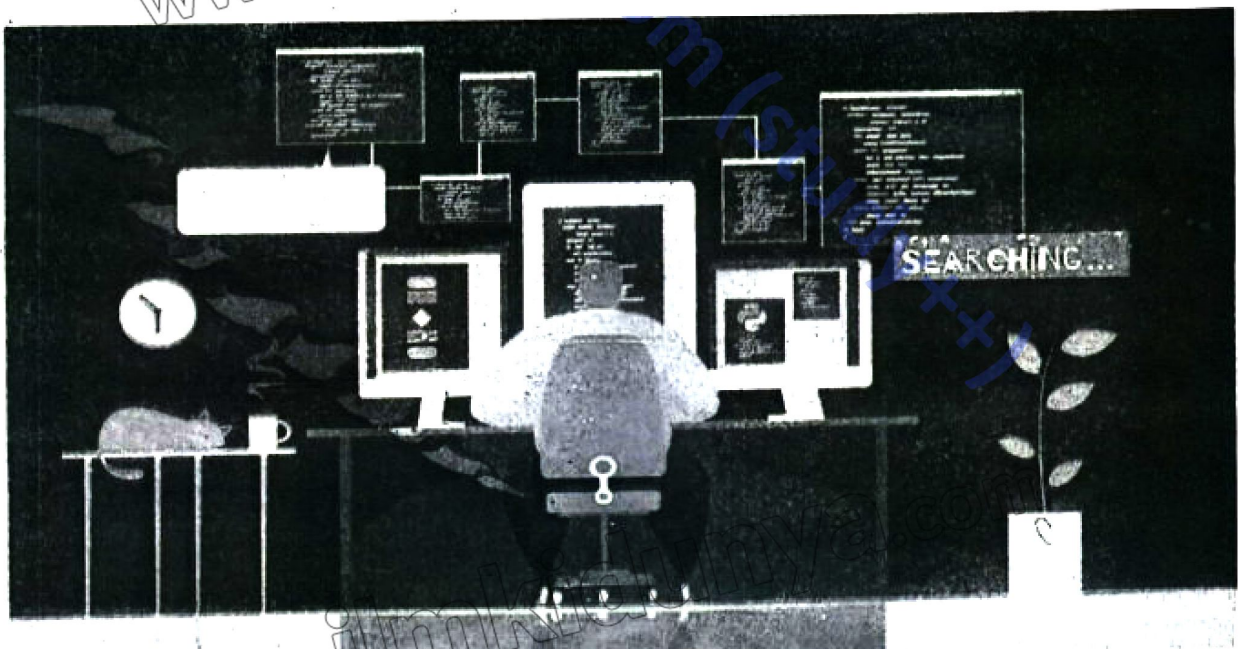


Unit 03

PROGRAMMING FUNDAMENTALS

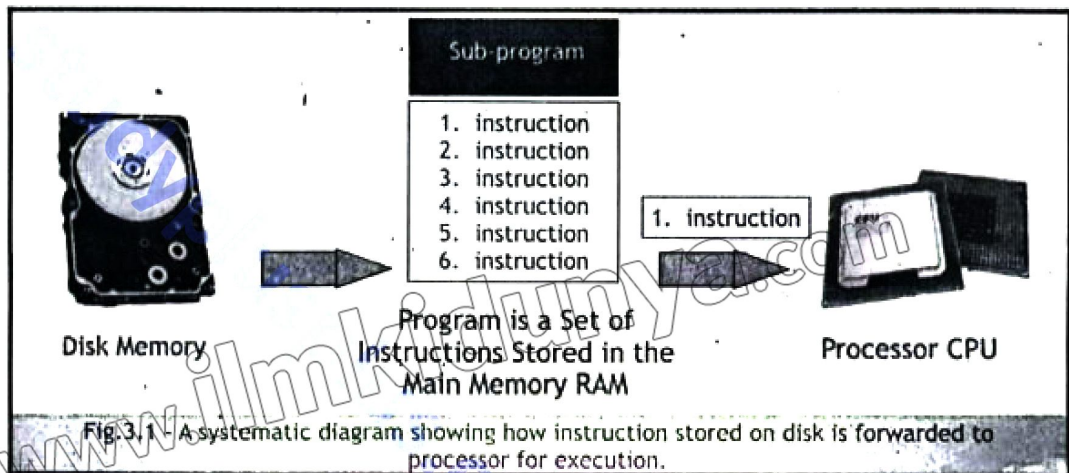
After completing this lesson, you will be able to:

- understand the importance of computer programming and applications
- write and execute simple programs in Python.
- draw shapes using Turtle Graphics functions in Python
- understand the need for libraries and learn the use of some simple libraries in Python.
- translate simple algorithms that use sequence and repetition in Python.
- decompose a problem into sub-problems and implement those sub-problems using functions in Python
- determine ways of debugging their code in Python



3.1 Computer Program

A set of well-defined instructions which a computer can execute to solve a problem is called a computer program. The program interacts with the user for some input, then executes one instruction at a time from set of instructions (called code) and delivers the result. In its simplest form, the program requires one algorithm to build but large programs can contain multiple algorithms for various tasks in a single program. When the program is executed, the program is loaded into main memory from where one instruction at a time is fetched and provided to the processor which executes it, as shown in Fig 3.1. Keep in mind that the program itself and the data it operates upon both are kept in the main memory for faster execution

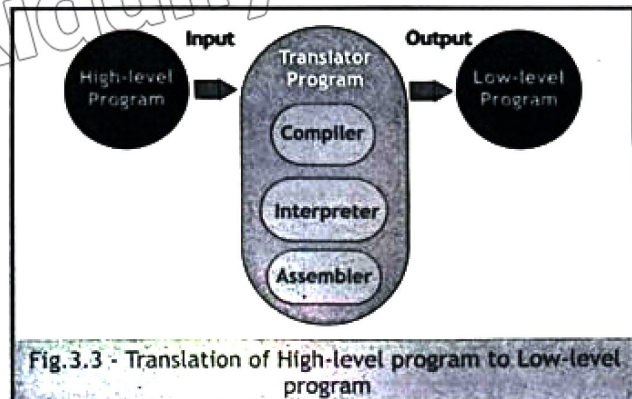


Generally, computer programs are categorized into interactive and batch programs. The former, accepts some input which is either from a user or some other program, like a web browser. Whereas the latter can be executed by user or an interactive program, where multiple commands are bunched together which are processed sequentially and then halts. For example print jobs on a printer lines up in a queue, processed one after the other and halts when no more jobs are in queue, as shown in Fig 3.2.

Computer Programs are developed in programming languages which provide the development environment via set of rules and keywords to use. With the help of keywords, you can create set of instructions (called code) which need to follow the rules of syntax of the said programming language. Thereafter, when you compile your code the compiler checks whether the syntax is as per the rules and converts your code into an object file, which comprises of binary data and is called machine



language, as shown in Fig 3.3. The machine language is understandable to the processor and executes your code. Examples of such programming languages which use compiler are C++, Java, Assembly Language, Fortran etc. Apart from compiler, languages like Python, JavaScript and Ruby use interpreter which are more flexible in a sense that the code written in high level language is converted into machine language line-by-line, as the code executes.



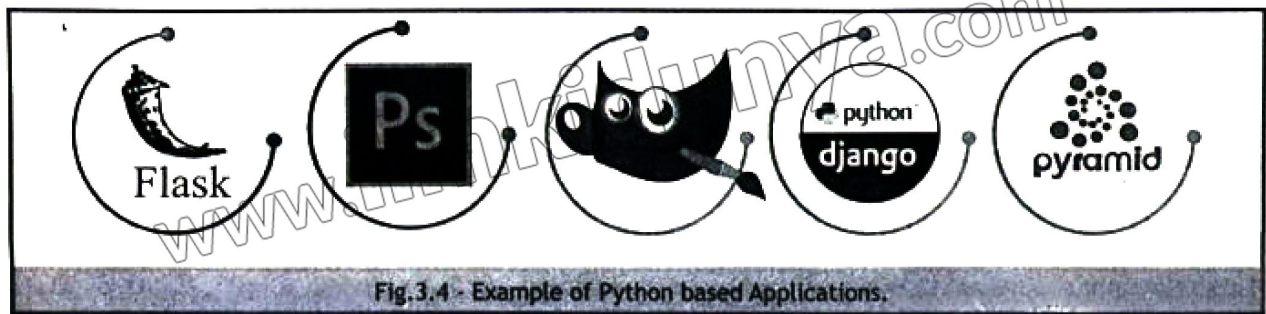
3.2 Python: An Overview

One of the most popular and widely used programming languages today is Python. Python is used for data analytics, machine learning, and software development. Python is a high-level, object-oriented (data-driven), and human-friendly (easier to grasp) computer language. It was first introduced in 1992 and is designed to be reasonably simple to write and comprehend. As a result, it is the perfect coding language for individuals looking for rapid development. Python's widespread use is due to a variety of factors, including:

- Python may be a great starting point for individuals who are unfamiliar with the art of programming, due to its simplicity.
- Python's syntax is quite similar to that of English, making it very simple to read and comprehend. You can determine what each line of code does just by looking at it.
- Python is an open-source programming language, meaning anybody can use it and help is available on different sites and platforms, as developers around the world share their experiences.

Python allows you to program a wide range of applications. Data scientists frequently use Python, which has a large number of machine learning and artificial intelligence tools and packages. Python programming is used in the quickly evolving field of data analytics. People who can gather, manage, and arrange information are needed at a time when we are producing higher quantities of data like never before. Python is excellent for web development, mainly because there are several Python website development frameworks available, like Django, Pyramid, and Flask. Python may be used to create apps for graphic design as well, as show in Fig 3.4.

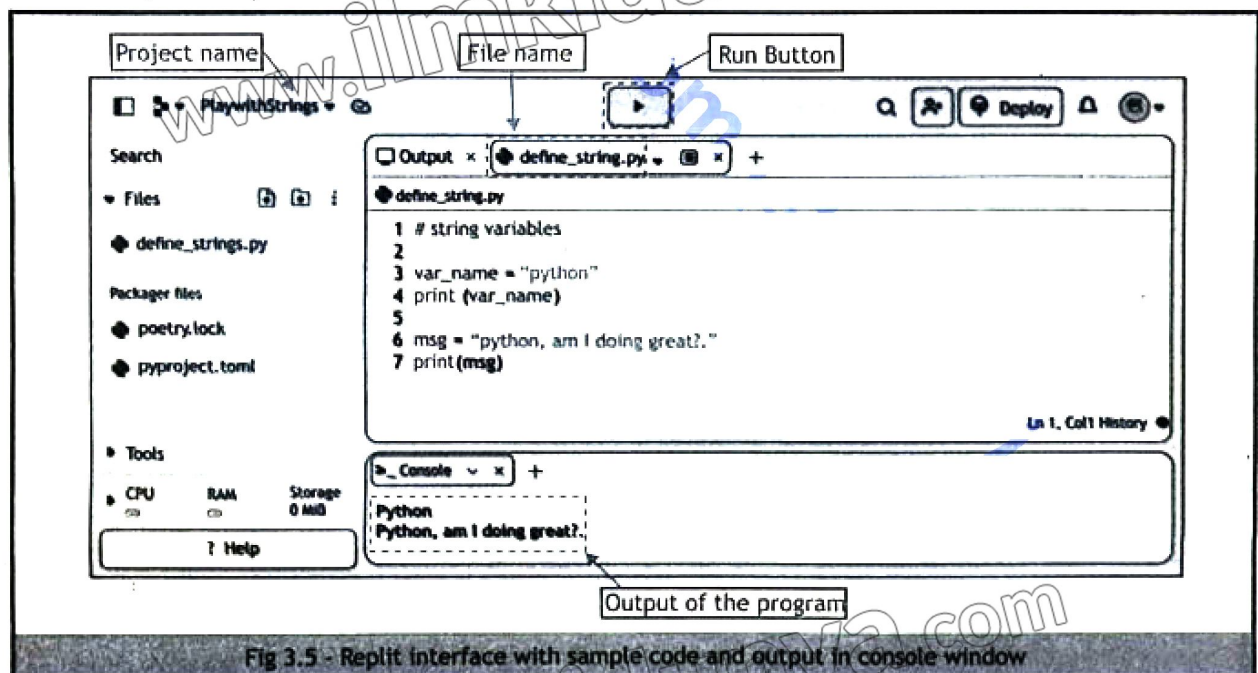
Python is also employed extensively in the creation of online educational materials and programs. For novices, learning this programming language is simple since it has an English-like syntax. It makes the learning process easier by providing a beginner with a standard library and a number of resources to comprehend the language. This is probably the main reason that Python is considered by novices as first choice to enter the programming paradigm. Last but not the least, Python programming language may be used to make small games, making it a helpful tool for quickly creating a prototype.



3.3 Python Integrated Development Environments

Python Integrated Development Environment (IDE) can be downloaded as IDLE from <https://www.python.org/downloads>. It has a simple command line interface with an interactive console. This means you can give commands on the console prompt as well as write your whole code in the form of a block, save in a file and run it.

Besides this, there are online IDEs available through which you can write and run your code, without installing it on your computer. One of them is Replit, which also provides an interactive console and easy to use environment, which are feasible for small blocks of code, especially for beginners. You just create your login or you may use your Gmail account to log in. Create a REPL (Read-Eval-Print-Loop) by pressing the 'Create Repl' button. Create a new project and assign a name to it, else Replit will provide one by default.



You can write code in the code area and then press the green button which runs the program. The corresponding output is displayed in console area.

A sample code of two variables are assigned string data type values and printed in Replit out, as

shown in Fig 3.5. The filename is 'define_strings.py' and respective project is titled 'PlaywithStrings'. Assigning such meaningful names to files, folders and projects is a good practice. This way, even without opening the file, you will recall what the code in the file is all about.

3.4 TURTLE GRAPHICS

The turtle graphics is installed with Python and we have just to invoke it and use it. No need to install anything to use it. It provides functions which can be easily used even from the shell prompt. Through these functions we can design basic drawings and two dimensional geometrical shapes. It is easy to use and provide interactive session and motivation for the novice. Nevertheless, turtle provides advanced features which are quite useful for the development of animations, simulations and computer games.

On the Shell, first statement needs to be "Import turtle". This command will invoke the turtle library and all the functions of the said library can now be used. The very next line as shown in figure is "turtle.Screen()", which loads the canvas of the turtle graphics. As soon as, a white canvas loads, you will be able to see a '>' which is the pen that will draw our lines and shapes. This pen is called turtle and correspondingly the name turtle graphics arose.

As line 3 highlights that we can resize the canvas and its color. Keep in mind that all measurements that are used in turtle graphics are in terms of pixel. To emphasize this, we print a dot of blue color of size 10 pixels in line 4. Another important point to remember that every instruction is executed from the point where the pen is currently located. Line 5, reflects that by default the pen color is "Black", as the circle() function is used and the circumference of the circle is drawn in black. Additionally, the circle() function takes radius as input and draws the resultant circle. To distinguish, we next change the pen color to "Blue" and draw another circle, whose resulting circumference is of blue color, as shown in Fig 3.6.

```
1. import turtle
2. turtle.Screen()
3. turtle.screensize(200,150,"coral")
4. turtle.dot(10,"Blue")
5. turtle.circle(50)
6. turtle.pencolor("Blue")
7. turtle.circle(-50)
```

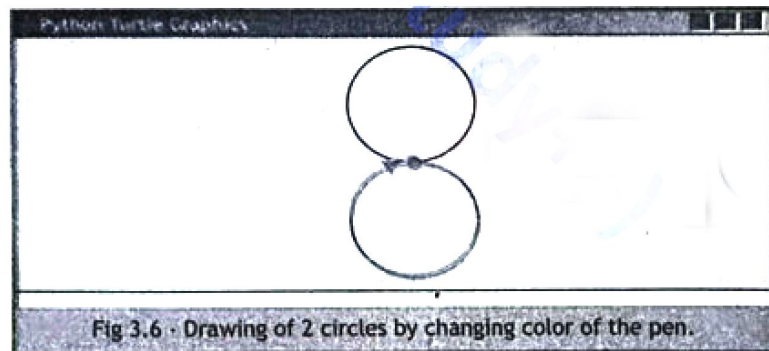


Fig 3.6 · Drawing of 2 circles by changing color of the pen.

To draw a triangle, you need to play with angles. The drawing of first line with forward() function is easy, as earlier. Thereafter, depending upon what type of triangle you want to draw, you will set the angle and length of line accordingly. This way, you will be thankful to your school teachers who taught you all the trigonometric formulae and the resultant shape of the triangle is visible on screen, as shown in Fig 3.7.

1. turtle.reset()
2. turtle.forward(140)
3. turtle.left(135)
4. turtle.forward(100)
5. turtle.left(90)
6. turtle.forward(100)
7. turtle.left(45)
8. turtle.left(90)



Fig 3.7 - Example of another linear shape drawing

To clear up the canvas, “turtle.reset()” is used and now you may have your next drawing. This time, let's use the most basic line drawing functions of `forward()` and `back()`. These functions take distance in pixels that how far you want to draw a line and in which direction. By now, it should be obvious enough that `forward()` will draw in the direction of the pen and `back()` will draw in the reverse direction of the pen. So, as the code shows, the `forward()` function with 100 pixels is called which draws a line and thereafter, `left()` function is called which changes the direction of pen. The functions `left()` and `right()` take input in terms of angles. Though, the standard of angles is radian, but since degrees are the more frequent in use, therefore Python by default uses degrees, but also provides the provision for radians. Hence, moving forward and rotating at 90 degree angles, we have created a basic layout of a car. To provide the car with tyres, let's use the above example scenario and adjust the size of circle accordingly. After, the first tyre is drawn, let's use the `penup()` function and relocate your pen. Next, `pendown()` function is called and the pen is again ready to draw the second circle, as shown in Fig 3.8.

1. turtle.forward(100)	18. turtle.left(90)
2. turtle.left(90)	19. turtle.forward(25)
3. turtle.forward(25)	20. turtle.pendown()
4. turtle.left(90)	21. turtle.circle(-10)
5. turtle.forward(25)	22. turtle.penup()
6. turtle.right(90)	23. turtle.right(90)
7. turtle.forward(25)	24. turtle.forward(10)
8. turtle.left(90)	25. turtle.dot(5, "Blue")
9. turtle.forward(50)	26. turtle.back(10)
10. turtle.left(90)	27. turtle.left(90)
11. turtle.forward(25)	28. turtle.forward(50)
12. turtle.right(90)	29. turtle.pendown()
13. turtle.forward(25)	30. turtle.circle(-10)
14. turtle.left(90)	31. turtle.penup()
15. turtle.forward(25)	32. turtle.right(90)
16. turtle.pencolor("Blue")	33. turtle.forward(10)
17. turtle.penup()	34. turtle.dot(5, "Blue")

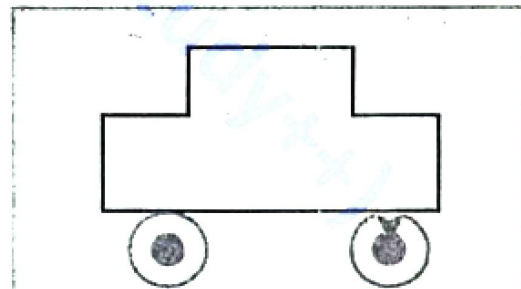


Fig 3.8 - Drawing a shape, like a car, by lifting the pen in between.

Turtle graphics is easy to learn and simple to use. Later in the chapter, when we will explore Selection and Iterations, using which you will be able to use better and complex shapes with comparatively lesser number of lines of code.

3.5 Libraries

A set of code which are meant for a particular task more proficient, is called library. We can produce result, with just one line of code via using a library, rather than writing our own tens of lines of code. For example, as discussed earlier, the Turtle library is already available in Python, which provides already written and tested functions that we can use. We simply have to import library and use from the available functions, with the right set of arguments. For any additional library, it can be downloaded and imported similarly.

By default date is not the type of data supported by Python, but 'datetime' library can be downloaded and imported. This way, date and time in different formats can be accommodated.

Rather than manually installing a package and trace for missing dependencies by going through all the required files one by one and later install them as well, package managers are quite useful. Pip is a package manager for installing Python packages which is simple to use and assures that all dependencies of the said package are also installed.

For example, if you want to install "datetime" package to avail various options to display date in Python, simply type in command prompt:

```
>pip install datetime
```

So, let's consider the first example which displays the current date and time. The date returns year, month, day of the week along with the time till millisecond as shown in Fig 3.9 (a).

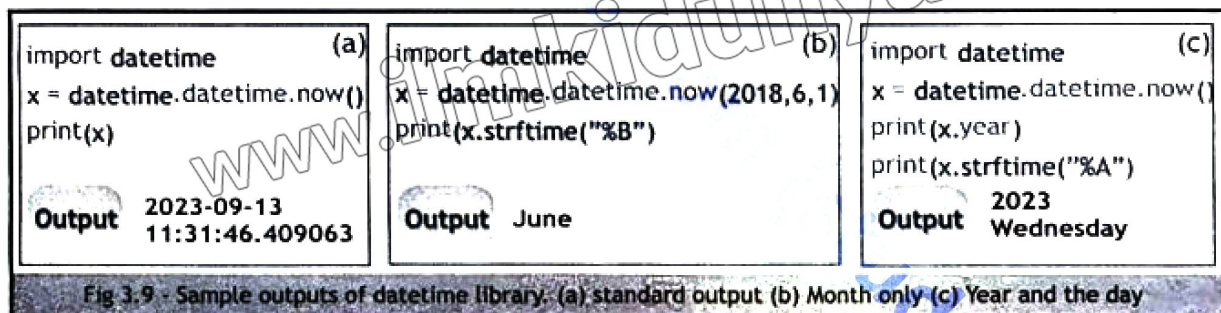


Fig 3.9 - Sample outputs of datetime library. (a) standard output (b) Month only (c) Year and the day

Second example in Fig 3.9 (b) shows how to extract year from the date, in line 3. Line 4 is of special consideration, which uses 'strftime()' function and formats according to the argument. The strftime() function takes the previous example's output and treats it as a string. This way, based on the argument, the function is able to extract the particular output. In the Fig 3.9 (c) argument '%A' returns the day. Similarly, '%B' argument returns the name of the month, only. The most common arguments used for date-time format, are listed in Table 3.1.

Table -3.1- Sample list of arguments available for use in the library		
Argument	Description	Example
%A	Weekday, full version	Friday
%B	Month name,full version	August
%C	Century	19
%d	Day of month 01-31	21
%H	Hour 00-23	18
%I	Hour 00-12	03
%S	Second 00-59	48
%Y	Year, full version	2023

3.6 Python Variables

A variable stores some information which when allocated in the program, can be used later on. For example, rather than solving the same equation which is used multiple times in a program, we just store the resultant in a variable, and use it without recalculating the formula. A good programming practice is to keep meaningful variable name.

The name of a variable:

- can be alphanumeric i.e. combination of alphabets and numbers, but cannot start with a number.
- are Case Sensitive, i.e. 'Num' and 'num' are treated as two separate variables.
- cannot contain spaces, so a good technique is to use underscore ("_"), like 'Temp_var' which reflects a meaningful message that the value assigned to this variable is temporary.

Generally, instructions are written in the order they are supposed to execute, in all programming language. This way, it is easier to apply the logic which is meant to be implemented via those instructions. So, if for some reason, the sequence of instructions changes, it may result in undesirable results. And when you look again at your code the syntax of the code seems all right but the instructions are not behaving according to your expectations. Such a scenario is called **Logical Bug** or **Bug**. Bug is hard to detect just by looking at the code and this capability comes with experience. But the simplest method is to **Dry-Run** your code.

Let's write a code to swap the values of two variables, namely 'x' and 'y', with the help of a temporary variable 'temp', as shown in Fig 3.10. For some reason, if the sequence changes like as follows, then the corresponding output will change too. Hence, both x and y will have same values. In other words, if the sequence of instructions are not carefully structured and placed, the desired output is not likely to happen and data may be lost.

A variable needs to be defined as what type of data it can hold and thus programming language needs to define support for various data types. Python also provides support for different data types by default. Numbers are stored in numeric data types, like int, float. Sequence of characters are handled by string data type. To manoeuvre a series of similar values, list can be used represented by "[_,_]". Tuple is identical to list, but are immutable, i.e. once defined the values in a tuple cannot be changed. A tuple holds the values in "(_,_)".



Activity 1

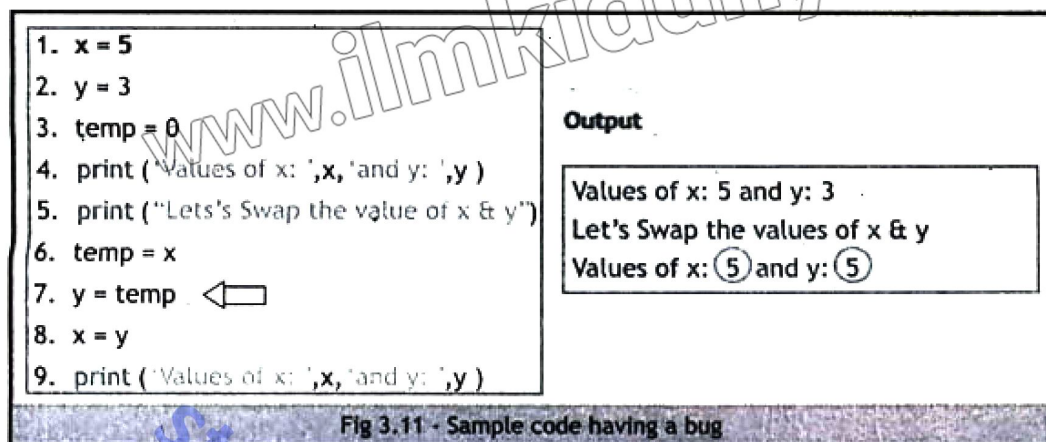
In context of Fig 3.10, rewrite code to swap the values of 2 variables without using temporary variable.

```
1. x = 5
2. y = 3
3. temp = 0
4. print ('Values of x: ',x,'and y: ',y )
5. print ("Let's Swap the value of x & y")
6. temp = x
7. x = y
8. y = temp
9. print ('Values of x: ',x,'and y: ',y )
```

Output

```
Values of x: 5 and y: 3
Let's Swap the values of x & y
Values of x: 3 and y: 5
```

Fig 3.10 - Code to swap the values of 2 variables



So, let's try to fix the code of Fig 3.11 by Dry-Run, as shown in Table 3.2. Dry-Run is where you execute your program on paper, and keep track of the variables. Initially, we break down our code. First 3 lines are variable initialization. Line 4 and 5 are print commands and so is the last line. Thus, we are left with lines 6-8 which need to be focused on and examined.

Table 3.2 - Table showing the values of variables during a Dry-Run

Line	Variables		
	x	y	temp
Before 6	5	3	0
6	5	3	5
7	5	5	5
8			

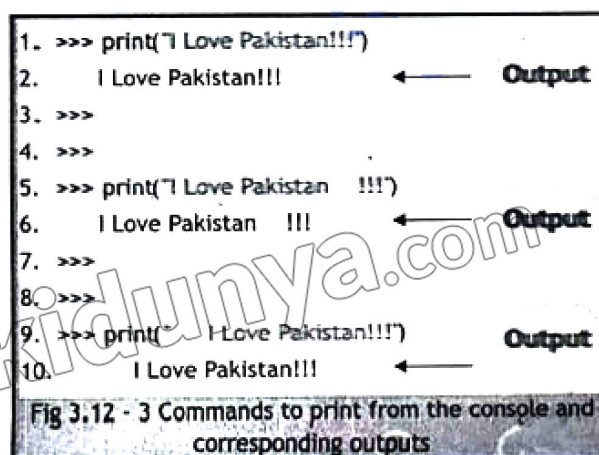
In view of that before executing line 6, the values of the three variables are the same as initialized. In line 6, value of temp gets updated to 5. As soon as, we check line 7, we are able to highlight that value of y updates to 5, which though is the desired one but all other variables have the same values, as well. Hence, value of '3' is lost and cannot be assigned to x.

As a result, we now know where the problem lies, and we can fix it.

3.7 Python Input/Output (I/O)

The Python Shell is interactive and accepts commands. So, result of arithmetic operations and desired output using the print function can be used. Parenthesis followed by print, i.e. print() function instructs Python to output whatever is inside the parenthesis, within quotes. Following are 3 outputs shown to emphasize that even if a space is entered before or after the sentence within the quotes, it will be displayed as it is; as shown in Fig 3.12.

Computer Programs are meant to be processed while taking some input and generating



corresponding output. User can enter data through `input()` function, via which data will be stored in a variable. This way, the same input does not need to enter again and again by the user and whenever the data is required, the corresponding variable is used. In the following example, a variable named 'country' takes the input in the form of character-string and the same variable is displayed using the `print()` function, in the next line. Steps are show in Fig 3.13.

```
1. country = input ( 'Enter the name of your country: ' )
2. print ( 'I love ', country)
```

Output

```
Enter the name of your country : Pakistan
I love Pakistan
```

Fig 3.13 - Command to take input from the console and respective output

Python provides basic variable types as char, integer, float, boolean, etc. The `input()` function treats the entered data as a set of characters. Hence, if a numeral is entered, it needs to be converted to number, explicitly using the `eval()` function and can be used in conjunction with `input()`. Fig 3.14 highlights the importance of `eval()` function while dealing with numbers. In Python, `eval()` function evaluates expressions which are generally mathematical in nature. However, it also assesses strings/ set of characters and responds with an integral value, as highlighted in Fig 3.14 where with `eval()` the input is treated as integer. Note that the first output shows that when two variables are added, while the values are characters the add function joins the values rather than adding them up. The respective terminology is 'concatenation' of two strings or characters to merge them up.

```
1. num = input ("Enter a Number: ")
2. print ( "Double of the said number is", num+num)
3. num=eval (input("Enter a number:"))
4. print ("Double of the said number is", num+num)
```

Output

```
Enter a Number: 123
Double of the said number is: 123123
```

```
Enter a Number: 123
Double of the said number is: 246
```

Fig 3.14 - Code highlighting how Python treats input



Activity 2

Take three inputs as day, month & year, e.g. your date of birth, store in three variables and print in the form of:

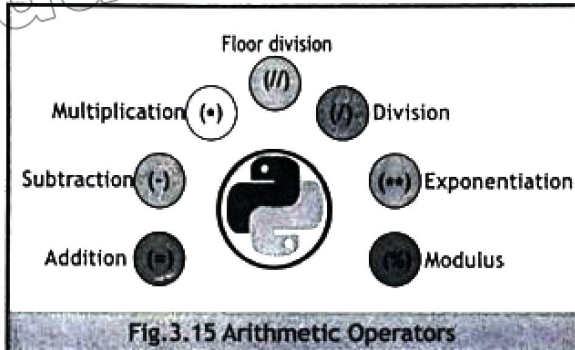
- 13-09-2023
- 09-13-2023
- 2023-09-13

3.8 Operators in Python

3.8.1 Arithmetic Operators

Other than the assignment operator (=) and normal arithmetic operators i.e. addition (+), subtraction (-), multiplication (*) and division (/); as shown in Fig 3.15, Python provides the option of:

- Modulo operator (%) which returns remainder of a division. e.g. $5/4=1.25$ but result of $5\%4=1$.
- Integer division (//), e.g. $5//4$ will return 1, only integral value and will discard the decimal portion.
- "**", which is used for exponentiation.



Python considers operator precedence in the sequence of: *,/,//,%,+,-. Operators having same precedence are evaluated from left to right. However, parentheses have the highest precedence and will be evaluated first.

The assignment operator can be used in Python with arithmetic operators, as well. e.g. $i=i+1$ can also be written as $i+=1$, $i=i-2$ as $i-=2$, $i=i*3$ as $i*=3$, etc.

3.8.2 Bitwise Operators

Python provides operators for bitwise operations, which are used for comparison purposes of binary numbers. An overview of these is already discussed in chapter 1, under the heading of 'Basic Logic Gates'.

- AND (&) - Bit is fixed to 1 if both comparing bits are found as 1.
- OR (|) - Bit is fixed to 1 if at least one of the comparing bits is found as 1.
- XOR (^) - Bit is fixed to 1 if only one of the comparing bits is found as 1.
- NOT (~) - Flips all the bits, i.e. 0s are flipped to 1s and vice versa. It is a single operand operator.
- Left Shift (<<) - Shifts all the bits one place to the left, while the left most bit is discarded and right most bit is set to a 0.
- Right Shift (>>) - Shifts all the bits one place to the right, while the right most bit is discarded and left most bit is set to a 0.

3.8.3 Membership Operators

Membership operators in Python are used to check whether a specified value either exists in an Object or not. The operators are 'in' and 'not in' and return true if a value is present and not present, respectively.

3.8.4 Comparison Operators

Consider the scenario, where you need to log in to your system/ cell phone and you enter password. If the password matches, you will be logged in else it will ask you to re-enter the password. Similar is the case with the programs, we develop. In a normal execution of the program, the instructions are executed one-by-one in a top-down manner and the program should terminate properly. But, there may be scenarios where the program has to make a selection based on some conditions. For someone reason, if the program is unable to execute the next instruction, the program will reach state of 'hang' and will not terminate properly.

In the system log in scenario, if you have setup the password as 1234 but while logging in a typing error occurs and the password you entered is 1123. So, what should the log-in program do? Obviously it should compare the typed-in password with that of saved one, if it does not matches it should ask the user to re-enter the password. Alternatively, it should allow the user to log-in anyways. What do you think?

3.8.5 Logical Operators

The flow of the program is controlled via conditional statements which rely mainly on the comparison statements, the result of which either positive or negative will glide the program in respective direction. However, there are scenarios where more than one comparisons needs to be placed in a single conditional statement. For example, checking of a value for lower and upper limits, like mathematical constraint of $(1 < x < 10)$ can be incorporated in a program code like $(x > 1)$ and $(x < 10)$. So, to incorporate such multiple constraints in a single conditional statement, logical operators are used. Python provides the support of 3 types of logical operators, as:

- and - all conditions needs to be met, for a true outcome.
- or - any one condition needs to be met, for a true outcome.
- not - negates the outcome, i.e. opposite of the outcome will be the result.

The code in Fig 3.16 initializes three variables and in the following lines checks each of the variables whether their respective conditions are met in terms of the three logical operators. This way, the use of operator and corresponding outputs are highlighted for better understanding.

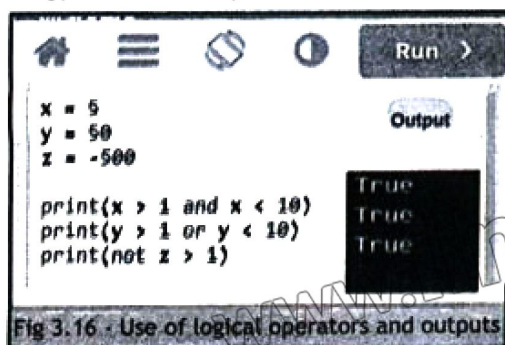


Fig 3.16 - Use of logical operators and outputs

Table 3.3 - List of Comparison Operators		
Operator	Name	Example
==	is equal	x==y
!=	is not equal	x!=y
>	Greater than	x>y
<	Less than	x<y
>=	Greater than or equal to	x>=y
<=	Less than or equal	<=y



Teacher's Guide

For quick reference or checking output of a code-block, online Python interpreters can be used, like:

- www.online-python.com
- www.onecompiler.com/python
- www.pythononlinecompiler.com

3.8.6 Conditional Statements

Let us consider another example, where we take a number, input from the user and we have to determine whether the entered number is even or odd. So, we apply comparison with the help of 'if' statement and based on the result of the condition/ comparison, we will display our result to the user. For such comparisons, Python provides us with the six comparison operators, as shown in table 3-3.

Now, we can write a program as shown in Fig 3.17, to determine whether a number is even or odd.

<pre> 1. num=eval (input("Enter a Number: ")) 2. if(num%2==0): 3: print("Entered Number Is Even. ") 4. if(num%2==1): 5. print("Entered Number Is Odd. ") </pre>	Run-1 Output <pre> Enter a Number : 7 Entered a Number is Odd. </pre>	Run-2 Output <pre> Enter a Number: 4 Entered a Number is Even. </pre>
---	---	---

Fig 3.17 - multiple 'if' statements.

The first line of program is same, as the last example program. Next, we put an 'if condition' where we compare two values, i.e. if remainder of the number when divided by two is zero, then it is an even number. Remember to put a ":" after the condition, in an 'if' statement. Then we repeat the same process to check whether the number is odd. Let's run the program twice and by entering an even and odd number, check whether the conditions are working. This way, we know now that both if conditions are working. Additionally, we also learnt that if the condition does not matches, the corresponding 'if statement' is not executed.

IF-ELSE

Instead, a good choice to write the same code is as follows, where we check for a condition in 'if statement' and the alternate condition is handled by an 'else:' statement, which handles any other possible outcomes, as shown in Fig 3.18.

<pre> 1. num=eval (input("Enter a Number: ")) 2. if(num%2==0): 3: print("Entered Number Is Even. ") 4. else: 5. print("Entered Number Is Odd. ") </pre>

Fig 3.18 - 'if-else' statement.

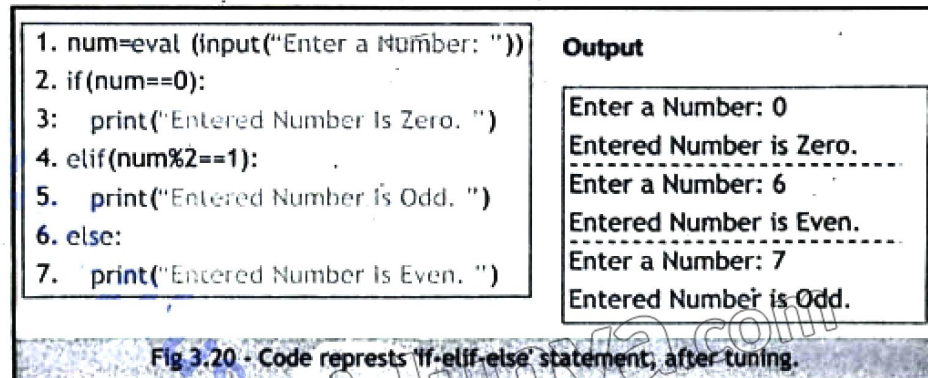
IF-ELIF-ELSE

Let's take a step ahead. Some people treat Zero neither as even nor odd. It means, now we have 3 possibilities. The last program will treat zero as even integer. So, let's change it by incorporating 'elif' which is short of 'else-if statement'. We changed the above program and tried to incorporate all possible outcomes of the program by using 'if-elif-else', as shown in Fig 3.19.

<pre> 1. num=eval (input("Enter a Number: ")) 2. if(num%2==0): 3: print("Entered Number Is Zero. ") 4. elif(num%2==1): 5. print("Entered Number Is Odd. ") 6. else: 7. print("Entered Number Is Even. ") </pre>	Output <pre> Enter a Number: 0 Entered Number is Even. </pre>
---	--

Fig 3.19 - 'if-elif-else' statement, which does not provide the desired result.

However, it did not work, and the following outcome was displayed. What could have gone wrong? Let's discuss. We checked the condition if remainder is 0, then it is even. But since our number is zero, so is its remainder. And hence, this highlights that in 'if-elif-else' statement. If all other fail, only then the last part executes. Let's change our program accordingly where we check first if the input from user is zero, else-if the remainder is 1 for odd, otherwise it will be an even number; as shown in Fig.3.20. Corresponding output after running the program thrice, to check for all possible answers:

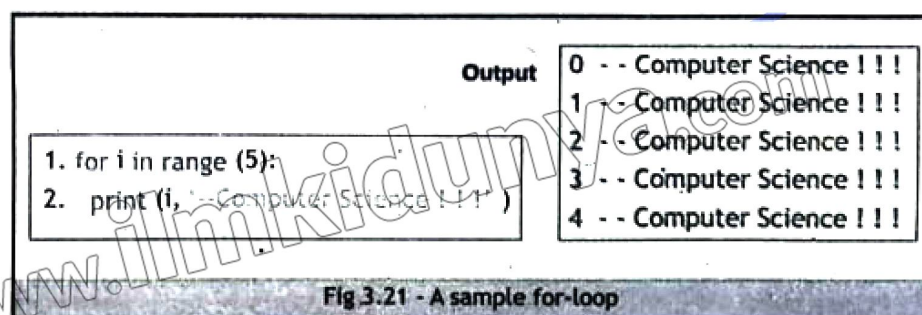


Alternate method is to use logical operators '&&' (and) '||' (or) and '!=' (not), to combine multiple conditions in 'if-else' statement. This we leave as an activity for practice.

3.9 Iteration and Loop

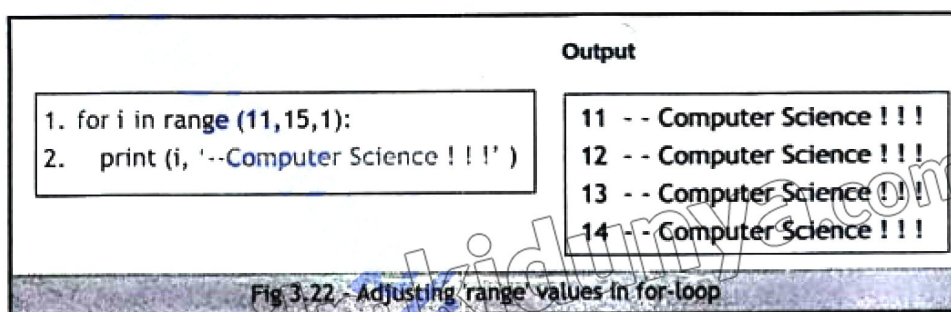
Computer programs are the most useful in daily life where same task is to be executed in iterative fashion. Rather than write code for every part of same iteration, incorporate iterations in the programming language, to write small code for iterations and execute effectively. The most commonly used is the 'for loop'.

In 'for loop', we need to assign the starting point and the terminating condition, and the loop will execute the instruction(s) for the defined range. For example, you want to display your name five times. So, rather than writing the print() function with a subject name five times, better alternative is to place the print() function once within the for loop and define the range to be five; as shown in Fig 3.21.



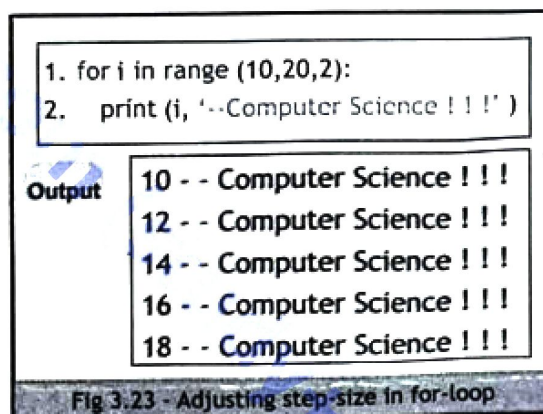
The output reflects that by default, the index (i) of the for loop starts with 0 with step of 1. Alternatively, we can define the starting point and terminating condition along with the step-size for a 'for loop'. Here another important aspect to note is the indentation of the code, which not only provides better readability and visibility of the code as the lines of code increases but also helps in outline the composition of the code and ultimately the overall program. It is quite critical to follow the 'indentation rules' in Python, as the line/ lines of code if not correctly indented can result in logical bugs and unforeseen outcome of the program. The ':' after the for loop statement shows that the lines of code that follow are to be executed in the loop; and hence need to be indented for the understanding of Python interpreter.

The program in Fig 3.22 starts the loop index from 11, terminating value is provided to be 15 with a step-size of 1.

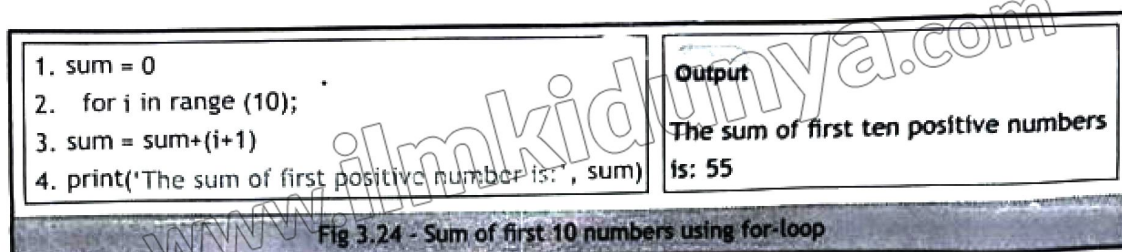


When the terminating condition in a loop met, the loop stop. In other words, the value we provide for terminating condition of the loop is not inclusive. As far as the step-size is concerned, we can provide any non-zero value, according to which the index value will update itself. Fig 3.23 shows a for loop with step-size 2.

For the sake of practice, you may explore for-loops further, by assigning a negative value to step-size, in range function. Thus, for any series and sequence, the value of range can be set accordingly.

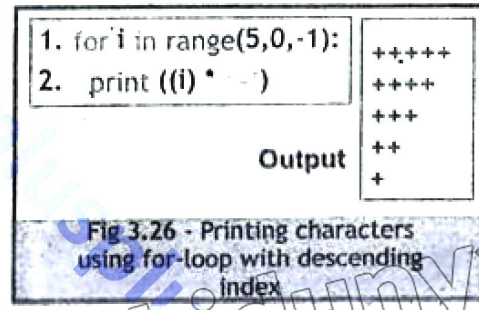
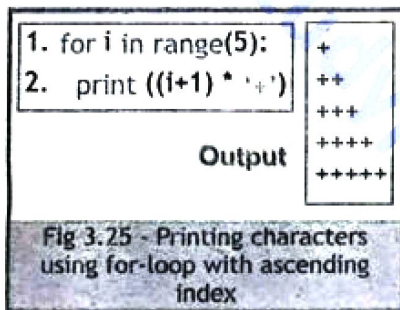


After understanding the syntax and corresponding outputs, let's explore in what other ways, we can utilize 'for loop'.



In the Fig 3.24, first, we initialized a variable namely 'sum' to 0. Next, we set the range of the for-loop and with a single statement; we were able to sum-up the first 10 positive integers. Just, change the range value to 100 and the same program of 4 lines of code, will calculate the sum of first 100 positive integers.

To emphasize, how to use index effectively, programs in Fig 3.25 & 3.26 use a character which is displayed, based on the index number in iterations. The range value defines the height of the triangles in both these figures which will execute 5 times. Since, the index starts with 0, therefore index is incremented before multiplying with character to be displayed as 'i+1'. Otherwise, the first line would have been blank as index value of 0 multiplied by anything will result in nothing. Consequently, the same print statement reflects an additional character in each line, specially in first and last line while using descending index. So, in the latter, 'i+1' is not required.



Activity 3

Activity: Take input from the user which determines the height of the diamond and prints a hollow diamond (◇).

3.10 List

Now that we have a pretty good idea about variables and how to use them for I/O, let's take things to the next level. Assuming that in a class of 25 students, the teacher takes a test and marks are entered in a computer program. So 25 variables are required to store the numbers of each student in the program. In case of increase in number of students, additional variables need to be created and remembered. Though, it is correct but now effective specially how many variable names can we create and keep track of? Python provides a data type of 'List', which is alterable and can contain many values.

A list is defined with a name assigned with '[]', later we can assign the values. Alternatively, it is a good idea to initialize the list at the time of creation; if user input is not desired to run the program. For example,

```
Sngl_Dgt_Odd_Nums = [1,3,5,7,9]
```

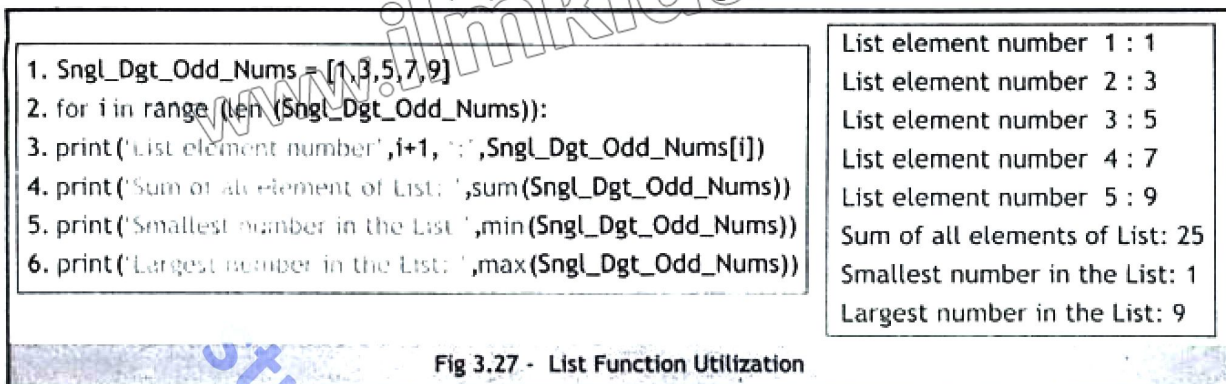
Printing the list is simply straight forward, we have to pass the name of the list to print() and the whole list will be displayed. However, if a particular value needs to be accessed or displayed, the corresponding index of the list can be directly accessed and needs to be assigned like:

```
Print(Sngl_Dgt_Odd_Nums[2])
```

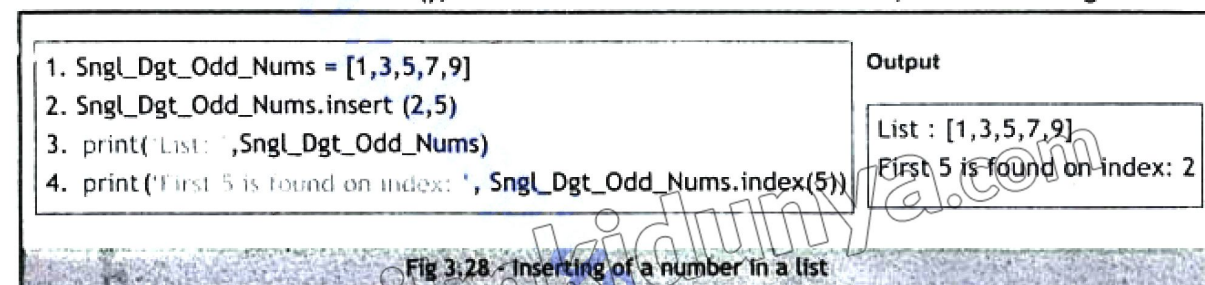
and the output will be '5', which is the value on 2nd index where index value starts from 0.

Another aspect of using list is that it provides some basic functions via which handling and manipulating a list is fun to learn and use. The len() function provides with the total number of

elements in a list. The `min()` and `max()` functions can be used to extract the smallest and largest element of the list, respectively. While `sum()` function when applied to the list, sums up all the elements of the list and result is displayed as shown in Fig 3.27.



Another useful function is `insert()`, which inserts an element in the list, as shown in Fig 3.28.



A list can have duplicate values, too. If we insert another value of 5, other than the one at index number 2, let's say at the end of the list; the list will now hold more than one values of 5, i.e. on index numbers 2 and 5. In Fig 3.28 the `index()` function on the last line returns the position of a number in the list. In case of multiple occurrences of the said number, the placement of first match is returned.

Next, let's introduce some dynamicity in our code by introducing random numbers, as shown in the first line of the following code. Among the modules that Python provides for convenience, is the `random` module which consists of many inbuilt functions like `randint()` and can be used by importing in the code, with the 'import' keyword. Once defined in the start of the program, the function can be used anywhere in the program to generate a random number, in the given range. Like in line 4 which is inside a for-loop that will be executed 10 times where in every iteration the `randint()` function will generate a number between 1 and 99.

Activity 4

- Take a random 2 digit number.
- Subtract it from 100.
- Calculate the difference.
- Print the difference (magnitude only).

Tip

`abs()` function returns the value/magnitude of an integer only and discards "-ve" sign. This means that `abs()` will always returns a positive number

So, initially an empty list is defined and with the help of for loop, the list is populated with 10 random numbers. The outputs of running the same code shows different randomly generated values in each execution, as shown in Fig 3.29.

```
1. from random import randint
2. Rnums = [ ]
3. for i in range (10):
4.   Rnums.append(randint(1,99))
5. print(Rnums)
```

Output

```
[ 66, 1, 6 , 15, 83, 8, 73, 15, 30, 7]
-----
[ 94, 49, 96, 16, 97, 57, 24, 5, 87, 10]
-----
[ 79, 36, 92, 53, 86, 94, 36, 68, 34, 29]
```

Fig 3.29 - Code for Generating random numbers and adding to the list

Let us extend the same code, where we initially populate random numbers and later, try to find a particular number in the list. The two possible outcomes are either the number is in the list or not. In Fig 3.30 shown is the code along with the both possible outcomes.

As discussed earlier, the membership operators can be used for checking whether a particular element exists in an object, like list. For better understanding, in Fig 3.31, a list is defined in the line 1 which consists of some members. In line 2, we check for whether the value 5 is in the list and corresponding result of either true / false gets printed. The last 2 lines are commented using '#' and are highlighted in a different color by the IDE. Since, the value of 5 exists in the list, therefore the output reflects 'True'. The membership operators are quite useful to control the flow of the program, dynamically by incorporating respective logic based on the values.

```
1. L = [1, 3, 5, 7]
2. print(5 in L)
3. # should return True
4. # because value 5 is in the list
```

Output **True**

Fig 3.31 - Membership operators in list

```
1. from random import randint
2. numseek = 7
3. temp = 0
4. Rnum = [ ]
5. for i in range (10):
6.   Rnums.append(randint(1,99))
7.   print(Rnums)
8.
9. for i in range (len (Rnums)):
10.  if(numseek == Rnums[i]):
11.    temp= temp + 1
12.  if temp >0:
13.    print( "7 is on index: ". Rnums.index(7))
14.  else:
15.    print('7 is not in the list')
```

Output [27, 20, 34, 56, 36, 48, 47, 13, 9, 12]
7 is not in the list

```
-----
[ 7, 81, 77, 83, 41, 67, 58, 75, 65, 97]
7 is on index: 0
```

Fig 3.30 - Code for Searching a number in the list

3.11 Functions in Python

So far, as you have experienced throughout this chapter that we started from a single line of code and now can develop tens of it. Additionally, via conditions and iterations we are able to not only write it effectively but also control the flow of the program. However, as the code lengthens it is hard to keep track of the logic you have applied, especially in case of if multiple algorithms co-exist.

So, the solution is to divide the code into segments through which it becomes handy to retain and understand large program. Each segment does a separate task and can be called upon to fulfill that particular task, if required from the main program or another segment. Such segments are called **functions**. In Python 'def' is used to define a function with a unique name given to each function, followed by ':'. For example,

1. Def my_first_func():

2. print('This line is printed, when my_first_func() is called')

```
1. def draw_line ():  
2. print( '_' * 10)
```

So, let's consider another simple example where we want to draw a line on the screen, which is required to be done frequently and hence we define a draw_line() function.

However, if we want to extend it further and divide the drawing of vertical lines too, we will define another function and change the names of the functions, such that they are meaningful and easy to portray about their respective working. Example code and corresponding output is as , as shown in Fig 3.32.

```
1. def draw_hline ():  
2. print( '_' * 10)  
3. def draw_vline ():  
4. print( '|')  
5. draw_hline  
6. draw_vline
```

Output

```
-----  
|
```

Fig 3.32 - A sample function definition to draw a line on console

As a next step, let's adjust the number of vertical lines to be two, such that one is in the start and other in the end of the horizontal line and place them just under the horizontal line. Since, the functionality of the function changed, so we may change the name of the functions as well, as reflected in the Fig 3.33.

Here, the draw_v_parallel_line() function for drawing vertical lines is called 4 times. The functionality is defined once in the function and the same result is obtained, every time. The draw_h_line() function is called in lines 5 and 10.

```
1. def draw_h_line ():  
2. print( '_' * 10)  
3. def draw_v_parallel_line ():  
4. print( '|', '|', *26, '|')  
5. draw_h_line()  
6. draw_v_parallel_lines()  
7. draw_v_parallel_lines()  
8. draw_v_parallel_lines()  
9. draw_v_parallel_lines()  
10. draw_h_lines()
```

Output

```
-----  
|      |  
|      |  
|      |  
|      |  
-----
```

Fig 3.33 - Multiple Functions in Program



Activity 5

Take a number as input from the user, using a function, print a mathematical table.

3.11.1 Arguments of Function

The beauty of defining function is that, the same functionality can be achieved for different values. This is done by passing arguments to the function. The function needs to be defined in such a way, that it accepts a value and stores it in a variable and uses it inside the function. The point to remember is that, the variable defined inside the function, does not exist outside the function and is called **local variable**; as its scope is limited to the function itself. And that is the very reason; to test a variable inside the function is easy as compared to the variables in the main program, called **global variables**.

So, let us define an argument in each of the functions, as shown in Fig 3.34. Arguments are variables and we can name them anyway we want, as per the rules defined by Python. As you might have observed that whenever the function is called, rather than a **variable**, a value is being passed. This way, the value is assigned to that argument in the function and is used inside the function. The output is same as that of Fig 3.33. Try it and experience it yourself.

To make our functions more **dynamic**, why not use the argument portion of a function, to its **potential**. Let's extend the number of arguments and also allow the functionality to print the character which is also passed in the **argument**. This way, each function now, accepts a character and a number and the function will display accordingly, as shown in Fig 3.35.

Note that, in both the functions, the **argument sequence** is not the same. This is to emphasize that sequence of arguments is the choice of the **developer**. But, good programming technique is to stick to the same sequence this way; fewer errors are encountered and are easier to call the function without referring to the function, again and **again**. The important aspect is the order of arguments defined in the function, should be the same while **calling** the function.

```
1. def draw_hline (n):
2.     print( '-' * 10)
3. def draw_v_parallel_line (m):
4.     print( '|', '|', '|', '|', '|', '|', '|', '|', '|', '|')
5. draw_hline(10)
6. draw_v_parallel_lines(26)
7. draw_v_parallel_lines(26)
8. draw_v_parallel_lines(26)
9. draw_v_parallel_lines(26)
10. draw_hlines()
```

Fig 3.34 - Function with Single Argument

```
1. def draw_hline (n,o):
2.     print(o*n)
3. def draw_v_parallel_lines (l,m):
4.     print( '|', '|', '|', '|', '|', '|', '|', '|', '|', '|')
5. draw_hline(10, '=')
6. draw_v_parallel_lines('||',24)
7. draw_v_parallel_lines('|||',24)
8. draw_v_parallel_lines('|||',24)
9. draw_v_parallel_lines('|||',24)
10. draw_hlines(10, '=')
```

Output



Fig 3.35 - Sample code and output of functions with multiple arguments

```
1. L_of_num = [7,8,9,10,11]
2. def list_sumup ():
3.     { sum = 0
4.     for i in range(5):
5.         sum += L_of_num [i]
6.     return (sum) }
7. print (list_sumup())
```

Output

45

Fig 3.36 - Sample code and output of functions with multiple arguments

3.11.2 Function Returns Result

Function is not always, about displaying or printing. Functions are also used to solve an equation and return the result. This is done by using 'return'. For example, the function in the Fig 3.36 sums up the elements of a list and returns the value, which is printed in the main program. The point to note is that a function can return a single value only. A good practice is to enclose the variable or value in '()' after return.

3.12 Debugging

Bug, is an unanticipated error which is generally logical in nature. Some are simple, as we discussed earlier in the example where we swapped the values of 2 variables. And to fix it we dry ran the code. However, for complex ones which exist in a code having large number of lines of code and many variables of various data types, manual dry-run will be tedious and hard. Hence, the solution is to debug the code.

Debugger in IDE

In the Python IDLE menu, locate the Debug and select 'Debugger'. Correspondingly, you will notice a 'Debug On' message on the shell prompt before the cursor. Next, run your program and a window titled 'Debug Control' will pop-up as shown in Fig 3.37 which allows you to monitor and examine the defined variables and the corresponding values, after each instruction. The pop-up window comprises of some buttons that will help you in debugging the code.

- **Step button** is the one that you will use initially, quite frequently; as it allows you to execute the code line-by-line. Every time you press it, the next line of code will execute.
- **Go button** allows you to jump to the breakpoint directly. And after that you can press step iteratively to seek the bug.
- **Over button** is used when there is a function call and you want to avoid it completely. This way, the function is fully executed and value is returned but the debugger skips it.
- **Out button** is used, when you have reached in a function and are examining it in debugger, but wants to skip the remaining portion of the function and come out of it.

Besides this, there are some checkboxes as well. 'Globals' and 'Locals' reflect the global and local data respectively. 'Source' is the file, currently running in the debugger. Lastly, 'Stack' highlights the function currently executed in the debugger.

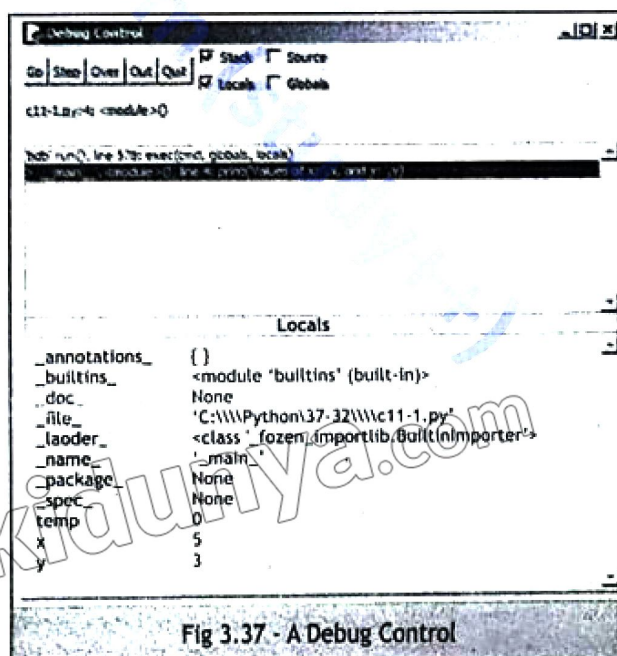


Fig 3.37 - A Debug Control

```

1. x = 5
2. y = 3
3. temp = 0
4. print ('Values of x: ',x, 'and y: ',y)
5. print ("Let's Swap the value of x & y")
6. temp = x
7. y = temp
8. x = y
9. print ('Values of x: ',x, 'and y: ',y)

```

Fig 3.38 - Breakpoint in code of Fig.3.11

The most important element, we have kept for the end, the Breakpoint. A breakpoint is the line in the code which needs close examination, so the debugger will make a stopover there definitely. Although there may be multiple breakpoints, and debugger will make a stopover on each one of them. To assign a breakpoint is quite easy. After activating the debugger, just right click on the line of code, you want to set as a breakpoint and select 'Set Breakpoint'. The said line will be highlighted yellow and is now marked as a breakpoint. For every breakpoint, you want to set, repeat the same process on the lines of codes, you want to examine. Next, run and debug your code. To clear the breakpoint, right click on the highlighted line and select 'Clear Breakpoint'.

For better understanding, let's revisit the earlier code of example Fig 3.11 as an example scenario. Here, we set the breakpoint on the line, where we doubt the values are not as expected and need to examine it for confirmation i.e. line 7, as shown in Fig 3.38.

As soon as we run and start to step-over the code, the values of the variables are initialized and are shown in the Fig 3.37.

However, when we reach the breakpoint, we observe the values of the variables and get a confirmation about the bug, as shown in Fig 3.39. The corresponding output on the shell is also visible.

Python Debugger (pdb)

Python provides a debugger library namely Python Debugger (pdb). From the command prompt, you may execute the program with 'pdb' parameter, by giving the command:

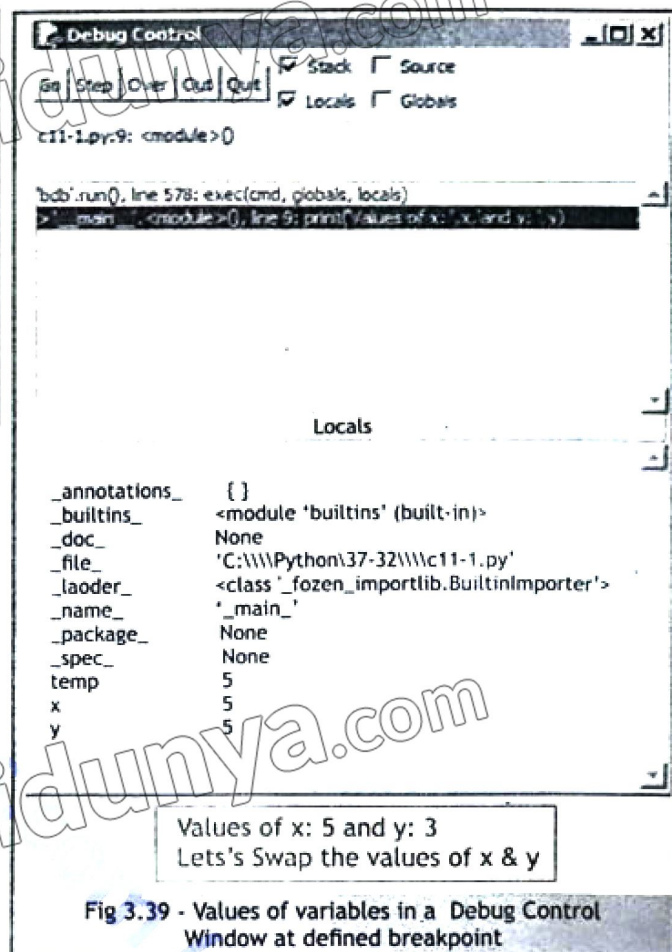


Fig 3.39 - Values of variables in a Debug Control Window at defined breakpoint

> `Python -m pdb swap_var.py`

Adding '-m switch' allows python to run the imported module as script. Thereafter, you can execute through the code line by line using 'step' or 'next' command, as shown in Fig 40.

You may assign breakpoints via 'break "line_number"', like break 7 can be typed in to inspect line 7, in our example. Subsequently, if 'continue' command is used, the debugger will go directly to the breakpoint. For convenience and clarity, 'list' command is used in Fig 41, such that defined breakpoint and effect of continue command can be shown along with the code, simultaneously.

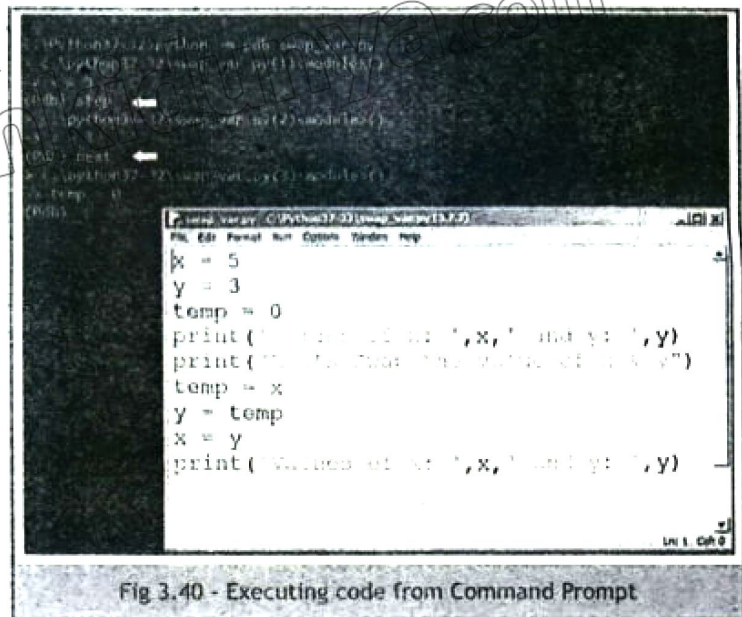


Fig 3.40 - Executing code from Command Prompt

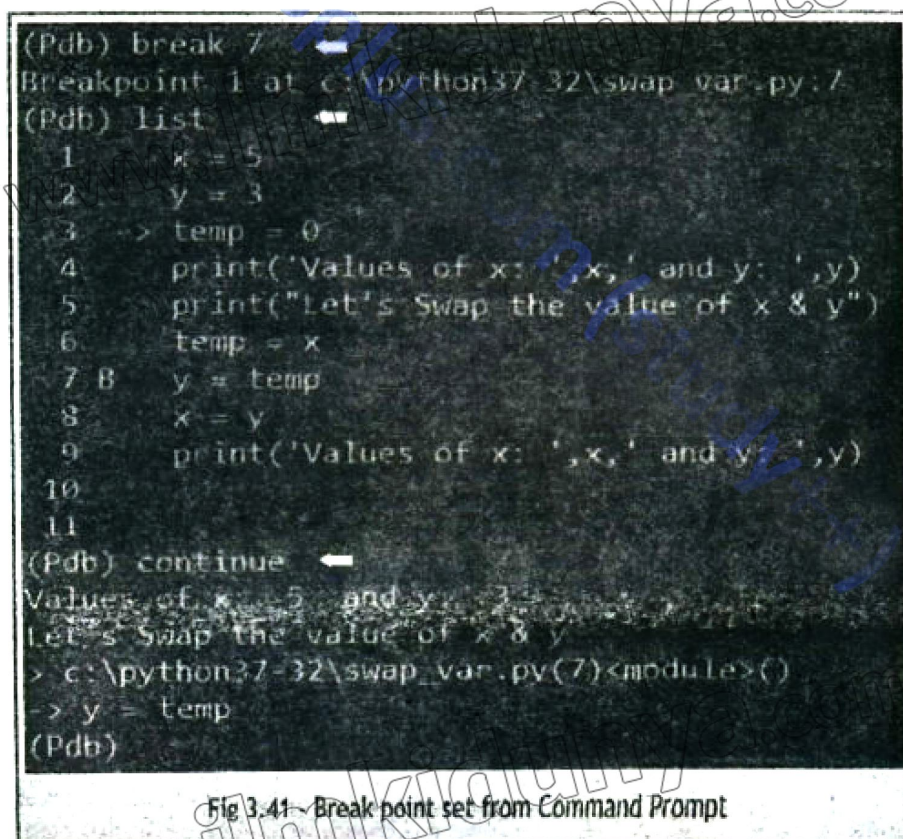


Fig 3.41 - Break point set from Command Prompt

Alternatively, Python also gives the option to import `pdb` in the code, as the first line of the program. When the program executes, the debugger stops at the first line. Next, 'step' command can be used.

You may assign breakpoints in the code, using '`pdb.set_trace()`' instruction. As soon as the program executes, it stops on the very first line after '`pdb.set_trace()`' instruction allowing the code to be inspected and executed line-by-line, as shown in Fig 42.

Thereafter, you may type in 'step' on the shell to execute the code line by line. Fig 43 shows the similar sequence of commands that were executed from command prompt, as reflected in Fig 41.

```

Values of x: 5 and y: 3
Let's Swap the value of x & y
> c:\python37-32\swap_var.py(8)<module>()
-> temp = x
(Pdb)
(Pdb) pdb
x = 5
y = 3
temp = 0
print('Values of x: ',x,' and y: ',y)
print("Let's Swap the value of x & y")
pdb.set_trace()
temp = x
y = temp
x = y
print('Values of x: ',x,' and y: ',y)

```

Fig 3.42 - Sample code having pdb in code

```

Values of x: 5 and y: 3
Let's Swap the value of x & y
> c:\python37-32\swap_var.py(8)<module>()
-> temp = x
(Pdb) list
  3     y = 3
  4     temp = 0
  5     print('Values of x: ',x,' and y: ',y)
  6     print("Let's Swap the value of x & y")
  7     pdb.set_trace()
  8 -> temp = x
  9     y = temp
 10     x = y
 11     print('Values of x: ',x,' and y: ',y)
(Eof)
(Pdb) next
> c:\python37-32\swap_var.py(9)<module>()
-> y = temp

```

Fig 3.43 - Debugging code in Python Shell

Debugging via print statements

While writing a program, the first statement you will learn is how to print and if used adequately is the easiest way to debug a program. By placing multiple print statements thoughtfully, a smooth traceability of any bug can be achieved. In Fig.44, we have placed multiple print statements to display the x, y and temp variables, after each statement where we are assigning new values to any of the said variables. The corresponding output allows us to identify the bug.

```

x = 5
y = 3
temp = 0
print('Values of x: ',x,' and y: ',y)
print("Let's Swap the value of x & y")
temp = x
print('Values of x: ',x,' and y: ',y,' and temp: ',temp)
y = temp
print('Values of x: ',x,' and y: ',y,' and temp: ',temp)
x = y
print('Values of x: ',x,' and y: ',y,' and temp: ',temp)

```

Output

```

Values of x: 5 and y: 3
Let's Swap the value of x & y
Values of x: 5 and y: 3 and temp: 5
Values of x: 5 and y: 5 and temp: 5
Values of x: 5 and y: 5 and temp: 5

```

Fig 3.44 - Debug using print statements

Assert

Python also provides 'assert' keyword which just like a boolean checks for a condition and returns True or False. If the assert statement is True, the statement will execute normally otherwise an 'AssertionError' is returned. Mainly, assert is used to locate logical bugs in code, or any illogical suppositions are taken into consideration. For example, if there is a typing error or miscalculation of multiplying the number with a negative number, in the code as shown in Fig. 45; then using assert before output can help in pinpointing the error.

```
x= 5 # initialize
x = 5 * -1 # may be typing error
assert x> 0
print(x)
```

Output

```
assert x> 0
AssertionError
```

Fig 3.45 - An Assert example

Tip

You can type in 'help' on the shell prompt, at any point in time and a list of available debugging commands will be available. You may type in specifically the 'help "command_name"' for further guidance. Like 'help step'.

```
(Pdb) help step
s(step)
```

Execute the current line, stop at the first possible occasion (either in a function that is called or in the current function).



Activity 6

Take input from the user which determines the height of the diamond and print a filled diamond (◆).



Activity 8

Take an odd number as input from the user and draw a 'X' with the help of a character like '+'. The input number defines the size of 'X' to be displayed.



Activity 7

Write a turtle graphics code to draw a Tic-Tac-Toe board.



Activity 9

Write a program to calculate how many iterations it takes to subtract a small number from a larger one, where both the numbers are positive integrals.

Summary

- A set of well-defined instructions which a computer can execute to solve a problem is called a computer program.
- When the program is executed, the program is loaded into main memory from where one instruction at a time is fetched and provided to the processor which executes it.
- Computer Programs are developed in programming languages which provide the development environment via set of rules and keywords to use.
- Python is an open-source programming language. Provisions are provided in code with the help of conditional statements like 'if-elif-else' statements.
- To implement an iterative task, Python provides loops like the for-loop to accomplish the said task effectively and efficiently with lesser lines of code.
- Turtle graphics provides functions to design basic drawings and two dimensional geometrical shapes.
- With the help of a library we can produce result, with just one line of code, rather than writing our own tens of lines of code.
- Functions help in separating the code which is frequently occurring, this way the written function can be called in a single statement, rather than writing the same code again and again.
- The variable inside a function is a local variable and its scope is limited inside the function only, whereas variables in the main program are global variables and its scope covers the whole program.
- Arguments can be passed to functions for more dynamicity of the functionality.
- Function can return the result of the task it performed in a single value.
- Python provides a data type of List, which is alterable and can contain many values.
- Bug, is an unanticipated error which is generally logical in nature.
- Debugger allows examining the defined variables and the corresponding values, after each instruction.
- A breakpoint is the line in the code which needs close examination, so the debugger will make a stopover there definitely.

Exercise



Select the best answer for the following Multiple-Choice Questions (MCQs).

1. Computer Programs are categorized into _____.
 - a) Interactive Program and Batch Program
 - b) Interactive Program and Badge Program
 - c) Inactive Program and Batch Program
 - d) Inactive Program and Badge Program
2. Python was first introduced in _____.
 - a) 1990
 - b) 1994
 - c) 1992
 - d) 2022
3. Operator `!=` is used for _____.
 - a) Equal
 - b) Logical AND
 - c) Not equal
 - d) Logical OR
4. We can design 2D geometrical shapes in Python, using _____.
 - a) Operators
 - b) Iterations
 - c) Variables
 - d) Turtle Graphics
5. What will the following program do:

```
for i in range(10,0,-1):  
    print('iteration no-',i)
```

 - a) It will give error
 - b) It will run 9 times
 - c) It will run 10 times
 - d) It will run 11 times
6. The interactive computer program accept some input which is from a user program like _____.
 - a) Sub program
 - b) Web browser
 - c) Print job
 - d) Page break
7. _____ provide basic variable types as str, integer, float, etc.
 - a) Print job
 - b) Page break
 - c) Python
 - d) Procedure
8. The input() function treats the entered data as a set of _____.
 - a) Digits
 - b) Characters
 - c) Symbols
 - d) Arithmetic operators
9. Python provide the option of Modulo operator represented as _____.
 - a) =
 - b) ==
 - c) !
 - d) %



Give short answers to the following Short Response Questions (SRQs).

1. What are the applications of computer programming in daily life?
2. Write code to take input a number from user and print its mathematical table on screen from 1 to 10.
3. Take an odd number as input from the user, check if it is odd, otherwise ask the user to re-enter an odd number.
4. Write down the main examples of Python based application.
5. Differentiate between global and local variables with the help of suitable example.



Give long answers to the following Extended Response Questions (ERQs).

1. Explain the applications of Python in different business and technical domains.
2. What are the basic functions that 'List' provides. Elaborate each of them with an example.
3. Write a program for a Dice Rolling Race game for 2 players. Alternatively, generate a random number for each player. If the number rolls out to be 6, then the player gets another roll. Rolls of every iteration is summed up to the previous rolls. The player who reaches 100 first, wins.
4. How to locate and select Debugger in IDLE? Write steps by taking an example into consideration.
5. Write code to print the multiplication of first 10 odd numbers and first 10 even numbers and find the difference of the two. [Hint: Use functions]



Mini Project 1: Number Guessing Game

Generate a Random Number and ask the user to guess the number.

For each input, respond the user, whether the guess is right or wrong.

If the guess was right, congratulate the user and inform about how many guesses it took till the right guess.

However for each wrong answer, inform the user to try again. Additionally, also provide a hint whether the guess was on the higher or lower side.

Give the user appropriate number of tries. e.g. if the guess is between 1 to 20, you may provide with 3 retries.



Mini Project 2: Motorcycle Rent Calculator

Motorcycles can be rented either based on distance travelled or the time the motorcycle was being used by the client. Both are separately calculated and added together, to generate total bill.

For example hourly rate is Rs. 100 while the distance rate is of Rs. 5/ km. A client travelled only 20 km but the possession of the motorcycle was for 3 hours. This way, the bill of travelled-distance will be Rs. 100/- but the possession-charges will be Rs. 300/-. This way, total bill will be of Rs. 400/-.

However, for existing members, the lesser of the two charges is treated as a "discount".

This way, if the client in above example was already a member, only Rs. 300/- needs to be paid by him.