :
    print("Ending loop after printing all elements of sequence")
```

The above will give the following output :

```
Element is 1
Element is 2
Element is 3
Ending loop after printing all elements of sequence.
```

*This line is printed because the else clause of given for loop executed when the for loop was terminating, normally.*

If, in above code, you change the line

```
if a % 8 == 0 :
```

with

```
if a % 2 == 0 :
```

Then the output will be :

```
Element is 1
```

*Notice that for a = 2, the break got executed and loop terminated. Hence just one element got printed (for a = 1). As break terminated the loop, the else clause also did not execute, so no line after the printing of elements.*

**NOTE**

The else clause of a Python loop executes when the loop terminates normally, not when the loop is terminating because of a break statement.

The **else clause** works identically in **while loop**, *i.e.*, executes if the test-condition goes *false* and in case of **break** statement, the **loop-else clause** is not executed.

## 1.13.2 Nested Loops

A loop may contain another loop in its body. This form of a loop is called nested loop. But in a nested loop, the inner loop must terminate before the outer loop.

The following is an example of a nested loop :

```
for i in range(1, 6) :
    for j in range (1, i ) :
        print("*", end = ' ')
print( )
```

In nested loops, a **break** statement will terminate the very loop it appears in. That is, if **break** statement is inside the inner loop then only the inner loop will terminate and outer loop will continue. If however, the **break** statement is in outer loop, the outer loop will terminate.

## LET US REVISE

- *Statements are the instructions given to the computer to perform any kind of action.*
- *Python statements can be on one of these types : empty statement, single statement and compound statement.*
- *A compound statement represents a group of statements executed as a unit.*
- *Every compound statement of Python has a header and an indented body below the header. Some examples of compound statements are : functions, if statement, while statement etc.*
- *Python provides one selection statement* **if** *in many forms –* ***if..else*** *and* ***if..elif..else.***
- *The statements that allow a set of instructions to be performed repeatedly are iteration statements.*
- *Python provides two looping constructs – for and while. The* **for** *loop is a counting loop and* **while** *is a conditional loop.*
- *The while loop is an entry-controlled loop as it has a control over entry in the loop in the form of test condition.*
- *Loops in Python can have* **else clause** *too. The else clause of a loop is executed in the end of the loop only when loop terminates normally.*
- *The* **break** *statement can terminate a loop immediately and the control passes over to the statement following the statement containing break.*
- *In nested loops, a* **break** *statement terminates the very loop it appears in.*
- *The* **continue** *statement abandons the current iteration of the loop by skipping over the rest of the statements in the loop-body. It immediately transfers control to the beginning of the next iteration of the loop.*

## Solved Problems

1. *What is the difference between a keyword and an identifier ?*

   Solution. **Keyword** is a special word that has a special meaning and purpose. Keywords are reserved and are a few. *For example,* if, elif, else etc. are keywords.

   **Identifier** is the user-defined name given to a part of a program *viz.* variable, object, function etc. Identifiers are not reserved. These are defined by the user but they can have letters, digits and a symbol underscore. They must begin with either a letter or underscore. *For instance, _chk, chess, trial* etc. are identifiers in Python.

2. *What are literals in Python ? How many types of literals are allowed in Python ?*

   Solution. Literals mean constants *i.e.,* the data items that never change their value during a program run. Python allows *five* types of literals :

   (*i*) String literals (*ii*) Numeric literals
   (*iii*) Boolean literals (*iv*) Special Literal *None*
   (*v*) Literal Collections like tuples, lists etc.

3. *How many ways are there in Python to represent an integer literal ?*

   Solution. Python allows *three* types of integer literals :

   (*a*) Decimal (base 10) integer literals
   (*b*) Octal (base 8) integer literals
   (*c*) Hexadecimal (base 16) integer literals

(*a*) **Decimal Integer Literals.** An integer literal consisting of a sequence of digits is taken to be decimal integer literal unless it begins with 0 (digit zero).

*For instance,* 1234, 41, +97, –17 are decimal integer literals.

(*b*) **Octal Integer Literals.** A sequence of digits starting with 0 (digit zero) is taken to be an octal integer. *For instance,* decimal integer 8 will be written as 010 as octal integer. ($8_{10} = 10_8$) and decimal integer 12 will be written as 014 as octal integer ($12_{10} = 14_8$).

(*c*) **Hexadecimal Integer Literals.** A sequence of digits preceded by ***0x*** or ***0X*** is taken to be an hexadecimal integer.

*For instance,* decimal 12 will be written as 0XC as hexadecimal integer.

Thus number 12 will be written either as 12 (as decimal), 014 (as octal) and 0XC (as hexadecimal).

4. *How many types of strings are supported in Python ?*

Solution. Python allows two string types :

| | | |
|---|---|---|
| (*i*) | *Single line Strings* | Strings that are terminated in single line |
| (*ii*) | *Multiline Strings* | Strings storing multiple lines of text. |

5. *What is None literal in Python ?*

Solution. Python has one special literal called **None**.

The **None** literal is used to indicate something that has not yet been created. It is a legal empty value in Python.

6. *The following code is not giving desired output. We want to input value as 20 and obtain output as 40. Could you pinpoint the problem ?*

```
Number = input( "Enter Number" )
DoubleTheNumber = Number * 2
Print (DoubleTheNumber)
```

Solution. The problem is that input( ) returns value as a string, so the input value 20 is returned as string '20' and not as integer 20. So the output is 2020 in place of required output 40.

Also **Print** is not legal statement of Python ; it should be **print**.

7. *Why is following code giving errors ?*

```
name = "Rehman"
print("Greetings !!!")
    print("Hello", name)
    print("How do you do ?")
```

Solution. The problem with above code is inconsistent indentation. In Python, we cannot indent a statement unless it is inside a suite and we can indent only as much as required.

Thus, corrected code will be :

```
name = "Rehman"
print("Greetings !!!")
print("Hello", name)
print("How do you do ?")
```

8. *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

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

Solution. Python provides following data types of handle numbers (version 3.x) :

(*i*) Plain integers (*ii*) Long integers
(*iii*) Boolean (*iv*) Floating-point numbers
(*v*) Complex numbers

10. *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* as 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.

11. *What is the role of comments and indentation in a program ?*

Solution. Comments provide explanatory notes to the readers of the program. Compiler or interpreter ignores the comments but they are useful for specifying additional descriptive information regarding the code and logic of the program.

Indentation makes the program more readable and presentable. Its main role is to highlight nesting of groups of control statements.

12. *What is a statement ? What is the significance of an empty statement ?*

Solution. A statement is an instruction given to the computer to perform any kind of action.

An empty statement is useful in situations where the code requires a statement but logic does not. To fill these two requirements simultaneously, empty statement is used.

Python offers **pass** statement as an empty statement.

13. *If you are asked to label the Python loops as determinable or non-determinable, which label would you give to which loop ? Justify your answer.*

Solution. The 'for loop' can be labelled as ***determinable loop*** as number of its iterations can be determined beforehand as the size of the sequence, it is operating upon.

The 'while loop' can be labelled as ***non-determinable loop,*** as its number of iterations cannot be determined beforehand. Its iterations depend upon the result of a test-condition, which cannot be determined beforehand.

14. *There are two types of else clauses in Python. What are these two types of else clauses ?*

Solution. The two types of Python else clauses are :

(*a*) else in an if statement (*b*) else in a loop statement

The *else* clause of an *if* statement is executed when the condition of the if statement results into *false.*

The *else* clause of a *loop* is executed when the loop is terminating normally *i.e.,* when its test-condition has gone *false* for a *while* loop or when the *for* loop has executed for the last value in sequence.

15. *Write a program that asks the user to input number of seconds and then expresses it in terms of many minutes and seconds it contains.*

Solution.

```
#get the number of seconds from the user
numseconds = input("Enter number of seconds")
numseconds_int = int(numseconds)

# extract the number of minutes using integer division
numminutes = numseconds_int//60

# extract the number of seconds remaining since the last minute using the modulo
remainingseconds = numseconds_int % 60

print('minutes:', numminutes)
print('seconds:', remainingseconds)
```

16. *Write a program that repeatedly asks from users some numbers until string 'done' is typed. The program should print the sum of all numbers entered.*

Solution.

```
#Unknown Number of Numbers to Sum
total = 0
s = input('Enter a number or "done": ')
while s != 'done' :
    num = int(s)
    total = total + num
    s = input('Enter a number or "done": ')
print('The sum of entered numbers is', total)
```

17. *Write a program to print a square multiplication table as shown below :*

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
| 2 | 4 | 6 | 8 | 10 | 12 | 14 | 16 | 18 |
| 3 | 6 | 9 | 12 | 15 | 18 | 21 | 24 | 27 |
| 4 | 8 | 12 | 16 | 20 | 24 | 28 | 32 | 36 |
| 5 | 10 | 15 | 20 | 25 | 30 | 35 | 40 | 45 |
| 6 | 12 | 18 | 24 | 30 | 36 | 42 | 48 | 54 |
| 7 | 14 | 21 | 28 | 35 | 42 | 49 | 56 | 63 |
| 8 | 16 | 24 | 32 | 40 | 48 | 56 | 64 | 72 |
| 9 | 18 | 27 | 36 | 45 | 54 | 63 | 72 | 81 |

Solution.

```
for row in range(1, 10):
    for col in range(1, 10):
        prod = row * col
        if prod < 10:
            print(' ', prod,' ', end = ' ')    #This adds a space if the number is single digit
        else:
            print(prod, ' ', end = ' ')
    print( )
```

## GLOSSARY

| | |
|---|---|
| ***Block*** | A group of consecutive statements having same indentation level. |
| ***Coercion*** | Implicit Type Conversion. |
| ***Constant*** | A data item that never changes its value during a program run. |
| ***Explicit Type Conversion*** | Forced data type conversion by the user. |
| ***Empty statement*** | A statement that appears in the code but does nothing. |
| ***Identifier*** | Name given by user for a part of the program. |
| ***Implicit Type Conversion*** | Automatic Internal Conversion of data type (lower to higher type) by Python. |
| ***Keyword*** | Reserved word having special meaning and purpose. |
| ***Lexical Unit*** | Other name of token. |
| ***Literal*** | Constant. |
| ***Operator*** | Symbol/word that triggers an action or operation. |
| ***Token*** | The smallest individual unit in a program. |
| ***Type Casting*** | Explicit Type Conversion. |
| ***Variable*** | Named stored location whose value can be manipulated during program run. |

## Assignments

### Type A : Short Answer Questions/Conceptual Questions

1. What are tokens in Python ? How many types of tokens are allowed in Python ? Exemplify your answer.
2. How are keywords different from identifiers ?
3. What are literals in Python ? How many types of literals are allowed in Python ?
4. Can nongraphic characters be used and processed in Python ? How ? Give examples to support your answer.
5. Out of the following, find those identifiers, which cannot be used for naming Variables or Functions in a Python program : [CBSE D 2016]

```
Price*Qty, class, For, do,
4thCol, totally, Row31, _Amount
```

6. How are floating constants represented in Python ? Give examples to support your answer.
7. How are string-literals represented and implemented in Python ?
8. What are operators ? What is their function ? Give examples of some unary and binary operators.
9. What is an expression and a statement ?
10. What all components can a Python program contain ?
11. What are variables ? How are they important for a program ?
12. Describe the concepts of block or suite and body. What is indentation and how is it related to block and body ?
13. What are data types ? How are they important ?
14. How many integer types are supported by Python ? Name them.
15. What are immutable and mutable types ? List immutable and mutable types of Python.
16. What is the difference between implicit type conversion and explicit type conversion ?

17. An immutable data type is one that cannot change after being created. Give three reasons to use immutable data.
18. What is entry controlled loop ? Which loop is entry controlled loop in Python ?
19. Explain the use of the pass statement. Illustrate it with an example.
20. Below are seven segments of code, each with a part coloured. Indicate the data type of each coloured part by choosing the correct type of data from the following type.

(a) int (b) float (c) bool (d) str (e) function (f) list of int (g) list of str

(i)
```
if temp < 32 :
    print("Freezing")
```

(ii)
```
L = [ 'Hiya', 'Zoya', 'Preet' ]
print(L[1])
```

(iii)
```
M = [ ]
for i in range (3) :
    M . append(i)
    print(M)
```

(iv)
```
L = [ 'Hiya', 'Zoya', 'Preet' ]
n = len (L)
if 'Donald' in L[1 : n] :
    print(L)
```

(v)
```
if n % 2 == 0 :
    print("Freezing")
```

(vi)
```
L = inputline.split( )
while L != ( ) :
    print(L)
    L = L[1 :]
```
*it converts the read line into words*

(vii)
```
L = [ 'Hiya', 'Zoya', 'Preet' ]
print(L[0] + L[1] )
```

## Type B : Application Based Questions

1. Fill in the missing lines of code in the following code. The code reads in a limit amount and a list of prices and prints the largest price that is less than the limit. You can assume that all prices and the limit are positive numbers. When a price 0 is entered the program terminates and prints the largest price that is less than the limit.

```
#Read the limit
limit = float(input("Enter the limit"))
max_price = 0
# Read the next price
next_price = float(input("Enter a price or 0 to stop:"))
while next_price > 0 :
    <write your code here>
    #Read the next price
    <write your code here>
if max_price > 0:
    <write your code here>
else :
    <write your code here>
```

2. Predict the outputs of the following programs :

(a)
```
count = 0
while count < 10:
    print('Hello')
    count += 1
```

(b)
```
x = 10
y = 0
while x > y:
    print(x, y)
    x = x - 1
    y = y + 1
```

(c)
```
keepgoing = True
x = 100
while keepgoing:
    print(x)
    x = x - 10
    if x < 50:
        keepgoing = False
```

(d)
```
x = 45
while x < 50:
    print(x)
```

(e)
```
for x in [1,2,3,4,5]:
    print(x)
```

(f)
```
for p in range(1,10):
    print(p)
```

(g)
```
for z in range(-500,500,100):
    print(z)
```

(h)
```
x = 10
y = 5
for i in range(x - y * 2):
    print("%", i)
```

(i)
```
c = 0
for x in range(10):
    for y in range(5):
        c += 1
print(c)
```

(j)
```
x = [1,2,3]
counter = 0
while counter < len(x):
    print(x[counter] * '%')
    for y in x:
        print(y * '* ')
    counter += 1
```

(k)
```
for x in 'lamp':
    print(str.upper(x))
```

(l)
```
x = 'one'
y = 'two'
counter = 0
while counter < len(x):
    print(x[counter], y[counter])
    counter += 1
```

(m)
```
x = "apple, pear, peach"
y = x.split(", ")
for z in y :
    print(z)
```

(n)
```
x = 'apple, pear, peach, grapefruit'
y = x.split(',')
for z in y:
    if z < 'm':
        print(str.lower(z))
    else:
        print(str.upper(z))
```

3. Find and write the output of the following python code :

```
for Name in ['Jayes', 'Ramya', 'Taruna', 'Suraj'] :
    print (Name)
    if Name[0] == 'T' :
        break
    else :
        print ('Finished!')
print ('Got it!')
```

4. How many times will the following for loop execute and what's the output ?

(i)
```
for i in range(-1,7, -2):
    for j in range (3):
        print(1, j)
```

(ii)
```
for i in range(1,3,1):
    for j in range(i+1):
        print('*')
```

5. Is the loop in the code below infinite ? How do you know (for sure) before you run it ?

```
m = 3
n = 5
while n < 10:
    m = n - 1
    n = 2 * n - m
    print(n, m)
```

## Type C : Programming Practice/Knowledge based Questions

1. Write a program to print one of the words negative, zero, or positive, according to whether variable *x* is less than zero, zero, or greater than zero, respectively.
2. Write a program that returns True if the input number is an even number, False otherwise.
3. Write a Python program that calculates and prints the number of seconds in a year.
4. Write a Python program that accepts two integers from the user and prints a message saying if first number is divisible by second number or if it is not.
5. Write a program that asks the user the day number in a year in the range 2 to 365 and asks the first day of the year – Sunday or Monday or Tuesday etc. Then the program should display the day on the day-number that has been input.
6. One foot equals 12 inches. Write a function that accepts a length written in feet as an argument and returns this length written in inches. Write a second function that asks the user for a number of feet and returns this value. Write a third function that accepts a number of inches and displays this to the screen. Use these three functions to write a program that asks the user for a number of feet and tells them the corresponding number of inches.
7. Write a program that reads an integer N from the keyboard computes and displays the sum of the numbers from N to (2 * N) if N is nonnegative. If N is a negative number, then it's the sum of the numbers from (2 * N) to N. The starting and ending points are included in the sum.
8. Write a program that reads a date as an integer in the format MMDDYYYY. The program will call a function that prints print out the date in the format <Month Name> <day>, <year>.

   Sample run :

```
Enter date : 12252019
December 25, 2019
```

9. Write a program that prints a table on two columns – table that helps converting miles into kilometres.
10. Write another program printing a table with two columns that helps convert pounds in kilograms.
11. Write a program that reads two times in military format (0900, 1730) and prints the number of hours and minutes between the two times.

    A sample run is being given below :

```
Please enter the first time : 0900
Please enter the second time : 1730
8 hours 30 minutes
```


# Viva Technologies

☎: 94257-01888,02888,03888

✉: vivatechgwl@gmail.com