



Chapter (5)

Code Testing and Debugging

Student Learning Outcomes:

By the end of this chapter, students will be able to :

- Use advanced techniques such as unit tests, breakpoints, and watches for testing and debugging Python applications.

5.1 Introduction

In this chapter, students will learn the essential techniques of code testing and debugging to ensure their programs function correctly. They will understand the importance of testing for creating reliable applications and explore common types of programming errors and bugs. The chapter covers unit testing using Python's unittest and pytest modules, guiding students on how to write and execute test cases. Students will also be introduced to the process of debugging, including setting breakpoints, using watch expressions, and performing step-by-step debugging. Furthermore, the chapter explains exception handling and how to manage multiple exception types effectively. Finally, students will explore profiling and optimization techniques to measure performance and identify potential bottlenecks in their code.

5.2 Testing and Debugging Techniques

Advanced testing and debugging techniques help create reliable and efficient applications. They go beyond basic error fixing to ensure program quality. These techniques identify hidden problems that may affect functionality. Testing and debugging improve performance and user satisfaction. They also make future maintenance easier.

5.2.1 Importance of Testing and Debugging

Testing and debugging are essential to ensure applications work correctly under all conditions. They help detect and remove errors before the software is released. Regular testing prevents unexpected failures during operation. Debugging locates the cause of errors and fixes them efficiently. Together, these practices increase reliability and user confidence.



CLASS ACTIVITY

Discuss scenarios where errors cause failure, and explain testing importance.

Why testing is essential for reliable applications

Testing checks whether an application meets its requirements. It ensures that all functions produce correct results. Testing reduces the risk of crashes or incorrect outputs. It also helps verify that changes do not break existing features. Applications that are thoroughly tested are more trustworthy and stable.

Common types of programming errors and bugs

Syntax errors occur when code does not follow language rules. Logic errors happen when the program produces incorrect results. Runtime errors appear while the program is running. Other

common bugs include data errors and resource leaks. Identifying these errors is key to building stable applications (as shown in Figure 5.1).

Types of Program Errors

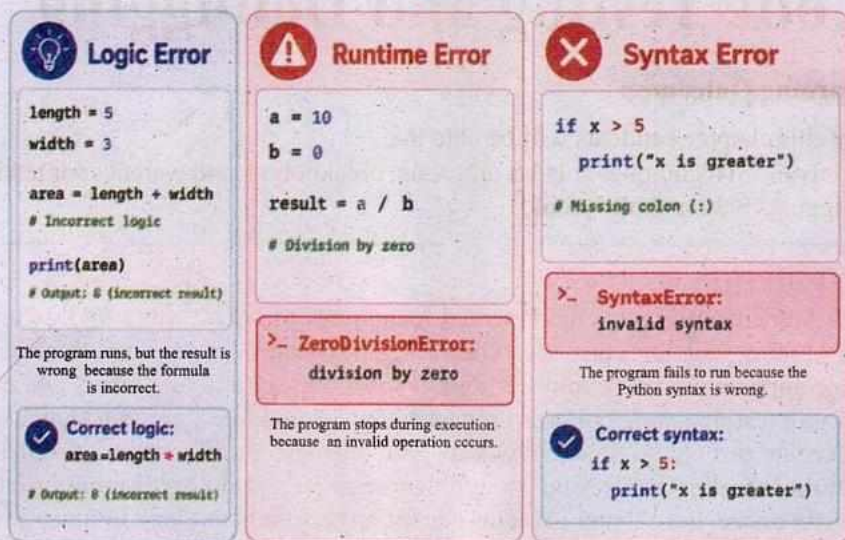


Figure 5.1: Example showing logic error, runtime error and syntax error

Unit Testing

Unit testing is the process of testing individual parts or units of a program to ensure they work correctly. It helps catch errors early and makes debugging easier. Python provides modules like unittest and pytest for writing and running tests. These modules allow automated testing of functions and methods. Unit tests check whether each component produces the expected output. Writing clear and simple test cases improves code reliability.

Introduction to Python's unittest and pytest modules

The unittest module is built into Python and provides a framework for creating test cases. Pytest is an external module that simplifies writing and running tests. Both modules help verify that code behaves as expected. They can test individual functions, classes, or entire modules. Using these tools improves program stability and reduces errors in larger applications.

Writing and executing test cases

Test cases define specific inputs and the expected output for a function. For example, a calculator function can be tested for addition and subtraction results. Python executes these tests and reports failures or errors. Repeated testing ensures that changes in code do not break existing functionality. Automated test execution saves time and improves accuracy in development.

Example:

```
# Function to test
# This function takes two input values, a and b.
# It returns the sum of both values.
def add(a, b):
    return a + b
```



```

# This function takes two input values, a and b.
# It returns the product of both values.
def multiply(a, b):
    return a * b

# Using pytest to test functions
# In pytest, every test function usually starts with the word "test_".
# Pytest automatically detects and runs these test functions.

def test_add():
    # This test checks whether the add() function gives the correct result.
    # Here, 2 and 3 are passed as input values.
    # The expected output is 5.
    assert add(2, 3) == 5 # Test passes because 2 + 3 = 5

def test_multiply():
    # This test checks whether the multiply() function gives the correct result.
    # Here, 2 and 3 are passed as input values.
    # The expected output is 6.
    assert multiply(2, 3) == 6 # Test passes because 2 * 3 = 6

```

- pytest simplifies testing. It doesn't require test case classes or boilerplate code like unittest.
- The tests are written as simple functions, and assert is used to check if the output is correct.
- To run the tests, simply execute pytest in the command line in the directory where the file is located.

Using Breakpoints and Watches

Breakpoints and watches are tools used in debugging to analyze program behavior. They allow pausing program execution and examining data at specific points. These tools help identify errors and understand how code flows. Using breakpoints and watches makes debugging faster and more precise.

Setting breakpoints in IDEs like PyCharm or VS Code

Breakpoints are markers that pause program execution at a chosen line of code. IDEs like PyCharm and VS Code allow setting breakpoints with a simple click. When the program reaches a breakpoint, execution stops, and the current state can be examined. Breakpoints help focus on specific sections of code without running the entire program.

Monitoring variable values with watch expressions

Watch expressions track the values of variables while the program is running. They update automatically when variable values change. This helps identify incorrect or unexpected data during execution. Watches allow observing how data changes step by step in real time.

Step-by-step debugging process

Step-by-step debugging executes code one line or function at a time. It helps follow the program's logic and detect where errors occur. Developers can step into functions, step over lines, or continue execution after a breakpoint. This process makes it easier to locate and fix bugs accurately (as shown in Figure 5.2).

Example:

```
def calculate_area(radius):
    # Set a breakpoint here to pause and check the value of 'radius'
    area = 3.14 * radius ** 2
    return area

def main():
    # Step-by-step debugging: starting from here
    radius = 5
    print("Area of circle:", calculate_area(radius)) # Set another breakpoint here to monitor
    return value

    radius = 10 # Change the radius value
    print("Area of circle:", calculate_area(radius))

# Start the program
if __name__ == "__main__":
    main()
```

- The function `calculate_area(radius)` calculates the area of a circle using the formula $3.14 * radius ** 2$ and returns the calculated value.
- The `main()` function tests the area calculation twice by using two different radius values: 5 and 10.
- Breakpoints can be placed inside `calculate_area()` and near the `print()` statements to check the value of radius, observe the calculated area, and understand step-by-step debugging.

1. Setting Breakpoints in IDEs like PyCharm:

- In PyCharm, you can set breakpoints by clicking the line number next to the code (in the margin).
- When the program runs and reaches a breakpoint, it pauses. At this point, you can inspect variable values and see the current state of the program.
- Example: Set breakpoints at `radius = 5` and `radius = 10` to pause the program execution at these lines.

2. Monitoring Variable Values with Watch Expressions:

- In your IDE, you can use watch expressions to track variables like radius during execution.
- When you run the debugger, watch expressions will display how the value of radius changes as the program runs.
- Example: You can add a watch for radius and see its value change from 5 to 10 as the program progresses.

3. Step-by-Step Debugging Process:

- Step-by-step debugging is available in PyCharm. You can choose to:
 - Step Into: Go inside a function to see how it's working (`calculate_area` in the example).
 - Step Over: Execute the current line and move to the next one without going inside any functions.

- Continue: Continue the execution until the next breakpoint.
- Example: You can step over the line `area = 3.14 * radius ** 2` to skip calculating the area, or step into `calculate_area()` to examine how the area is calculated.

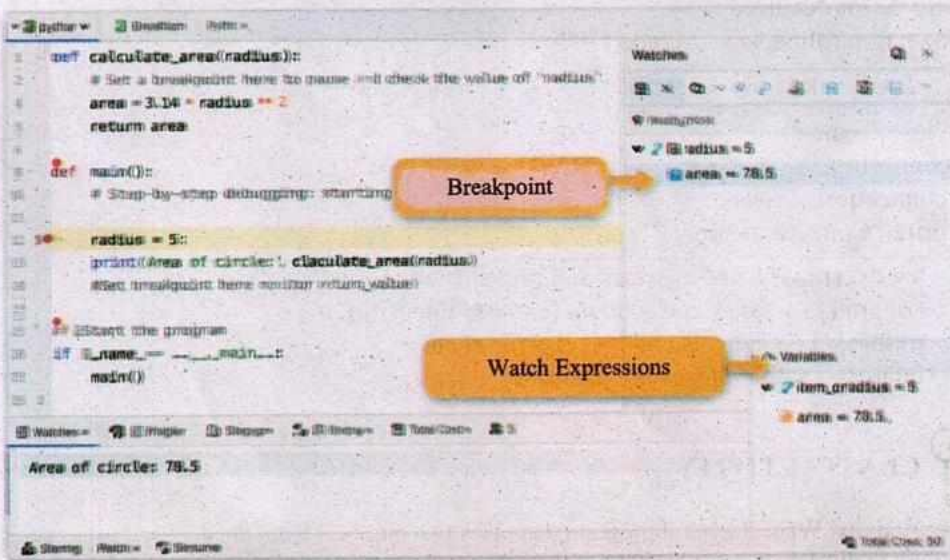


Figure 5.2: Debugging in PyCharm

Exception Handling

Exception handling is used to manage errors that occur during program execution. It prevents the program from crashing and allows it to respond gracefully to unexpected situations. Proper exception handling improves program reliability and user experience. Python provides built-in tools to catch and handle exceptions efficiently.



TIDBIT

Exception handling helps programs manage unexpected problems without stopping suddenly.

Example:

```
import sqlite3

try:
    # Connect to database
    connection = sqlite3.connect("school.db")
    cursor = connection.cursor()

    # Insert data into table
    cursor.execute("INSERT INTO students (ID, Name, Age) VALUES (?, ?, ?)", (1, "Ali", 20))

    # Save changes (ensures data integrity)
    connection.commit()
```



```
print("Data inserted successfully")
```

```
except sqlite3.Error as e:
```

```
    # Handle database errors
```

```
    print("Error occurred:", e).
```

```
    connection.rollback() # Undo changes if error occurs
```

```
finally:
```

```
    # Close connection
```

```
    if connection:
```

```
        connection.close()
```

```
    print("Connection closed")
```

- try-except → handles errors and prevents crashes
- commit() → saves correct data (ensures integrity)
- rollback() → cancels changes if error occurs
- finally → safely closes the database



CLASS ACTIVITY

Class Activity: Write a Python program that takes two numbers from the user and uses exception handling to display an error message if the user enters invalid input or tries to divide by zero.

Handling multiple exception types effectively

A single program may encounter different types of exceptions. Python allows catching multiple exceptions in one try-except block. Each exception type can be handled with a specific response. This approach prevents unexpected crashes and ensures proper error recovery. Handling multiple exceptions improves program stability and control over errors.

5.3 Profiling and Optimization

Profiling and optimization help improve the speed and efficiency of a program. Profiling measures how much time and resources each part of the code uses. Optimization focuses on making slow or resource-heavy parts faster. These techniques are important for building high-performance applications. Regular profiling ensures that code remains efficient as it grows.

? DO YOU KNOW?

Profiling helps identify slow parts of a program so they can be improved for better performance.

Example:

The following code shows a nested loop:

```
for i in range(10000):  
    for j in range(10000):  
        # Some repeated task  
        pass
```



In this code, the outer loop runs 10,000 times. For every single iteration of the outer loop, the inner loop also runs 10,000 times. Therefore, the total number of executions becomes:

$$10000 \times 10000 = 100,000,000$$

This means the inner statement may run one hundred million times, which can make the program very slow. Such a part of the code is called a performance bottleneck. Profiling tools help programmers identify these slow sections so that they can optimise the code and improve program efficiency.

Measuring performance using profiling tools

Profiling tools track the execution time and memory usage of different functions in a program. Python provides modules like cProfile and timeit for this purpose. These tools generate reports showing which parts of the code are slow or resource-intensive (as shown in Figure 5.3). Measuring performance helps prioritize areas for optimization.

Call Count	Time	Cumulative Time		Description
process_data	10	2.500	2.500	Bottleneck
read_file	1	2.400	2.400	Bottleneck
analyze_results	10	0.100	0.100	



Figure 5.3: Diagram showing profiling output highlighting bottlenecks

Identifying and fixing performance bottlenecks

Performance bottlenecks are sections of code that slow down the program. Profiling reports help locate these bottlenecks. Optimization techniques include improving algorithms, reducing unnecessary operations, or using efficient data structures. Fixing bottlenecks improves overall program speed and responsiveness (as shown in Figure 5.4).

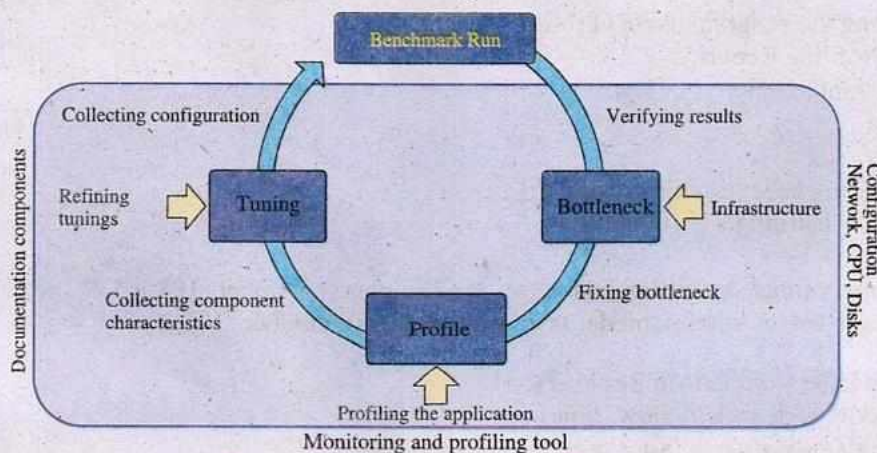


Figure 5.4: Performance Test Cycle



Example:

```
import cProfile
import timeit

# Slow code example
def slow_sum(numbers):
    total = 0

    # Adding numbers one by one using a loop
    for num in numbers:
        total = total + num

    return total

# Fast code example
def fast_sum(numbers):
    # Python built-in sum() is faster and optimised
    return sum(numbers)

# Main function
def main():
    numbers = list(range(1, 1000000))

    print("Slow code result:", slow_sum(numbers))
    print("Fast code result:", fast_sum(numbers))

# Profiling the program using cProfile
print("Profiling Report:")
cProfile.run("main()")

# Comparing execution time using timeit
numbers = list(range(1, 1000000))

slow_time = timeit.timeit(lambda: slow_sum(numbers), number=10)
fast_time = timeit.timeit(lambda: fast_sum(numbers), number=10)

print("\nTime Comparison Report:")
print("Slow code time:", slow_time)
print("Fast code time:", fast_time)
```



- cProfile generates a profiling report that shows how much time each function takes during program execution.
- `slow_sum()` is slower because it adds numbers manually using a loop.
- `fast_sum()` is faster because it uses Python's built-in `sum()` function, which is already optimised.
- The function taking more execution time is called the bottleneck.
- `timeit` helps compare the execution time of slow and fast code.

Output: (The exact time values may be different on each computer, but the output will look like this)

Profiling Report:

Slow code result: 499999500000

Fast code result: 499999500000

6 function calls in 0.080 seconds

Ordered by: standard name

ncalls	tottime	percall	cumtime	percall	function
1	0.060	0.060	0.080	0.080	<module>
1	0.050	0.050	0.050	0.050	slow_sum
1	0.010	0.010	0.010	0.010	{built-in sum}
1	0.010	0.010	0.010	0.010	fast_sum
1	0.000	0.000	0.080	0.080	main

Time Comparison Report:

Slow code time: 0.42

Fast code time: 0.09

- Both functions give the same result: 499999500000.
- `slow_sum()` takes more time because it uses a manual loop.
- `fast_sum()` takes less time because it uses Python's built-in `sum()` function.
- Therefore, `slow_sum()` is the bottleneck in this program.



Summary

Testing

Testing is the process of checking a program to make sure that it works correctly and produces the expected output.

Debugging

Debugging is the process of finding and fixing errors or bugs in a program.

Bug

A bug is an error, mistake, or fault in a program that causes it to produce an incorrect result or behave unexpectedly.

Programming Error

A programming error is a mistake in code that may prevent the program from running correctly. Common errors include syntax errors, logic errors, and runtime errors.

Unit Testing

Unit testing is a testing method in which small parts of a programme, such as individual functions, are tested separately to check whether they work correctly.

Test Case

A test case is a set of inputs, expected outputs, and conditions used to check whether a program or function is working properly.

unittest

unittest is a built-in Python module used to create and run test cases for checking program correctness.

Pytest

pytest is a Python testing framework used to write and execute simple and powerful test cases.

Breakpoint

A breakpoint is a point in a program where execution is paused so that the programmer can check the values of variables and understand program behavior.

Watch Expression

A watch expression is used during debugging to monitor the value of a variable or expression while the program is running step by step.

Exception Handling

Exception handling is a method used to manage runtime errors so that a program does not stop suddenly during execution.

Profiling

Profiling is the process of measuring a program's performance to find which parts of the code take more time or use more resources.

Optimization

Optimization is the process of improving code so that it runs faster, uses fewer resources, or performs more efficiently.

Performance Bottleneck

A performance bottleneck is a part of a program that slows down the overall execution and reduces efficiency.



EXERCISE

✓ Multiple Choice Questions

1. Testing is essential for reliable applications because it helps to:
(a) Optimise code performance (b) Identify and fix bugs
(c) Reduce the program size (d) Make code run faster
2. Common types of programming errors include:
(a) Syntax error (b) Logic error
(c) Runtime error (d) All of the above
3. Unit testing focuses on:
(a) Testing the entire application
(b) Testing individual components or functions
(c) Testing the user interface
(d) Testing database connections
4. The Python module used for unit testing is:
(a) numpy (b) matplotlib
(c) unittest (d) os
5. The main purpose of using testing tools like unittest and pytest:
(a) to increase program size (b) to verify code works correctly
(c) to design layouts (d) to compile programs
6. Breakpoints in debugging are used to:
(a) Run the program faster
(b) Pause the program at specific points and inspect its state
(c) Skip certain code sections
(d) Optimise the code
7. Watch expressions in debugging are used to:
(a) Change variable values
(b) Monitor variable values during program execution
(c) Pause the program
(d) Display output messages
8. Step-by-step debugging is used to:
(a) Run the program faster
(b) Execute code one line at a time for easier error identification
(c) Increase the program's performance
(d) Compile the code
9. The Python keyword pair used for exception handling is:
(a) try-except (b) try-finally
(c) try-catch (d) handle-throw



10. Profiling in programming is used to:

- (a) Improve the user interface design
- (b) Measure performance and identify bottlenecks
- (c) Optimise memory usage only
- (d) Test for security vulnerabilities



Short Questions

1. Why is testing essential for ensuring reliable applications?
2. What are some common types of programming errors and bugs?
3. What is unit testing, and why is it important in programming?
4. How do Python's unittest and pytest modules help in unit testing?
5. What is the purpose of writing and executing test cases?
6. How do you set breakpoints in IDEs like PyCharm or VS Code?
7. What is the role of monitoring variable values with watch expressions during debugging?
8. What is the step-by-step debugging process in IDEs?
9. How does exception handling help in managing multiple exception types effectively?
10. What tools can be used for profiling and measuring performance in Python?



Long Questions

1. Why is testing essential for reliable applications, and what role does it play in ensuring program stability?
2. Discuss the common types of programming errors and bugs that occur during software development.
3. What is unit testing, and how do Python's unittest and pytest modules support effective testing?
4. Explain the process of writing and executing test cases using Python's unittest or pytest modules.
5. How do breakpoints and watch expressions help in debugging? Discuss their role in identifying issues in the code.
6. Describe how to set breakpoints in IDEs like PyCharm or VS Code, and explain their significance in the debugging process.
7. What is step-by-step debugging, and how does it help in identifying and fixing bugs in Python programs?
8. How does exception handling work in Python? Explain how to handle multiple exception types effectively to prevent program crashes.



Answer Key

- | | |
|------|-------|
| 1. B | 6. B |
| 2. D | 7. B |
| 3. B | 8. B |
| 4. C | 9. A |
| 5. B | 10. B |