

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:
 - o Username length must be between 5 and 15 characters.
 - o Must contain only alphabets and digits.
 - o Must not start with a digit.
 - o No spaces allowed.

Example Assert Test Cases:

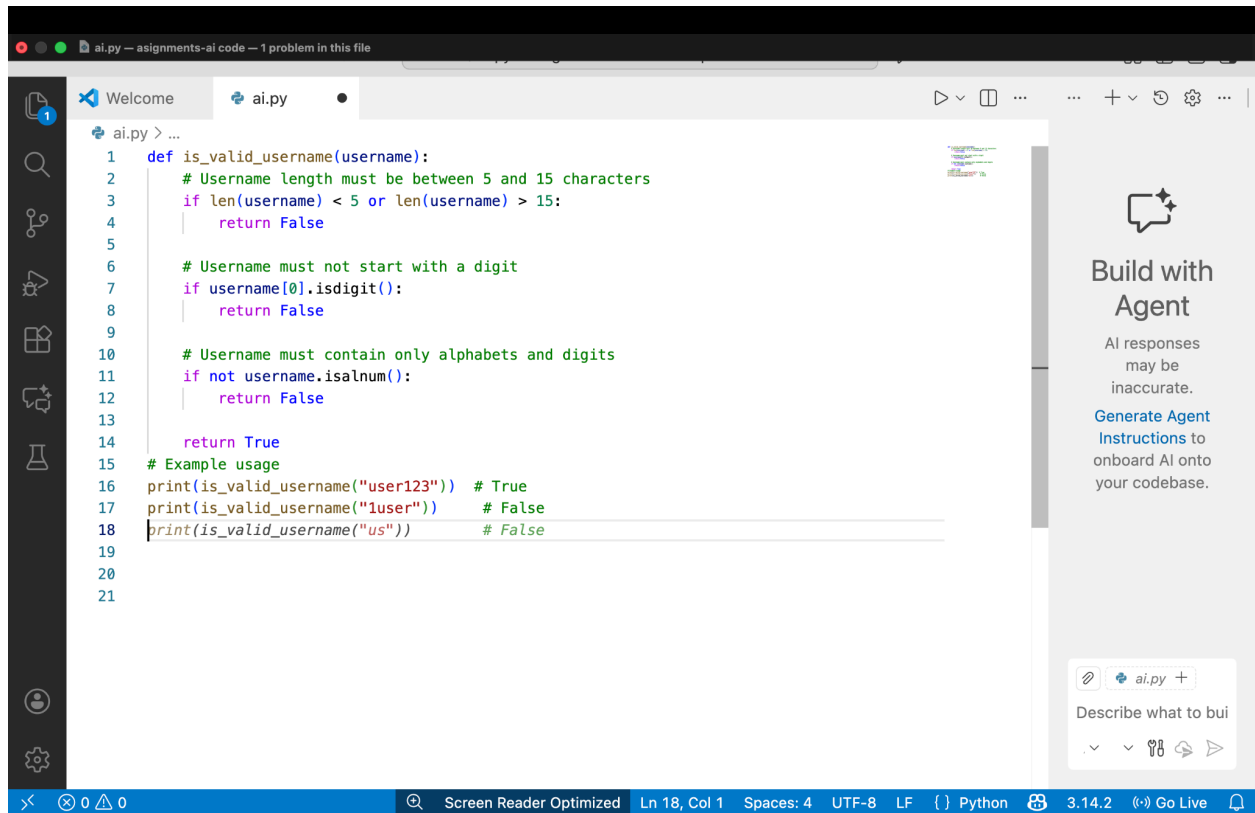
```
assert is_valid_username("User123") == True
```

```
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 with a file named `ai.py`. The code defines a function `is_valid_username(username)` that checks if a username is valid based on three criteria: length (5-15 characters), starting character (not a digit), and character set (alphanumeric). It includes example usage prints for `"user123"`, `"luser"`, and `"us"`. The status bar at the bottom indicates the file is Python 3.14.2, UTF-8, and LF line endings.

```
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("luser"))  # False
18 print(is_valid_username("us"))     # False
19
20
21
```

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

Example Assert Test Cases:

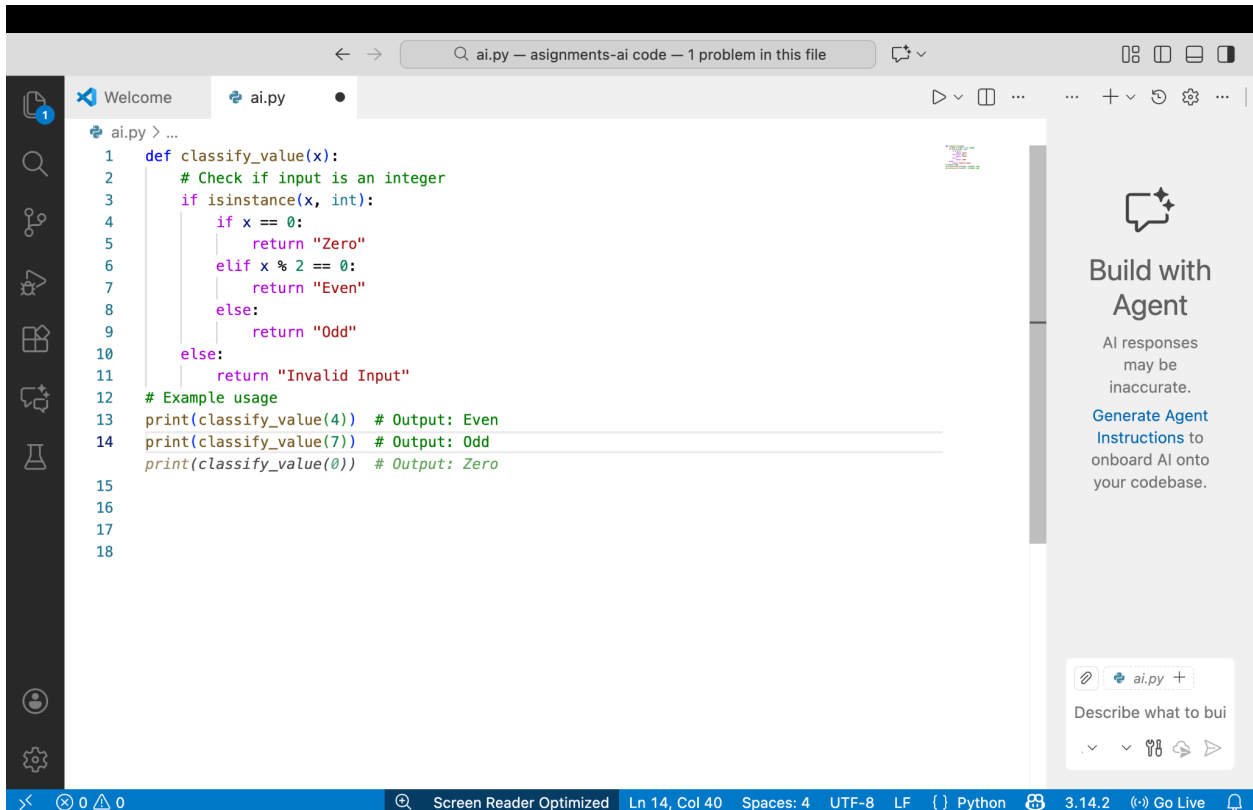
```
assert classify_value(8) == "Even"
```

```
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 with a file named 'ai.py'. The code defines a function 'classify_value(x)' that checks if the input is an integer. If it is, it returns 'Zero' for 0, 'Even' for even numbers, and 'Odd' for odd numbers. If the input is not an integer, it returns 'Invalid Input'. Below the function, there are three example usage lines: 'print(classify_value(4)) # Output: Even', 'print(classify_value(7)) # Output: Odd', and 'print(classify_value(0)) # Output: Zero'. The editor interface includes a search bar at the top, a sidebar on the left with various icons, and a status bar at the bottom showing 'Ln 14, Col 40', 'Spaces: 4', 'UTF-8', 'LF', 'Python', and '3.14.2'. On the right side, there is a panel titled 'Build with Agent' with a warning that 'AI responses may be inaccurate' and a button to 'Generate Agent Instructions to onboard AI onto your codebase.' At the bottom right, there is a small input field with the text 'Describe what to bui' and some icons.

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

Example Assert Test Cases:

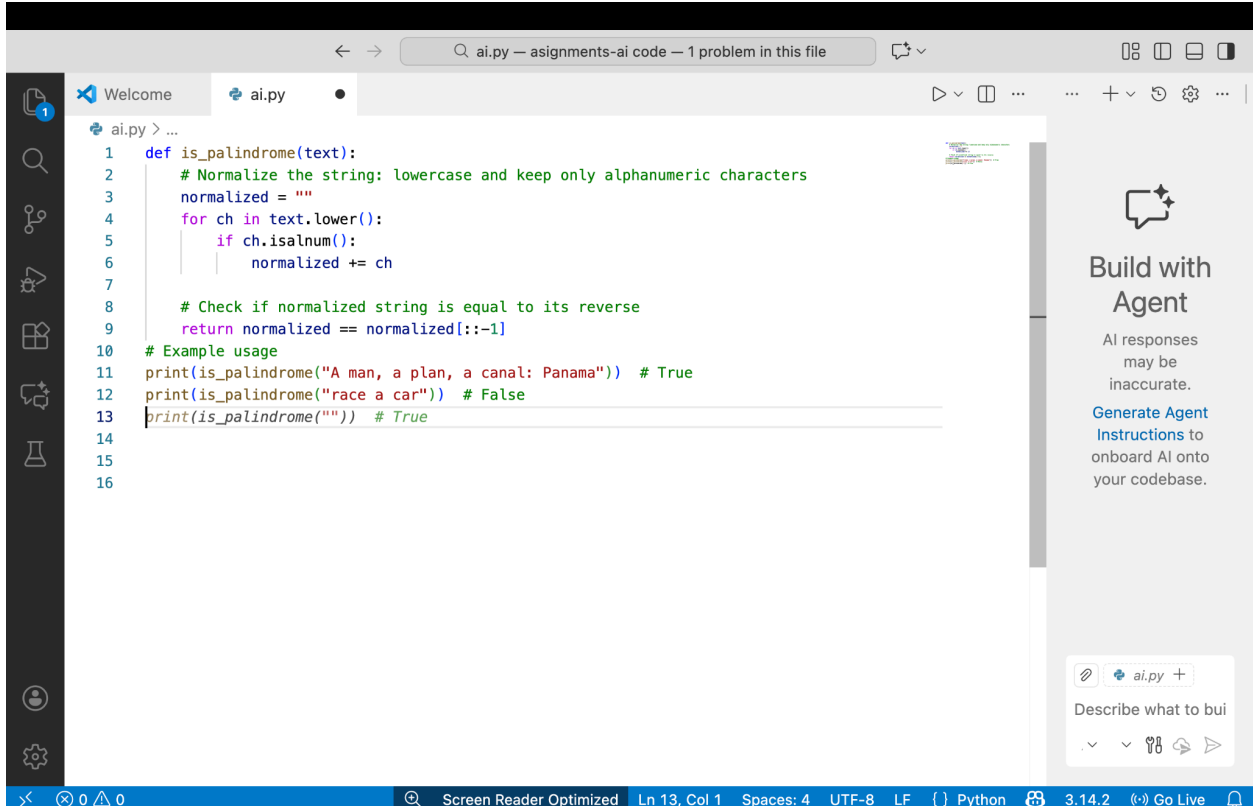
```
assert is_palindrome("Madam") == True
```

```
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 with a file named `ai.py`. The code defines a function `is_palindrome(text)` that normalizes the input string (lowercase, alphanumeric) and checks if it is equal to its reverse. Below the function, there are three example usage lines with comments indicating the expected output: `print(is_palindrome("A man, a plan, a canal: Panama")) # True`, `print(is_palindrome("race a car")) # False`, and `print(is_palindrome("")) # True`. The editor interface includes a sidebar with icons, a top bar with search and file management, and a bottom status bar showing "Screen Reader Optimized" and other details.

```
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
11 # Example usage
12 print(is_palindrome("A man, a plan, a canal: Panama")) # True
13 print(is_palindrome("race a car")) # False
14 print(is_palindrome("")) # True
```

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:

o `deposit(amount)`

o `withdraw(amount)`

o `get_balance()`

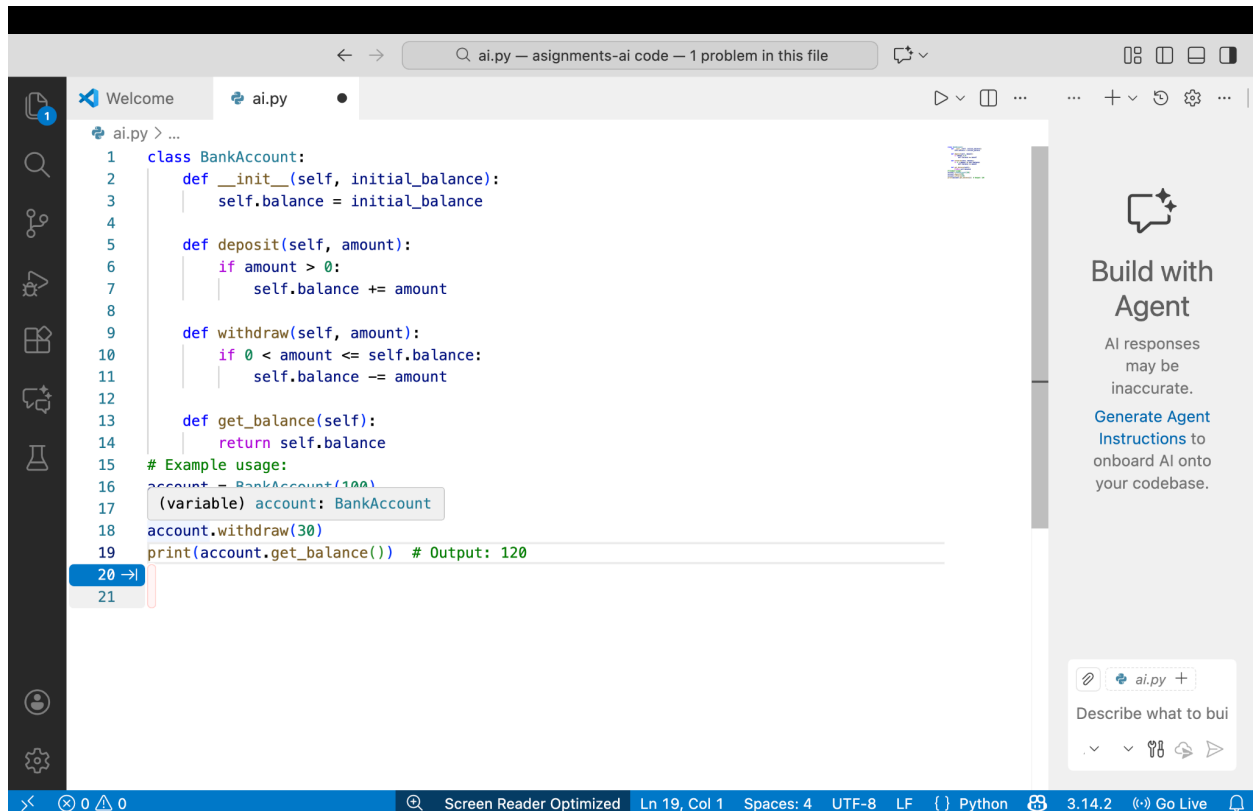
Example Assert Test Cases:

```
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.



The screenshot shows a code editor with a file named 'ai.py'. The code defines a class 'BankAccount' with methods for deposit, withdraw, and get_balance. It also includes example usage code that creates an account, withdraws 30, and prints the balance, which is 120. The editor interface includes a sidebar with icons, a top bar with search and file management, and a bottom status bar showing 'Screen Reader Optimized', 'Ln 19, Col 1', 'Spaces: 4', 'UTF-8', 'LF', 'Python', '3.14.2', and 'Go Live'.

```
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
16    # Example usage:
17    account = BankAccount(100)
18    (variable) account: BankAccount
19    account.withdraw(30)
20    print(account.get_balance()) # Output: 120
21
```

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:

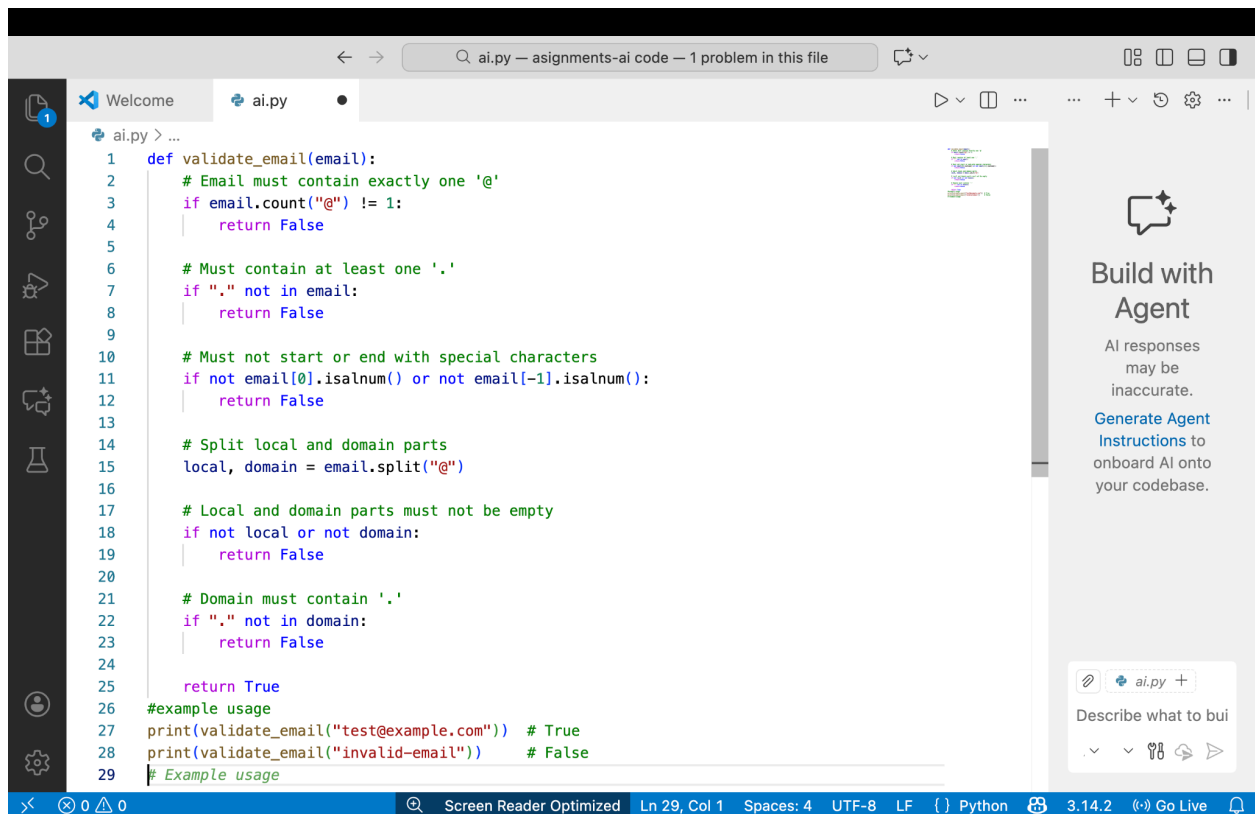
```
assert validate_email("user@example.com") == True
```

```
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 with a file named `ai.py`. The code defines a `validate_email(email)` function that checks for valid email format. It includes comments for each validation step: checking for exactly one '@', at least one '.', no special characters at the start or end, non-empty local and domain parts, and a domain with at least one dot. Below the function, there are example usage lines: `print(validate_email("test@example.com"))` (commented as `# True`) and `print(validate_email("invalid-email"))` (commented as `# False`). The status bar at the bottom indicates the file is Python 3.14.2, screen reader optimized, and has 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
27 #example usage
28 print(validate_email("test@example.com")) # True
29 print(validate_email("invalid-email")) # False
30 # Example usage
```