ANNEXURE

4

Testing and Debugging

A4.1 TESTING

Software testing is a very important activity in software development that aims at investigating, evaluating, and ascertaining the completeness and quality of computer software. The objectives of testing include the following:

- Verify software completeness with respect to user's requirements
- Identify bugs or errors to ensure that the software is error-free
- Assess usability, performance, security, and compatibility of the software

To realize these objectives, different types of testing techniques (a few of them have been already covered in Chapter 1) are applied to ensure that the software performs its intended task correctly. These include —

- **Black box testing:** In this testing, the internal system design is not considered. Tests are conducted based on requirements and functionality. Since the test data is generated without knowledge of the implementation, it does not change when the implementation changes.
- **White box (or Glass box) testing:** While black box testing concentrates on what a program does, white box testing, on the other hand, concentrates on how a program does its intendended task. For this type of testing, the tester should have knowledge about the implementation details of the program so that he can design the test cases accordingly to execute each part of the code. Tests are based on coverage of code statements, branches, paths, and conditions.
- **Unit testing:** In unit testing, individual software components or modules are tested by the programmer (not by testers), as it requires detailed knowledge of the internal program design and code. During unit testing, stubs and drivers are created. While drivers simulate parts of the program that use the unit being tested, stubs on the other hand, simulate parts of the program used by the unit being tested. Stubs are useful as they enable programmers to test units that depend upon other software codes that does not yet exist.

Note ***Stubs*** are dummy modules that simulate "called programs" (or low-level modules) during integration testing (top-down approach). They are used when sub-programs are under construction. ***Drivers*** are dummy modules that simulate "calling programs" (or high-level modules) during bottom up integration testing. They are used when main programs are under construction.

- **Integration testing:** In integration testing, integrated modules are tested to verify the combined functionality of all the modules. It is carried out after unit testing is completed. The purpose of integration testing is to expose faults in the interaction between integrated units. It involves both white box as well as black box testing techniques. Therefore, integration testing is said to be a logical extension of unit testing.
- **Functional testing:** This type of testing ignores the internal parts and focuses on the output to check whether it is as per the requirement or not. It uses black box testing technique to ensure that the system is in line with user's requirements. Functional testing does not test individual functions in the program but is used to test a slice of functionality of the whole system.

- **System testing:** In system testing, the entire system is tested as per the requirements. It uses black box type testing technique to design test cases to cater to the overall requirements' specifications (and not on individual module) so that the combined parts of the system is tested. While functional testing verifies a program by checking it against user's specifications, system testing, on the other hand, validates a program by checking it against the published system requirements specifications.

Note **Verification** means evaluating a software in the development phase to find out whether it meats the specified requirement. **Validation** means to ensure that the software meets the user's requirements and check whether the specifications were correct in the first place.

- **Sanity testing:** Sanity testing is done to determine if a new software version is performing well enough to accept it for a major testing effort. If an application crashes during initial use, then the system is not stable and should be corrected. Therefore, there is no point in doing further testing.
- **Regression testing:** Regression testing is done on an application when there has been modification(s) in any module or functionality. This type of testing is usually done through automation tools.
- **Acceptance testing:** Acceptance testing is done by users (and not by programmers or testers) to verify if the system meets their specified requirements. If the system satisfies its users in acceptance testing, then it is accepted, otherwise, it is sent back for required changes.

Note Testing is done to show that bugs exist, not to show that a program is bug-free.

A4.2 DEBUGGING

We have learnt about the basic approaches applied while debugging a code in Chapter 1. In this annexure, we will discuss about the preconditions, principles, aids, and frequent errors while debugging as well as debugging using Python IDLE.

Debugging is a method that involves testing of a code during its execution and code correction. During debugging, a program is executed several times with different inputs to ensure that it performs its intended task correctly. While other forms of testing aims to demonstrate correctness of the program(s), testing during debugging, on the other hand is primarily aimed at locating errors.

Preconditions for Effective Debugging To minimize the time spent on debugging, the programmer should remember certain points which are as follows:

- *Understand the algorithm* – To debug a code effectively, you must understand its algorithm. This knowledge will help you to know what the module is supposed to do and thus write the test cases to locate the errors.
- *Check correctness* – Before debugging, make sure that the code is correct. For this, check the preconditions, terminating conditions, and post conditions for a loop. Even if these checks don't reveal any error, the programmer will at least get a better understanding of the algorithm after making these checks.
- *Code tracing* – Errors can be detected by tracing through the execution of function calls. However, tracing may not catch all errors, but definitely enhances understanding of the algorithms.
- *Peer reviews* – A peer review means getting your code examined by a peer (who is familiar with the algorithm) to identify error(s) in it. It is an important activity to ensure software quality. For best results, peer review should be restricted to short segments of code and the peer should be an outsider so that the programmer can get a different perspective to discover blind spots that seem to be inherent in evaluating his own work.

Principles of Debugging Report error conditions immediately. The earlier an error is detected, the easier it is to find the cause. An error detected in the client interface may be difficult to narrow down the list of possible causes.

Debugging Aids The debugging aids are built into the programming language. Some of them are given below.

- ***Assert statements:*** Make use of assert statements to detect and report error conditions. The assert statement evaluates a Boolean expression and returns the result. If the result is True, nothing happens and if it is False, the program terminates with an error message.
- ***Tracebacks:*** Whenever a run-time error occurs, the Python compiler generates a traceback reporting the currently active subroutine, line number, and a message indicating the cause and location of the run-time error.
- ***Debugger:*** Debugger helps the programmer to step through the program line-by-line and run it with breakpoints set by him. When a line with a breakpoint is about to be executed, the program is interrupted so that the programmer can examine or modify program data. When a run-time error occurs, the debugger generates tracebacks to help the programmer locate the error and indicate its cause.
- ***Print statements:*** Programmers can insert print statements in the code to examine the value of certain variables in different parts in the program. This enables them to check the value of a variable and ensure whether it is being modified or not and if being modified, then the current value if correct or not.

Hence, once an error is detected, its cause (and not the symptom) should be fixed. Consider the following examples for understanding this concept.

Example A4.1 Consider the code given below which finds greater of two numbers.

```
a = int(input("Enter the first number:"))
b = int(input("Enter the first number:"))
if a>b:
    print(a, "is larger than", b)
else:
    print(b, "is larger than", a)
```

Testing this program with all possible values will be a tedious job, so it is better to run this program on pairs of values that have a higher probability of producing wrong results. This can be done by partitioning the domain of all possible inputs into subsets that provide an equivalent result and then forming test cases such that one input comes from each partition.

For the above program, we can partition the pair of input values into seven subsets:

- a positive, b positive
- a positive, b negative
- a negative, b positive
- a=0, b=0
- a negative, b negative
- a=0, b≠0
- a≠0, b=0

If we test any two values from each set, then the probability of exposing a bug (if any) will be quite high.

Example A4.2 Consider another program to count all the prime and composite numbers entered by the user.

```
total_prime = 0
total_composite = 0

while(1):
    num = int(input('Enter no. '))
    if(num == -1):
        break
    is_composite = 0
    for i in range(2,num):
        if(num%i == 0):
            is_composite = 1
            break
    if(is_composite):
        total_composite+=1
    else:
        total_prime+=1
print('total composite : ',total_composite)
print('total prime : ',total_prime)
```


For the aforementioned program, it is best to use glass box testing as we will have to create at least one test case for each path.

Key points to remember

- Create test cases for both branches of all `if` statements.
- If you have a `try-except` block, then write test cases to examine how the code behaves with or without exception.
- For each `for` and `while` loop, create test cases for the following situations:
  - The loop is not entered.
  - The body of the loop is executed exactly once.
  - The body of the loop is executed more than once.
  - The body of the loop is exited.
- For recursive functions, create test cases that cause the function to return with no recursive calls, exactly one recursive call, and more than one recursive call.
- If the program makes a function call, then create a test case to check the order of arguments passed to the function.
- When you are not able to find the bug, either stop debugging and start writing documentation again, or try again after a break.

Frequent Errors

- Wrongly spelt variable names (this would create another separate variable)
- Changing the case of variable names (this would create another separate variable)
- Failure to initialize or reinitialize a variable
- Tested for value equality when you actually wanted to test object equality. For example, given two lists `L1` and `L2`, writing expression as `L1==L2` instead of `id(L1)==id(L2)`
- Created an unintentional alias
- Instead of concentrating on why the program is not doing its intended task, ask yourself why it is doing and what it is doing.

A4.2 Debugging using Python IDLE

Debugging is an important activity and Python IDLE has a built-in debugger to help you debug your programs. The debugger allows the programmer to step through a program and see how the variables change values. To use the debugger, start the IDLE and open the program to debug. Then follow the steps given below.

Step 1: In the shell window, click on *Debug* menu and select *Debugger* as shown in Figure A4.1.

A *Debug Control* window (as shown in Figure A4.2) will open. You may also see the [DEBUG ON] written on the shell. Now, you need to set a breakpoint in the program source code before running the program.

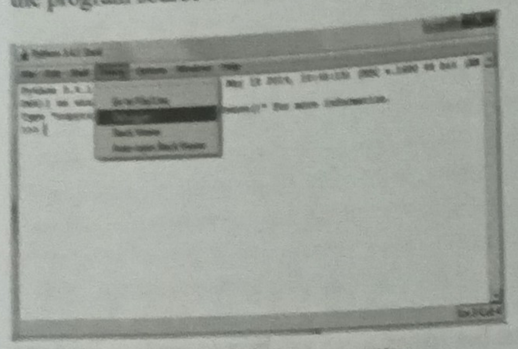

Figure A4.1 Debugger option

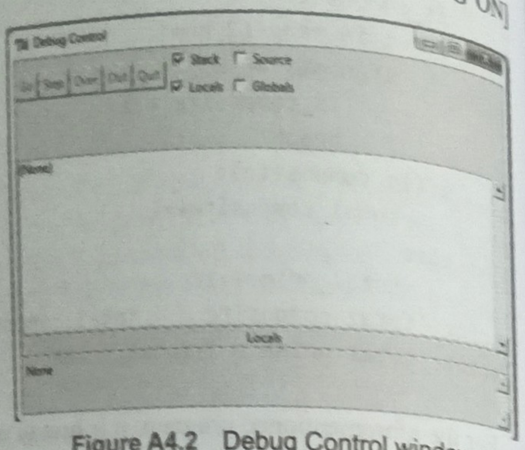

Figure A4.2 Debug Control window

A *breakpoint* is a marker on the source code that tells the debugger to run to this point at normal speed and then pause and let the programmer have control over the program.

As a programmer, you can have many breakpoints in your program at different places.

Step 2: To set a breakpoint, right-click on a line of program code and choose *Set Breakpoint* (as shown in Figure A4.3).

You will see that the background color of the line has become yellow. This indicates that the line has been marked as a breakpoint.

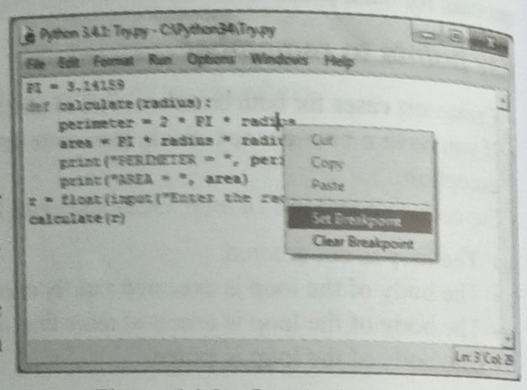

Figure A4.3 Setting breakpoint

Step 3: Press F5 to run the program.

The *Debug Control* window will open and the blue line indicates the code line that is currently being executed.

From this point, click on the *Go* button to make the program run at normal speed until a breakpoint is encountered (or input is requested or the program finishes).

You can also click on the *Step* button to step through the lines in program code (one line at a time). This button is especially important when the line has a function call. The *Step* button makes the execution control go to the first line of the function definition.

If the program line is asking for some input, then enter the value in the Shell window and press the *Enter* key (refer Figure A4.4).

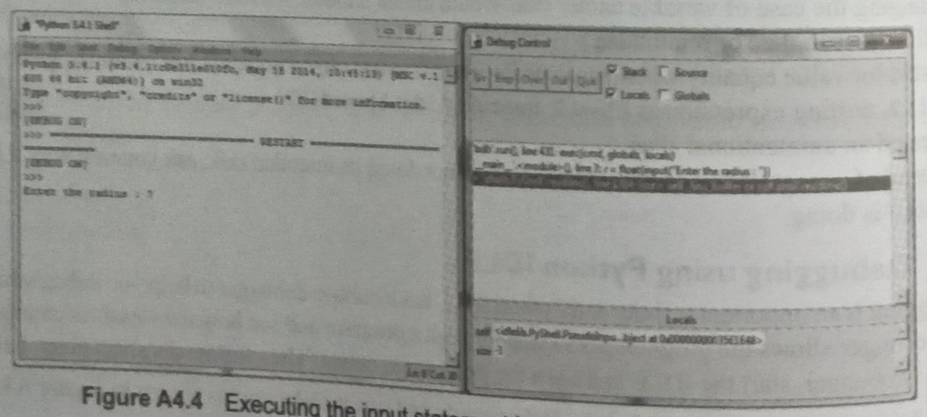

Figure A4.4 Executing the input statement by clicking the *Step* button

The bottom of the Debug Control window has a pane `"Locals"` which shows the value of radius to be 7.0 (Figure A4.5). The *Locals* pane displays the values of variables as they change, and also shows the *types* of those variables. For example, if the input value is of string type, then it would be enclosed within quotes.

Step 4: Continue to click on the *Step* button till the program execution completes. After completion, the output window and Debug Control window will look as shown in Figure A4.6.

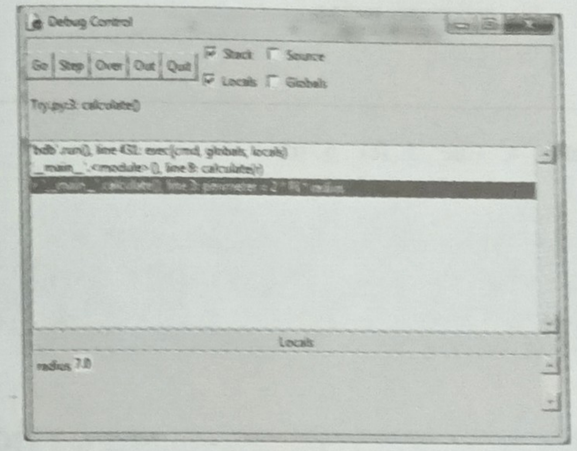

Figure A4.5 Input value being shown at the bottom of the Debug Control window

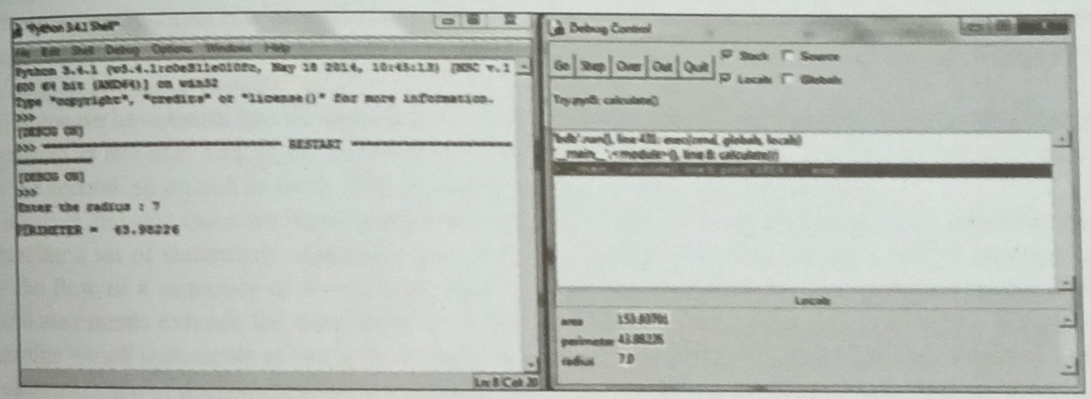

Figure A4.6 Output window and Debug Control window

Other options in the Debug Control window

- You can click on the *Over* button if the statement to be executed has a function call and you want the function to be executed without showing any details of the execution or variables.
- Clicking on *Out* button assumes that you are in a function's code. The button allows the function execution to be finished at normal speed and then return from the function and then give the human control again.
- *Quit* button is pressed to stop the execution of the entire program.

Exercises

1. In _____ testing, individual software components or modules are tested by the programmer.
2. _____ testing is done on an application when there has been modification(s) in any module or functionality.