Mark as done

Description Submission view

Available from: Wednesday, 29 October 2025, 9:15 AM
Due date: Wednesday, 29 October 2025, 10:45 AM
Requested files: p1.py, p2.py, p2_input.txt, p1_input.txt, input1.txt (Download)
Type of work: Individual work

Problem 1 - Advanced Function and Data Handling for testing Symmetric Strings

Description:

Write a Python program that implements **four distinct functions** working together to read, process, and organize textual data into a **map**. The goal is to analyze input containing key-value pairs, determine whether each key is *symmetric*, and return a **map of maps** representing the evaluation results.

Notes on Testing:

Your reader function will read input form the terminal, but you will not need to type it by hand. Instead, we will redirect the data in the file `input1.txt` to your program, and `process` will be called with `sep` = ' ' (space). You may change the contents of the file to test with different input values. You may also test different separators by modifying the sep argument inside your process function.

Further, the current auto-grader for the convert function will assign certain marks for completely wrong code. This grader be updated after the test for finalizing the marks. If the grader shows 1/1 for this part, your code will be accepted also after the update.

Concept
This problem evaluates your ability to design and compose higher-order and recursive functions in Python without using loops. You are expected to demonstrate understanding of:

- **Closures:** Creating functions that return other functions which capture variables from their enclosing scope.
- **Higher-Order Functions:** Using `map()` and `filter()` to process collections without loops.
- **Recursion:** Implementing logic that handles repeated operations without iterative constructs.
- **String Validation and Pattern Recognition:** Verifying key-value pair structures in input lines.
- **Composite Functional Programming:** Combining multiple function outputs into a structured data representation.

Task:
Go to the provided file `p1.py` and implement the following four functions. Do not change function signatures, as this will result in grading errors.

1. `make_reader() -> Callable` :
   Define and return a parameter-less inner function. This inner function, when called, must **read lines until there are no more lines to read** and return those lines as a list of strings. Note that attempting to read input when there is no more input causes an `EOFError` exception. You must handle it.
2. `tokens(s: str, sep: str) -> filter`:
   This function must have **a single statement** which must be a **return** statement. It returns a filter containing the *valid* tokens from the input string `s`. Tokens in `s` are separated by the given string `sep`.
   You may define an internal helper function to perform validation but cannot use any loops (not even in helper functions).

   A **valid token** is a string of the form `Key=Value` satisfying the following:

   - `Key` consists only of English alphabetic characters.
   - `Value` is a valid integer (represented as a string).
   - `Key` must be at least 2 characters long.

   Return a filter object containing all valid tokens.

?

3. `convert(s: str) → bool`:
   This **recursive** function evaluates whether the `Key` in a string with `Key=Value` pattern is **symmetric**.

   A key is symmetric if:

   - It has an even number of characters
   - The first half of the key is either **equal to the second half** or **equal to the reverse** of the second half and each half is symmetric.

   Example:
   ```
   AA → symmetric (length = 2)
   XYYX → symmetric (first half = reverse of second half, each of length 2)

   ABCD → not symmetric
   XYZZYX → not symmetric as each half XYZ and ZYX are not symmetric
   ```

   Note: Recursion checking is currently turned off. Please implement the function recursively—it will be re-evaluated later.
4. `process(sep: str)`:
   This function orchestrates the other three.
   Steps:

   1. Calls `make_reader()` to obtain a reader function that reads input lines.
   2. Calls that reader to get a list of input lines.
   3. Uses `map()` to apply `tokens()` to each line, splitting it into valid key-value pairs using the given `sep`.
   4. Recursively maps `convert()` to each valid token to determine symmetry.
   5. Returns the final **map of maps**, where each inner map represents the converted tokens of one input line.

   > **Note:** Returning a list of lists instead of a map of maps will yield **50% marks** for this part.

Partial Credit: If your `process` function works differently but does return Python container*(eg: dict, list)* of containers, where each of the inner containers contains the converted valid tokens of one line, you will receive 50% of the total marks for part 4. See the examples below.

Example 1

File: `p2_input.txt`
```
2
input1.txt
input2.txt ,
```

(No separator given for `input1.txt`, so use space as default.)

File: `input1.txt`
```
KeyA=12 KeyB=23 KeyC=34
AA=100 ABAB=200 XYXY=300
```

File: `input2.txt`
```
QW=11,QWQW=22,W=5
ABAABA=55,AAABAA=66,XY=77
```

Output:

For `input1.txt`:
```
{ 1: {'KeyA': False, 'KeyB': False, 'KeyC': False},
2: {'AA': True, 'ABAB': True, 'XYXY': True} }
```

For `input2.txt`:
```
{ 1: {'QW': True, 'QWQW': True, 'W': False},
2: {'ABAABA': False, 'AAABAA': False, 'XY': True} }
```

Explanation of Output:

- **Line 1:** Keys `KeyA`, `KeyB`, `KeyC` are not symmetric.
- **Line 2:**
  - `AA` is symmetric (length = 2).
  - `ABAB` is symmetric (first half equals second half).

- `XYXY` is symmetric (first half equals reverse of second half).

If an additional line in `input.txt` is:
`K=45 ZY=67 XYZZYX=20`

Then:
`{ 3: {'K': False, 'ZY': True, 'XYZZYX': False} }`

Restrictions:

- You must not change any function signatures.
- **No loops** (for, while, or list comprehensions) are allowed in `tokens` or `convert`.
- `tokens()` must be implemented with **a single return statement** after defining its helper function.
- `convert()` must be **recursive**.
- The returned object from `process()` must be a **map of maps** (dictionary of dictionaries).

Problem 2: Function Invocation Counter

Description:
Write a higher-order function `counter(f: Callable[..., Any])`, which returns a wrapper function for any given function `f`.
This wrapper function maintains the count of invocations of `f` and stores it in a global dictionary `calls: dict[str, int]`, where:

- the **key** is the function name (`f.__name__`), and
- the **value** is the total number of times that specific function has been called.

This function must return the updated dictionary `calls` each time the wrapped function is invoked.

Goal
Construct a function that:

1. Keeps track of how many times each wrapped function has been called.
2. Updates a global dictionary `calls` accordingly.
3. Executes the original function normally and preserves its return value (if required by the program logic).

Function Definition

```
def counter(f: Callable[..., Any]) -> Callable[..., Any]:
```

Input Parameters

- **f** — a callable (function) that can accept any number of arguments and return any type.

Returns

A **wrapper function** that behaves exactly like `f`, but:

- Maintains the global dictionary `calls` containing the invocation counts of all wrapped functions.
- Updates `calls[f.__name__]` each time `f` is called.
- Returns the global dictionary `calls` after each invocation.

Implementation Details

- The `calls` dictionary must be defined globally.
- Each wrapped function updates only its own entry in the dictionary.
- Function names (`f.__name__`) act as keys in `calls`.
- The wrapper function must **update** the dictionary (it does not print anything).
- Do not modify the function name or the required structure.

Input File (`p2_input.txt`) Format

The input file simulates multiple function calls wrapped by `counter`.

- **First line** → integer `N`, number of test cases (function invocations).
- **Next N lines** → function calls in the form:
  `function_name(argument)`

Example p2_input.txt

```
8
is_even(5)
is_positive(0)
is_even(10)
has_char_e(elephant)
is_even(30)
is_positive(-1)
is_positive(3)
has_char_e(in)
```

Expected Output Example

- `is_even(5)` → First call for is_even:
  `{'is_even': 1}`
- `is_positive(0)` → First call for is_positive:`{'is_even': 1, 'is_positive': 1}`
- `is_even(10)` → Second call for is_even:`{'is_even': 2, 'is_positive': 1}`

Notes

- `calls` must be defined globally as `calls: dict[str, int] = {}`.
- The wrapper must always return the full `calls` dictionary, reflecting the cumulative state after each call.

Requested files

p1.py

```python
""" Boilerplate Code for Problem 1 of LabTest4-Day3
"""
import pprint # Do not change
from typing import Callable, Dict, List # Do not change

def make_reader() -> Callable[[], List[str]]:
    """
    Creates and returns a parameter-less function that reads all input lines
    from the specified input file.

    Args:
        None.

    Returns:
        function: A parameter-less function that reads all lines until EOF.

    Analogy:
        Think of this as a 'book reader' who reads all the pages (lines)
        from a specified book.

    Example:
        >>> reader = make_reader()
        >>> lines = reader()  # Reads all lines from input1.txt
    """
    pass


def tokens(s: str, sep: str) -> filter:
    """
    Extracts valid Key=Value tokens separated by a given separator.

    Args:
        s (str): Input string containing key-value pairs.
        sep (str): Separator between tokens (e.g., space or comma).

    Returns:
        filter: A filtered iterable of valid tokens.

    Rules:
        - Each token must be in the form 'Key=Value'.
        - Key must consist of only English alphabets (A–Z, a–z).
        - Value must be a valid integer string.
        - Key must be at least 2 characters long.

    Constraints:
        - Must be implemented as a SINGLE statement (using map or filter).
        - No loops allowed.

    Analogy:
        Imagine this as a filter that only allows correctly formed 'Key=Value'
        passes through a security gate.

    Example:
        >>> list(tokens("AA=12 BB=34", " "))
        ['AA=12', 'BB=34']
    """
    pass


def convert(s: str) -> bool:
    """
    Recursively checks if the Key part of 'Key=Value' is symmetric.

    Args:
        s (str): String of the form 'Key=Value'.

    Returns:
        bool: True if the key is symmetric, False otherwise.

    Rules:
        - A string is symmetric if:
            * It has even length AND
            * The first half equals the second half OR its reverse and each half is symmetric
        - Must be implemented recursively.

    Analogy:
        Think of symmetry like folding a paper in half — if both sides match,
        it’s symmetric.

    Example:
        >>> convert("AA=100")
        True
        >>> convert("XYZZYX=10")
        False
    """
    pass


def process(sep: str) -> Dict[int, Dict[str, bool]]:
    """
    Orchestrates the workflow of reading, tokenizing, converting,
    and structuring results into a map of maps.

    Args:
        sep (str): Separator for tokens (e.g., space " " or comma ",").

    Returns:
        dict: Dictionary of dictionaries containing Boolean results.

    Steps:
        1. Create a reader using make_reader().
        2. Read input lines from the file.
        3. Tokenize each line using tokens().
        4. Convert each token using convert().
        5. Structure results as a map of maps.

    Analogy:
        This function is like the 'factory manager' who ensures every machine
        (reader, tokenizer, converter) does its job in sequence, and collects
        the finished products neatly.

    Example:
        Input File (input1.txt):
            3
            Key1=12 Key2=23 Key3=34
            AA=100 ABAB=200 XYXY=300
            K=45 ZZ=67 XYZZYX=20

        Output:
            {
              1: {'Key1': False, 'Key2': False, 'Key3': False},
              2: {'AA': True, 'ABAB': True, 'XYXY': True},
              3: {'K': False, 'ZZ': True, 'XYZZYX': False}
            }
    """
    pass



############################## DO NOT CHANGE ########################################
def solution(tc):
    """
    Solution entry point for test case handling.
    """
    if isinstance(tc, str):
        # For backward compatibility, assume space separator
        return process(" ", tc)
    elif isinstance(tc, tuple) and len(tc) == 2:
        # New format: (input_filename, separator)
        input_filename, separator = tc
        return process(separator)
    else:
        raise ValueError("Test case must be a string (filename) or tuple (filename, separator)")


def process_input(filename: str):
    """
    Reads the test case file and extracts input filenames and separators for the solution.
    """
    lines = open(filename, 'r').read().splitlines()
    num_tests = int(lines[0])
    test_data = []
    for i in range(1, num_tests + 1):
        line = lines[i]
        parts = line.split(' ', 1)  # Split on first space
        input_filename = parts[0]
        if len(parts) > 1:
            separator = parts[1] if parts[1] else " "  # If separator is empty string, use space
        else:
            separator = " "  # Default to space if no separator
        test_data.append((input_filename, separator))
    return test_data

if __name__ == "__main__":
    from pprint import pprint
    Input = process_input('p1_input.txt')
    for tc in Input:
        result = solution(tc)
        if result != "":
            pprint(result)

```

p2.py

```python
"""Boilerplate for Problem 2: Function Call Counter Decorator."""

from typing import Callable, Any, Dict # Do not change

# Global dictionary to store function call counts
calls: Dict[str, int] = {} # Do not change

def counter(f: Callable[..., Any]) -> Callable[..., Any]:
    """
    A HOF that counts and prints how many times a single-arg function is called.

    This is a Higher-Order Function (HOF). It must return a NEW function
    (a "wrapper") that replaces the original function 'f'.

    Args:
        f (Callable): The original single-argument function to wrap.
                      Example: is_even, is_positive, etc.

    Returns:
        Callable: A new wrapper function to keep track of each function's call count
    """
    # todo: your code: Be sure to set and update count appropriately

    return None


############################ Do Not Change #################################

def solution(tc):
    func, arg = tc
    if func is None:
        print("Error: counter not implemented correctly.")
        return None

    # Call the function (this will update the global calls dictionary)
    result = func(arg) if arg is not None else func()

    # Return the global calls dictionary
    return calls


def process_input(filename: str):
    """Parses input file and dynamically loads + decorates only required functions."""
    import p2_utils
    with open(filename, "r") as f:
        lines = [line.strip() for line in f if line.strip()]

    num_tests = int(lines[0])
    test_lines = lines[1:num_tests + 1]
    func_names = {line.split("(", 1)[0].strip() for line in test_lines}

    functions = {}
    for name in func_names:
        if hasattr(p2_utils, name):
            functions[name] = counter(getattr(p2_utils, name))
        else:
            print(f"Warning: Function '{name}' does not exist")
            functions[name] = None

    parsed_tests = []
    for line in test_lines:
        func_name, rest = line.split("(", 1)
        arg_str = rest.rstrip(")").strip()
        func = functions.get(func_name.strip())

        if not arg_str:
            arg = None
        elif (arg_str.startswith('"') and arg_str.endswith('"')) or (arg_str.startswith("'") and arg_str.endswith("'")):
            arg = arg_str[1:-1]
        else:
            try:
                arg = int(arg_str)
            except ValueError:
                arg = arg_str

        parsed_tests.append((func, arg))

    return parsed_tests

if __name__ == "__main__":
    for tc in process_input("p2_input.txt"):
        print(solution(tc))
```

p2_input.txt

```
8
is_even(5)
is_positive(0)
is_even(10)
has_char_e(elephant)
is_even(30)
is_positive(-1)
is_positive(3)
has_char_e(in)
```

p1_input.txt

```
1
input1.txt
```

input1.txt

```
KeyA=12 KeyB=23 KeyC=34
AA=100 ABAB=200 XYXY=300
```

VPL