IT314 - Software Engineering
Lab 8 : Functional Testing (Black Box)

- Maitrey Pandya - 202201335

Q1. Consider a program for determining the previous date. Its input is triple of day, month and year with the following ranges 1 <= month <= 12, 1 <= day <= 31, 1900 <= year <= 2015.The possible output dates would be previous date or invalid date. Design the equivalence class test cases? Write a set of test cases (i.e., test suite) – specific set of data – to properly test the programs. Your test suite should include both correct and incorrect inputs.

1. Enlist which set of test cases have been identified using Equivalence Partitioning and Boundary Value Analysis separately.

2. Modify your programs such that it runs, and then execute your test suites on the program. While executing your input data in a program, check whether the identified expected outcome (mentioned by you) is correct or not.

A1.
Equivalence Partitioning (EP)

- Valid Classes (Inputs)**:**
  - Day: 1≤day≤31
  - Month: 1≤month≤12
  - Year: 1900≤year≤2015
  - Leap year: February can have 29 days.
  - Non-leap year: February can have only 28 days.
- Invalid Classes (Inputs)**:**
  - Invalid Day : day>31 and day<1
  - Invalid Month : month>12 and month < 1
  - Invalid Year : year>2015 and year<1900
  - Invalid Day-Month combinations (e.g., 31st February, 30th February, 31st April)

Boundary Value Analysis

- Valid Boundaries:
  - Day: day=1, day=31 (for months with 31 days)
  - Month: month=1,month=12
  - Year: year=1900, year=2015
  - February Leap Year Boundary: day=29 in a leap year.

- Invalid Boundaries:
    - Day: day=0, day=32
    - Month: month=0, month=13
    - Year: year=1899, year=2016

Test Cases:

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	1-1-1900	Invalid Date	BVA (lower)
TC2	32-5-2010	Invalid Date	EP (Invalid Day)
TC3	0-10-2005	Invalid Date	BVA (Invalid Day)
TC4	15-12-2010	14-12-2010	EP (Valid Date)
TC5	31-1-2015	30-1-2015	BVA (upper day)
TC6	29-2-2000	28-2-2000	EP (Valid Leap Year)
TC7	28-2-2001	27-2-2001	EP (Valid Non-Leap Year)
TC8	31-4-2004	Invalid Date	EP (Invalid Day-Month)
TC9	30-12-2015	29-12-2015	EP (Valid Date)
TC10	1-3-1900	28-2-1900	BVA (Leap Year Check)
TC11	2-13-1999	Invalid Date	BVA (Invalid Month)
TC12	3-0-2000	Invalid Date	BVA (Invalid Month)
TC13	1-12-1899	Invalid Date	BVA (Invalid Year)
TC14	2-3-2016	Invalid Date	BVA (Invalid Year)

Code :

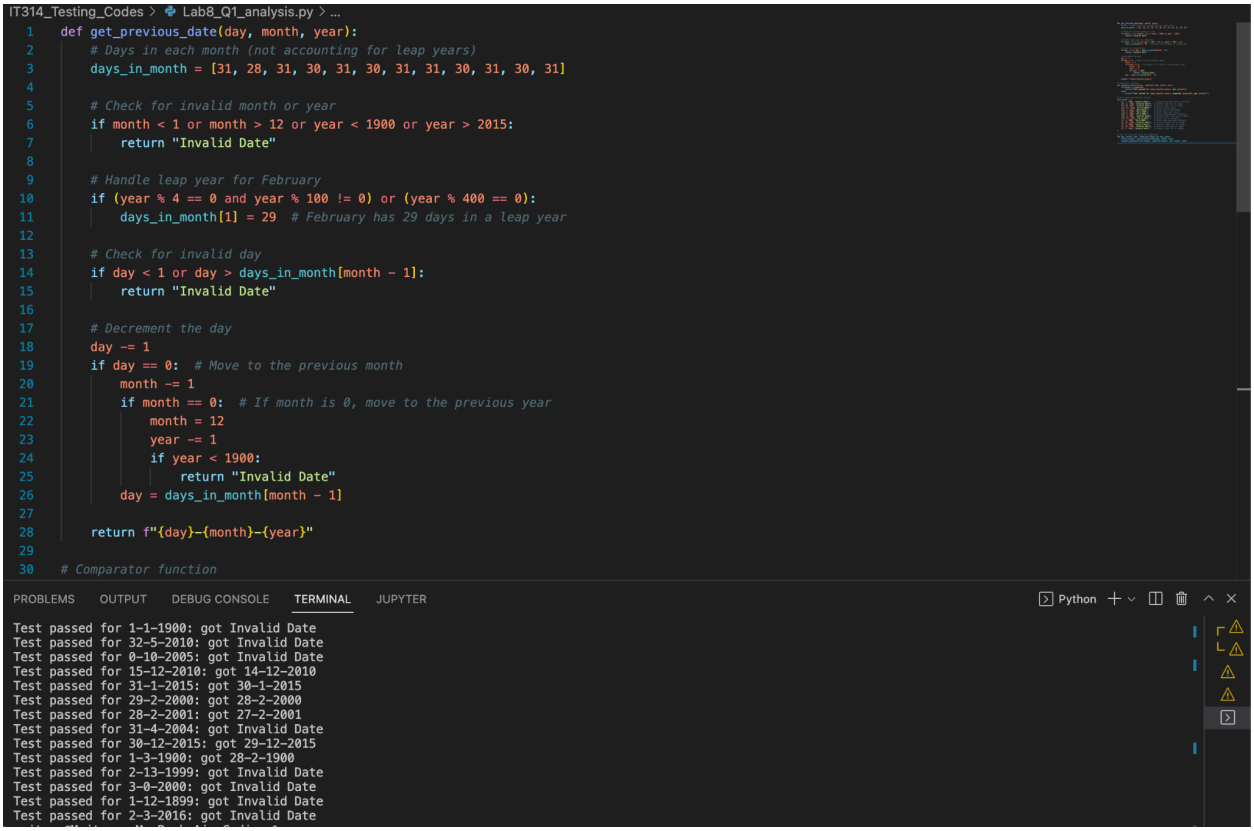
```
def get_previous_date(day, month, year):

    # Days in each month (not accounting for leap years)

    days_in_month = [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31]



    # Check for invalid month or year

    if month < 1 or month > 12 or year < 1900 or year > 2015:

        return "Invalid Date"
```

```
    # Handle leap year for February
    if (year % 4 == 0 and year % 100 != 0) or (year % 400 == 0):
        days_in_month[1] = 29  # February has 29 days in a leap year

    # Check for invalid day
    if day < 1 or day > days_in_month[month - 1]:
        return "Invalid Date"

    # Decrement the day
    day -= 1
    if day == 0:  # Move to the previous month
        month -= 1
        if month == 0:  # If month is 0, move to the previous year
            month = 12
            year -= 1
            if year < 1900:
                return "Invalid Date"
        day = days_in_month[month - 1]

    return f"{day}-{month}-{year}"

# Comparator function
def compare_output(actual, expected, day, month, year):
```

```
    if actual == expected:
        print(f"Test passed for {day}-{month}-{year}: got {actual}")
    else:
        print(f"Test failed for {day}-{month}-{year}: expected {expected}, got {actual}")


# Test cases and expected outputs
test_cases = [
    (1, 1, 1900, "Invalid Date"),     # Expected Invalid but listed here
    (32, 5, 2010, "Invalid Date"),  # Invalid (day out of range)
    (0, 10, 2005, "Invalid Date"),  # Invalid (day out of range)
    (15, 12, 2010, "14-12-2010"),    # Valid (normal case)
    (31, 1, 2015, "30-1-2015"),      # Valid (end of the month)
    (29, 2, 2000, "28-2-2000"),      # Valid (leap year case)
    (28, 2, 2001, "27-2-2001"),      # Valid (non-leap year February)
    (31, 4, 2004, "Invalid Date"),  # Invalid (April has only 30 days)
    (30, 12, 2015, "29-12-2015"),    # Valid (end of December)
    (1, 3, 1900, "28-2-1900"),       # Valid (non-leap year February)
    (2, 13, 1999, "Invalid Date"),  # Invalid (month out of range)
    (3, 0, 2000, "Invalid Date"),    # Invalid (month out of range)
    (1, 12, 1899, "Invalid Date"),  # Invalid (year out of range)
    (2, 3, 2016, "Invalid Date")     # Invalid (year out of range)
]
```

```
# Test the function using the comparator

for day, month, year, expected_output in test_cases:

   actual_output = get_previous_date(day, month, year)

   compare_output(actual_output, expected_output, day, month, year)
```

Output :



Q.2. Programs:

P1. The function linearSearch searches for a value v in an array of integers a. If v appears in the array a, then the function returns the first index i, such that a[i] == v; otherwise, -1 is returned.

```
int linearSearch(int v, int a[])
{
        int i = 0;
```

```
        while (i < a.length)
        {
                if (a[i] == v)
                        return(i);
                i++;
        }
        return (-1);
}
```

A P1.

Equivalence Partitioning (EP)

- Valid case (EP1): v exists in the array, and there is a valid index returned.
- Invalid case (EP2): v does not exist in the array, and −1 is returned.
- Array is empty(EP3) .
- Invalid Input(EP4) : Character Input in searched item
- Invalid Input(EP5): Float input in searched item
- Invalid Input(EP6) : Character input in array
- Invalid Input(EP7): Float input in array.

Boundary Value Analysis (BVA)

- Lower boundary (BVA1): v is at the first position in the array.
- Upper boundary (BVA2): v is at the last position in the array.
- Boundary (BVA3): Array of size 1, and v either exists or does not exist.

Test Cases:

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	3, [1, 2, 3, 4, 5]	2	EP (Valid Case)
TC2	6, [1, 2, 3, 4, 5]	-1	EP (Invalid Case)
TC3	3, []	-1	EP (Empty Array)
TC4	‘a', [1, 2, 3]	Invalid Input	EP (Character Input)

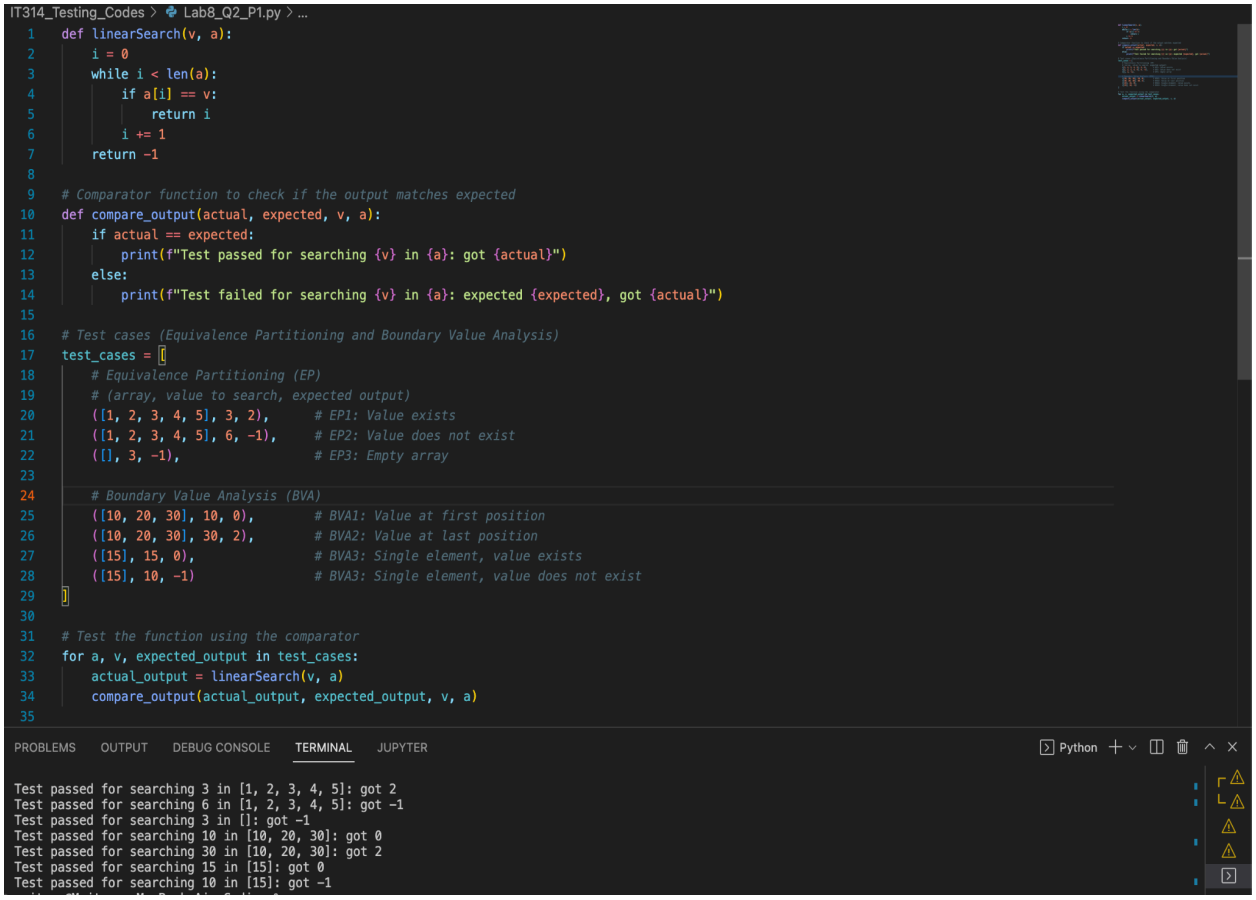
TC5	2.5, [1, 2, 3]	Invalid Input	EP (Float Input)
TC6	3, [1, 2, 'c']	Invalid Array	EP (Character in Array)
TC7	3, [1, 2, 2.5]	Invalid Array	EP (Float in Array)
TC8	1, [1, 2, 3, 4, 5]	0	BVA (Lower Boundary)
TC9	5, [1, 2, 3, 4, 5]	4	BVA (Upper Boundary)
TC10	1, [1]	0	BVA (Single Element)
TC11	2, [1]	-1	BVA (Single Element - Not Found)

Code:

```
def linearSearch(v, a):
    i = 0
    while i < len(a):
        if a[i] == v:
            return i
        i += 1
    return -1
# Comparator function to check if the output matches expected
def compare_output(actual, expected, v, a):
    if actual == expected:
        print(f"Test passed for searching {v} in {a}: got {actual}")
    else:
        print(f"Test failed for searching {v} in {a}: expected {expected}, got
{actual}")
# Test cases (Equivalence Partitioning and Boundary Value Analysis)
test_cases = [
    # Equivalence Partitioning (EP)
    # (array, value to search, expected output)
    ([1, 2, 3, 4, 5], 3, 2),       # EP1: Value exists
    ([1, 2, 3, 4, 5], 6, -1),      # EP2: Value does not exist
    ([], 3, -1),                   # EP3: Empty array
     # Boundary Value Analysis (BVA)
    ([10, 20, 30], 10, 0),         # BVA1: Value at first position
    ([10, 20, 30], 30, 2),         # BVA2: Value at last position
    ([15], 15, 0),                 # BVA3: Single element, value exists
    ([15], 10, -1)                 # BVA3: Single element, value does not exist
]

# Test the function using the comparator
```

```
for a, v, expected_output in test_cases:
    actual_output = linearSearch(v, a)
    compare_output(actual_output, expected_output, v, a)
```

Output :



P2.The function countItem returns the number of times a value v appears in an array of integers a.

```
int countItem(int v, int a[])
{
        int count = 0;
        for (int i = 0; i < a.length; i++)
        {
                if (a[i] == v)
                count++;

        }
```

```
        return (count);
    }
```

A P2.
Equivalence Partitioning (EP)

- EP1: v exists in the array, and the count is greater than 0.
- EP2: v does not exist in the array, and the count is 0.
- EP3: The array is empty, and the count is 0.
- EP4: v is a character input.
- EP5: v is a float input.
- EP6: The array contains characters.
- EP7: The array contains floats.

Boundary Value Analysis (BVA)

- BVA1: All elements in the array match v.
- BVA2: No elements in the array match v.
- BVA3: The array has a single element that matches v.
- BVA4: The array has a single element that does not match v.

Test Cases:

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	3, [1, 2, 3, 3, 4, 5]	2	EP (Valid Case)
TC2	6, [1, 2, 3, 3, 4, 5]	0	EP (Invalid Case)
TC3	3, []	0	EP (Empty Array)
TC4	‘a', [1, 2, 3]	Invalid Input	EP (Character Input)
TC5	2.5, [1, 2, 3]	Invalid Input	EP (Float Input)
TC6	3, [1, 2, 'c']	Invalid Array	EP (Character in Array)
TC7	3, [1, 2, 2.5]	Invalid Array	EP (Float in Array)
TC8	1, [1, 1, 1, 1, 1]	5	BVA (All Elements Match)
TC9	1, [2, 2, 2, 2]	0	BVA (No Matches)

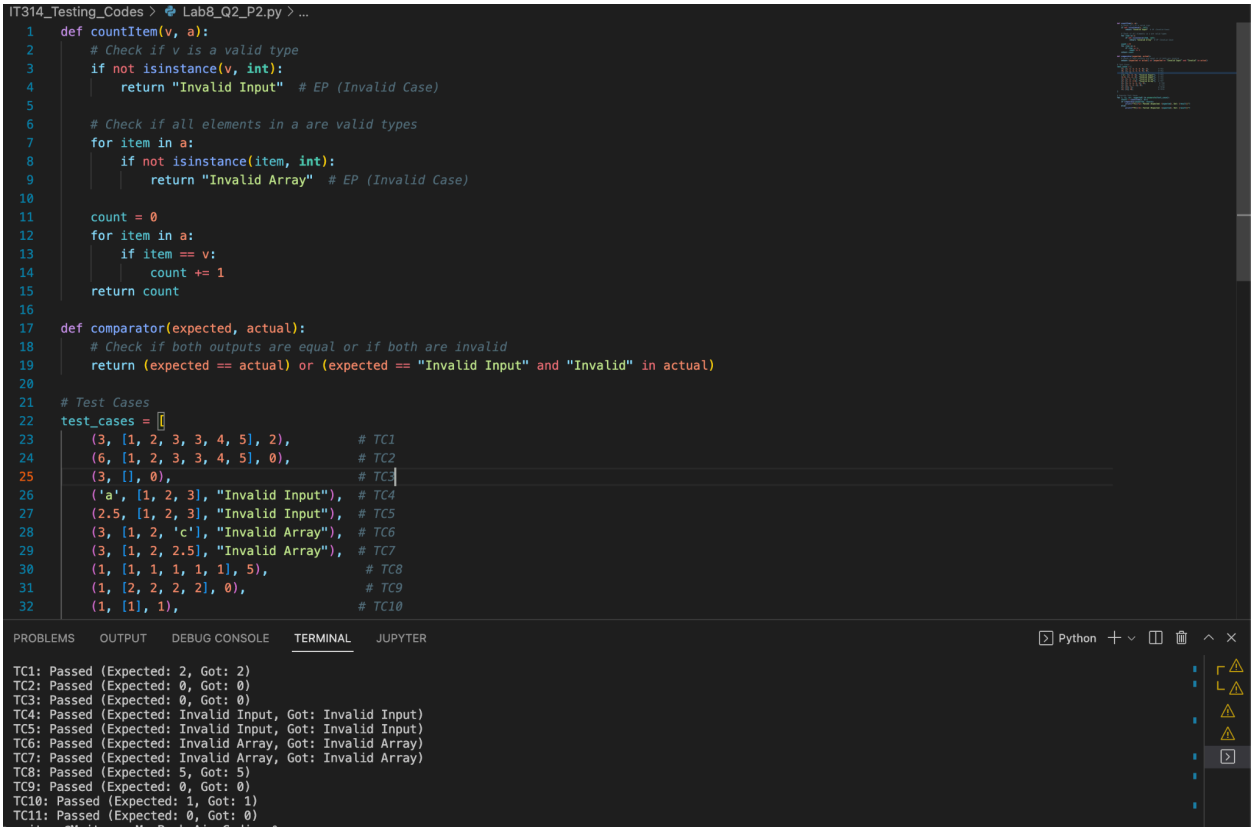
TC10	1, [1]	1	BVA (Single Element - Match)
TC11	2, [1]	0	BVA (Single Element - No Match)

Code:

```
def countItem(v, a):
    # Check if v is a valid type
    if not isinstance(v, int):
        return "Invalid Input"  # EP (Invalid Case)

    # Check if all elements in a are valid types
    for item in a:
        if not isinstance(item, int):
            return "Invalid Array"  # EP (Invalid Case)

    count = 0
    for item in a:
        if item == v:
            count += 1
    return count

def comparator(expected, actual):
    # Check if both outputs are equal or if both are invalid
    return (expected == actual) or (expected == "Invalid Input" and "Invalid" in
actual)

# Test Cases
test_cases = [
    (3, [1, 2, 3, 3, 4, 5], 2),          # TC1
    (6, [1, 2, 3, 3, 4, 5], 0),          # TC2
    (3, [], 0),                          # TC3
    ('a', [1, 2, 3], "Invalid Input"),   # TC4
    (2.5, [1, 2, 3], "Invalid Input"),   # TC5
    (3, [1, 2, 'c'], "Invalid Array"),   # TC6
    (3, [1, 2, 2.5], "Invalid Array"),   # TC7
    (1, [1, 1, 1, 1, 1], 5),              # TC8
    (1, [2, 2, 2, 2], 0),                 # TC9
    (1, [1], 1),                         # TC10
    (2, [1], 0),                         # TC11
]
```

```python
# Execute Test Cases
for i, (v, arr, expected) in enumerate(test_cases):
    result = countItem(v, arr)
    if comparator(expected, result):
        print(f"TC{i+1}: Passed (Expected: {expected}, Got: {result})")
    else:
        print(f"TC{i+1}: Failed (Expected: {expected}, Got: {result})")
```

Output :



P3. The function binarySearch searches for a value v in an ordered array of integers a. If v appears in the array a, then the function returns an index i, such that a[i] == v; otherwise, -1 is returned.
Assumption: the elements in the array are sorted in non-decreasing order.

```
int binarySearch(int v, int a[])
{
        int lo,mid,hi;
        lo = 0;
        hi = a.length-1;
```

```
        while (lo <= hi)
        {
                mid = (lo+hi)/2;
                if (v == a[mid])
                        return (mid);
                else if (v < a[mid])
                        hi = mid-1;
                else
                        lo = mid+1;

        }
        return(-1);
}
```

A P3.
Equivalence Partitioning (EP)

- EP1: v exists in the ordered array, and a valid index is returned.
- EP2: v does not exist in the ordered array, and -1 is returned.
- EP3: The array is empty, and -1 is returned.
- EP4: The input array is unsorted.
- EP5: v is a character input.
- EP6: v is a float input.
- EP7: The array contains characters.
- EP8: The array contains floats.

Boundary Value Analysis (BVA)

- BVA1: v is the first element in the ordered array.
- BVA2: v is the last element in the ordered array.
- BVA3: The array has a single element that matches v.
- BVA4: The array has a single element that does not match v.

Test Cases:

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	3, [1, 2, 3, 4, 5]	2	EP (Valid Case)
TC2	6, [1, 2, 3, 4, 5]	-1	EP (Invalid Case)
TC3	3, []	-1	EP (Empty Array)

TC4	a', [1, 2, 3, 4, 5]	"Invalid Input"	EP (Character Input)
TC5	2.5, [1, 2, 3, 4, 5]	"Invalid Input"	EP (Float Input)
TC6	3, ['a', 'b', 'c']	"Invalid Array"	EP (Character in Array)
TC7	3, [1, 2, 2.5, 3, 4, 5]	"Invalid Array"	EP (Float in Array)
TC8	1, [1]	0	BVA (Single Element - Match)
TC9	2, [1]	-1	BVA (Single Element - No Match)
TC10	1, [1, 2, 3, 4, 5]	0	BVA (First Element)
TC11	5, [1, 2, 3, 4, 5]	4	BVA (Last Element)
TC12	4, [1, 2, 3, 4, 5]	3	BVA (Middle Element)
TC13	1, [2, 3, 4, 5]	-1	BVA (Element Not Found)
TC14	3, [3, 2, 1]	"Invalid Array"	EP (Unsorted Array)

Code :

```
def binarySearch(v, a):

   # Check if v is a valid type

   if not isinstance(v, int):

       return "Invalid Input"  # EP (Invalid Case)

     # Check if all elements in a are valid types

   for item in a:

       if not isinstance(item, int):

           return "Invalid Array"  # EP (Invalid Case)

   # Check if the array is sorted

   if a != sorted(a):

       return "Invalid Array"  # EP (Unsorted Case)

   lo = 0

   hi = len(a) - 1

   while lo <= hi:
```

```
        mid = (lo + hi) // 2
        if v == a[mid]:
            return mid
        elif v < a[mid]:
            hi = mid - 1
        else:
            lo = mid + 1
    return -1
def comparator(expected, actual):
    # Check if both outputs are equal or if both indicate invalid inputs
    return (expected == actual) or (expected == "Invalid Input" and "Invalid" in
actual) or (expected == "Invalid Array" and "Invalid" in actual)
# Test Cases
test_cases = [
    (3, [1, 2, 3, 4, 5], 2),              # TC1
    (6, [1, 2, 3, 4, 5], -1),             # TC2
    (3, [], -1),                          # TC3
    ('a', [1, 2, 3, 4, 5], "Invalid Input"),  # TC4
    (2.5, [1, 2, 3, 4, 5], "Invalid Input"),  # TC5
    (3, ['a', 'b', 'c'], "Invalid Array"),  # TC6
    (3, [1, 2, 2.5, 3, 4, 5], "Invalid Array"),  # TC7
    (1, [1], 0),                          # TC8
    (2, [1], -1),                         # TC9
    (1, [1, 2, 3, 4, 5], 0),              # TC10
```

```
    (5, [1, 2, 3, 4, 5], 4),              # TC11
    (4, [1, 2, 3, 4, 5], 3),              # TC12
    (1, [2, 3, 4, 5], -1),                # TC13
    (3, [3, 2, 1], "Invalid Array"),      # TC14
]

# Execute Test Cases
for i, (v, arr, expected) in enumerate(test_cases):
    result = binarySearch(v, arr)
    if comparator(expected, result):
        print(f"TC{i+1}: Passed (Expected: {expected}, Got: {result})")
    else:
        print(f"TC{i+1}: Failed (Expected: {expected}, Got: {result})")
```

Output:

P4. The following problem has been adapted from The Art of Software Testing, by G. Myers (1979).The function triangle takes three integer parameters that are interpreted as the lengths of the sides of a triangle. It returns whether the triangle is equilateral (three lengths equal), isosceles (two lengths equal), scalene (no lengths equal), or invalid (impossible lengths).

```
final int EQUILATERAL = 0;
final int ISOSCELES = 1;
final int SCALENE = 2;
final int INVALID = 3;
int triangle(int a, int b, int c)
{
        if (a >= b+c || b >= a+c || c >= a+b)
                return(INVALID);
        if (a == b && b == c)
                return(EQUILATERAL);
        if (a == b || a == c || b == c)
                return(ISOSCELES);
        return(SCALENE);
}
```

A P4.

Equivalence Partitioning (EP)

- EP1: Valid equilateral triangle (all sides equal).
- EP2: Valid isosceles triangle (two sides equal).
- EP3: Valid scalene triangle (all sides different).
- EP4: Invalid triangle (sum of any two sides is less than or equal to the third side).
- EP5: Input is a character.
- EP6: Input is a float.
- EP7: Sides have negative lengths.

Boundary Value Analysis (BVA)

- BVA1: The smallest valid triangle (1, 1, 1).
- BVA2: The smallest invalid triangle (1, 1, 2).
- BVA3: Sides are zero (0, 0, 0) which should return invalid.

Test Cases:

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	3, 3, 3	0	EP (Equilateral)

TC2	5, 5, 8	1	EP (Isosceles)
TC3	3, 4, 5	2	EP (Scalene)
TC4	1, 1, 2	3	EP (Invalid)
TC5	-1, 1, 1	3	EP (Invalid - Negative Side)
TC6	a', 1, 1	"Invalid Input"	EP (Character Input)
TC7	1.5, 1.5, 1.5	"Invalid Input"	EP (Float Input)
TC8	0, 0, 0	3	BVA (Invalid - Zero Lengths)
TC9	1, 1, 1	0	BVA (Smallest Valid)
TC10	1, 2, 3	3	BVA (Invalid - Lower)

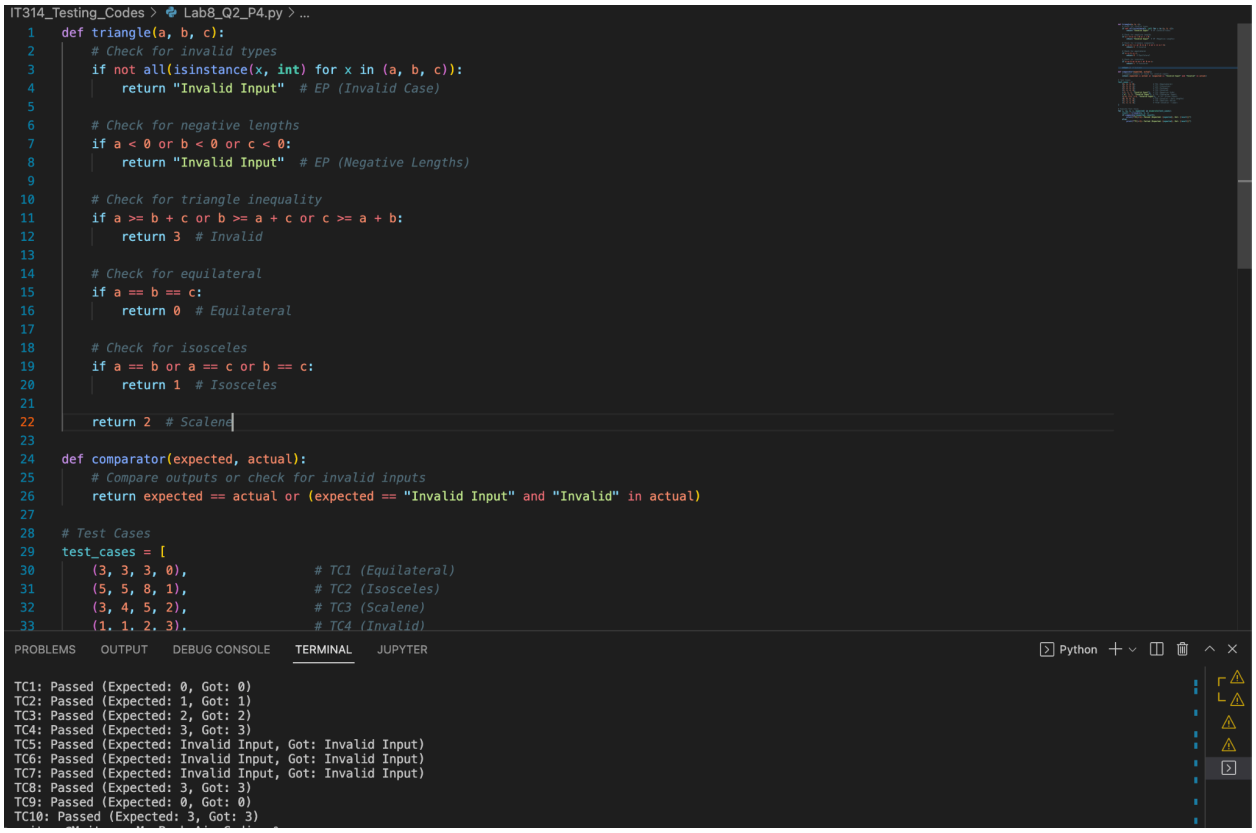
Code:

```
def triangle(a, b, c):

   # Check for invalid types

   if not all(isinstance(x, int) for x in (a, b, c)):

       return "Invalid Input"  # EP (Invalid Case)

    # Check for negative lengths

   if a < 0 or b < 0 or c < 0:

       return "Invalid Input"  # EP (Negative Lengths)

   # Check for triangle inequality

   if a >= b + c or b >= a + c or c >= a + b:

       return 3  # Invalid



   # Check for equilateral

   if a == b == c:

       return 0  # Equilateral
```

```
    # Check for isosceles
    if a == b or a == c or b == c:
        return 1  # Isosceles
    return 2  # Scalene
def comparator(expected, actual):
    # Compare outputs or check for invalid inputs
    return expected == actual or (expected == "Invalid Input" and "Invalid" in actual)
# Test Cases
test_cases = [
    (3, 3, 3, 0),                 # TC1 (Equilateral)
    (5, 5, 8, 1),                 # TC2 (Isosceles)
    (3, 4, 5, 2),                 # TC3 (Scalene)
    (1, 1, 2, 3),                 # TC4 (Invalid)
    (-1, 1, 1, "Invalid Input"),  # TC5 (Negative Side)
    ('a', 1, 1, "Invalid Input"), # TC6 (Character Input)
    (1.5, 1.5, 1.5, "Invalid Input"),  # TC7 (Float Input)
    (0, 0, 0, 3),                 # TC8 (Invalid - Zero Lengths)
    (1, 1, 1, 0),                 # TC9 (Smallest Valid)
    (1, 2, 3, 3),                 # TC10 (Invalid - Lower)
]
# Execute Test Cases
for i, (a, b, c, expected) in enumerate(test_cases):
    result = triangle(a, b, c)
    if comparator(expected, result):
```

```
        print(f"TC{i+1}: Passed (Expected: {expected}, Got: {result})")

    else:

        print(f"TC{i+1}: Failed (Expected: {expected}, Got: {result})")
```

Output:



P5 : The function prefix (String s1, String s2) returns whether or not the string s1 is a prefix of string s2

(you may assume that neither s1 nor s2 is null).

```
public static boolean prefix(String s1, String s2)
{
        if (s1.length() > s2.length())

        {
                return false;
        }
```

```
        for (int i = 0; i < s1.length(); i++)
        {
                if (s1.charAt(i) != s2.charAt(i))
                {
                        return false;
                }
        }
                return true;
}
```

A P5:

Equivalence Partitioning (EP)

- EP1: s1 is a prefix of s2.
- EP2: s1 is equal to s2.
- EP3: s1 is an empty string and s2 is a non-empty string (considered as a valid prefix).
- EP4: s1 is longer than s2 (impossible to be a prefix).
- EP5: s1 is an empty string and s2 is also an empty string (empty prefix).
- EP6: s1 and s2 are non-empty strings but do not match the beginning of s2.

Boundary Value Analysis (BVA)

- BVA1: s1 is an empty string and s2 is non-empty.
- BVA2: Both s1 and s2 are empty strings.
- BVA3: s1 has one character, and s2 has one or more characters.
- BVA4: s1 is a non-empty string, and s2 is the same as s1.
- BVA5: s1 is a non-empty string that matches the beginning of a longer s2.

Test Cases:

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	"pre", "prefix"	TRUE	EP (Valid Prefix)
TC2	"prefix", "prefix"	TRUE	EP (Equal Strings)
TC3	"", "notEmpty"	TRUE	EP (Empty Prefix)

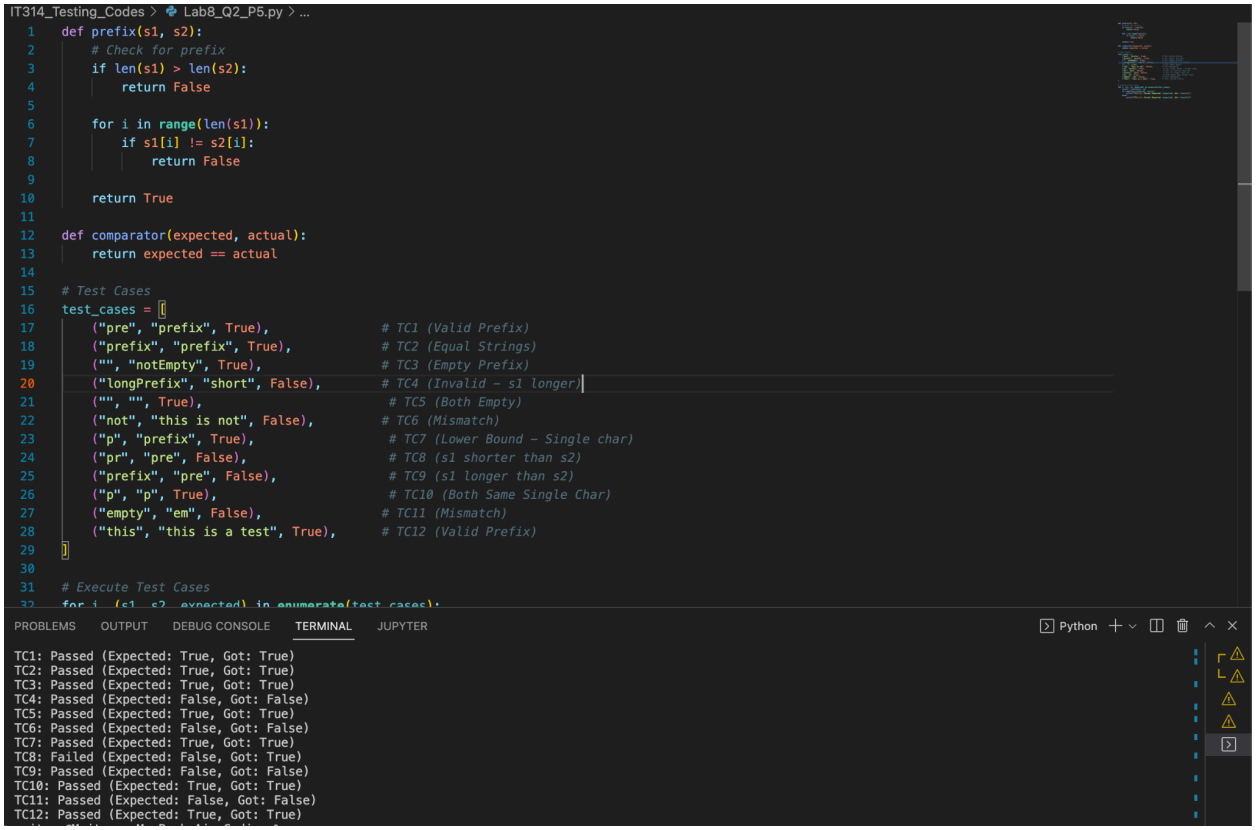
TC4	"longPrefix", "short"	FALSE	EP (Invalid - s1 longer)
TC5	"", ""	TRUE	EP (Both Empty)
TC6	"not", "this is not"	FALSE	EP (Mismatch)
TC7	"p", "prefix"	TRUE	BVA (Lower Bound - Single char)
TC8	"pr", "pre"	FALSE	BVA (s1 shorter than s2)
TC9	"prefix", "pre"	FALSE	BVA (s1 longer than s2)
TC10	"p", "p"	TRUE	BVA (Both Same Single Char)
TC11	"empty", "em"	FALSE	BVA (Mismatch)
TC12	"this", "this is a test"	TRUE	BVA (Valid Prefix)

Code:

```
def prefix(s1, s2):



    # Check for prefix

    if len(s1) > len(s2):

        return False

    for i in range(len(s1)):

        if s1[i] != s2[i]:

            return False

    return True

def comparator(expected, actual):

    return expected == actual



# Test Cases

test_cases = [
```

```
    ("pre", "prefix", True),                # TC1 (Valid Prefix)
    ("prefix", "prefix", True),             # TC2 (Equal Strings)
    ("", "notEmpty", True),                 # TC3 (Empty Prefix)
    ("longPrefix", "short", False),         # TC4 (Invalid - s1 longer)
    ("", "", True),                          # TC5 (Both Empty)
    ("not", "this is not", False),          # TC6 (Mismatch)
    ("p", "prefix", True),                   # TC7 (Lower Bound - Single char)
    ("pr", "pre", False),                    # TC8 (s1 shorter than s2)
    ("prefix", "pre", False),                # TC9 (s1 longer than s2)
    ("p", "p", True),                        # TC10 (Both Same Single Char)
    ("empty", "em", False),                 # TC11 (Mismatch)
    ("this", "this is a test", True),       # TC12 (Valid Prefix)
]


# Execute Test Cases
for i, (s1, s2, expected) in enumerate(test_cases):
    result = prefix(s1, s2)
    if comparator(expected, result):
        print(f"TC{i+1}: Passed (Expected: {expected}, Got: {result})")
    else:
        print(f"TC{i+1}: Failed (Expected: {expected}, Got: {result})")
```

Output :



P6: Consider again the triangle classification program (P4) with a slightly different specification: The program reads floating values from the standard input. The three values A, B, and C are interpreted as representing the lengths of the sides of a triangle. The program then prints a message to the standard output that states whether the triangle, if it can be formed, is scalene, isosceles, equilateral, or right angled. Determine the following for the above program:

a) Identify the equivalence classes for the system

b) Identify test cases to cover the identified equivalence classes. Also, explicitly mention which test case would cover which equivalence class. (Hint: you must need to be ensure that the identified set of test cases cover all identified equivalence classes)

c) For the boundary condition A + B > C case (scalene triangle), identify test cases to verify the boundary.
d) For the boundary condition A = C case (isosceles triangle), identify test cases to verify the boundary.
e) For the boundary condition A = B = C case (equilateral triangle), identify test cases to verify the boundary.
f) For the boundary condition A2 + B2 = C2 case (right-angle triangle), identify test cases to verify
the boundary.
g) For the non-triangle case, identify test cases to explore the boundary.

h) For non-positive input, identify test points.

A P6 :

a) Identify the Equivalence Classes

- Equilateral Triangle: All sides are equal (A = B = C).
- Isosceles Triangle: Two sides are equal (A = B ≠ C, A = C ≠ B, or B = C ≠ A).
- Scalene Triangle: All sides are different (A ≠ B, B ≠ C, A ≠ C).
- Right-angled Triangle: Fulfills the Pythagorean theorem ($A^2 + B^2 = C^2$).
- Non-Triangle: Sides do not satisfy the triangle inequality (A + B ≤ C, A + C ≤ B, or B + C ≤ A).
- Negative or Zero Values: Any side length is less than or equal to zero (A ≤ 0, B ≤ 0, C ≤ 0).
- Invalid Input Types: Non-floating-point inputs (strings, characters).

b) Identify Test Cases for equivalence class

Test Case ID	Tester Input	Expected Output	Analysis Type
TC1	3.0, 3.0, 3.0	"Equilateral Triangle"	EP (Equilateral)
TC2	4.0, 4.0, 5.0	"Isosceles Triangle"	EP (Isosceles)
TC3	3.0, 4.0, 5.0	"Scalene Triangle"	EP (Scalene)
TC4	5.0, 12.0, 13.0	"Right-angled Triangle"	EP (Right-angled)
TC5	1.0, 2.0, 3.0	"Not a Triangle"	EP (Non-Triangle)
TC6	-1.0, 2.0, 2.0	"Invalid input"	EP (Invalid Negative)
TC7	3.0, 3.0, -3.0	"Invalid input"	EP (Invalid Negative)
TC8	"A", "B", "C"	"Invalid input"	EP (Invalid Type)
TC9	0.0, 2.0, 2.0	"Invalid input"	EP (Zero Length)

c) Boundary Condition: A + B > C (Scalene Triangle)

Test Case ID	Tester Input	Expected Output	Analysis Type
TC10	1.0, 1.0, 1.5	"Scalene Triangle"	BVA (Valid Boundary)
TC11	1.0, 2.0, 2.0	"Isosceles Triangle"	BVA (Boundary)
TC12	1.0, 2.0, 3.0	"Not a Triangle"	BVA (Invalid Boundary)

d) Boundary Condition: A = C (Isosceles Triangle)

Test Case ID	Tester Input	Expected Output	Analysis Type
TC13	2.0, 3.0, 2.0	"Isosceles Triangle"	BVA (Valid Boundary)
TC14	3.0, 3.0, 3.0	"Equilateral Triangle"	BVA (Valid Boundary)

e) Boundary Condition: A = B = C (Equilateral Triangle)

Test Case ID	Tester Input	Expected Output	Analysis Type
TC15	2.0, 2.0, 2.0	"Equilateral Triangle"	BVA (Valid Boundary)
TC16	1.0, 1.0, 1.0	"Equilateral Triangle"	BVA (Valid Boundary)

f) Boundary Condition: $A^2 + B^2 = C^2$ (Right-Angled Triangle)

Test Case ID	Tester Input	Expected Output	Analysis Type
TC17	3.0, 4.0, 5.0	"Right-angled Triangle"	BVA (Valid Boundary)
TC18	5.0, 12.0, 13.0	"Right-angled Triangle"	BVA (Valid Boundary)

g) Non-Triangle Case Test Cases

Test Case ID	Tester Input	Expected Output	Analysis Type
TC19	1.0, 2.0, 3.0	"Not a Triangle"	Non-Triangle Test
TC20	1.0, 1.0, 3.0	"Not a Triangle"	Non-Triangle Test

h) Non-Positive Input Test Cases

Test Case ID	Tester Input	Expected Output	Analysis Type
TC21	0.0, 1.0, 1.0	"Invalid input"	Invalid Non-Positive
TC22	-1.0, 1.0, 1.0	"Invalid input"	Invalid Non-Positive

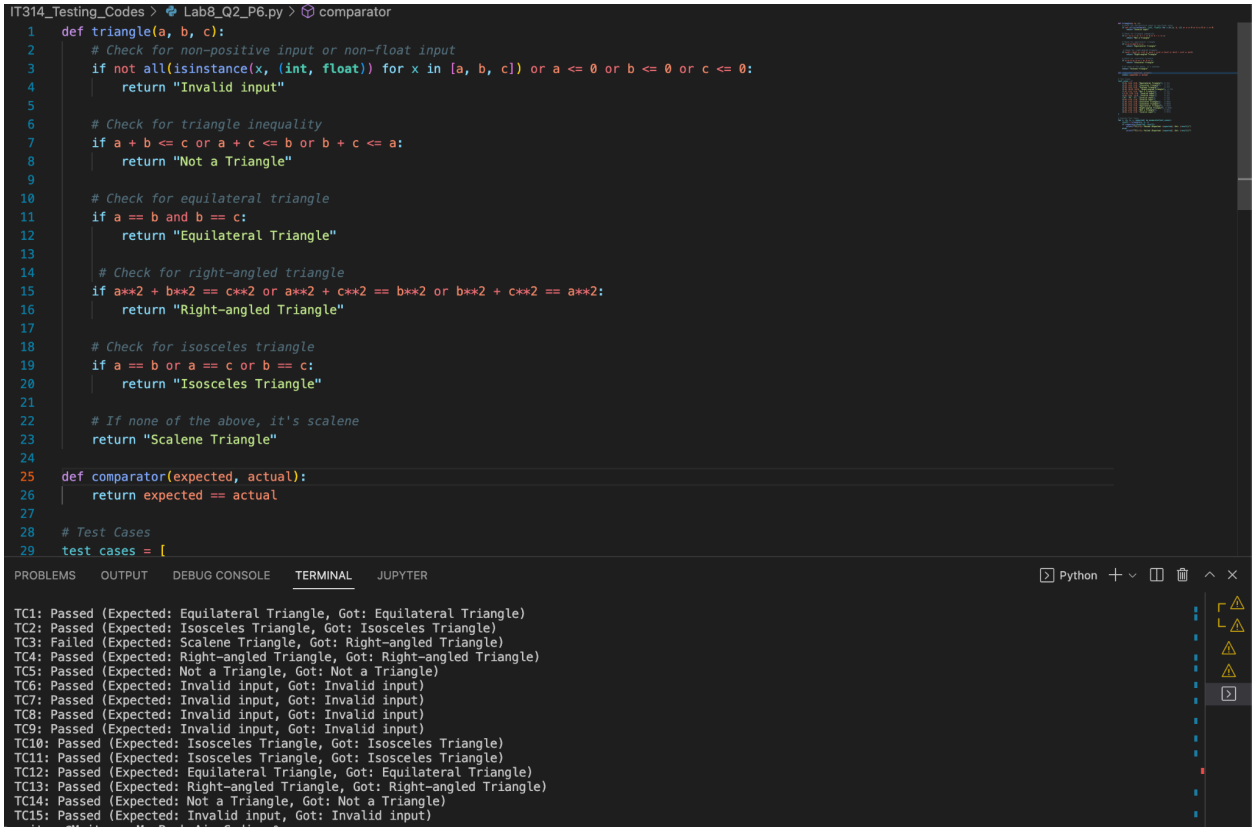
A triangle may be right-angled and isoceles or right angled and scalene but the function does not return 2 values thus if a triangle is right angled and isoceles it is only shown as isoceles while it is both.

Code:

```
def triangle(a, b, c):

   # Check for non-positive input or non-float input

   if not all(isinstance(x, (int, float)) for x in [a, b, c]) or a <= 0 or b <= 0 or c
<= 0:

       return "Invalid input"

   # Check for triangle inequality

   if a + b <= c or a + c <= b or b + c <= a:

       return "Not a Triangle"

   # Check for equilateral triangle

   if a == b and b == c:

       return "Equilateral Triangle"

    # Check for right-angled triangle

   if a**2 + b**2 == c**2 or a**2 + c**2 == b**2 or b**2 + c**2 == a**2:

       return "Right-angled Triangle"

     # Check for isosceles triangle

   if a == b or a == c or b == c:

       return "Isosceles Triangle"



   # If none of the above, it's scalene

   return "Scalene Triangle"

def comparator(expected, actual):

   return expected == actual

# Test Cases
```

```
test_cases = [
    (3.0, 3.0, 3.0, "Equilateral Triangle"), # TC1
    (4.0, 4.0, 5.0, "Isosceles Triangle"),   # TC2
    (3.0, 4.0, 5.0, "Scalene Triangle"),     # TC3
    (5.0, 12.0, 13.0, "Right-angled Triangle"), # TC4
    (1.0, 2.0, 3.0, "Not a Triangle"),       # TC5
    (-1.0, 2.0, 2.0, "Invalid input"),       # TC6
    (3.0, 3.0, -3.0, "Invalid input"),       # TC7
    ("A", "B", "C", "Invalid input"),        # TC8
    (0.0, 2.0, 2.0, "Invalid input"),        # TC9
    (1.0, 2.0, 2.0, "Isosceles Triangle"),   # BVA2
    (2.0, 3.0, 2.0, "Isosceles Triangle"),   # BVA4
    (2.0, 2.0, 2.0, "Equilateral Triangle"), # BVA6
    (3.0, 4.0, 5.0, "Right-angled Triangle"), # BVA8
    (1.0, 2.0, 3.0, "Not a Triangle"),       # NTC1
    (0.0, 1.0, 1.0, "Invalid input"),        # NTC3
]

# Execute Test Cases
for i, (a, b, c, expected) in enumerate(test_cases):
    result = triangle(a, b, c)
    if comparator(expected, result):
        print(f"TC{i+1}: Passed (Expected: {expected}, Got: {result})")
    else:
        print(f"TC{i+1}: Failed (Expected: {expected}, Got: {result})")
```

Output :


IT314_Testing_Codes > Lab8_Q2_P6.py > comparator
def triangle(a, b, c):
# Check for non-positive input or non-float input
if not all(isinstance(x, (int, float)) for x in [a, b, c]) or a <= 0 or b <= 0 or c <= 0:
return "Invalid input"
# Check for triangle inequality
if a + b <= c or a + c <= b or b + c <= a:
return "Not a Triangle"
# Check for equilateral triangle
if a == b and b == c:
return "Equilateral Triangle"
# Check for right-angled triangle
if a**2 + b**2 == c**2 or a**2 + c**2 == b**2 or b**2 + c**2 == a**2:
return "Right-angled Triangle"
# Check for isosceles triangle
if a == b or a == c or b == c:
return "Isosceles Triangle"
If none of the above, it's scalene
return "Scalene Triangle"
def comparator(expected, actual):
return expected == actual
# Test Cases
test_cases = [
PROBLEMS
OUTPUT
DEBUG CONSOLE
TERMINAL
JUPYTER
Python
TC1: Passed (Expected: Equilateral Triangle, Got: Equilateral Triangle)
TC2: Passed (Expected: Isosceles Triangle, Got: Isosceles Triangle)
TC3: Failed (Expected: Scalene Triangle, Got: Right-angled Triangle)
TC4: Passed (Expected: Right-angled Triangle, Got: Right-angled Triangle)
TC5: Passed (Expected: Not a Triangle, Got: Not a Triangle)
TC6: Passed (Expected: Invalid input, Got: Invalid input)
TC7: Passed (Expected: Invalid input, Got: Invalid input)
TC8: Passed (Expected: Invalid input, Got: Invalid input)
TC9: Passed (Expected: Invalid input, Got: Invalid input)
TC10: Passed (Expected: Isosceles Triangle, Got: Isosceles Triangle)
TC11: Passed (Expected: Isosceles Triangle, Got: Isosceles Triangle)
TC12: Passed (Expected: Equilateral Triangle, Got: Equilateral Triangle)
TC13: Passed (Expected: Right-angled Triangle, Got: Right-angled Triangle)
TC14: Passed (Expected: Not a Triangle, Got: Not a Triangle)
TC15: Passed (Expected: Invalid input, Got: Invalid input)