Inclass Exercise - 3
Saksham Bansal
33076427

I. Part-1:

1.

is_palindrome(s: str) → bool
is_prime(n: int) → bool

Natural language descriptions:

is_palindrome:
A function that determines whether a given string is a palindrome. A palindrome reads the same forward and backward after normalizing the string by removing spaces, ignoring punctuation, and converting all characters to lowercase.

is_prime:
A function that checks whether an integer n is a prime number. A prime is greater than 1 and has no divisors other than 1 and itself. Negative numbers, 0, and 1 are not prime. Even numbers greater than 2 are not prime. For n > 3, primality depends on whether any integer from 2 up to root(n) divides n.

LLM Prompts:

- is_palindrome:
  Please write formal specifications as Python `assert` statements that describe the correct behavior of this method.
  Use the variable `res` to denote the expected logical output of is_palindrome(s).
  Do NOT call is_palindrome() in any assertion.
  Do NOT use any method with side effects such as print(), input(), file I/O, randomness, or modifying data structures.
  Express relationships using ONLY pure logic (string properties, normalization rules, equality conditions, arithmetic, Boolean logic, etc.).
  Produce around 5 assertions. Each assertion should be independent and describe a logical property of palindromes after normalization.

- is_prime:
  Please write formal specifications as Python `assert` statements that describe the correct behavior of this method.
  Use the variable `res` to denote the expected logical output of is_prime(n).
  Do NOT call is_prime() in any assertion.
  Do NOT use side-effect methods such as print(), input(), reading/writing files, randomness, or mutation operations.
  Use ONLY pure arithmetic, divisibility logic, inequalities, and Boolean relationships.
  Produce around 5 assertions. Each assertion should describe a correct logical property of prime numbers.

2. Assertion Table

- is_palindrome:

```
# 1. res is True exactly when normalized equals its reverse
assert res == (normalized == normalized[::-1])

# 2. Symmetric characters must match when res is True
assert (not res) or (normalized == normalized[::-1])

# 3. Empty normalized string is always a palindrome
assert (normalized != "") or res

# 4. For odd-length strings, removing the middle char should not change palindrome property
if len(normalized) % 2 == 1:
    mid = len(normalized) // 2
    reduced = normalized[:mid] + normalized[mid+1:]
    assert res == (reduced == reduced[::-1])
```

Assertion Number	Incorrect / Correct	Notes	Corrected Assertion
1	Correct		
2	Correct		
3	Incorrect	Logical form is too weak and does not correctly enforce that empty normalized strings must imply res ==	assert (normalized != "") or (res == True)

		True	
4	Incorrect	Contains control flow (if) and string reconstruction, which violates the requirement for simple, non-computational formal specifications.	assert (not res) or (normalized == normalized[::-1])

Accuracy Rate = 2/4 = 50%

- is_prime

```
# 1. Integers <= 1 are not prime.
assert (n <= 1) == (res == False)

# 2. 2 is the only even prime.
assert (n == 2) == (res == True)

# 3. Even numbers > 2 are not prime.
assert (n > 2 and n % 2 == 0) == (res == False)

# 4. If n is prime, then no d in [2, sqrt(n)] divides n.
assert (not res) or all(n % d != 0 for d in range(2, int(n**0.5) + 1))

# 5. If n is composite (>1), some d in [2, sqrt(n)] divides n.
assert (res == False and n > 1) == any(n % d == 0 for d in range(2, int(n**0.5) + 1))
```

Assertion No	Incorrect / Correct	Notes	Corrected Assertion
1	Correct		
2	Correct		

3	Correct		
4	Incorrect	Uses procedural iteration instead of pure logical specification.	assert (not res) or (for all integers d such that 2 <= d <= int(n**0.5), n % d != 0)
5	Incorrect	Uses generator computation instead of declarative logic.	assert (not(res == False and n > 1)) or (there exists d with 2 <= d <= int(n**0.5) and n % d == 0)

Accuracy Rate = 3/5 = 60%

II. PART - 2

1. LLM Prompts based on final specifications:

- is_palindrome

  Using the corrected specifications below, generate Python pytest unit tests for the function is_palindrome(s: str).
  Use ONLY these logical rules. Do NOT invent new rules or behaviors beyond these specifications.

  Corrected specifications:
  1. assert res == (normalized == normalized[::-1])
  2. assert (not res) or (normalized == normalized[::-1])
  3. assert (normalized != "") or (res == True)
  4. assert (not res) or (normalized == normalized[::-1])

  Task:
  Generate test cases that thoroughly reflect these logical specifications.
  Include edge cases such as: empty normalized string, strings whose normalized version is a palindrome,
  strings whose normalized version is not a palindrome, and examples involving uppercase, punctuation, and spacing.

- is_prime:
  Using the corrected specifications below, generate Python pytest unit tests for the function is_prime(n: int).
  Use ONLY these logical rules. Do NOT invent new rules or behaviors beyond these specifications.

  Corrected specifications:
  1. assert (n <= 1) == (res == False)
  2. assert (n == 2) == (res == True)
  3. assert ((n > 2 and n % 2 == 0) == (res == False))
  4. assert (not res) or (for all integers d such that 2 <= d <= int(n**0.5), n % d != 0)
  5. assert (res == False and n > 1) implies (there exists an integer d such that 2 <= d <= int(n**0.5) and n % d == 0)

  Task:
  Generate test cases that thoroughly reflect these logical specifications.
  Include: numbers <= 1, the number 2, even numbers > 2, odd primes, composite numbers, and boundary values.

SPEC GUIDED TEST CASES:

1. is_palindrome:

```python
import pytest
from cs520_inclass_exercise1codes import is_palindrome

# Helper to compute normalized version (same definition used in specification)
def norm(s):
    return "".join(ch.lower() for ch in s if ch.isalnum())
# 1. Empty normalized string → must be palindrome (res == True)
def test_palindrome_spec_empty_normalized():
    s = "!!! ???"
    assert norm(s) == ""
    assert is_palindrome(s) == True
# 2. Normalized string equals its reverse → res must be True
def test_palindrome_spec_true_basic():
    s = "A man, a plan, a canal: Panama"
    assert norm(s) == norm(s)[::-1]
    assert is_palindrome(s) == True
# 3. Normalized palindrome with numbers
def test_palindrome_spec_numeric_palindrome():
    s = "12321"
    assert norm(s) == norm(s)[::-1]
    assert is_palindrome(s) == True
# 4. Normalized not equal to reverse → res must be False
def test_palindrome_spec_false_basic():
    s = "Hello"
    assert norm(s) != norm(s)[::-1]
    assert is_palindrome(s) == False
# 5. Uppercase + punctuation: still must follow normalized palindrome rule
def test_palindrome_spec_case_insensitive():
    s = "RaceCar!!!"
    assert norm(s) == "racecar"
    assert is_palindrome(s) == True
# 6. Mixed alphanumeric where normalized breaks symmetry
def test_palindrome_spec_mixed_false():
    s = "abc123"
    assert norm(s) != norm(s)[::-1]
    assert is_palindrome(s) == False
```

2. is_prime:

```python
import pytest
from cs520_inclass_exercise1codes import is_prime

# 1. All n <= 1 must return False
def test_prime_spec_nonpositive():
    for n in [-10, -1, 0, 1]:
        assert is_prime(n) == False

# 2. 2 must be True (only even prime)
def test_prime_spec_two():
    assert is_prime(2) == True

# 3. Even numbers > 2 must be False
def test_prime_spec_even_composites():
    for n in [4, 6, 8, 10, 100]:
        assert n % 2 == 0 and n > 2
        assert is_prime(n) == False

# 4. Odd primes → must satisfy rule that no d divides n
def test_prime_spec_odd_primes():
    for n in [3, 5, 7, 11, 13, 17, 19]:
        assert is_prime(n) == True

# 5. Composite odd numbers → must return False
def test_prime_spec_composite_odd():
    for n in [9, 15, 21, 25, 27, 33]:
        assert is_prime(n) == False

# 6. Specification boundary: large primes and large composites
def test_prime_spec_large_values():
    assert is_prime(97) == True    # prime
    assert is_prime(99) == False   # composite
```

Case Specific Insights:

is_palindorme:

The specification-guided tests revealed multiple failing cases for is_palindrome, specifically for inputs where the normalized string should be considered a palindrome. The original implementation did not remove punctuation, ignore spaces, or convert characters to lowercase, which caused the tests for empty-normalized strings, mixed-case palindromes, and punctuation-heavy inputs to fail. These failures demonstrate that the formal specifications uncovered real gaps in the function's logic that were not detected by the baseline tests from Exercise 2. Even though coverage increased, the spec-guided tests identified correctness issues rather than coverage issues, validating the importance of specification-based testing.

is_prime:

The spec-guided tests for is_prime covered all major categories required by the specifications, including nonpositive numbers, even composites, the special case of 2, odd primes, and composite odds. All tests passed, which confirmed that the implementation correctly followed each logical rule derived from the formal specifications. Unlike the palindrome problem, the prime function's behavior was already aligned with the specification, so the main benefit was ensuring complete branch coverage rather than discovering faults.

Function	Old %	New %	Old Branch %	New Branch %
is_palindrome	93%	62%	85%	9%
is_prime	98%	62%	85%	9%