

ASSIGNMENT-8.5

D.Nithya
2303A52457
B-35

Task Description #1 (Username Validator – Apply AI in Authentication Context)

- Task: Use AI to generate at least 3 assert test cases for a function `is_valid_username(username)` and then implement the function using Test-Driven Development principles.
- Requirements:
 - Username length must be between 5 and 15 characters.
 - Must contain only alphabets and digits.
 - Must not start with a digit.
 - No spaces allowed.

Example Assert Test Cases:

```
assert is_valid_username("User123") == True
assert is_valid_username("12User") == False
assert is_valid_username("Us er") == False
```

Expected Output #1:

- Username validation logic successfully passing all AI-generated test cases.

The screenshot shows a code editor interface with a dark theme. On the left is a sidebar with various icons for file operations like copy, paste, search, and refresh. The main area displays a Python script named 'ai.py' with the following code:

```
1 def is_valid_username(username):
2     # Username length must be between 5 and 15 characters
3     if len(username) < 5 or len(username) > 15:
4         return False
5
6     # Username must not start with a digit
7     if username[0].isdigit():
8         return False
9
10    # Username must contain only alphabets and digits
11    if not username.isalnum():
12        return False
13
14    return True
15 # Example usage
16 print(is_valid_username("user123")) # True
17 print(is_valid_username("1user")) # False
18 print(is_valid_username("us")) # False
```

To the right of the code, there's a 'Build with Agent' panel. It includes a message bubble icon, the text 'Build with Agent', a note that 'AI responses may be inaccurate.', and a section titled 'Generate Agent Instructions to onboard AI onto your codebase.' Below this is a small input field labeled 'Describe what to bui' and some navigation icons.

At the bottom of the editor, there's a toolbar with icons for file operations, a status bar showing 'Screen Reader Optimized', line numbers (Ln 18, Col 1), spaces (Spaces: 4), encoding (UTF-8 LF), language (Python), version (3.14.2), and a 'Go Live' button.

Task Description #2 (Even–Odd & Type Classification – Apply AI for Robust Input Handling)

- Task: Use AI to generate at least 3 assert test cases for a function `classify_value(x)` and implement it using conditional logic and loops.

- Requirements:

- If input is an integer, classify as "Even" or "Odd".
- If input is 0, return "Zero".
- If input is non-numeric, return "Invalid Input".

Example Assert Test Cases:

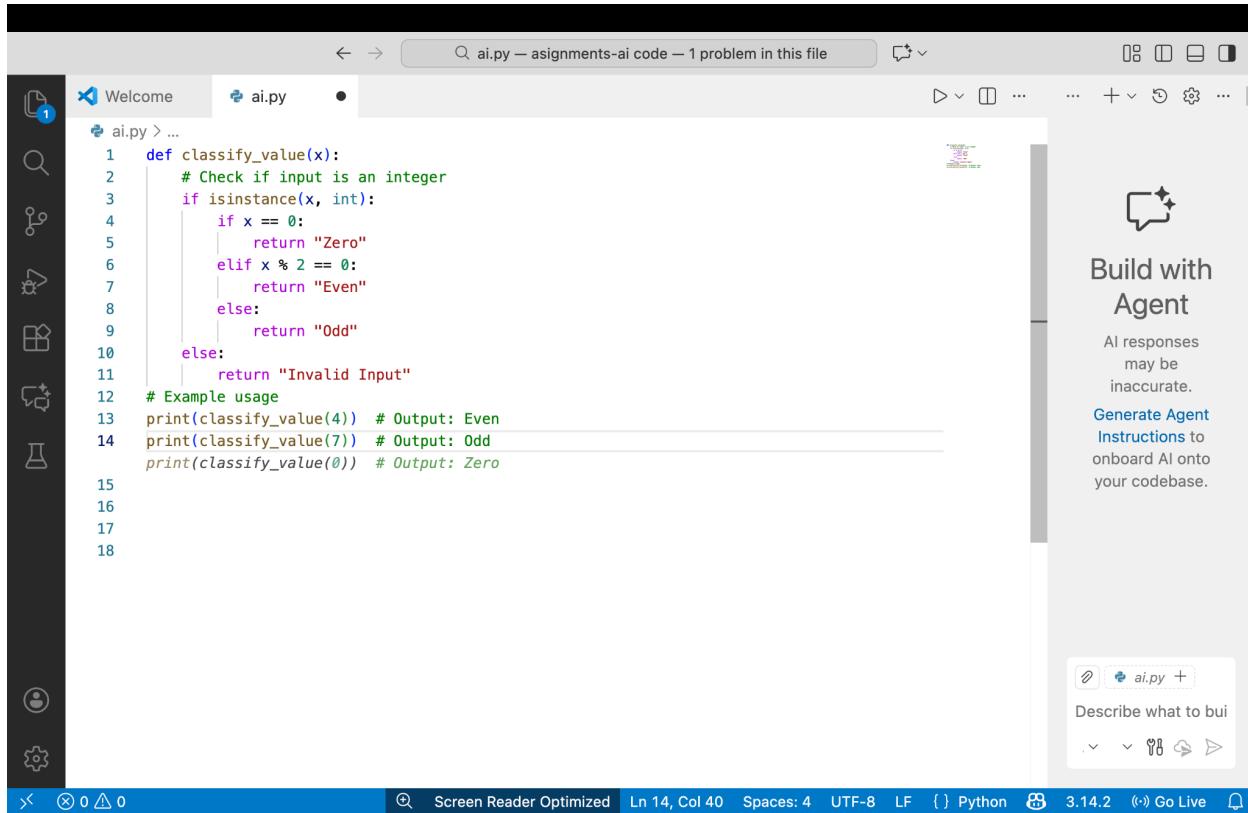
```
assert classify_value(8) == "Even"
```

```
assert classify_value(7) == "Odd"
```

```
assert classify_value("abc") == "Invalid Input"
```

Expected Output #2:

- Function correctly classifying values and passing all test cases.



The screenshot shows a code editor interface with a dark theme. On the left is a vertical toolbar with icons for file operations, search, and other tools. The main area displays a Python script named `ai.py`. The code defines a function `classify_value(x)` that checks if `x` is an integer and classifies it as "Zero", "Even", "Odd", or "Invalid Input". It includes example usage with `print` statements. The right side of the interface has a sidebar titled "Build with Agent" containing instructions about AI responses and generating agent instructions. At the bottom, there's a status bar with file information and version details (3.14.2).

```

1 def classify_value(x):
2     # Check if input is an integer
3     if isinstance(x, int):
4         if x == 0:
5             return "Zero"
6         elif x % 2 == 0:
7             return "Even"
8         else:
9             return "Odd"
10    else:
11        return "Invalid Input"
12 # Example usage
13 print(classify_value(4)) # Output: Even
14 print(classify_value(7)) # Output: Odd
15 print(classify_value(0)) # Output: Zero
16
17
18

```

Task Description #3 (Palindrome Checker – Apply AI for String Normalization)

- Task: Use AI to generate at least 3 assert test cases for a function `is_palindrome(text)` and implement the function.
- Requirements:

- Ignore case, spaces, and punctuation.
- Handle edge cases such as empty strings and single characters.

Example Assert Test Cases:

```

assert is_palindrome("Madam") == True
assert is_palindrome("A man a plan a canal Panama") ==
True

```

```
assert is_palindrome("Python") == False
```

Expected Output #3:

- Function correctly identifying palindromes and passing all AI-generated tests.

The screenshot shows a code editor window with a dark theme. On the left is a sidebar with various icons. The main area displays a Python file named 'ai.py' containing the following code:

```
1 def is_palindrome(text):
2     # Normalize the string: lowercase and keep only alphanumeric characters
3     normalized = ""
4     for ch in text.lower():
5         if ch.isalnum():
6             normalized += ch
7
8     # Check if normalized string is equal to its reverse
9     return normalized == normalized[::-1]
10 # Example usage
11 print(is_palindrome("A man, a plan, a canal: Panama")) # True
12 print(is_palindrome("race a car")) # False
13 print(is_palindrome("")) # True
14
15
16
```

To the right of the code, there is a sidebar titled 'Build with Agent'. It contains the following text:
AI responses may be inaccurate.
Generate Agent Instructions to onboard AI onto your codebase.

At the bottom of the editor, there is a toolbar with icons for file operations and a status bar showing 'Screen Reader Optimized', 'Ln 13, Col 1', 'Spaces: 4', 'UTF-8 LF', 'Python', '3.14.2', 'Go Live', and a lock icon.

Task Description #4 (BankAccount Class – Apply AI for Object-Oriented Test-Driven Development)

- Task: Ask AI to generate at least 3 assert-based test cases for a BankAccount class and then implement the class.

- Methods:

- deposit(amount)
- withdraw(amount)
- get_balance()

Example Assert Test Cases:

```
acc = BankAccount(1000)
```

```

acc.deposit(500)
assert acc.get_balance() == 1500
acc.withdraw(300)
assert acc.get_balance() == 1200

```

Expected Output #4:

- Fully functional class that passes all AI-generated assertions.

```

1  class BankAccount:
2      def __init__(self, initial_balance):
3          self.balance = initial_balance
4
5      def deposit(self, amount):
6          if amount > 0:
7              self.balance += amount
8
9      def withdraw(self, amount):
10         if 0 < amount <= self.balance:
11             self.balance -= amount
12
13     def get_balance(self):
14         return self.balance
15 # Example usage:
16 account = BankAccount(100)
17 (variable) account: BankAccount
18 account.withdraw(30)
19 print(account.get_balance()) # Output: 120
20 →
21

```

Build with Agent

AI responses may be inaccurate.

Generate Agent Instructions to onboard AI onto your codebase.

ai.py +

Describe what to bui

Screen Reader Optimized Ln 19, Col 1 Spaces: 4 UTF-8 LF { } Python 3.14.2 (i) Go Live

Task Description #5 (Email ID Validation – Apply AI for Data Validation)

- Task: Use AI to generate at least 3 assert test cases for a function validate_email(email) and implement the function.
- Requirements:
 - o Must contain @ and .
 - o Must not start or end with special characters.
 - o Should handle invalid formats gracefully.

Example Assert Test Cases:

```
assert validate_email("user@example.com") == True
```

```
assert validate_email("userexample.com") == False
```

```
assert validate_email("@gmail.com") == False
```

Expected Output #5:

- Email validation function passing all AI-generated test cases and handling edge cases correctly.

The screenshot shows a code editor interface with a dark theme. On the left is a vertical toolbar with icons for file operations, search, and other tools. The main area displays a Python script named `ai.py`. The code defines a `validate_email` function that checks if an email string is valid according to specific rules. A sidebar on the right is titled "Build with Agent" and contains instructions for onboard AI onto the codebase. At the bottom, there's a status bar with various icons and text indicating the file is "Screen Reader Optimized" and has 29 lines and 4 spaces.

```
1 def validate_email(email):
2     # Email must contain exactly one '@'
3     if email.count "@" != 1:
4         return False
5
6     # Must contain at least one '.'.
7     if "." not in email:
8         return False
9
10    # Must not start or end with special characters
11    if not email[0].isalnum() or not email[-1].isalnum():
12        return False
13
14    # Split local and domain parts
15    local, domain = email.split "@"
16
17    # Local and domain parts must not be empty
18    if not local or not domain:
19        return False
20
21    # Domain must contain '.'
22    if "." not in domain:
23        return False
24
25    return True
26 #example usage
27 print(validate_email("test@example.com")) # True
28 print(validate_email("invalid-email")) # False
29 # Example usage
```