PytechArena buildathon

Level 1 Beginner

Problem Set

Duration: 60 Minutes

Total Questions: 10

Points: 100

Department of Information Technology
February 15, 2026

Instructions

- **Duration:** 16-17(2 Days)
- **Total Marks:** 100 (All questions carry equal marks – 10 points each)
- Write clean and readable Python code.
- Handle all edge cases mentioned in the problem statement.
- Do not include unnecessary print statements unless specified.
- Solutions must be your own work.

Submission Process

1. Fork the provided GitHub repository.
2. The **Team Leader** must create the GitHub account using the team name.
3. Create solution files named: `q1.py` to `q10.py`.
4. Each file should contain only the required function(s) for that question.
5. At the top of each file, include:
   - Team Name
   - Team ID
6. Test your code with the provided sample test cases.
7. Push all solutions to your forked repository before the deadline.

Evaluation Criteria

- **Correctness (60%):** Passing all visible and hidden test cases
- **Code Quality (20%):** Readability, structure, and meaningful naming
- **Efficiency (10%):** Time and space complexity
- **Edge Case Handling (10%):** Robust and complete solutions

Good Luck!

1 Question 1: Precise Tip Calculator (10 points)

Difficulty: Easy

Write a function `calculate_total_bill(amount, tip_percent)` that takes a bill amount and a tip percentage, then returns the total bill rounded to exactly two decimal places.

The function should handle the following:

- Convert both inputs to `float` to ensure decimal precision.
- Calculate the total using the formula: $Total = Amount + (Amount \times \frac{Tip\_Percent}{100})$
- Return the result as a `float` rounded to 2 decimal places.

Function Signature

```python
def calculate_total_bill(amount: float, tip_percent: int) -> float:
    """
    Calculate the total bill including tip.

    Args:
        amount: The initial bill amount (numeric)
        tip_percent: The tip percentage (integer)

    Returns:
        The total bill rounded to 2 decimal places.
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Standard tip
assert calculate_total_bill(100.0, 15) == 115.0

# Test Case 2: Decimal amount
assert calculate_total_bill(55.50, 20) == 66.6

# Test Case 3: Zero tip
assert calculate_total_bill(200, 0) == 200.0

# Test Case 4: Small amount with precision
assert calculate_total_bill(12.99, 10) == 14.29

# Test Case 5: Free item
assert calculate_total_bill(0, 15) == 0.0
```

Constraints

- $0 \leq \text{amount} \leq 10,000$
- $0 \leq \text{tip\_percent} \leq 100$

Constraints

- $0 \leq \text{length of string} \leq 1000$
- String may contain letters, digits, spaces, and special characters

2 Question 2: Time Converter (10 points)

Difficulty: Easy

Write a function `convert_seconds(total_seconds)` that takes a non-negative integer representing a total number of seconds and returns a formatted string in the style of `"Xm Ys"`.

The function should use basic operators to:

- Determine the number of full minutes using integer division (//).
- Determine the remaining seconds using the modulo operator (%).

Function Signature

```python
def convert_seconds(total_seconds: int) -> str:
    """
    Convert total seconds into minutes and remaining seconds.

    Args:
        total_seconds: An integer representing time in seconds.

    Returns:
        A string formatted as "Xm Ys".
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Standard conversion
assert convert_seconds(125) == "2m 5s"

# Test Case 2: Exactly one minute
assert convert_seconds(60) == "1m 0s"

# Test Case 3: Less than a minute
assert convert_seconds(45) == "0m 45s"

# Test Case 4: Multiple minutes
assert convert_seconds(3600) == "60m 0s"

# Test Case 5: Zero seconds
assert convert_seconds(0) == "0m 0s"
```

Constraints

- $0 \leq \text{total\_seconds} \leq 86,400$ (Seconds in a full day)
- The output must be a string with the exact format `"Xm Ys"`.

3 Question 3: Grade Filter (5 points)

Difficulty: Easy

Write a function `average_passing_grades(grades)` that takes a list of numerical grades and returns the **average** of all grades that are 50 or above. If there are no passing grades or the list is empty, return `0.0`.

The function should:

- Iterate through the list to find grades ≥ 50.
- Calculate the sum and count of these passing grades.
- Return the average as a float.

Function Signature

```python
def average_passing_grades(grades: list[int]) -> float:
    """
    Calculate the average of grades that are 50 or higher.

    Args:
        grades: A list of integers representing scores.

    Returns:
        The average of passing grades as a float, or 0.0 if none exist.
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Mixed grades
assert average_passing_grades([40, 60, 80, 20]) == 70.0

# Test Case 2: All passing
assert average_passing_grades([50, 100]) == 75.0

# Test Case 3: All failing
assert average_passing_grades([10, 20, 30]) == 0.0

# Test Case 4: Single passing grade
assert average_passing_grades([85]) == 85.0

# Test Case 5: Empty list
assert average_passing_grades([]) == 0.0
```

Constraints

- $0 \leq$ length of list ≤ 1000
- $0 \leq$ grade ≤ 100

4 Question 4: Ticket Pricer (10 points)

Difficulty: Easy

Write a function `get_ticket_price(age, is_student)` that determines the cost of a movie ticket based on a person's age and student status.

The pricing rules are as follows:

- **Children (under 12):** $8
- **Seniors (65 and older):** $10
- **Adults (12 to 64):**
  - If they are a student: $12
  - If they are NOT a student: $15

Function Signature

```python
def get_ticket_price(age: int, is_student: bool) -> int:
    """
    Determine ticket price based on age and student status.

    Args:
        age: Integer representing the person's age.
        is_student: Boolean indicating if the person is a student.

    Returns:
        The ticket price as an integer.
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Child
assert get_ticket_price(10, False) == 8

# Test Case 2: Senior
assert get_ticket_price(70, True) == 10

# Test Case 3: Adult Student
assert get_ticket_price(20, True) == 12

# Test Case 4: Adult Non-Student
assert get_ticket_price(25, False) == 15

# Test Case 5: Boundary Case (Age 12)
assert get_ticket_price(12, False) == 15
```

Constraints

- $0 \leq \text{age} \leq 120$
- `is_student` is always a boolean value.

5 Question 5: Smart Calculator (10 points)

Difficulty: Medium

Write a function `calculate(expression)` that evaluates a mathematical expression given as a string. The expression contains:

- Integers (positive or negative)
- Operators: +, -, *, /
- Spaces (should be ignored)

Follow standard operator precedence (* and / before + and -). Return the result as a float rounded to 2 decimal places.

Constraint: You **CANNOT** use `eval()`, `exec()`, or any similar built-in evaluation functions.

Function Signature

```python
def calculate(expression: str) -> float:
    """
    Evaluate mathematical expression without using eval().

    Args:
        expression: Mathematical expression as string

    Returns:
        Result rounded to 2 decimal places
    """
    pass
```

Sample Test Cases

```python
# Test Case 1
assert calculate("2 + 3") == 5.0

# Test Case 2
assert calculate("10 - 5 * 2") == 0.0

# Test Case 3
assert calculate("20 / 4 + 3 * 2") == 11.0

# Test Case 4
assert calculate("100 / 3") == 33.33

# Test Case 5
assert calculate("5") == 5.0
```

Constraints

- Expression will always be valid
- No parentheses in the expression
- Division by zero will not occur
- **Cannot use:** `eval()`, `exec()`, `compile()`

6 Question 6: Temperature Converter (10 points)

Difficulty: Easy

Write a function `convert_temperature(value, unit)` that converts a temperature from Celsius to Fahrenheit or vice versa.

The conversion formulas are:

- Celsius to Fahrenheit: $F = (C \times \frac{9}{5}) + 32$
- Fahrenheit to Celsius: $C = (F - 32) \times \frac{5}{9}$

The function should return the converted value rounded to **one decimal place**. If the unit provided is not `’C’` or `’F’`, return the string `"Invalid Unit"`.

Function Signature

```
def convert_temperature(value: float, unit: str) -> float | str:
    """
    Convert temperature between Celsius and Fahrenheit.

    Args:
        value: The temperature value to convert.
        unit: The unit of the input value (’C’ for Celsius, ’F’ for Fahrenheit).

    Returns:
        The converted temperature as a float (rounded to 1 decimal),
        or "Invalid Unit" if the unit is unknown.
    """
    pass
```

Sample Test Cases

```
# Test Case 1: Celsius to Fahrenheit
assert convert_temperature(0, ’C’) == 32.0

# Test Case 2: Fahrenheit to Celsius
assert convert_temperature(100, ’F’) == 37.8

# Test Case 3: Boiling point in Celsius
assert convert_temperature(100, ’C’) == 212.0

# Test Case 4: Negative temperature
assert convert_temperature(-40, ’F’) == -40.0

# Test Case 5: Invalid unit
assert convert_temperature(25, ’K’) == "Invalid Unit"
```

Constraints

- unit will be a single character string.
- Round the result using the `round(result, 1)` function.

Bugs to Find

Write your corrected code in `q6.py` and include comments explaining each bug you found.

Constraints

- $2 \leq n \leq 1000$
- Function should be reasonably efficient

7 Question 7: Fruit Stand Inventory (10 points)

Difficulty: Easy

Write a function `count_inventory(fruit_list)` that takes a list of strings representing fruits and returns a dictionary where the **keys** are the fruit names and the **values** are the number of times each fruit appears in the list.

Function Signature

```
def count_inventory(fruit_list: list[str]) -> dict[str, int]:
    """
    Create a frequency dictionary from a list of fruits.

    Args:
        fruit_list: A list of strings.

    Returns:
        A dictionary with fruit names as keys and counts as values.
    """
    pass
```

Sample Test Cases

```
# Test Case 1: Standard list
assert count_inventory(["apple", "banana", "apple", "cherry"]) == {"apple": 2,
    "banana": 1, "cherry": 1}

# Test Case 2: All same items
assert count_inventory(["orange", "orange"]) == {"orange": 2}

# Test Case 3: Single item
assert count_inventory(["grape"]) == {"grape": 1}

# Test Case 4: Empty list
assert count_inventory([]) == {}

# Test Case 5: Case sensitivity (Note: treat "Apple" and "apple" as different)
assert count_inventory(["Apple", "apple"]) == {"Apple": 1, "apple": 1}
```

Constraints

- $0 \leq$ length of list ≤ 1000
- All list elements will be strings.

Constraints

- $1 \leq$ rows ≤ 100
- $1 \leq$ columns ≤ 100
- Matrix contains integers

8 Question 8: Email Sanitizer (10 points)

Difficulty: Easy

Write a function `sanitize_email(raw_input)` that cleans up a user-submitted string intended to be an email address.

The function must perform the following steps in order:

- Remove any leading or trailing whitespace using `.strip()`.
- Convert the entire string to lowercase using `.lower()`.
- Check if the string contains exactly one ”@” symbol.
- If valid, return the cleaned string. If invalid, return `"Invalid Email"`.

Function Signature

```python
def sanitize_email(raw_input: str) -> str:
    """
    Clean an email string and validate basic structure.

    Args:
        raw_input: A string containing a potential email address.

    Returns:
        The cleaned lowercase email or "Invalid Email".
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Standard cleaning
assert sanitize_email("  User@Example.com  ") == "user@example.com"

# Test Case 2: No whitespace
assert sanitize_email("test@domain.org") == "test@domain.org"

# Test Case 3: Missing @ symbol
assert sanitize_email("myname-website.com") == "Invalid Email"

# Test Case 4: Multiple @ symbols
assert sanitize_email("admin@@company.com") == "Invalid Email"

# Test Case 5: Empty input after stripping
assert sanitize_email("   ") == "Invalid Email"
```

Constraints

- $0 \leq$ length of input ≤ 500
- You should use string methods to ensure the final output is uniform.

9 Question 9: Step Sequence Generator (10 points)

Difficulty: Easy

Write a function `generate_threes(start, end)` that creates a list of numbers starting from `start` up to (but not including) `end`, counting by **threes**.

If the `start` value is already greater than or equal to the `end` value, return an empty list.

Function Signature

```python
def generate_threes(start: int, end: int) -> list[int]:
    """
    Generate a list of numbers from start to end, skipping by 3.

    Args:
        start: The starting integer.
        end: The integer to stop before.

    Returns:
        A list of integers incremented by 3.
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Standard range
assert generate_threes(1, 11) == [1, 4, 7, 10]

# Test Case 2: Exact multiple
assert generate_threes(0, 9) == [0, 3, 6]

# Test Case 3: Start equals end
assert generate_threes(5, 5) == []

# Test Case 4: Start greater than end
assert generate_threes(20, 10) == []

# Test Case 5: Starting from a negative number
assert generate_threes(-5, 5) == [-5, -2, 1, 4]
```

Constraints

- $-1000 \leq \text{start}, \text{end} \leq 1000$
- You must use the built-in `range()` function.

10 Question 10: High Score Organizer (10 points)

Difficulty: Easy

Write a function `organize_scores(scores, descending)` that takes a list of integers and a boolean flag.

The function should:

- Return a **new list** containing the scores sorted.
- If `descending` is `True`, the list should be sorted from highest to lowest.
- If `descending` is `False`, the list should be sorted from lowest to highest.
- The original `scores` list must remain unchanged.

Function Signature

```python
def organize_scores(scores: list[int], descending: bool) -> list[int]:
    """
    Sort scores without modifying the original list.

    Args:
        scores: A list of integers.
        descending: Boolean indicating sort order.

    Returns:
        A new sorted list of integers.
    """
    pass
```

Sample Test Cases

```python
# Test Case 1: Ascending order
assert organize_scores([10, 5, 8], False) == [5, 8, 10]

# Test Case 2: Descending order
assert organize_scores([10, 5, 8], True) == [10, 8, 5]

# Test Case 3: Verify original list is not changed
original = [3, 1, 2]
organize_scores(original, True)
assert original == [3, 1, 2]

# Test Case 4: Already sorted
assert organize_scores([1, 2, 3], False) == [1, 2, 3]

# Test Case 5: Empty list
assert organize_scores([], False) == []
```

Constraints

- $0 \leq$ length of list ≤ 5000
- You must use the `sorted()` function to ensure the original list is preserved.