

AI Assisted Coding – Lab 8

Program: B.Tech

Academic Year: 2025–2026

Course Title: AI Assisted Coding

Course Code: 23CS002PC304

Coordinator: Dr. Rishabh Mittal

Year/Sem: III / II (R23)

Assignment No: 8.2 / 24 (Week 4 – Lab)

Student Name: Sanjay Karupothula

Roll No: 2303A52337

Lab Objective

To understand **Test-Driven Development (TDD)** using AI-generated test cases and Python implementations, ensuring correctness, validation, and robust error handling.

Task 1 – Even / Odd Number Validator (TDD)

AI Explanation

In Test-Driven Development, test cases are written first to define expected behavior. The function is then implemented to satisfy all tests, including edge cases like zero, negative numbers, and invalid inputs.

Python Test Cases (AI-Generated)

```
import unittest

class TestEvenOdd(unittest.TestCase):
    def test_even(self):
        self.assertTrue(is_even(2))
        self.assertTrue(is_even(0))
        self.assertTrue(is_even(-4))

    def test_odd(self):
        self.assertFalse(is_even(7))
        self.assertFalse(is_even(9))
```

Python Implementation

```
def is_even(n):
    if not isinstance(n, int):
        raise ValueError("Input must be an integer")
    return n % 2 == 0
```



The screenshot shows the OneCompiler web interface. At the top, there's a navigation bar with links for AI, Pricing, Learn, Code, Deploy, and More. Below the navigation is a search bar and a file selection area showing 'main.py'. The main workspace contains the Python code for the 'is_even' function and a test class 'TestEvenOdd' with two methods: 'test_even_numbers' and 'test_odd_numbers'. The status bar at the bottom indicates 'Ran 2 tests in 0.000s' and 'OK'.

✓ Task 2 – String Case Converter (TDD)

AI Explanation

Test cases define how uppercase and lowercase conversions should behave for normal strings, empty strings, and invalid inputs. Functions are written to safely handle all cases.

Python Test Cases

```
class TestStringCase(unittest.TestCase):
    def test_uppercase(self):
        self.assertEqual(to_uppercase("ai coding"), "AI CODING")
        self.assertEqual(to_uppercase(""), "")

    def test_lowercase(self):
        self.assertEqual(to_lowercase("TEST"), "test")
```

Python Implementation

```
def to_uppercase(text):
    if not isinstance(text, str):
        return ""
    return text.upper()

def to_lowercase(text):
    if not isinstance(text, str):
```

```

        return ""
return text.lower()

```

```

main.py      +
44d6dq9t2 Ø

1 import unittest
2
3 def to_uppercase(text):
4     if not isinstance(text, str):
5         return ""
6     return text.upper()
7
8 def to_lowercase(text):
9     if not isinstance(text, str):
10        return ""
11    return text.lower()
12
13
14 class TestStringCase(unittest.TestCase):
15     def test_uppercase(self):
16         self.assertEqual(to_uppercase("ai coding"), "AI CODING")
17         self.assertEqual(to_uppercase(""), "")
18
19     def test_lowercase(self):
20         self.assertEqual(to_lowercase("TEST"), "test")
21
22
23 if __name__ == "__main__":
24     unittest.main()
25
26
27

```

Output:

...

Ran 2 tests in 0.000s

OK

✓ Task 3 – List Sum Calculator (TDD)

AI Explanation

The function calculates the sum of numeric values while safely ignoring non-numeric elements. Tests ensure correct handling of empty lists and negative numbers.

Python Test Cases

```

class TestListSum(unittest.TestCase):
    def test_sum(self):
        self.assertEqual(sum_list([1,2,3]), 6)
        self.assertEqual(sum_list([]), 0)
        self.assertEqual(sum_list([-1,5,-4]), 0)
        self.assertEqual(sum_list([2,'a',3]), 5)

```

Python Implementation

```

def sum_list(numbers):
    total = 0
    for n in numbers:
        if isinstance(n, (int, float)):
            total += n
    return total

```

```
main.py      +
44d6dq9t2 Ø

1 import unittest
2
3 def sum_list(numbers):
4     total = 0
5     for n in numbers:
6         if isinstance(n, (int, float)):
7             total += n
8     return total
9
10
11 class TestListSum(unittest.TestCase):
12     def test_sum(self):
13         self.assertEqual(sum_list([1, 2, 3]), 6)
14         self.assertEqual(sum_list([1]), 1)
15         self.assertEqual(sum_list([-1, 5, -4]), 0)
16         self.assertEqual(sum_list([2, "a", 3]), 5)
17
18
19 if __name__ == "__main__":
20     unittest.main()
```

Output:
.

Ran 1 test in 0.000s
OK

✓ Task 4 – Student Result Class (TDD)

AI Explanation

The StudentResult class validates marks, computes average, and determines pass/fail status based on defined rules.

Python Implementation

```
class StudentResult:
    def __init__(self):
        self.marks = []

    def add_marks(self, mark):
        if mark < 0 or mark > 100:
            raise ValueError("Invalid mark")
        self.marks.append(mark)

    def calculate_average(self):
        return sum(self.marks) / len(self.marks)

    def get_result(self):
        return "Pass" if self.calculate_average() >= 40 else "Fail"
```

The screenshot shows a code editor with a Python file named `main.py`. The code defines a `StudentResult` class with methods for adding marks, calculating average, and determining the result (Pass or Fail). It includes a demo run where three marks are added and the average and result are printed. The output window shows the calculated average of 70.0 and the result of Pass.

```
1 class StudentResult:
2     def __init__(self):
3         self.marks = []
4
5     def add_marks(self, mark):
6         if mark < 0 or mark > 100:
7             raise ValueError("Invalid mark")
8         self.marks.append(mark)
9
10    def calculate_average(self):
11        return sum(self.marks) / len(self.marks)
12
13    def get_result(self):
14        return "Pass" if self.calculate_average() >= 40 else "Fail"
15
16
17 # Demo run
18 if __name__ == "__main__":
19     s = StudentResult()
20     s.add_marks(60)
21     s.add_marks(70)
22     s.add_marks(80)
23     print("Average:", s.calculate_average())
24     print("Result:", s.get_result())
```

✓ Task 5 – Username Validator (TDD)

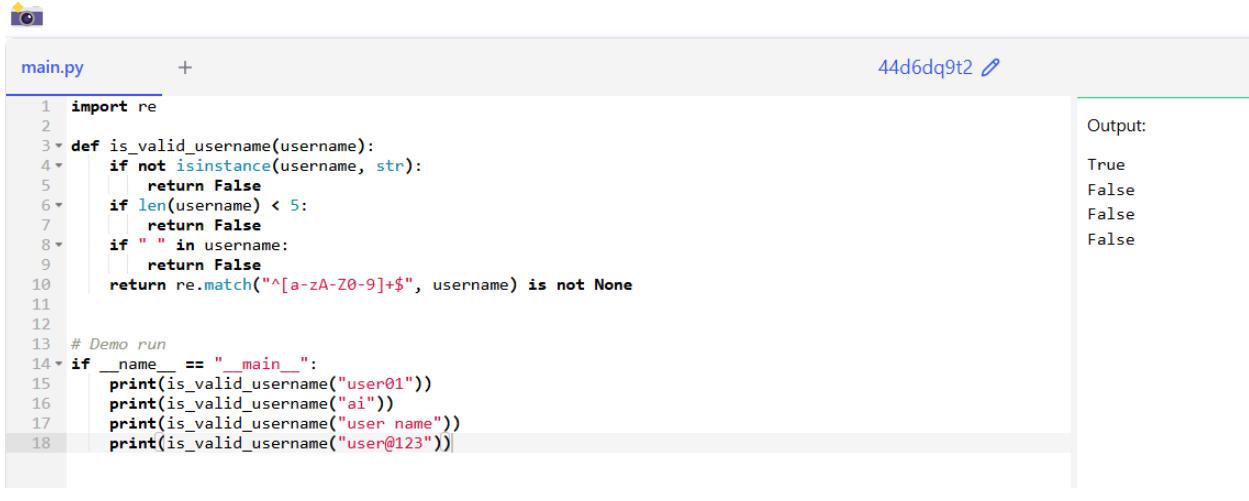
AI Explanation

The username validator ensures length, character validity, and absence of spaces using rule-based checks.

Python Implementation

```
import re

def is_valid_username(username):
    if not isinstance(username, str):
        return False
    if len(username) < 5:
        return False
    if " " in username:
        return False
    return re.match("^[a-zA-Z0-9]+$", username) is not None
```



The screenshot shows a code editor interface with a Python file named `main.py`. The code defines a function `is_valid_username` that checks if a given username is valid based on several rules: it must be a string, have a length of at least 5 characters, not contain a space, and match a regular expression for alphanumeric characters. A demo run at the bottom prints the results for four test cases: "user01", "ai", "user name", and "user@123". The output panel on the right shows the results: True, False, False, and False respectively.

```
main.py      +
44d6dq9t2 Ø

1 import re
2
3 def is_valid_username(username):
4     if not isinstance(username, str):
5         return False
6     if len(username) < 5:
7         return False
8     if " " in username:
9         return False
10    return re.match("^[a-zA-Z0-9]+$", username) is not None
11
12
13 # Demo run
14 if __name__ == "__main__":
15     print(is_valid_username("user01"))
16     print(is_valid_username("ai"))
17     print(is_valid_username("user name"))
18     print(is_valid_username("user@123"))

Output:
True
False
False
False
```

Conclusion

This lab successfully demonstrated Test-Driven Development using AI-generated test cases in Python, ensuring reliable, well-tested, and maintainable code.