2 Python Fundamentals

In This Chapter

2.1 INTRODUCTION

You must have heard the term IPO – Input, Process, Output. Most (in fact, nearly all) daily life and computer actions are governed by IPO cycle. That is, there is certain **I**nput, certain kind of **P**rocessing and an **O**utput.

Do you know that ***programs*** make IPO cycle happen ?

Anywhere and everywhere, where you want to transform some kind of input to certain output, you have some kind of input to certain output, you have to have a *program*. A program is a set of instructions that govern the processing. In other words, a program forms the base for processing.

In this chapter, we shall be talking about all basic elements that a Python program can contain. You'll be learning about Python's basics like *character set, tokens, expressions, statements, simple input and output* etc. So, are we all ready to take our first sincere step towards Python programming ? And, here we go :-).

2.2 PYTHON CHARACTER SET

Character set is a set of valid characters that a language can recognize. A character represents any letter, digit or any other symbol. Python supports Unicode encoding standard. That means Python has the following character set :

- Letters — A-Z, a-z
- Digits — 0-9
- Special symbols — space + - * / ** \ ( ) [ ] { } // = != == < , > . ' " "" , ; : % ! & # <= >= @ _(underscore)
- Whitespaces — Blank space, tabs (→), carriage return (↵), newline, formfeed
- Other characters — Python can process all ASCII and Unicode characters as part of data or literals.

2.3 TOKENS

In a passage of text, individual words and punctuation marks are called *tokens* or *lexical units* or lexical elements. 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***.

Let us talk about these one by one.

2.3.1 Keywords

Keywords are the words that convey a special meaning to the language compiler/interpreter. These are reserved for special purpose and must not be used as normal identifier names.

Python programming language contains the following keywords :

KEYWORD

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

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	

2.3.2 Identifiers (Names)

Identifiers are fundamental building blocks of a program and are used as the general terminology for the names given to different parts of the program *viz.* variables, objects, classes, functions, lists, dictionaries etc.

Identifier forming rules of Python are being specified below :

- An identifier is an arbitrarily long sequence of letters and digits.
- The first character must be a letter; the underscore (_) counts as a letter.

- Upper and lower-case letters are different. All characters are significant.
- The digits 0 through 9 can be part of the identifier except for the first character.
- Identifiers are unlimited in length. *Case* is significant *i.e.*, Python is case sensitive as it treats upper and lower-case characters differently.
- An identifier must not be a keyword of Python.
- An identifier cannot contain any special character except for underscore (_).

The following are some *valid* identifiers :

```
Myfile      DATE9_7_77     Z2T0Z9
MYFILE      _DS            _HJI3_JK
_CHK        FILE13
```

NOTE

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

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 (.)

2.3.3 Literals / Values

Literals (often referred to as constant-Values) are data items that have a fixed value. Python allows several kinds of literals :

(*i*) String literals (*ii*) Numeric literals (*iii*) Boolean literals

(*iv*) Special Literal *None* (*v*) Literal Collections

2.3.3A String Literals

The text enclosed in quotes forms a string literal in Python. *For example*, *'a'*, *'abc'*, *"abc"* are all string literals in Python. Unlike many other languages, both single character enclosed in quotes such as *"a"* or *'x'* or multiple characters enclosed in quotes such as *"abc"* or *'xyz'* are treated as String literals.

As you can notice, one can form string literals by enclosing text in both forms of quotes – single quotes or double quotes. Following are some valid string literals in Python :

```
'Astha'          "Rizwan"
'Hello World'    "Amy's"
"129045"
'1-x-0-w-25'
"112FBD291"
```

NOTE

In Python, one can form string literals by enclosing text in both forms of quotes – single quotes or double quotes.

Check Point 2.1

1. What is meant by token ? Name the tokens available in Python.
2. What are keywords ? Can keywords be used as identifiers ?
3. What is an identifier ? What are the identifier forming rules of Python ?
4. Is Python case sensitive ? What is meant by the term 'case sensitive' ?
5. Which of the following are valid identifiers and why/why not :

   Data_rec, _data, 1 data, data1, my.file, elif, switch, lambda, break ?

Python allows you to have certain *nongraphic-characters* in String values. ***Nongraphic characters*** are those characters that cannot be typed directly from keyboard *e.g.*, backspace, tabs, carriage return etc. (No character is typed when these keys are pressed, only some action takes place). These *nongraphic-characters* can be represented by using escape sequences. An escape sequence is represented by a backslash (\) followed by one or more characters.[1]

STRING LITERALS

A *string literal* is a sequence of characters surrounded by quotes (single or double or triple quotes).

Following table (Table 2.1) gives a listing of escape sequences.

Table 2.1 *Escape Sequences in Python*

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[1] database (Unicode only)		

In the above table, you see sequences representing \, ', ". Though these characters can be typed from the keyboard but when used without escape sequence, these carry a special meaning and have a special purpose, however, if these are to be typed *as it is*, then escape sequences should be used. (In Python, you can also directly type a double-quote inside a single-quoted string and vice-versa. *e.g.*, "anu's" is a valid string in Python)

NOTE

An *escape sequence* represents a single character and hence consumes one byte in ASCII representation.

String Types in Python

Python allows you to have *two* string types :

(*i*) Single-line Strings

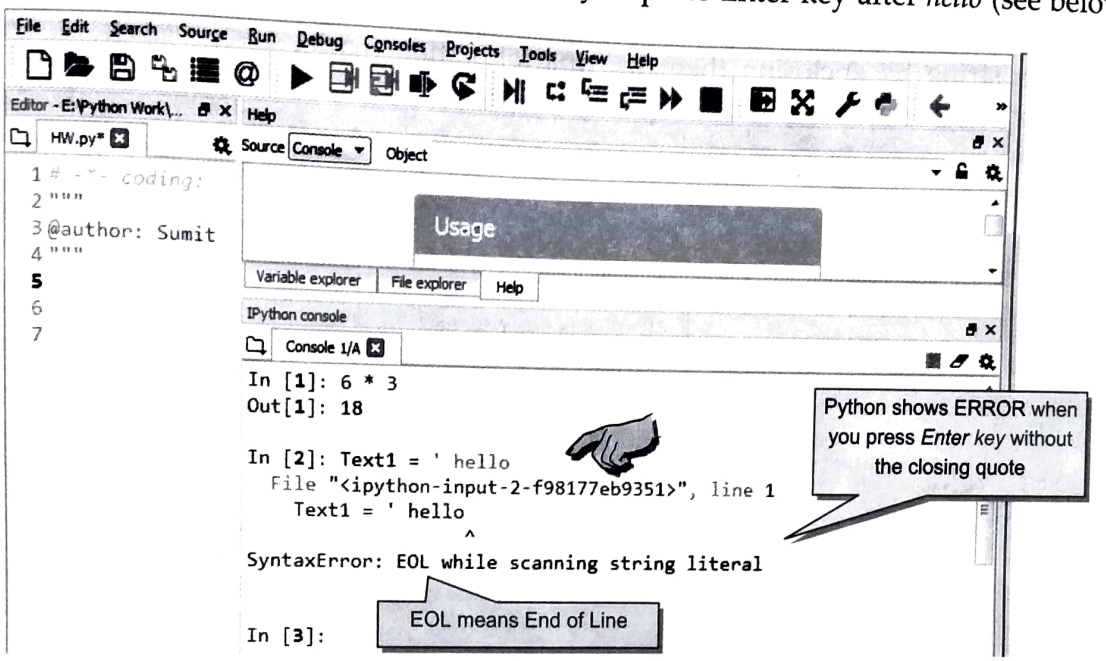
(*ii*) Multiline Strings

(*i*) ***Single-line Strings (Basic strings).*** The strings that you create by enclosing text in single quotes (' ') or double quotes (" ") are normally single-line strings, *i.e.*, they must terminate in one line. To understand this, try typing the following in IDLE window and see yourselves :

```
Text1= 'hello ↵
there'
```

1. Unicode and ASCII are two character encodings discussed later in Chapter 13 (Data Representation)

Python will show you an error the moment you press Enter key after *hello* (see below)



The reason for the above error is quite clear – Python by default creates single-line strings with both single or double quotes. So, if at the end of a line, there is no closing quotation mark for an opened quotation mark, Python shows an error.

(*ii*) ***Multiline Strings.*** Sometimes you need to store some text spread across multiple lines as one single string. For that Python offers multiline strings.

Multiline strings can be created in *two* ways :

(*a*) *By adding a backslash at the end of normal single-quote / double-quote strings.* In normal strings, just add a backslash in the end before pressing **Enter** to continue typing text on the next line. *For instance,*

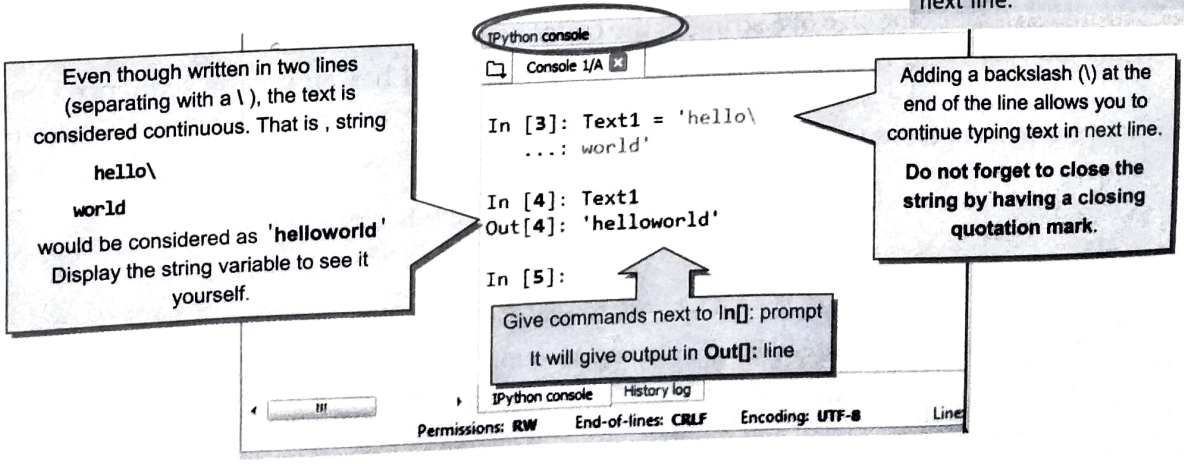
```
Text1 = 'hello\
world'
```

Do not indent when continuing typing in next line after '\'.

NOTE

A basic string must be completed on a single line, or continued with a backslash (\) as the very last character of a line if it is to be closed with a closing quote in next line.

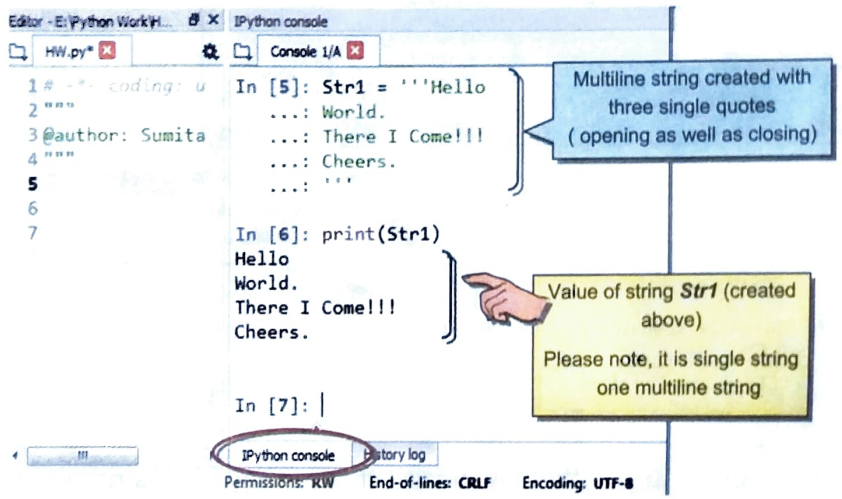
Following figure shows this :



(b) *By typing the text in triple quotation marks. (No backslash needed at the end of line).* Python allows to type multiline text string by enclosing them in triple quotation marks (both triple apostrophe or triple quotation marks will work).

NOTE

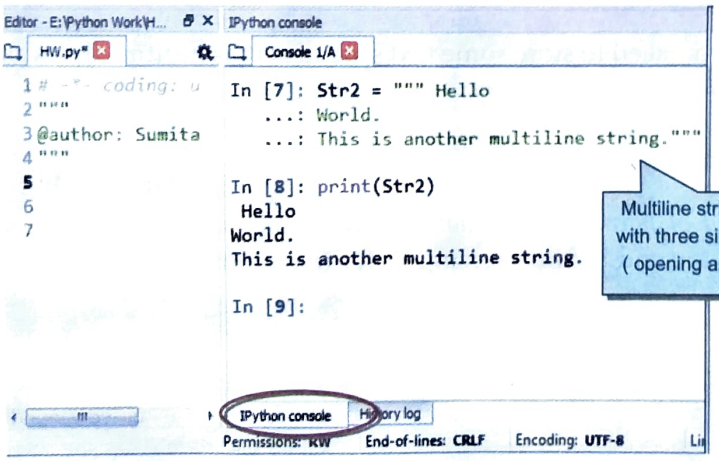
A multiline string continues on until the concluding triple-quote or triple-apostrophe.



For example,

```
Str1 = '''Hello
World.
There I Come !!!
Cheers.
'''
```

Or




```
Str2 = """Hello
World.
This is another multiline string."""
```

Size of Strings

Python determines the size of a string as the count of characters in the string. *For example,* size of string *"abc"* is 3 and of *'hello'* is 5. But if your string literal has an escape sequence contained within it, then make sure to count the escape sequence as one character. Consider some examples given below :

`'\\'`	size is 1	(\\ is an escape sequence to represent backslash)
`'abc'`	size is 3	
`"\ab"`	size is 2	(\a is an escape sequence, thus one character).
`"Seema\'s pen"`	size is 11	(For typing apostrophe (') sign, escape sequence \' has been used.).
`"Amy's"`	size is 4	Python allows a single quote (without escape sequence) in double-quoted string and vice-versa.

For multiline strings created with triple quotes, while calculating size, the EOL (end-of-line) character at the end of the line is also counted in the size. For example, if you have created a string Str3 as :

```
Str3 = '''a ↵
b ↵
c'''
```

These (enter keys) are considered as EOL (End-of-Line) characters and counted in the length of multiline string.

```
nstr = '''he\
ll\
o'''
```

But backslashes (\ ) at the end of intermediate lines are not counted in the size of the multiline strings.

then size of the string Str3 is 5 (three characters *a, b, c* and two *EOL* characters that follow characters *a* and *b* respectively).

For multiline strings created with single/double quotes and backslash character at end of the line, while calculating size, the backslashes are not counted in the size of the string ; also you cannot put EOLs using return key in single/double quoted multiline strings *e.g.*,

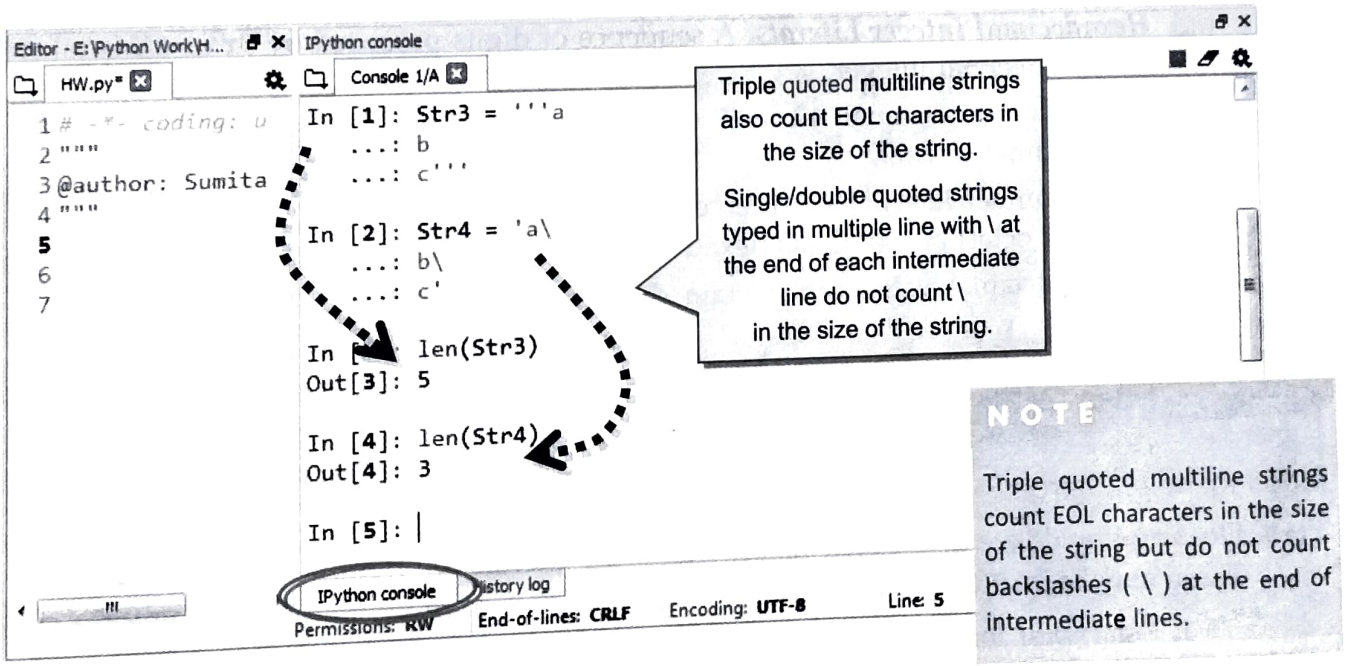
```
Str4 = 'a\
b\
c'
```

The size of string Str4 is 3 (only 3 characters, no backslash counted.)

To check the size of a string, you may also type len(<stringname>) command on the Python prompt in console window shell as shown in the following figure :

NOTE

Use len(<object name>) to get the size or length of an object.



NOTE

Triple quoted multiline strings count EOL characters in the size of the string but do not count backslashes (\ ) at the end of intermediate lines.

2.3.3B Numeric Literals

The numeric literals in Python can belong to any of the following *four* different numerical types :

int **(signed integers)**	often called just **integers** or **int**s, are positive or negative whole numbers with no decimal point.
float (floating point real values)	**floats** represent real numbers and are written with a decimal point dividing the integer and fractional parts.
complex (complex numbers)	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.

Let us talk about these literal types one by one.

Integer Literals

Integer literals are whole numbers without any fractional part. The method of writing integer constants has been specified in the following rule :

An integer constant must have at least one digit and must not contain any decimal point. It may contain either (+) or (–) sign. A number with no sign is assumed to be positive. Commas cannot appear in an integer constant.

Python allows *three* types of integer literals :

(*i*) ***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.

(*ii*) ***Octal Integer Literals.*** A sequence of digits starting with 0o (digit zero followed by letter o) is taken to be an octal integer.

For instance, decimal integer 8 will be written as 0o10 as octal integer. ($8_{10} = 10_8$) and decimal integer 12 will be written as 0o14 as octal integer ($12_{10} = 14_8$).

An octal value can contain only digits 0-7 ; 8 and 9 are invalid digits in an octal number i.e., 0o28, 0o19, 0o987 etc., are examples of invalid octal numbers as they contain digits 8 and 9 in them.

(*iii*) ***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), 0o14 (as octal) and 0XC (as hexadecimal).

A hexadecimal value can contain digits 0-9 and letters A-F only *i.e., 0XBK9, oxPQR, 0x19AZ* etc., are examples of invalid hexadecimal numbers as they contain invalid letters, *i.e.,* letters other than A-F.

NOTE

Many programming languages such as C, C++, and even Python 2.x too have two types for integers : **int** (for small integers) and **long** (for big integers). But in **Python 3.x**, there is only one integer type <class 'int'> that works like long integers and can support all small and big integers.

Floating Point Literals

Floating literals are also called real literals. Real literals are numbers having fractional parts. These may be written in one of the *two* forms called ***Fractional Form*** or the **Exponent Form.**

1. *Fractional form.* A real literal in Fractional Form consists of signed or unsigned digits including a decimal point between digits.

The rule for writing a real literal in fractional form is :

A real constant in fractional form must have at least one digit with the decimal point, either before or after. It may also have either + or – sign preceding it. A real constant with no sign is assumed to be positive.

The following are valid real literals in fractional form :

2.0, 17.5, – 13.0, – 0.00625, .3 (*will represent 0.3*), 7. (*will represent 7.0*)

The following are invalid real literals :

7	(*No decimal point*)
+17 / 2	(*/-illegal symbol*)
17,250.26.2	(*Two decimal points*)
17,250.262	(*comma not allowed*)

2. *Exponent form.* A real literal in *Exponent form* consists of *two* parts : ***mantissa*** and ***exponent***. For instance, 5.8 can be written as $0.58 \times 10^1 = 0.58\ E01$, where *mantissa* part is **0.58** (the part appearing before E) and *exponent* part is **1** (the part appearing after E). E01 represents 10^1 . The rule for writing a real literal in exponent form is :

> *A real constant in exponent form has two parts : a **mantissa** and an **exponent**. The mantissa must be either an integer or a proper real constant. The mantissa is followed by a letter E or e and the exponent. The exponent must be an integer.*

The following are the valid real literals in exponent form : 152E05, 1.52E07, 0.152E08, 152.0E08, 152E+8, 1520E04, –0.172E–3, 172.E3, .25E–4, *3.E3* (equivalent to 3.0E3)

(Even if there is no preceding or following digit of a decimal point, Python 3.x will consider it right)

The following are invalid real literals in exponent form :

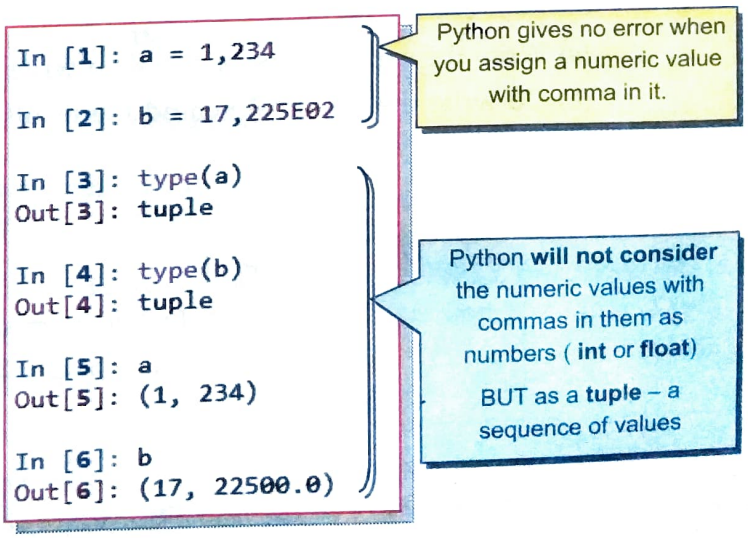
1.7E	(*No digit specified for exponent*)
0.17E2.3	(*Exponent cannot have fractional part*)
17,225E02	(*No comma allowed*) [Do read following discussion after it.]

Numeric values with commas are not considered **int** or **float** value, rather Python treats them as a tuple. A tuples is a special type in Python that stores a *sequence of values*. (You will learn about tuples in coming chapters – for now just understand a tuple as *a sequence of values only*.)

The last invalid example value given above (17,225e02) asks for a special mention here.

Any numeric value with a comma in its mantissa will not be considered a legal *floating point number*, **BUT if you assign this value, Python won't give you an error**. The reason being is that Python will not consider that as a *floating point value* rather a **tuple**. Carefully have a look at the adjacent figure that illustrates it.

We are not talking about Complex numbers here. These would be discussed later when the need arises.



2.3.3C 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*.

NOTE

True and **False** are the only two Boolean literal values in Python. **None** literal represents absence of a value.

2.3.3D Special Literal None

Python has one special literal, which is **None**. The **None** literal is used to indicate absence of value. It is also used to indicate the end of lists in Python.

The **None** value in Python means *"There is no useful information"* or *"There's nothing here."* Python doesn't display anything when asked to display the value of a variable containing value as **None**. Printing with print statement, on the other hand, shows that the variable contains **None** (see figure here).

```
In [13]: Value1 = 10

In [14]: Value2 = None

In [15]: Value1
Out[15]: 10

In [16]: Value2

In [17]: print(Value2)
None

In [18]:
```

Displaying a variable containing **None** does not show anything. However, with ***print()***, it shows the value contained as **None**.

Python supports literal collections also such as tuples and lists etc. But covering these here would make the discussion too complex for the beginning. So, we'll take them at a later time.

NOTE

Boolean literals **True**, **False** and special literal None are some built-in constants/literals of Python.

PriP BASICS ABOUT TOKENS — Progress In Python 2.1

Start ***Spyder IDE*** *through* ***Anaconda Navigator or any other IDE of your choice.***

1. *In front of the Python prompt* ***In[ ]:*** *in* ***IPythonConsole****, type the following statements one by one, in the same order.*

   (*a*) Write the expected result and then write the actual result that Python returned.

   (*b*) Do write the reason(s) behind the result returned by Python.

Solved Sample

Statement to be typed	*Expected result*	*Actual result*	*Reason*
abc123 =25	nothing	nothing	Python internally assigns the value to abc123 but shows nothing.
abc123	25	25	Value of abc123 is displayed on screen.
:	:	:	:

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.1 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

Check Point 2.2

1. What are literals ? How many types of literals are available in Python ?
2. How many types of integer literals are allowed in Python ? How are they written ?
3. Why are characters \, ', " and tab typed using escape sequences ?
4. Which escape sequences represent the newline character and backspace character ? An escape sequence represents how many characters ?
5. What are string-literals in Python ? How many ways, can you create String literals in Python ? Are there any differences in them ?
6. What is meant by a floating-point literal in Python ? How many ways can a floating literal be represented into ?
7. Write the following real constants into exponent form :

   23.197, 7.214, 0.00005, 0.319
8. Write the following real constants into fractional form :

   0.13E04, 0.417E-04, 0.4E-5,

   .12E02, 12.E02
9. What are the two Boolean literals in Python ?
10. Name some built-in literals of Python.
11. Out of the following literals, determine their type whether decimal / octal / hexadecimal integer literal or a floating point literal in fractional or exponent form or string literal or other ?

    123, 0o124, 0xABC, 'abc', "ABC",

    12.34, 0.3E-01, '"ftghjkjl"',

    None, True, False
12. What kind of program elements are the following ?

    'a', 4.38925, "a", "main" ?
13. What will **var1** and **var2** store with statements : **var1** = 2,121E2 and **var2** = 0.2,121E2 ? What are the types of values stored in **var1** and **var2** ?

2.3.4 Operators

Operators are tokens that trigger some computation when applied to variables and other objects in an expression. Variables and objects to which the computation is applied, are called **operands**. So, an operator requires some *operands* to work upon.

The following list gives a brief description of the operators and their functions / operators, in details, will be covered in next chapter – *Data Handling*.

OPERATORS

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

Unary Operators

Unary operators are those operators that require one operand to operate upon. Following are some unary operators :

+	Unary plus
–	Unary minus
~	Bitwise complement
not	logical negation

Binary Operators

Binary operators are those operators that require two operands to operate upon. Following are some binary operators :

Arithmetic operators

+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Remainder/ Modulus
**	exponent (raise to power)
//	Floor division

Bitwise operators

&	Bitwise AND
^	Bitwise exclusive OR (XOR)
|	Bitwise OR

Shift operators

<<	shift left
>>	shift right

Identity operators

is	is the identity same ?
is not	is the identity not same ?

Relational operators

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
==	Equal to
!=	Not equal to

Assignment operators

=	Assignment
/=	Assign quotient
+=	Assign sum
*=	Assign product
%=	Assign remainder
-=	Assign difference
**=	Assign Exponent
//=	Assign Floor division

Logical operators

and	Logical AND
or	Logical OR

Membership operators

in	whether variable in sequence
not in	whether variable not in sequence

More about these operators you will learn in the due course. Giving descriptions and examples is not feasible and possible right here at the moment.

2.3.5 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 :

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

PUNCTUATORS

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

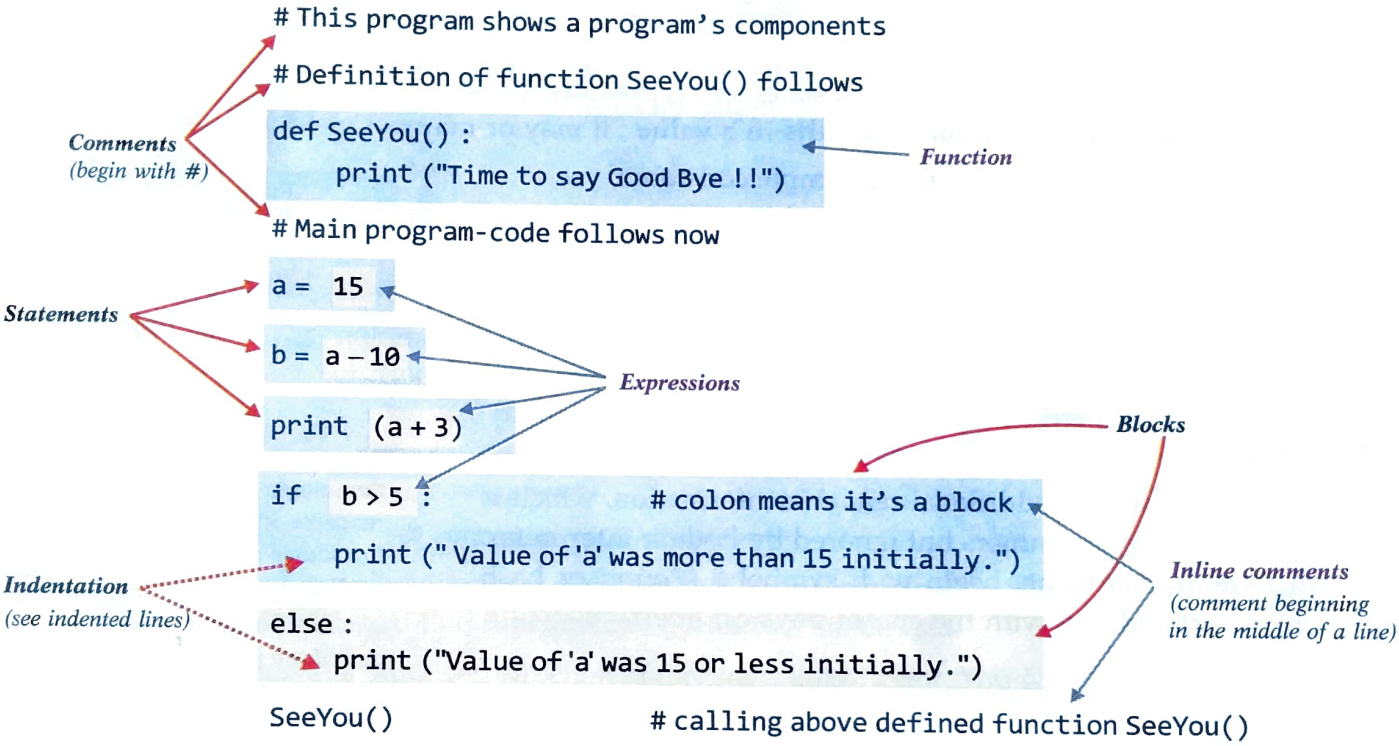
The usage of these punctuators will be discussed when the need arises along with normal topic discussions.

LET US REVISE

- *A token is the smallest individual unit in a program.*
- *Python provides following tokens : keywords, identifiers (names), Values (literals), punctuators, operators and comments.*
- *A keyword is a reserved word carrying special meaning and purpose.*
- *Identifiers are the user-defined names for different parts of the program.*
- *In Python, an identifier may contain* ***letters*** *(a-z, A-Z),* ***digits*** *(0-9) and a symbol underscore (_). However, an identifier must begin with a letter or underscore ; all letters/digits in an identifier are significant.*
- *Literals are the fixed values.*
- *Python allows following literals : string literal, numeric (integer, floating-point literals, Boolean literals, special literal* ***None*** *and literal collections).*
- *Operators are tokens that trigger some computation / action when applied to variables and other objects in an expression.*
- *Punctuators are symbols used to organize programing- sentence structures and indicate the rhythm and emphasis of expressions, statements and program-structure.*

2.4 BAREBONES OF A PYTHON PROGRAM

Let us take our discussion further. Now we are going to talk about the basic structure of a Python program – what all it can contain. Before we proceed, have a look at following sample code. Look at the code and then proceed to the discussion that follows. Don't worry if the things are not clear to you right now. They'll become clear when the discussion proceeds.




```
# This program shows a program's components
# Definition of function SeeYou() follows
def SeeYou() :
    print ("Time to say Good Bye !!")
# Main program-code follows now
a = 15
b = a - 10
print (a + 3)
if b > 5 :                    # colon means it's a block
    print ("Value of 'a' was more than 15 initially.")
else :
    print ("Value of 'a' was 15 or less initially.")
SeeYou()                      # calling above defined function SeeYou()
```

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

- expressions
- statements
- comments
- function
- blocks and indentation

Let us now discuss various components shown in above sample code.

(i) Expressions

An ***expression*** is any legal combination of symbols that ***represents a value.*** An expression represents something, which **Python evaluates** and which then produces a *value*.

Some examples of expressions are

```
15
2.9
```
} *expressions that are values only*

```
a + 5
(3 + 5) / 4
```
} *complex expressions that produce a value when evaluated.*

EXPRESSIONS

An ***expression*** is any legal combination of symbols that ***represents a value.***

Now from the above sample code, can you pick out all expressions ?

These are : 15, $a+10$, $a+3$, $b>5$

(ii) Statement

While an expression represents something, a statement is a programming instruction that does something *i.e.*, some action takes place.

Following are some examples of statements :

```
print ("Hello")   # this statement calls print function
if b > 5 :
    :
```

STATEMENT

A statement is a programming instruction that does something *i.e.*, some action takes place.

While an expression is evaluated, a statement is executed *i.e.*, some action takes place. And it is **not necessary that a statement results in a value** ; it may or may not yield a value.

Some statements from the above sample code are :

```
a = 15
b = a - 10
print (a + 3)
if b < 5 :
    :
```

NOTE

A statement executes and may or may not yield a value.

(iii) Comments

Comments are the additional readable information, which is read by the programmers but ignored by Python interpreter. In Python, comments begin with symbol # (Pound or hash character) and end with the end of physical line.

COMMENTS

Comments are the additional readable information to clarify the source code.

Comments in Python begin with symbol # and generally end with end of the physical line.

In the above code, you can see *four* comments :

(*i*) The physical lines beginning with # are the **full line comments**. There are *three* full line comments in the above program are :

```
# This program shows a program's components
# Definition of function SeeYou( ) follows
# Main program code follows now
```

(*ii*) The fourth comment is an ***inline comment*** as it starts in the middle of a physical line, after Python code (see below)

```
if b < 5 : # colon means it requires a block
```

NOTE

A *Physical line* is the one complete line that you see on a computer whereas a *logical line* is the one that Python sees as one full statement.

Multi-line Comments

What if you want to enter a ***multi-line comment*** or a ***block comment*** ?. You can enter a multi-line comment in Python code in *two* ways :

(*i*) Add a # symbol in the beginning of every physical line part of the multi-line comments, *e.g.*,

```
# Multi-line comments are useful for detailed additional information.
# Related to the program in question.
# It helps clarify certain important things.
```

(*ii*) Type comment as a triple-quoted multi-line string *e.g.*,

```
'''  Multi-line comments are useful for detailed additional
     information related to the program in question.
     It helps clarify certain important things
'''
```

This type of multi-line comment is also known as ***docstring***. You can either use triple-apostrophe (''') or triple quotes (""") to write *docstrings*. The docstrings are very useful in documentation – and you'll learn about their usage later.

NOTE

Comments enclosed in triple quotes (" " ") or triple apostrophe (' ' ') are called **docstrings**.

') Functions

A function is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed.

In the above sample program, there is one function namely **SeeYou( )**. The statements indented below its **def** statement are part of the function. [All statements indented at the same level below *def SeeYou( )* are part of *SeeYou( )*.] This function is executed in main code through following statement (Refer to sample program code given above)

FUNCTIONS

A **function** is a code that has a name and it can be reused (executed again) by specifying its name in the program, where needed .

```
SeeYou()      # function-call statement
```

Calling of a function becomes a statement *e.g.*, **print** is a function but when you call **print( )** to print something, then that function call becomes a statement.

For now, only this much introduction of functions is sufficient. You will learn about functions in details in Class 12.

(v) Blocks and Indentation

Sometimes a group of statements is part of another statement or function. Such a group of one or more statements is called **block** or **code-block** or **suite**. *For example,*

our spaces together mark e next indent-level

```
if b < 5 :
    print ("Value of 'b' is less than 5.")
    print ("Thank you.")
```

This is a block with all its statements at same indentation level.

Many languages such as C, C++, Java etc., use symbols like curly brackets to show blocks but Python does not use any symbol for it, rather it uses indentation.

Consider the following example :

BLOCK OR CODE-BLOCK OR SUITE

A group of statements which are part of another statement or a function are called *block* or *code-block* or *suite* in Python.

```
if b < a :
    tmp = a
    a = b
    b = tmp
print ("Thank you")
```

*This is a block, part of **if** statement. Notice, all statements in same block have same indentation level.*

This statement is not part of if's block as it is at different indentation level.

A group of individual statements which make a single *code-block* is also called a **suite** in Python. Consider some more examples showing indentation to create blocks :

Two different indentation-levels inside this code.

```
def check() :
    c = a + b
    if c < 50 :
        print ('Less than 50')
        b = b * 2
        a = a + 10
    else :
        print ('>= 50')
        a = a * 2
        b = b + 10
```

Block inside function check( )

Block / suite inside if statement

Block / suite inside else statement

NOTE

Python uses indentation to create blocks of code. Statements at same indentation level are part of same block/suite.

Statements requiring suite/code-block have a colon (:) at their end.

You cannot unnecessarily indent a statement ; Python will raise error for that.

Python Style Rules and Conventions

While working in Python, one should keep in mind certain style rules and conventions. In the following lines, we are giving some very elementary and basic style rules :

Statement Termination Python does not use any symbol to terminate a statement. When you end a physical code-line by pressing Enter key, the statement is considered terminated by default.

Maximum Line Length Line length should be maximum 79 characters.

Lines and Indentation Blocks of code are denoted by line indentation, which is enforced through 4 spaces (not tabs) per indentation level.

Blank Lines Use *two* blank lines between top-level definitions, *one* blank line between method/function definitions. Functions and methods should be separated with *two* blank lines and Class definitions with *three* blank lines.

Avoid multiple statements on one line Although you can combine more than one statements in one line using symbol semicolon (;) between two statements, but it is not recommended.

Whitespace You should always have whitespace around operators and after punctuation but not with parentheses. Python considers these 6 characters as whitespace : ' ' (space), '\n' (newline), '\t' (horizontal tab), '\v' (vertical tab), '\f' (formfeed) and '\r' (carriage return)

Case Sensitive Python is case sensitive, so case of statements is very important. Be careful while typing code and identifier-names.

Docstring Convention Conventionally triple double quotes (""") are used for docstrings.

Identifier Naming You may use underscores to separate words in an identifier *e.g.*, **loan_amount** or use *CamelCase* by capitalizing first letter of the each word *e.g.*, LoanAmount or loanAmount

Check Point 2.3

1. What is an expression in Python ?
2. What is a statement in Python ? How is a statement different from expression ?
3. What is a comment ? In how many ways can you create comments in Python ?
4. How would you create multi-line comment in Python ?
5. What is the difference between full-line comment and inline comment ?
6. What is a block or suite in Python ? How is indentation related to it ?

PriP COMPONENTS OF A PROGRAM — Progress In Python 2.2

⋮

This is another program with different components

⋮

```
def First5Multiples( ) :
    ⋮
# main code
    ⋮
```

Fill the appropriate components of program from the above code.

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.2 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

2.5 VARIABLES AND ASSIGNMENTS

A variable in Python represents named location that refers to a value and whose values can be used and processed during program run. For instance, to store name of a student and marks of a student during a program run, we require some labels to refer to these *marks* so that these can be distinguished easily. *Variables,* called as *symbolic variables,* serve the purpose. The variables are called symbolic variables because these are named labels. *For instance,* the following statement creates a variable namely ***marks*** of **Numeric** type :

```
marks = 70
```

VARIABLES

Named labels, whose values can be used and processed during program run, are called *Variables.*

2.5.1 Creating a Variable

Recall the statement we used just now to create the variable **marks** :

```
marks = 70
```

As you can see, that we just assigned the value of numeric type to an identifier name and Python created the variable of the type similar to the type of value assigned. In short, after the above statement, we can say that ***marks*** is a *numeric* variable.

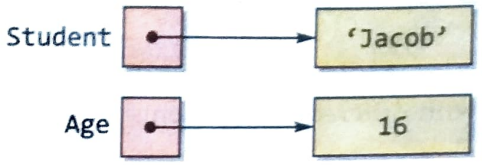
So, creating variables was just that simple only ? Yes, you are right. In Python, to create a variable, just assign to its name the value of appropriate type. *For example,* to create a variable namely ***Student*** to hold student's name and variable ***age*** to hold student's age, you just need to write somewhat similar to what is shown below :

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

NOTE

Python variables are created by assigning value of desired type to them, *e.g.*, to create a numeric variable, assign a numeric value to *variable_name* ; to create a string variable , assign a string value to *variable_name*, and so on.

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



Isn't it simple?? ☺ Okay, let's now create some more variables.

```
TrainNo = 'T#1234'          # variable created of String type
balance = 23456.75          # variable created of Numeric(floating point) type
rollNo = 105                # variable create of Numeric(integer) type
```

Same way, you can create as many variables as you need for your program.

IMPORTANT – Variables are Not Storage Containers in Python

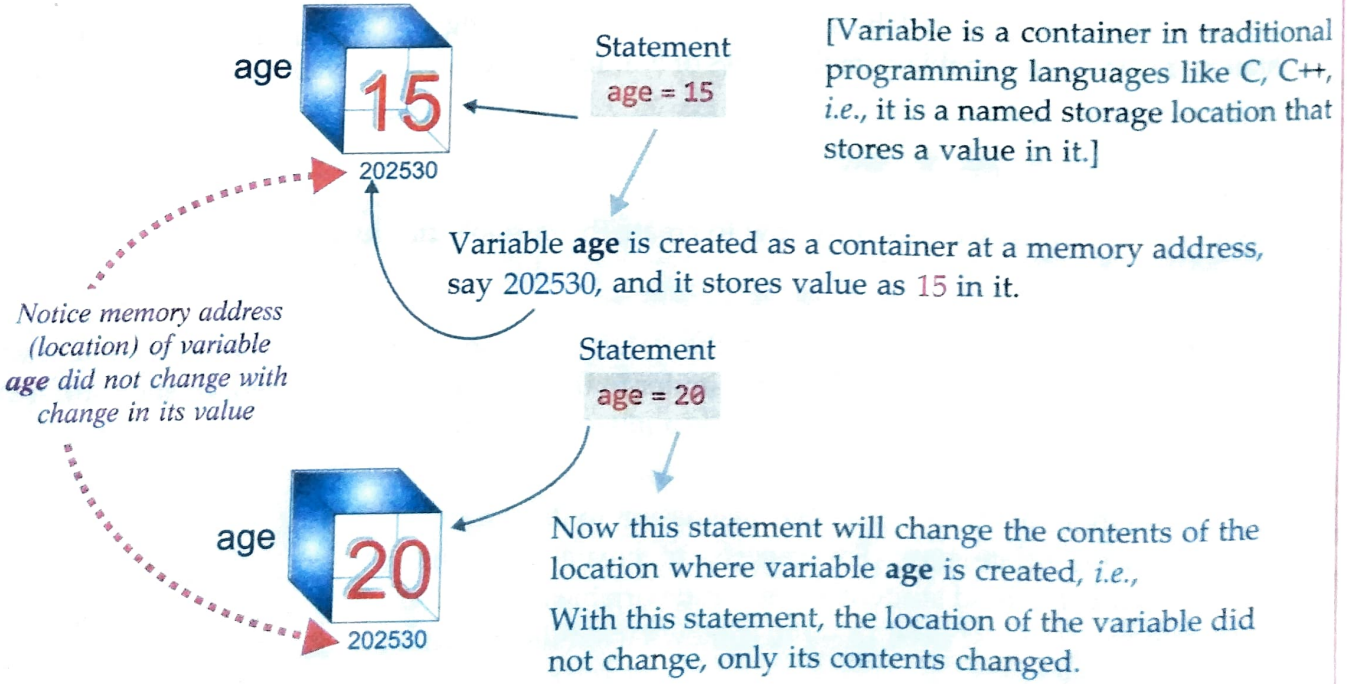
If you have an earlier exposure to programming, you must be having an idea of variables. BUT PYTHON VARIABLES ARE NOT CREATED IN THE FORM MOST OTHER PROGRAMMING LANGUAGES DO. Most programming languages create variables as storage containers *e.g.*,

Consider this :

```
age = 15
age = 20
```

Firstly value 15 is assigned to variable **age** and then value 20 is assigned to it.

Traditional Programming Languages' Variables



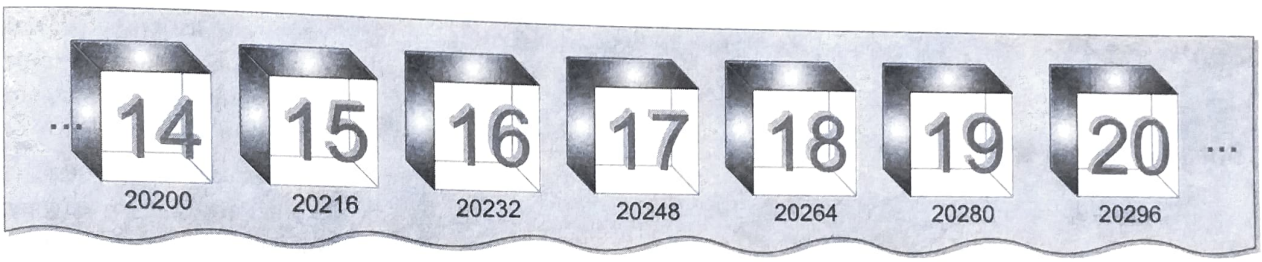
This is how traditionally variables were created in programming languages like C, C++, Java etc.

Python's Handling of Variables

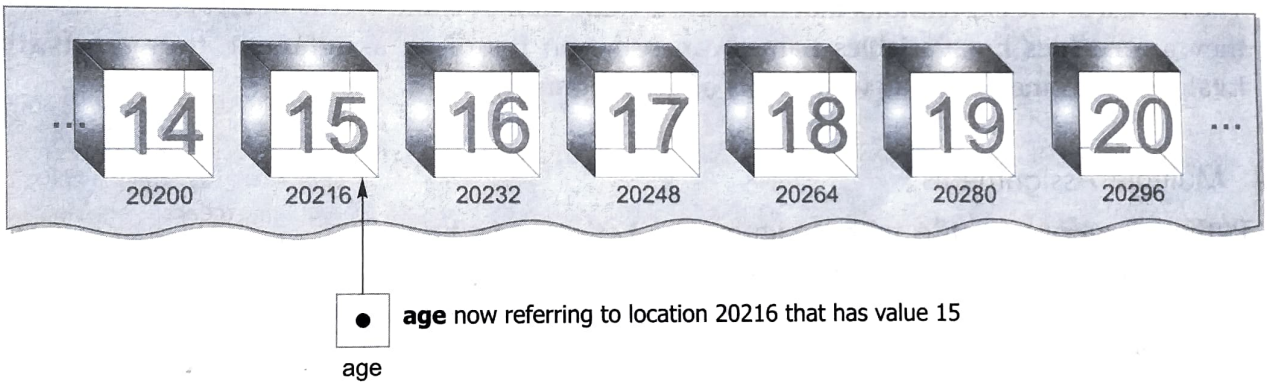
BUT PYTHON DOES THIS DIFFERENTLY

Let us see how Python will do it.

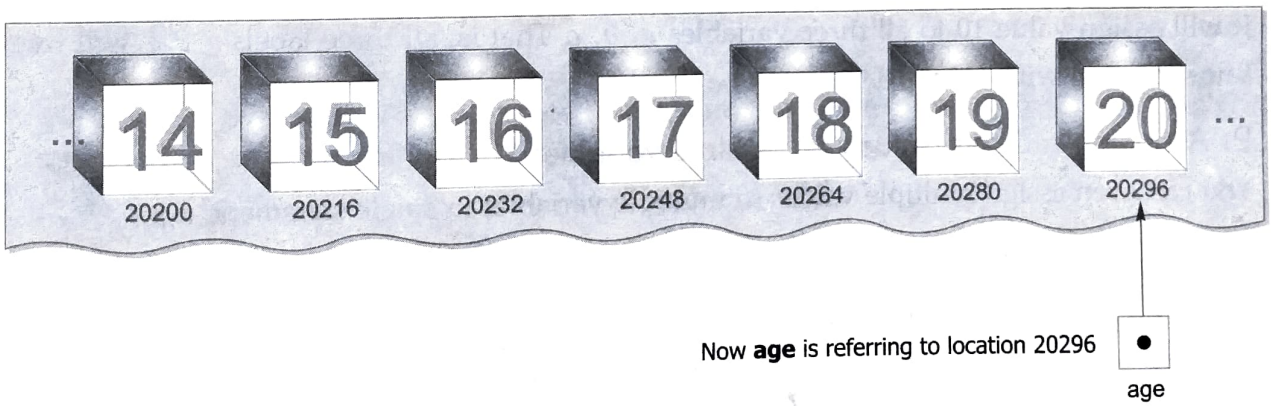
Memory has literals/values at defined memory locations, and each memory location has a memory address.



When you give statement **age = 15**, variable **age** will be created as a label pointing to memory location where value **15** is stored *i.e.*, as.



And when you give statement **age = 20**, the label **age** will not be having the same location as earlier. It will now refer to value 20, which is at different location *i.e.*,



So this time memory location of variable **age's** value is changed.

Thus variables in Python do not have fixed locations unlike other programming languages. The location they refer to changes everytime their values change (**This rule is not for all types of variables, though**). It will become clear to you in the next chapter, where we talk about *Mutable and Immutable types*.

Lvalues and Rvalues

Broadly *lvalue* and *rvalue* can be thought of as :

lvalue : expressions that can come on the **lhs** (left hand side) of an assignment.

rvalue : expressions that can come on the **rhs** (right hand side) of an assignment.

e.g., you can say that

```
a = 20
b = 10
```

But you cannot say

```
20 = a
```
or `10 = b`
or `a * 2 = b`

LVALUES AND RVALUES

Lvalues are the objects to which you can assign a value or expression. Lvalues can come on lhs or rhs of an assignment statement.

Rvalues are the literals and expressions that are assigned to lvalues. Rvalues can come on rhs of an assignment statement.

The literals or the expressions that evaluate a value cannot come on **lhs** of an assignment hence they are **rvalues** but variables names can come on **lhs** of an assignment, they are **lvalues**. Lvalues can come on lhs as well as rhs of an assignment.

NOTE

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

2.5.2 Multiple Assignments

Python is very versatile with assignments. Let's see in how many different ways, you can use assignments in Python :

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***. That is, all three labels a, b, c will refer to same location with value 10.

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.

This style of assigning values is very useful and compact. *For example*, consider the code given below :

```
x, y = 25, 50
print (x, y)
```

It will print result as

```
25  50
```

Because *x* is having value 25 and *y* is having 50. Now, if you want to swap values of *x* and *y*, you just need to write :

```
x, y = y, x
print (x,y)
```

Now the result will be

```
50   25
```

Because this time the values have been swapped and *x* is having value *25* and *y* is having *50*.

While assigning values through multiple assignments, please remember that Python first evaluates the RHS (right hand side) expression(s) and then assigns them to LHS, *e.g.*,

```
a, b, c = 5, 10, 7                    # statement1
b, c, a = a + 1, b + 2, c - 1         # statement2
print (a, b, c)
```

- Statement1 assigns 5, 10 and 7 to *a*, *b* and *c* respectively.
- Statement2 will first evaluate RHS *i.e.*, $a+1$, $b+2$, $c-1$ which will yield

```
5 + 1, 10 + 2, 7 - 1 = 6, 12, 6
```

Then it will make the statement (by replacing the evaluated result of RHS) as :

```
b, c, a = 6, 12, 6
```

Thus, $b = 6$, $c = 12$ and $a = 6$

- The third statement **print (a, b, c)** will print

```
6  6  12
```

Isn't this easy ? Now can you guess the output of following code fragment[2]

```
p, q = 3, 5
q, r = p - 2, p + 2
print (p, q, r)
```

Please note the expressions separated with commas are **evaluated from left to right and assigned in same order** *e.g.*,

```
x = 10
y, y = x + 2, x + 5
```

will evaluate to following (after evaluating expressions on rhs of = operator)

```
y, y = 12, 15
```

i.e., firstly it will assign *first RHS value* to *first LHS variable i.e.*,

```
y = 12
```

then it will assign second RHS value to second LHS variable *i.e.*,

```
y = 15
```

So if you print *y* after this statement *y* will contain 15.

Now, consider following code and guess the output :

```
x, x = 20, 30
y, y = x + 10, x + 20
print (x, y)
```

Well, it will print the output as

```
30  50
```

Now you know why ?☺

2.5.3 Variable Definition

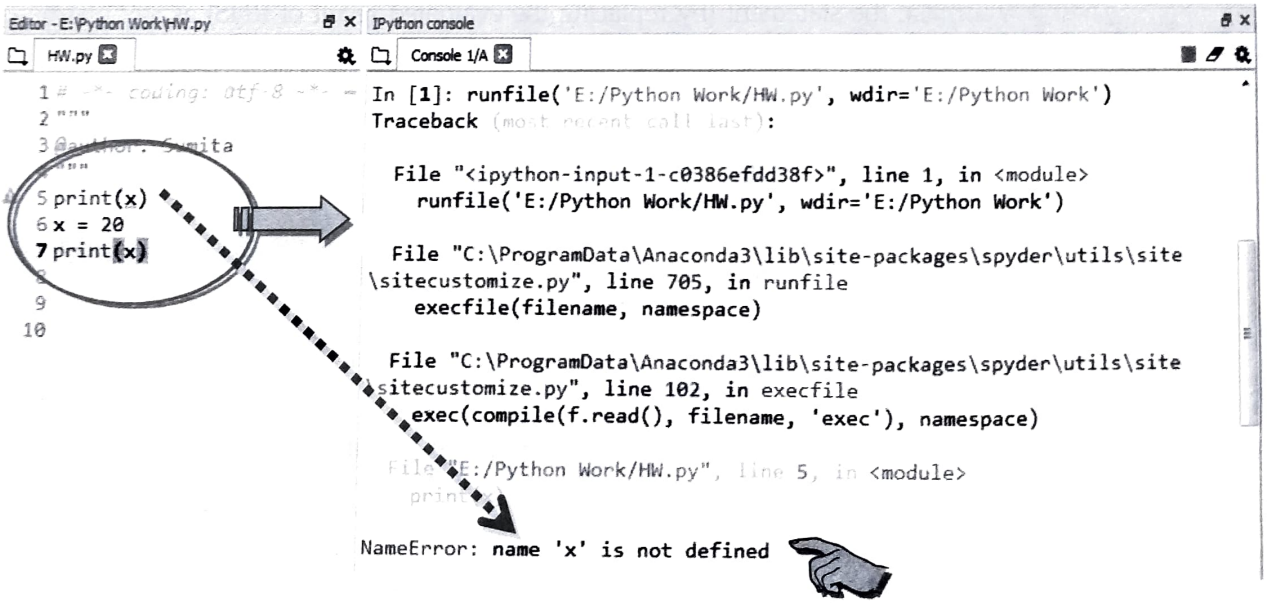
So, you see that in Python, a variable is ***created*** when you first assign a value to it.

It also means that **a variable is not created until some value is assigned to it.**

To understand it, consider the following code fragment. Try running it in script mode :

```
print (x)
x = 20
print (x)
```

when you run the above code, it will produce an error for the first statement (line 1) only – **name '*x*' not defined** (see figure below)



The reason for above error is implicit. As you know that a *variable is not created until some value is assigned to it.* So, variable *x* is not created and yet being printed in **line 1**. Printing/using an uncreated (undefined) variable results into error.

So, to correct the above code, you need to first assign something to *x* before using it in a statement, somewhat like :

```
x = 0                  # variable x created now
print (x)
x = 20
print (x)
```

Now the above code will execute without any error.

IMPORTANT

A variable is defined only when you assign some value to it. Using an undefined variable in an expression/statement causes an error called **Name Error**.

2.5.4 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 a Python variable as ***labels*** associated with objects (literal values in our case here) ; with dynamic typing, Python makes the label refer to new value (Fig. 2.1). Following figure illustrates it.

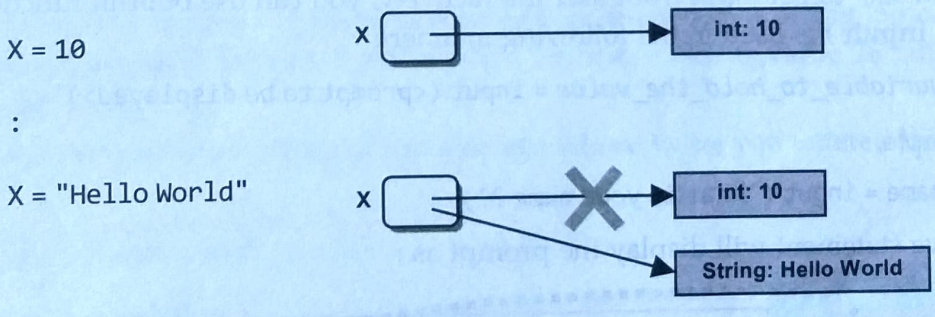


Figure 2.1 Dynamic typing in Python variables.

As you can see in Fig. 2.1, variable ***X*** is first pointing to/referring to an integer value **10** and then to a string value "**Hello world**".

Please note here that variable ***X*** does not have a type but the value it points to does have a type. So you can make a variable point to a value of different type by reassigning a value of that type ; Python will not raise any error. This is called **Dynamic Typing** feature of Python.

Caution with Dynamic Typing

Although Python is comfortable with changing types of a variable, the programmer is responsible for ensuring right types for certain type of operations. *For example,*

```
X = 10
Y = 0
Y = X/2
X = 'Day'
Y = X/2
```

legal, because two integers can be used in divide operation

Python is comfortable with dynamic typing

ERROR!! a string cannot be divided.

Features of print statement

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 (as it did in example statement 2 above)

  IMPORTANT : Wtih print( ), the objects/items that you give, must be convertible to string type.

  Covering it here with examples may derail our discussion so we have given examples of using *print( )* with different types of arguments in **Appendix B**. Please do refer to Appendix B.

- it inserts spaces between items automatically because the default value of ***sep*** argument is space(' '). The ***sep*** argument specifies the separator character. The **print( )** automatically adds the ***sep*** character between the items/objects being printed in a line. If you do not give any value for ***sep***, then by default the **print( )** will add a space in between the items when printing. 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.

So, a *print( )* statement appended a newline at the end of objects it printed, *i.e.*, in above code :

```
My name is Amit. ↵ or \n
I am 16 years old.
```

*Python automatically added a newline character in the end of a line printed so that the next **print( )** prints from the next line*

The **print( )** works this way only when you have not specified any ***end*** argument with it because by default **print( )** takes value for **end** argument as **'\n'** – the ***newline character***[4].

4. A newline is a character used to represent the end of a line of text and the beginning of a new 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, *e.g.*, the code

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

will print output as :

*This time the **print( )** ended the line with given **end** character, which is'$' here*

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

So the ***end*** argument determines the ***end*** character that will be printed at the end of print line.

Code fragment 1

```
a, b = 20, 30
print ("a =", a, end = ' ')
print ("b =", b)
```

*Notice first print statement has **end** set as a space*

Now the output produced will be like :

```
a = 20 b = 30
```

This space is because of end = ' ' in print( )

NOTE

In print( ) function, the default value of ***end*** argument is newline character ('\n') and of ***sep*** argument, it is space (' ').

The reason for above output is quite clear. Since there is **end** character given as a space (*i.e.*, **end = ' '**) in first print statement, the newline ('\n') character is not appended at the end of output generated by first print statement. Thus the output-position-cursor stays on the same line. Hence the output of second print statement appears in the same line. Now can you predict the output of following code fragment ?

```
Name = 'Enthusiast'
print ("Hello", end = ' ')
print (Name)
print ("How do you find Python ?")
```

Well, you guessed it right. ☺ It is :

```
Hello Enthusiast
How do you find Python ?
```

In Python you can **break any statement by putting a \ is the end and pressing Enter key, then completing the statement in next line.** For example, following statement is perfectly right.

```
print ("Hello", \
  end = ' ')
```

The backslash at the end means that the statement is still continuing in next line

Now consider following sample programs.

Check Point 2.4

1. What is a variable ?
2. Why is a variable called symbolic variable ?
3. Create variables for the following
   (*i*) to hold a train number
   (*ii*) to hold the name of a subject
   (*iii*) to hold balance amount in a bank account
   (*iv*) to hold a phone number
4. What do you mean by dynamic typing of a variable ? What is the caution you must take care of ?
5. What happens when you try to access the value of an undefined variable ?
6. What is wrong with the following statement ?

   ```
   Number = input("Number")
   Sqr = Number*Number
   ```

7. Write Python code to obtain the balance amount.
8. Write code to obtain fee amount and then calculate fee hike as 10% of fees (*i.e.*, fees × 0.10).

Program 2.1 *Program to obtain three numbers and print their sum.*

```
# to input 3 numbers and print their sum
num1 = int( input("Enter number 1 : "))
num2 = int( input("Enter number 2 : "))
num3 = int( input("Enter number 3 : "))
Sum = num1 + num2 + num3
print("Three numbers are : ", num1, num2, num3)
print("Sum is : ", Sum)
```

The output produced by above program is as shown below :

```
Enter number 1 : 7
Enter number 2 : 3
Enter number 3 : 13
Three numbers are :  7 3 13
Sum is :  23
```

Program 2.2 *Program to obtain length and breadth of a rectangle and calculate its area.*

```
# to input length and breadth of a rectangle and calculate its area
length = float( input("Enter length of the rectangle : "))
breadth = float( input("Enter breadth of the rectangle : "))

area = length * breadth

print ("Rectangle specifications ")
print ("Length = ", length, end = ' ')
print ("breadth = ", breadth)
print ("Area = ", area)
```

The output produced by above program is as shown below :

```
Enter length of the rectangle : 8.75
Enter breadth of the rectangle : 35.0

Rectangle specifications

Length =  8.75 Breadth =  35.0
Area =  306.25
```

Program 2.3 *Program to calculate BMI (Body Mass Index) of a person.*

Body Mass Index is a simple calculation using a person's height and weight.
The formula is **BMI = kg/m²** where *kg* is a person's weight in kilograms and m^2 is their height in *metres squared*.

```
# to calcu;ate BMI = kg /m square
weight_in_kg = float(input("Enter weight in kg : "))
```

```
height_in_meter = float(input("Enter height in meters : "))
bmi = weight_in_kg / ( height_in_meter * height_in_meter)
print("BMI is : ", bmi)
```

The output produced by above program is as shown below :

```
Enter weight in kg : 66
Enter height in meters : 1.6
BMI is :  25.781249999999996
```

VARIABLES, SIMPLE I/O

PriP — Progress In Python 2.3

*Start a **Python IDE of your choice***

1. *Click **File** → **New File...** and type the following code into it :*

```
myNumber = 10
print (myNumber + 1)          # output statement 1
print (myNumber)              # output statement 2
```

⋮

Please check the practical component-book – Progress in Computer Science with Python and fill it there in PriP 2.3 under Chapter 2 after practically doing it on the computer.

>>>❖<<<

LET US REVISE

- *A Python program can contain various components like expressions, statements, comments, functions, blocks and indentation.*
- *An expression is a legal combination of symbols that represents a value.*
- *A statement is a programing instruction.*
- *Comments are non-executable, additional information added in program for readability.*
- *In Python, comments begin with a # character.*
- *Comments can be single-line comment, multi-line comments and inline comments.*
- *Function is a named code that can be reused with a program.*
- *A block/suite/code-block is a group of statements that are part of another statement.*
- *Blocks are represented through indentation.*
- *A variable in Python is defined only when some value is assigned to it.*
- *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.*

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 a 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 0o (digit zero followed by letter o) is taken to be an octal integer. For instance, decimal integer 8 will be written as 0o10 as octal integer. ($8_{10} = 10_8$) and decimal integer 12 will be written as 0o14 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), 0o14 (as octal) and 0XC (as hexadecimal).

4. *What will be the sizes of following constants :*

```
'\a',  "\a",  "Reema\'s",  '\"',  "it's",  "XY \
YZ"  ,  """XY↵
   YZ"""  ?
```

Solution.

`'\a'` Size is 1 as there is 1 character (escape sequence) and it is a string literal enclosed in single quotes.

`"\a"` Size is 1 as there is 1 character enclosed in double quotes.

`"Reema\'s"` Size is 7 because it is a string having 7 characters enclosed in double quotes. (\' escape sequence for apostrophe and is considered a single character.)

`'\"'` Size is 1. It is a character constant and is containing just one character \".

`"it's"` Size is 4. Python allows single quote without escape sequence in a double quoted string and a double quote without escape sequence in a single quoted string *e.g.*, 'no."1tag'.

```
"xy\
yz"
```
Size is 4. It is a multi-line string create with \ in the basic string.

```
"""xy↵
  yz"""
```
Size is 5. Triple quoted multi-line string, EOL (↵) character is also counted in size.

5. *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*) *Multi-line Strings* Strings storing multiple lines of text.

6. *How can you create multi-line strings in Python ?*

Solution. Multi-line strings can be created in *two* ways :

(*a*) By adding a backslash at the end of normal single-quote or double-quote strings *e.g.*,

```
Text = "Welcome \
       To\
       Python"
```

(*b*) By typing the text in triple quotation marks. (No backslash needed at the end of line) *e.g.*,

```
Str1 = """Welcome
To
Python
"""
```

7. *What factors guide the choice of identifiers in programs ?*

Solution.

(*i*) An identifier must start with a letter or underscore followed by any number of digits and/or letters.

(*ii*) No reserved word or standard identifier should be used.

(*iii*) No special character (other than underscore) should be included in the identifier.

8. *Write the following real constants in exponent form : 17.251, 151.02, 0.00031, 0.452.*

Solution.

(*i*) $17.251 = 0.17251 \times 10^2$ = 0.17251E02

(*ii*) $151.02 = 0.15102 \times 10^3$ = 0.15102E03

(*iii*) $0.00031 = 0.31 \times 10^{-3}$ = 0.31E−3

(*iv*) $0.452 = 0.0452 \times 10^1$ = 0.0452E01

9. *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 in simple words, or absence of value. It is also used to indicate the end of lists in Python.

). *Identify the types of following literals ?*

23.789	23789	True	'True'	"True"
False	'"False"'	0XFACE	0o213	0o789

Solution.

23.789	Floating point
23789	integer
True	Boolean
'True'	String
"True"	String
False	Boolean
'"False"'	String
0XFACE	Integer (Hexadecimal)
0o213	Integer(Octal)
0o789	Invalid token (beginning with 0 means it is octal number but digits 8 and 9 are invalid digits in octal numbers)
None	None

11. *What is the difference between an expression and a statement in Python ?*

Solution.

Expression	Statement
Legal combination of symbols	Programming instruction as per Python syntax
Represents something	Does something
Python evaluates it	Python executes it
End result is a value	Need not result in a value
Example :	*Examples :*
`2.3`	`print ("Hello")`
`(3 + 5) / 4`	`if a > 0 :`

2. *Which of the following are syntactically correct strings? State reasons.*

(*a*) `"This course is great!"`

(*b*) `'She shouted "Hello!" very loudly.'`

(*c*) `"Goodbye'`

(*d*) `'This course is great!'`

(*e*) `"Hello`

(*f*) `"I liked the movie 'Bruce Almighty' very much."`

Solution. Strings (*a*), (*b*), (*d*) and (*f*) are syntactically correct. (Strings (*b*) and (*f*) are also valid as single quote strings can use double–quotes inside them and vice versa.)

String (*c*) ("Goodbye') is incorrect because opening and closing quotes don't match.

String (*e*) ("Hello) is invalid because it has no closing quotes.

13. *What is the error in following Python program with one statement ?*

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

Suggest a solution.

Solution. The above statement is trying to print the value of an undefined variable name. The solution to above problem is to define the variable name before using it, *i.e.*,

```
name = 'Tanya'
print ("My name is", name)
```

14. *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 not 40.

Also **Print( )** is not legal function of Python ; it should be **print( )**.

15. *What would be the correction for problem of previous question ?*

Solution. By using int( ) with input( ), we can convert the string into integer value, *i.e.*, as :

```
Number = int(input ("Enter Number" ) )
DoubleTheNumber = Number * 2
print (DoubleTheNumber)
```

Now the program will print desired output as 40 if 20 is typed as input.

16. *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 is required.

Thus, corrected code will be :

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

17. *Write a program to obtain temperature in Celsius and convert it into Fahrenheit using formula*

$$°C \times 9/5 + 32 = °F$$

Solution.

```
# Celsius to Fahrenheit
Cels = float(input( "Enter temp in Celsius :") )
print ("Temperature in Celsius is :", Cels)
Fahr = Cels * 9/5 + 32
print ("Temperature in Fahrenheit is :", Fahr)
```

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

```
value = 'Simar'
age = 17
print (name, ", you are ", 17, "now but", end = '')
print (" you will be ", age + 1, "next year")
```

Solution.

```
Simar, you are 17 now but you will be 18 next year.
```

19. *What will be the output of following code ?*

```
x, y = 2, 6
x, y = y, x + 2
print (x, y)
```

Solution.

```
6 4
```

Explanation. First line of code assigns values 2 and 6 to variables x and y respectively. Next line (second line) of code first evaluates right hand side *i.e.*, $y, x+2$ which is 6, 2 +2 *i.e.*, **6, 4** ; and then assigns this to x, y *i.e.*, $x, y = 6, 4$; so x gets **6** and y gets **4**. Third line prints x and y so the output is 6 4.

20. *Predict the output of following :*

```
x, y = 7, 2
x, y, x = x + 1, y + 3, x + 10
print (x, y)
```

Solution.

```
17 5
```

GLOSSARY

Constant	A data item that never changes its value during a program run.
Identifier	Name given by user for a part of the program.
Keyword	Reserved word having special meaning and purpose.
Lexical Unit	Other name of token.
Literal	Constant.
Token	The smallest individual unit in a program.
String literal	Sequence of characters enclosed in any type of quotes.
Variable	Named stored location whose value can be manipulated during program run.
Expression	Legal combination of symbols that represents a value.

Assignments

Type A : Short Answer Questions/Conceptual Questions

1. What are tokens in Python ? How many types of tokens are allowed in Python ? Examplify 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 in Python ? How ? Give examples to support your answer.
5. How are floating constants represented in Python ? Give examples to support your answer.
6. How are string-literals represented and implemented in Python ?
7. Which of these is not a legal numeric type in Python ? (*a*) int (*b*) float (*c*) decimal.
8. Which argument of print( ) would you set for :
   (*i*) changing the default separator (space) ? (*ii*) printing the following line in current line ?
9. What are operators ? What is their function ? Give examples of some unary and binary operators.
10. What is an expression and a statement ?
11. What all components can a Python program contain ?
12. What do you understand by block/code block/suite in Python ?
13. What is the role of indentation in Python ?
14. What are variables ? How are they important for a program ?
15. What do you understand by undefined variable in Python ?
16. What is Dynamic Typing feature of Python ?
17. What would the following code do : $X = Y = 7$?
18. What is the error in following code : $X, Y = 7$?
19. Following variable definition is creating problem $X = 0281$, find reasons.
20. "Comments are useful and easy way to enhance readability and understandability of a program." Elaborate with examples.

Type B : Application Based Questions

1. From the following, find out which assignment statement will produce an error. State reason(s) too.
   (*a*) $x = 55$ (*b*) $y = 037$ (*c*) $z = 0098$ (*d*) 56thnumber = 3300
   (*e*) length = 450.17 (*f*) !Taylor = 'Instant' (*g*) this variable = 87.E02
   (*h*) float = .17E – 03 (*i*) FLOAT = 0.17E – 03

2. Find out the error(s) in following code fragments :

   (*i*)
   ```
   temperature = 90
   print(temprature)
   ```

   (*ii*)
   ```
   a = 30
   b = a + b
   print (a And b)
   ```

   (*iii*)
   ```
   a, b, c = 2, 8, 9
   print (a, b, c)
   c, b, a = a, b, c
   print (a ; b ; c)
   ```

   (*iv*)
   ```
   X = 24
   4 = X
   ```

   (*v*)
   ```
   print ("X =" X)
   ```

   (*vi*)
   ```
   else = 21 - 5
   ```

3. What will be the output produced by following code fragment (s) ?

(*i*)
```
X = 10
X = X + 10
X = X - 5
print (X)
X, Y = X - 2, 22
print (X, Y)
```

(*ii*)
```
first = 2
second = 3
third = first * second
print (first, second, third)
first = first + second + third
third = second * first
print (first, second, third)
```

(*iii*)
```
side = int(input('side') )    #side given as 7
area = side * side
print (side, area)
```

4. What is the problem with the following code fragments ?

(*i*)
```
a = 3
    print (a)
b = 4
    print (b)
s = a + b
    print (s)
```

(*ii*)
```
name = "Prejith"
age = 26
print ("Your name & age are ", name + age)
```

(*iii*)
```
a = 3
s = a + 10
a = "New"
q = a / 10
```

5. Predict the output :

```
x = 40
y = x + 1
x = 20, y + x
print (x, y)
```

6. Predict the output

```
x, y = 20, 60
y, x , y = x, y - 10, x + 10
print (x, y)
```

7. Predict the output

(*a*)
```
a, b = 12, 13
c, b = a*2, a/2
print (a, b, c)
```

(*b*)
```
a, b = 12, 13
print (print(a + b))
```

8. Predict the output

```
a, b, c = 10, 20, 30
p, q, r = c - 5, a + 3, b - 4
print ('a, b , c :', a, b, c, end = '')
print ('p, q, r :', p, q, r)
```

9. Find the errors in following code fragment

(*a*)
```
y = x + 5
print (x, Y)
```

(*b*)
```
print (x = y = 5)
```

(*c*)
```
a = input("value")
b = a/2
print (a, b)
```

10. Find the errors in following code fragment : (The input entered is XI)

```
c = int(input( "Enter your class") )
print ("Your class is", c)
```

11. Consider the following code :

```
name = input("What is your name?")
print ('Hi', name, ',')
print ("How are you doing?")
```

was intended to print output as

```
Hi <name>, How are you doing ?
```

But it is printing the output as :

```
Hi <name>,
How are you doing?
```

What could be the problem ? Can you suggest the solution for the same ?

12. Find the errors in following code fragment :

```
c = input( "Enter your class" )
print ("Last year you were in class") c – 1
```

13. What will be returned by Python as result of following statements?

(*a*) `>>> type(0)` (*b*) `>>> type(int(0))` (*c*) `>>>.type(int('0'))`

(*d*) `>>> type('0')` (*e*) `>>> type(1.0)` (*f*) `>>> type(int(1.0))`

(*g*) `>>>type(float(0))` (*h*) `>>> type(float(1.0))` (*i*) `>>> type( 3/2)`

Match your result after executing above statements.

14. What will be the output produced by following code ?

(*a*) `>>> str(print())+"One"`

(*b*) `>>> str(print("hello"))+"One"`

Match your result after executing above statements. Can you explain why this result came?

15. What will be the output produced by following code ?

(*a*) `>>> print(print("Hola"))`

(*b*) `>>> print(print("Hola", end = " "))`

Match your result after executing above statements. Can you explain why this result came?

16. Carefully look at the following code and its execution on Python shell. Why is the last assignment giving error ?

```
>>> a = 0o12
>>> print(a)
10
>>> b = 0o13
>>> c = 0o78
    File "<python-input-41-27fbe2fd265f>", line 1
    c = 0o78
        ^
```

SyntaxError : invalid syntax

17. Predict the output

```
a, b, c = 2, 3, 4
a, b, c = a*a, a*b, a*c
print(a, b, c)
```

18. The id( ) can be used to get the memory address of a variable. Consider the following code and tell if the id( ) functions will return the same value or not (as the value to be printed via print( )) ? Why ?
[There are four print( ) function statements that are printing id of variable *num* below)

```
num = 13
print( id(num) )
num =  num + 3
print( id(num) )
num = num - 3
print( id(num) )
num = "Hello"
print( id(num) )
```

19. Consider below given two sets of codes, which are nearly identical, along with their execution in Python shell. Notice that first code-fragment after taking input gives error, while second code-fragment does not produce error. Can you tell why ?

(*a*)

```
>>> print(num = float(input("value1:")) )
value1:67
Traceback (most recent call last):
    File "<python-input-56-78b83d911bc6>", line 1, in <module>
        print(num = float(input("value1:")) )
TypeError: 'num' is an invalid keyword argument for this function
```

Input taken as per execution of input()

(*b*)

```
>>> print(float(input("value1:")) )
value1:67
67.0
```

Code successfully executed. No error reported.

20. Predict the output of the following code :

```
days = int(input("Input days: ")) * 3600 * 24
hours = int(input("Input hours: ")) * 3600
minutes = int(input("Input minutes: ")) * 60
seconds = int(input("Input seconds: "))
time = days + hours + minutes + seconds
print("The amounts of seconds", time)
```

If the input given is in this order : 1, 2, 3, 4

Type C : Programming Practice/Knowledge based Questions

1. Write a program that displays a joke. But display the punchline only when the user presses enter key. (**Hint.** You may use input( ))
2. Write a program to read today's date (only del part) from user. Then display how many days are left in the current month.
3. Write a program that generates the following output :

```
5
10
9
```

Assign value 5 to a variable using assignment operator (=) Multiply it with 2 to generate 10 and subtract 1 to generate 9.

4. Modify above program so as to print output as 5@10@9.
5. Write the program with maximum three lines of code and that assigns first 5 multiples of a number to 5 variables and then print them.
6. Write a Python program that accepts radius of a circle and prints its area.
7. Write Python program that accepts marks in 5 subjects and outputs average marks.
8. Write a short program that asks for your height in centimetres and then converts your height to feet and inches. (1 foot = 12 inches, 1 inch = 2.54 cm).
9. Write a program to read a number n and print n^2, n^3 and n^4.
10. Write a program to find area of a triangle.
11. Write a program to compute simple interest and compound interest.
12. Write a program to input a number and print its first five multiples.
13. Write a program to read details like *name, class, age* of a student and then print the details firstly in same line and then in separate lines.

    Make sure to have two blank lines in these two different types of prints.
14. Write a program to input a single digit(n) and print a 3 digit number created as <n(n+1)(n+2)> e.g., if you input 7, then it should print **789**. Assume that the input digit is in range 1-7.
15. Write a program to read three numbers in three variables and swap first two variables with the sums of first and second, second and third numbers respectively.