CLASS-XI

Study Notes

Python Fundamentals

Period-01

Learning Outcomes

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

Introduction:

Python programming language was developed by Guido Van Rossum in February 1991. Python is based on or influenced with two programming languages:

- ABC language, a teaching language created as a replacement of BASIC, and
- Modula-3

Python is an easy-to-learn yet powerful object oriented programming language. It is a very high level programming Language yet as powerful as many other middle-level not so high-level languages like C, C++, Java etc.

Though Python language came into being in early 1990's, yet it is competing with ever-popular languages such as C, C++, Java etc. in popularity index. Although, it is not perfect for every type of application, yet it has much strength that makes it a good choice for many situations. Let's see what these pluses of Python are.

- Pluses of Python:

- Easy to Use Object Oriented Language
- Expressive Language
- Interpreted Language
- Its Completeness
- Cross-platform Language
- Free and Open Source

1. Easy to Use:

Python is compact and very easy to use object oriented language with very simple syntax rules. It is a very high level language and thus very-very programmer-friendly.

2. Expressive Language:

Python's expressiveness means it is more capable to expressing the code's purpose than many other languages. Reason being- fewer lines of code, simpler syntax.

For example, consider following two sets of codes:

```
# In C++ : Swap Values
int a = 2, b = 3, tmp ;
tmp = a ;
a= b ;
b = tmp;
```

```
# In Python : Swap values
a, b = 2, 3
a, b = b, a
```

3. Interpreted Language:

Python is an interpreted language, not a compiled language. This means that the Python installation interprets and executes the code line by line at a time. It makes Python an easy-to-debug language and thus suitable for beginners to advanced user.

4. Its Completeness:

When you install Python, you get everything you need to do real work. You do not need download and install additional libraries ; all types of required functionality is available through various modules of Python standard library. For example, for diverse functionality such as emails, web-pages, databases, GUI development network connections and many more, everything is available in Python standard library. Thus, it is also called - Python follows - "Batteries Included" philosophy.

5. Cross-platform Language:

Python can run equally well on variety of platforms - Windows, Linux/UNIX, Macintosh, supercomputers, smart phones etc.2 Isn't that amazing ? And that makes Python a cross-platform language. Or in other words, Python is a portable language.

6. Free and Open Source:

Python language is freely available i.e., without any cost (from www.python.org). And not only it free, its source-code (i.e., complete program instructions) is also available, i.e., it is open-source also. Do you know, you can modify, improve/extend open-source software.

7. Variety of Usage/Applications:

Python has evolved into a powerful, complete and useful language over these years. These days Python is being used in many diverse fields/applications, some of which are:

- Scripting

- Web Applications
- Game development
- System Administrations
- Rapid Prototyping
- GUI Programs
- Database Applications

PYTHON - SOME MINUSES (SO HUMAN LIKE):

Although Python is very powerful yet simple language with so many advantages, it is not the Perfect Programming language. There are some areas where Python does not offer much or is not that capable. Let's see what these are:

1. Not the Fastest Language: Python is an interpreted language not a fully compiled one. Python is first semi-compiled into an internal byte-code, which is then exerted by a Python interpreter. Fully compiled languages are faster than their interpreted counterparts. So, here Python is little weaker though it offers faster development times but execution-times are not that fast compared to some compiled languages.

2. Lesser Libraries than C, Java, Perl:

Python offers library support for almost all computing programs, but its library is still competent with languages like C, Java, and Perl as

they have larger collections available. Some· in some cases, these languages offer better and multiple solutions than Python.

3. Not Strong on Type-binding:

Python interpreter is not very strong on catching 'Type-mismatch' issues. For example, if you declare a variable as integer but later store a string value in it, Python won't complain or pin-point it.

4. Not Easily Convertible:

Because of its lack of syntax, Python is an easy language to program in. But this advantage has a flip-side too: it becomes a disadvantage when it comes to translating a program into another programming language. This is because most other languages have structured defined syntax.

Since most other programming languages have strong-syntax, the translation from Python to another language would require the user to carefully examine the Python code and its structure and then implement the same structure into other programming language's syntax.

So, now you are familiar with what all Python offers. As a free and open-source language, its users are growing by leaps and bounds.

Period-02

Introduction:

Familiarization with the basics of Python Programming:

(Interactive & Script mode)

WORKING IN PYTHON:

Before you start working in Python, you need to install Python on your computers. There are multiple Python distributions available today.

- Default installation available from **www.python.org** is called CPython installation and comes with Python interpreter, Python IDLE (Python GUI) and Pip (package installer).
- There are many other Python distributions available these days. Anaconda Python distribution is one such highly recommended distribution that comes preloaded with many packages and libraries (e.g., NumPy, SciPy, Panda libraries etc),
- Many popular IDEs are also available e.g., Spyder IDE, PyCharm IDE etc. Of these, Spyder IDE is already available as a part of Anaconda Python distribution.

Once you have Python installed on your computers, you are ready to work on it. You can work in Python in following different ways:

(i) In Interactive mode (also called Immediate Mode)

(ii) In Script mode

ODM EDUCATIONAL GROUP Changing your Tomorrow

Working in Default CPython Distribution:

The default distribution, CPython, comes with Python interpreter, Python IDLE (GUI based) and pip (package installer). To work in interactive as well as script mode, you need to open Python IDLE.

Working in Interactive Mode (Python IDLE):

Interactive mode of working means you type the command - one command at a time, and the Python executes the given command there and then and gives you output. In interactive mode, you type the command in front of Python command prompt >>>.

For example, if you type 2 + 5 in front of Python prompt, it will give you result as 7 :

```
>>> 2 + 5   ←——— command/expression given here
7
```

Result returned by Python

To work in interactive mode, follow the process given below :

(i) Click **Start button -> All Programs -> Python 3.6.x -> IDLE (Python GUI)** [see Fig. 5.1(a)]

Or

Click **Start button -> All Programs -> Python 3.6.x -> Python (**command line)

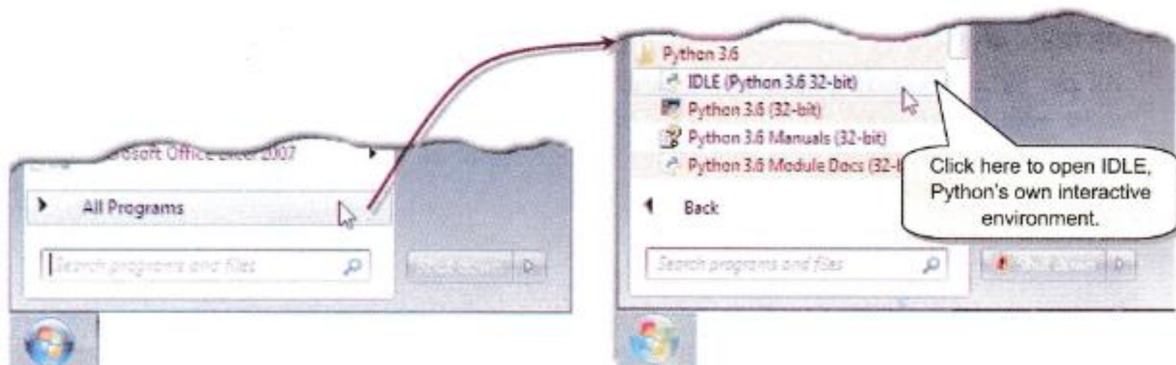


Figure 5.1 (a) Starting Python Shell.

(ii) It will open Python Shell [see Fig. 5.1(b)] where you'll see the Python prompt (three '>' signs i.e, >>>). The interactive interpreter of Python is also called Python Shell.

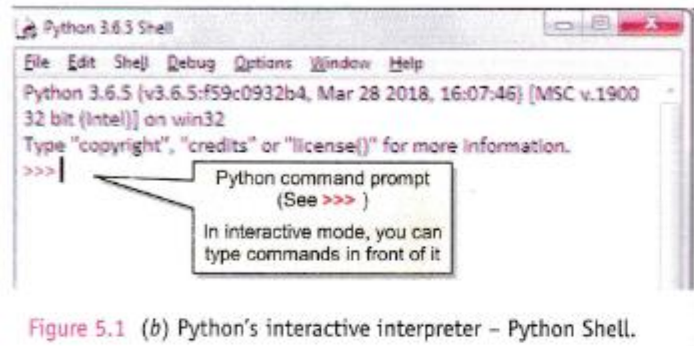


Figure 5.1 (*b*) Python's interactive interpreter – Python Shell.

(iii) Type command in front of this Python prompt and Python will immediately give you the result. [See Fig. 5.1(c)]

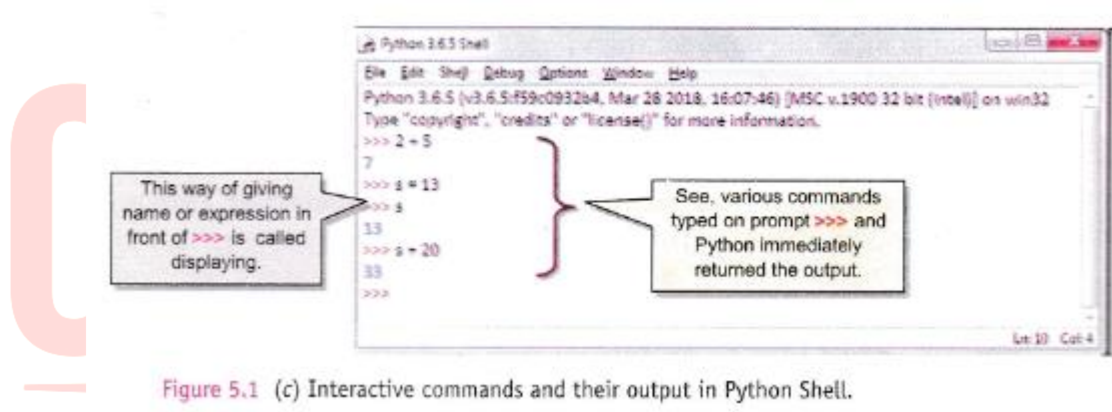


Figure 5.1 (*c*) Interactive commands and their output in Python Shell.

EDUCATIONAL GROUP
Changing your Tomorrow

For example, to print string "Hello" on the screen, you need to type the following in front of Python prompt (>>>).

```
>>> print ("Hello")
```

And Python interpreter will immediately display string Hello below the command. To display, you just need to mention name or expression [Fig. 5.1(c)] in front of the prompt.

Figure 5.1(c) shows you some sample commands that we typed in Python shell and the output returned by Python interpreter.

Interactive mode proves very useful for testing code; you type the commands one by one and get the result or error one by one.

Working in Script Mode (Python IDLE):

What if you want to save all the commands in the form of program file and want to see all output lines together rather than sandwiched between successive commands? With interactive mode, you cannot do so, for:

- Interactive mode does not save the commands entered by you in the form of a program.
- The output is sandwiched between the command lines [see Fig. 5.1(c)].

The solution to above problems is the Script mode. To work in a script mode, you need to do the following.

Step 1: Create Module / Script / Program File:

Firstly, you have to create and save a module / Script / Program file. To do so, follow these instructions:

(i) Click Start button ->All Programs -> Python 3.6.x -> IDLE. [Fig. 5.2(a)]

(ii) Click File -> **New** in IDLE Python Shell. [Fig. 5.2(a)]

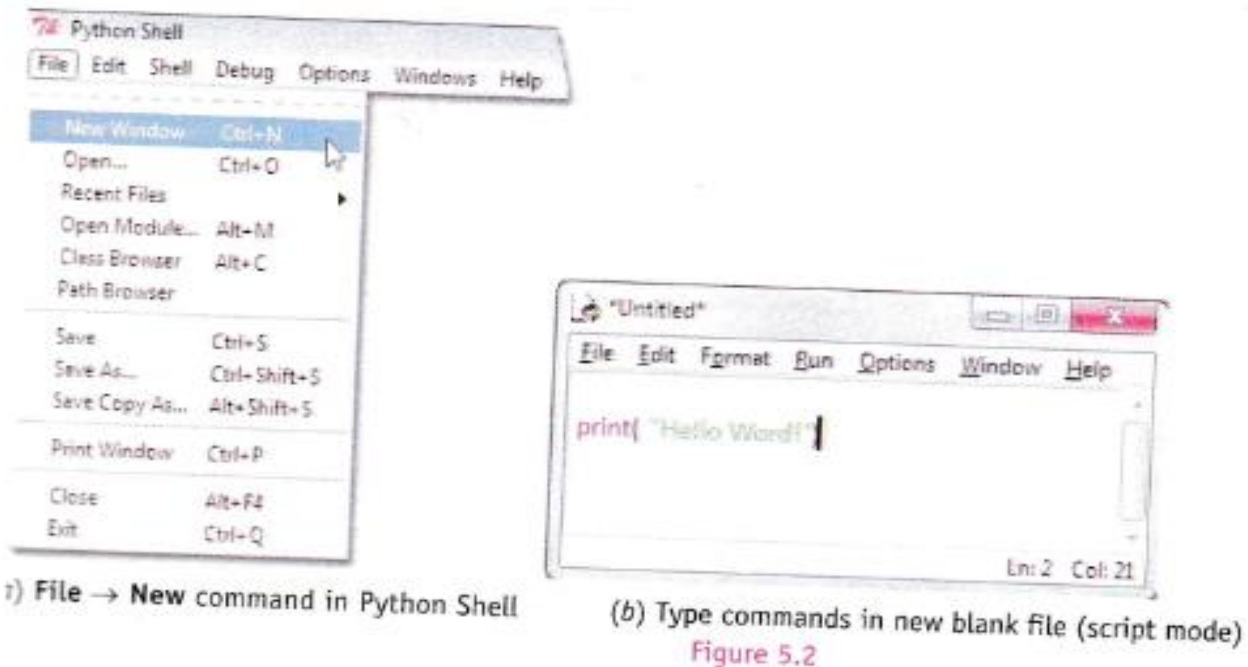
(iii) In the New window that opens, type the commands you want to save in the form of a program (or script). [Fig. 5.2(b)]

For instance, for the simple Hello World program, you'll need to type following line:

```
print (" Hello World ! ")
```

You can display as well as print values in interactive mode, but for script mode, print( ) command is preferably used to print results.

ODM EDUCATIONAL GROUP Changing your Tomorrow



(a) File → New command in Python Shell (b) Type commands in new blank file (script mode)

Figure 5.2

(iv)Click **File-> Save** and then save the file with an extension .py. The Python program has .py extension [Fig. 5.2(c)]. For instance, we gave the name to our program as Hello.py

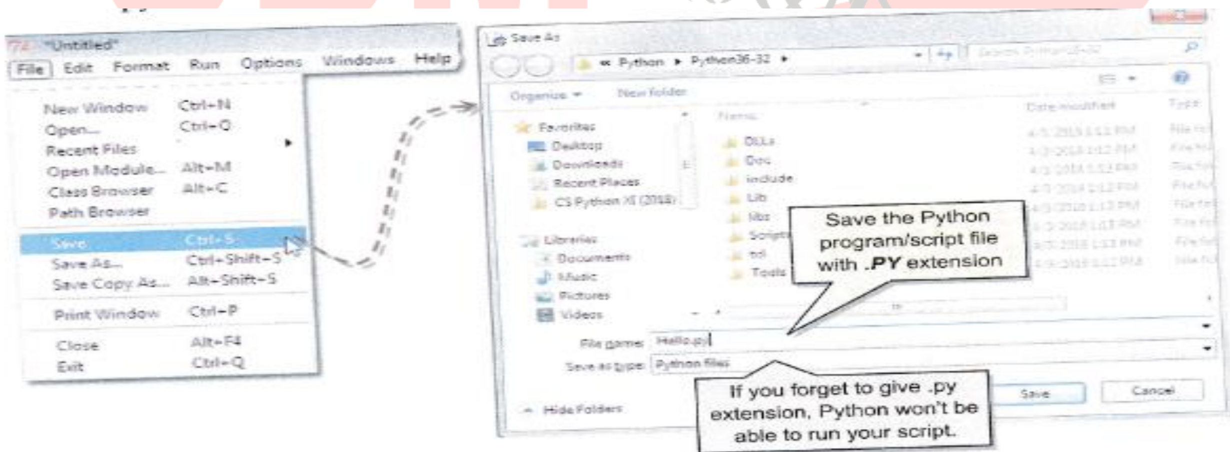


Figure 5.2 (c) Save file with .py extension with File → Save command (Script mode).

Now your program would be saved on the disk and the saved file will have .py extension.

Step 2: Run Module / Script / Program File:

After the program/script file is created, you can run it by following the given instructions:

(i) Open the desired program/script file that you created in previous Step 1 by using IDLE's File -> Open command.

If the program / script file is already open, you can directly move to next instruction

(ii) Click Run->Run Module command [Fig. 5.3(a)] in the open program / script file's window.

You may also press F5 key.

(iii) And it will execute all the commands stored in module I program I script that you had opened and show you the complete output in a separate Python Shell window. [Fig. 5.3(b)]

Figure 5.3 (a) Run → Run Module command (Script mode)

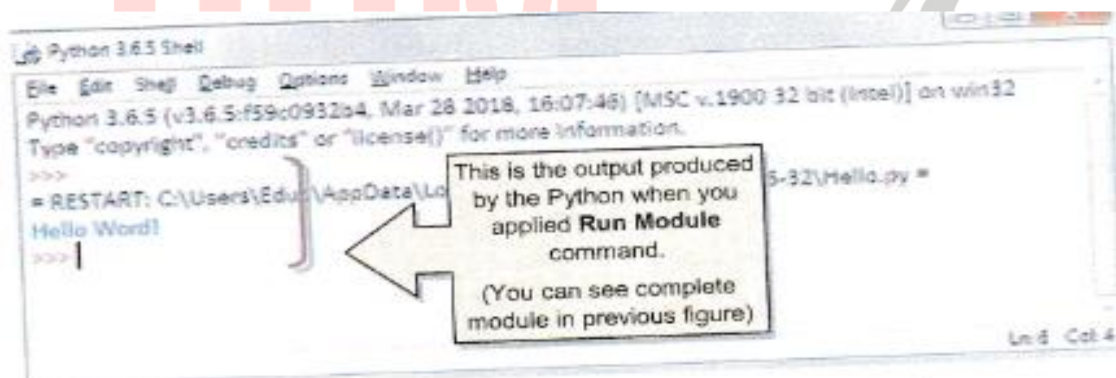


Figure 5.3 (b) Output of a module-run is shown in the shell window.

Period-03

Introduction:

Common Data types: Integer, float and strings

Numeric Literals:

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

int (**signed integers**)	often called just integers or ints, 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 + b j, where a and b are floats and j represents H, which is an imaginary number). a is the real part of the number, and b is the imaginary part.

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. (810 = 108) and decimal integer 12 will be written as 0o14 as octal integer (1210 = 148).

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, 0xPQR, 0x19AZ etc., are examples of invalid hexadecimal numbers as they contain invalid letters, i.e., letters other than A - F.

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 in fractional form:

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 0.58x 101 = 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 101.

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, l.52E07, 0.152E08, 152.0E05, 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

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)

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.)

Boolean Literals:

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.

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**.

ODM EDUCATIONAL GROUP Changing your Tomorrow

String Literals:

The text enclosed in quotes forms a string literal in Python. For example, 'a', 'obc', "abc" are all string literals in Python. Unlike any other languages, both single character enclosed in quotes such as "a" or 'x' or multi le 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“

Python allows you to have certain non-graphic-characters in String values. Non-graphic character is 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 non-graphic-characters can be represented by using escape sequences. An escape sequence represented by a backslash (\) followed by one or more characters.

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 va xxxx (Unicode only)
\a	ASCII Bell (BEL)	\Uxxxxxxxx	Character with 32-bit hex va 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 type from the keyboard but when used without escape sequence, these carry a special meaning an 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)

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'

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

```
In [2]: Text1 = ' hello
  File "<ipython-input-2-f98177eb9351>", line 1
    Text1 = ' hello
                   ^
SyntaxError: EOL while scanning string literal

In [3]:
```

EOL means End of Line

The reason for the above error is quite clear - Python by default creates single-line strings with both single and 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.

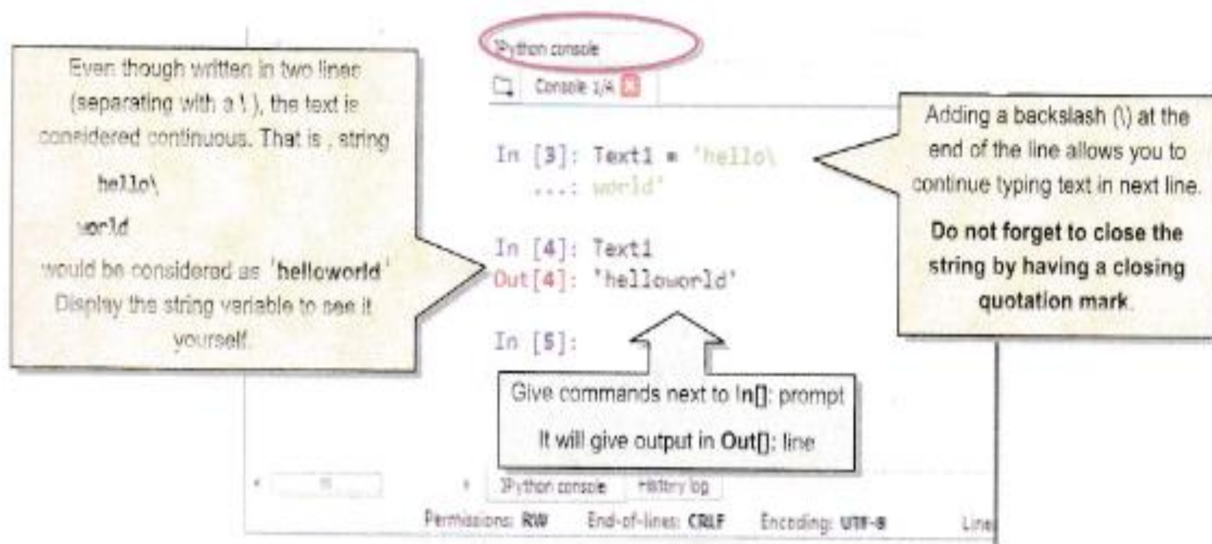
For instance,

```
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:



EDUCATIONAL GROUP
Changing your Tomorrow

(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).

For example,

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

Or

Python allows to type multiline text string by enclosing them in triple quotation marks (both triple apostrophe and triple quotation marks will work).

```
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\
11\
o'''
```

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

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(<string name>) command on the Python prompt in console window shell as shown in the following figure :

EDUCATIONAL GROUP
Changing your Tomorrow

Period-04

Introduction:

Features of Python - Python Character Set, Token & Identifiers, Keywords, Literals, Delimiters, Operators.

Features of Python:

- Python Character Set
- Token & Identifiers
- Keywords
- Literals

- Delimiters
- Operators

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 + - * I ** \ ( ) [ ] { } I I =! = == < , > & # <= >=@_(underscore) , , : % !
	Whitespaces Blank space, t abs (->), carriage return (.J), newline, form feed
Other characters	Python can process all ASCII and Unicode characters as part of data or literals.

TOKENS:

In a passage of text, individual words and punctuation marks are called tokens or lexical unit or lexical elements. The smallest individual unit in a program is known as a Token or a lexical unit·

Consider the following figure that tells what a token means.

An English Sentence

"Hurray!", he exclaimed, "We are the world champions."

The lexical units or tokens in an English sentence - here two types of tokens are used : words and punctuation characters

A Python Program

```
# A sample program
name = "Anya"
age = 15
contribution = 710.75
print(name, "aged", age, "has contributed", contribution)
```

A string value or string literal

An integer value or integer literal

A floating-point value or floating point literal

Keyword

Punctuator characters

Individual lexical units or tokens, together make components of a program such as statements or expressions etc. In above program you can see these tokens : literals, keywords, punctuators

Tokens are smallest individual units in a program.

Python has following tokens:

1. Keywords
2. Identifiers (Names)
3. Literals
4. Operators
5. Punctuators

ODM EDUCATIONAL GROUP Changing your Tomorrow

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

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:

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	

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 (_).

Identifiers (Names):

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 (.)

Literals I 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

(iv) Literal Collections

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.

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 Operator:

<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to
=	Equal to

Assignment Operator

=	Assignment
/=	Assign quotient
+=	Assign sum
*- =	Assign product
%=	Assign remainder

ODM EDUCATIONAL GROUP Changing your Tomorrow

!=	Not equal to	-=	Assign difference
		**=	Assign Exponent
		//=	Assign Floor division

Logical Operator

and	Logical AND
or	Logical OR

Membership Operator

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

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:

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

Period-05

Introduction:

Working with print statements

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 :

```
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.

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.

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.

```
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
```

Two different indentation-levels inside this code.
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.

Please note that values like 73 or 73. or .73 are all float convertible, hence Python will be able to convert them to float and no error shall be reported if you enter such values.

While entering numeric values through input( ) along with int() /float(), make sure that you enter values that are convertible to the target type otherwise Python will raise an error.

```
In [25]: marks = float ( input("Enter marks : ") )
Enter marks : 73
In [27]: marks = float ( input("Enter marks : ") )
Enter marks : 73.
```

Values like .73 or 73. or 73 etc. can be easily converted into float, hence Python report no error if you enter such values with **float( )** used with **input( )** .

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 syntax3 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
```

Output Through print( ) Statement:

```
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,"yearsold.") #multiple comma separated expressions

Consider some examples with outputs:

Example statement 1 :	*Example statement 2 :*	*Example statement 3 :*
`print ("Python is wonderful.")` will print the output as: `Python is wonderful.`	`print ("Sum of 2 and 3 is", 2 + 3)` will print the output as follows (2+3 is evaluated and its result is printed as 5): `Sum of 2 and 3 is 5`	`a = 25` `print ("Double of",a, "is", a*2)` will print the output as follows (a*2 is evaluated and its result is printed) : `Double of 25 is 50`

Now consider some more print statement examples:

print(obj)

print(objl, obj2, obj3)

print( )

print('Objectl has more value than Object2')

print(obj 1, 'is lesser than', obj2)

The output of these print() functions, ym-1'11 be able to determine if the values of variables obj, objl, obj2 and obj3 are known to you. (A print( ) without any value or name or expression prints a blank line.)

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

=> 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
```

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".

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.  ")
```

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

will print output as -> 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.

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

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( )

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

Find out the output for the following code:

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

Period- 06

Introduction:

Comments: (Single line & Multiline/ Continuation statements), Clarity & Simplification of expression

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

- expressions
- statements
- comments
- blocks and indentation
- Function

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     } expressions that are values only
2.9    }

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

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 do something i.e., some action takes place.

Following are some examples of statements:

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

While an expression is evaluated, a statement is executed i.e., some action takes place. And it 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:
:
```

:

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.

In the above code, you can see four comments:

(i) The physical lines beginning with # are the full line comments. There are three full 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
```

Multi-line Comments:

What if you want to enter a multi-line comment or a block comment ? You can enter 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 (Comments are enclosed in triple quotes (" " ") or triple apostrophe
(‘ ‘ ‘) are called docstrings) are very useful in documentation.

iv) 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)

```
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.

(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,

Four spaces together mark the 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 :

```
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.

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.

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.

```
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
```

Two different indentation-levels inside this code.

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.

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

Creating a Variable:

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

marks = 70

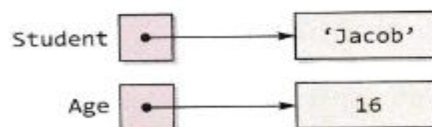
As you can see, that we just assigned the value of numeric type to an identifier name and Python created the variable of the type similar to the type of value assigned. In short, after the above statement, we can say that marks are a numeric variable.

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:



Period- 07

Introduction:

Introduce the notion of a variable and methods to manipulate it (concept of L-value and R-value)

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:

1. **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.
2. **Maximum Line Length**: Line length should be maximum 79 characters.
3. **Lines and Indentation**: Blocks of code are denoted by line indentation, which is enforced through 4 spaces (not tabs) per indentation level.

4. 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.

5. 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.

6. Whitespaces: 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' (form feed) and '\r' (carriage return)

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

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

ODM EDUCATIONAL GROUP Changing your Tomorrow

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

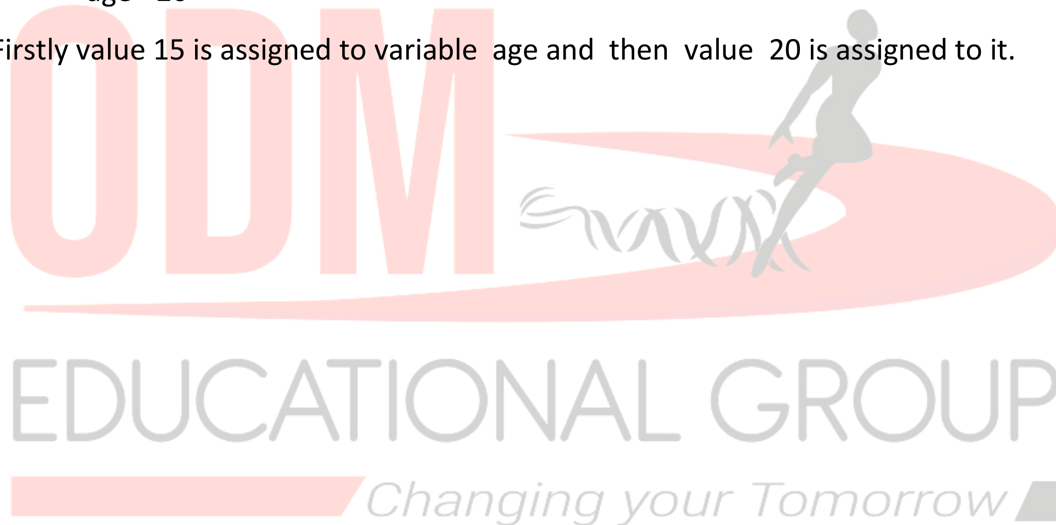
Variables are Not Storage Containers in Python:

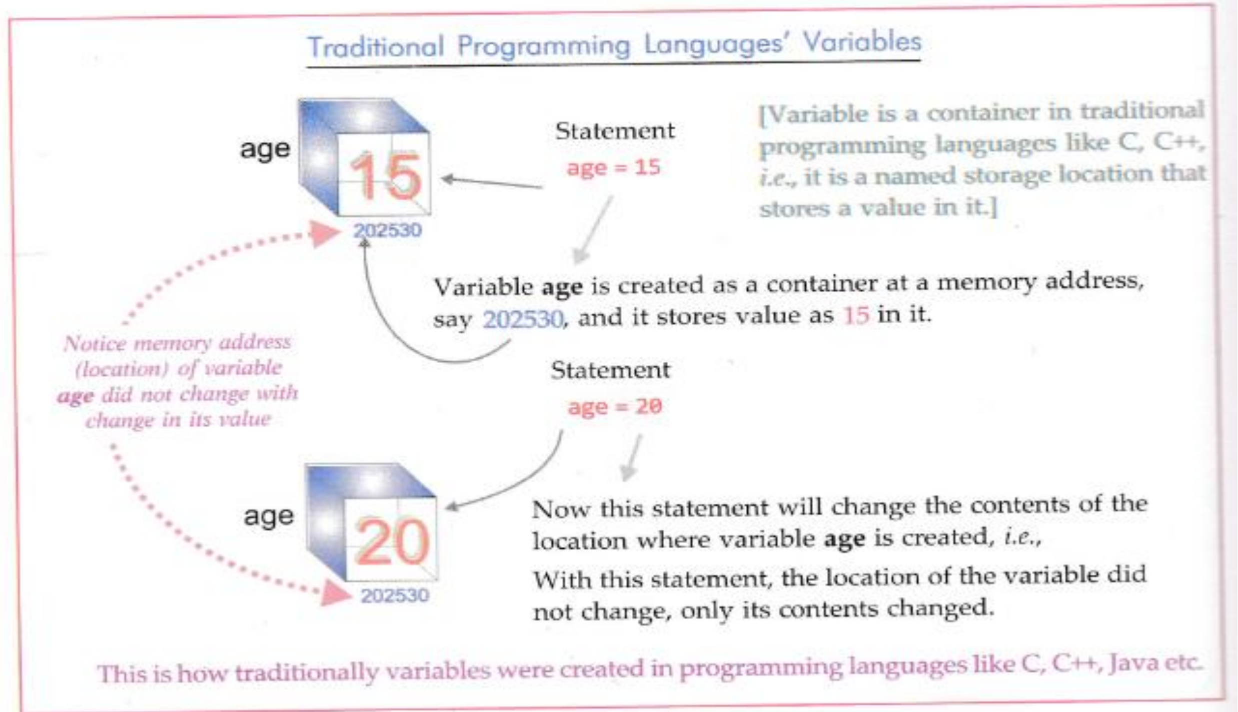
If you have an earlier exposure to programming, you must be having an idea of variables. BUT PYTHON VARIABLES ARE NOT CREATED IN THE FORM MOST OTHER PROGRAMMING LANGUAGES DO. Most programming languages create variables as storage containers e.g.,

Consider this:

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

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



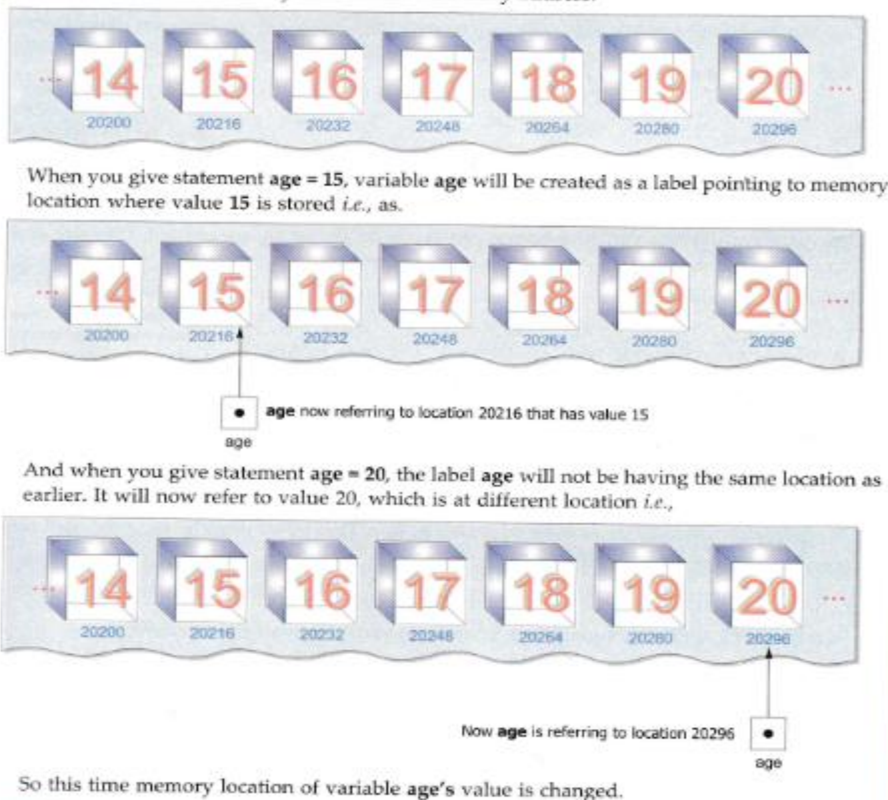


Python Variables in Memory:

BUT PYTHON DOES THIS DIFFERENTLY

Let us see how Python will do it. Python preloads some commonly used values (even before any identifier is created) in an area of memory. We can refer to this area as front-loaded dataspace.

The dataspace 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 every time their values change (This rule is not for all types of variables, though).

lvalue and rvalue:

lvalues are the objects to which you can assign a value or expression. lvalue 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.

e.g., we can say that a=30 & b=10

But we cannot say that 30=a, 10=b, 2*a=b is absolutely incorrect.

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 1 Lvalues can come on lhs as well as rhs of an assignment.

Multiple Assignments:

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

1. **Assigning some 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 re 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

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              # statement 1
b, c, a = a+ 1, b + 2, c – 1    # statement 2
print (a, b, c)
```

=> Statement 1 assigns 5, 10 and 7 to a, b and c respectively.

=> Statement 2 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

Now find out the output of following code fragment

```
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.

The output for the above should be: 3 1 5

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

ODM EDUCATIONAL GROUP Changing your Tomorrow

Variable Definition:

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

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 uncreated (undefined) variable results into error.

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

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.

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

```
x = 10
print(x)
x = "Hello World"
Print (x)
```

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 as described in the following figure.

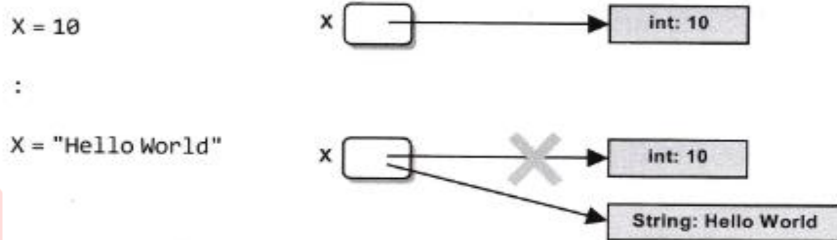


Figure 6.1 Dynamic typing in Python variables.

As you can see in Fig. 6.1, 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**.

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.

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          <- legal, because two integers can be used in divide operation
X = 'Day'        <- Python is comfortable with dynamic typing
Y = X/2          <- ERROR!! a string cannot be divided.
```

so as programmer, you need to ensure that variables with right type of values should be used in expressions.

If you want to determine the type of a variable i.e., what type of value does it point to?, you can use type() in following manner:

type (<variable name»)

For instance, consider the following sequence of commands that uses type( ) three times:

```
>>> a = 10
>>> type(a)
<class 'int'>        <- The type returned as int (integer)
>>> a = 20.5
>>> type(a)
<class 'float' >     <- The type returned as float (floating point number)
>>> a = "hello"
>>> type(a)
<class 'str'>        <- The type returned as str (string)
>>>
```

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.

Programming languages like C, C++ support static typing.

Simple Input and Output:

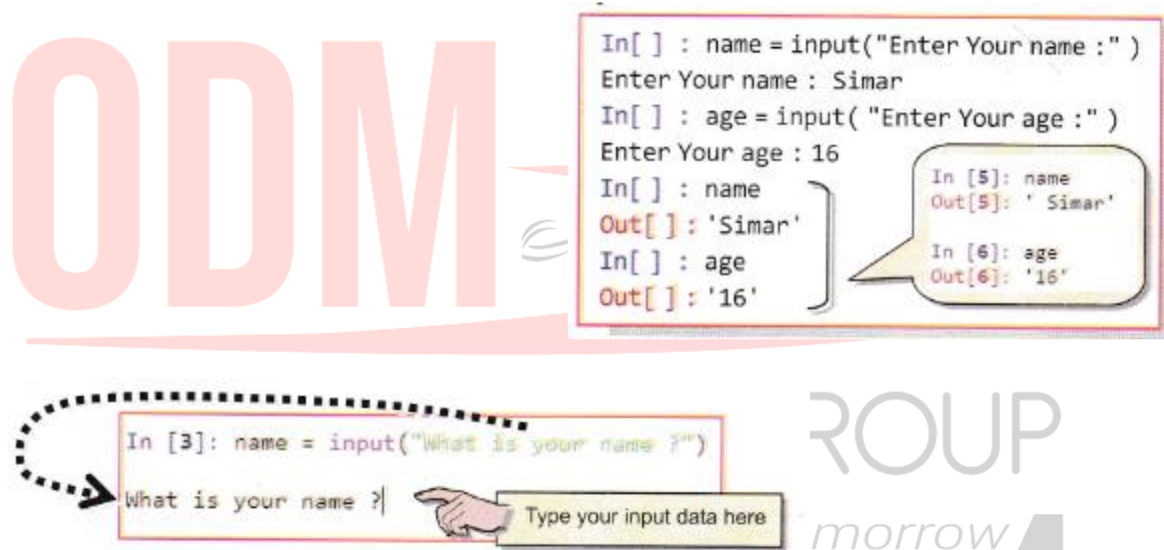
In Python 3.x, to get input from user interactively, you can use built-in function input( ). The function input( ) is used in the following manner :

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



in front of which you can type the name. The value that you type in front of the displayed prompt will be assigned to given variable, name in above case. Now consider the following command sequence:

In above code, we used input( ) function to input two values name and age.

Please note the In[ ]: is the prompt of Python shell(used with Spyder IDE) and >>> is shown as general prompt of a Python shell. You can use any of the available IDEs of Python.

But input( ) has a property, which you must be aware of. The input( ) function always returns a value of String type. Notice carefully in above code while displaying both values name and age, Python has enclosed both the values in quotes i.e., as 'Simar' and '16'. This is because whatever value you enter through input( ) function is treated as a String. The input( ) function always returns a value of String type.

Now what are its consequences, especially when you are entering a number? In order to understand this, consider the following code.

```
In [8]: age = input("What is your age ? ")

What is your age ? 16

In [9]: age + 1
Traceback (most recent call last):

  File "<ipython-input-9-eac256a954eb>", line 1, in <module>
    age + 1

TypeError: must be str, not int
```

Notice the error raised when you try to add a number to variable **age** which was given a value using ***input( )*** function.

See, Python raised an error when you tried to add 1 to age whose value you entered as 16. The reason is obvious and clear - Python cannot add on integer to a string. Since variable age received value 16 through input(), it actually had '16' in it i.e., string value '16'; thus you cannot add an integer to it.

You can check yourselves the type of variable age whose value you entered through input( ) Function:

When you try to perform an operation on a data type not suitable for it (e.g., dividing or multiplying a string), Python raises an error called TypeError.

ODM EDUCATIONAL GROUP Changing your Tomorrow

```
In [10]: age = input("What is your age ? ")

What is your age ? 16

In [11]: age
Out[11]: '16'

In [12]: type(age)
Out[12]: str
```

See it shows the type of variable ***age*** as **str**

Reading Numbers:

String values cannot be used for arithmetic or other numeric operations. For these operations, you need to have values of numeric types (integer or float).

But what you would do if you have to read numbers (int or float)? The function input( ) returns the entered value in string type only.

Python offers two functions int( ) and float( ) to be used with input( ) to convert the values received through input() into int and float types.

=> 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.

```
In [13]: age = input("What is your age ? ")

What is your age ? 16

In [14]: age = int(age)

In [15]: age + 1
Out[15]: 17
```

```
In [16]: marks = input("Enter marks : ")

Enter marks : 73.5

In [17]: marks = float(marks)

In [18]: marks + 1
Out[18]: 74.5
```

After inputting value through **input( )**, use **int( )** or **float( )** with variable to change its type to **int** or **float** type

See it gave no error because the type of **age** is not **string** but **int** and type of **marks** is **float**, not **string**

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>))

```
In [19]: marks = float ( input("Enter marks : ") )

Enter marks : 73.5

In [20]: age = int( input("What is your age ? ") )

What is your age ? 16

In [21]: type(marks)
Out[21]: float

In [22]: type(age)
Out[22]: int
```

Function **int( )** around **input( )** converts the read value into **int** type and function **float( )** around **input( )** function converts the read value into **float** type.

Possible Errors When Reading Numeric Values:

If you are planning to input an integer or floating-point number using input( ) inside int( ) or float( ) such as shown below :

age = int (input ('Enter Your age : '))

or

percentage= float (input ('Enter your percentage : '))

Then you must ensure that the value you are entering must be in a form that is easily convertible to the target type.

In other words:

. (i) While inputting integer values using int( ) with input( ), make sure that the value being entered must be int type compatible. Carefully have a look at example code given below to understand it.

```
In [23]: age = int( input("What is your age ? ") )

What is your age ? 17.5
Traceback (most recent call last):

  File "<ipython-input-23-f69b217c38eb>", line 1, in <module>
    age = int( input("What is your age ? ") )

ValueError: invalid literal for int() with base 10: '17.5'
```

Notice the value entered (17.5). It is not an ***int*** compatible value

Thus this error

```
In [24]: age = int( input("What is your age ? ") )

What is your age ? Seventeen
Traceback (most recent call last):

  File "<ipython-input-24-f69b217c38eb>", line 1, in <module>
    age = int( input("What is your age ? ") )

ValueError: invalid literal for int() with base 10: 'Seventeen'
```

Notice the value entered (Seventeen). It is not an ***int*** compatible value

Thus this error

(ii) while inputting floating point values using float( ) with input( ), make sure that the value being entered must be float type compatible. Carefully have a look at example code given below to understand it.

```
In [26]: marks = float ( input("Enter marks : ") )

Enter marks : 73.5.0
Traceback (most recent call last):

  File "<ipython-input-26-bae4b006eebf>", line 1, in <module>
    marks = float ( input("Enter marks : ") )

ValueError: could not convert string to float: '73.5.0'
```

Notice the value entered (73.5.0). It is not an ***float*** compatible value

Thus this error

```
In [28]: marks = float ( input("Enter marks : ") )

Enter marks : 73 percent
Traceback (most recent call last):

  File "<ipython-input-28-bae4b006eebf>", line 1, in <module>
    marks = float ( input("Enter marks : ") )

ValueError: could not convert string to float: '73 percent'
```

Notice the value entered (73 percent). It is not an ***float*** compatible value

Thus this error

Period- 08

Introduction:

Discussion of Output Questions

Find out the output for the following code:

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

****

ODM EDUCATIONAL GROUP Changing your Tomorrow