Mark as done

Description Submission view

Available from: Thursday, 30 October 2025, 9:15 AM
Due date: Thursday, 30 October 2025, 10:45 AM
Requested files: p1.py, p2.py, p2_input.txt, input.txt (Download)
Maximum number of files: 5
Type of work: Individual work

Problem 1

Description:

Write a Python program that implements four distinct functions working together to read and evaluate textual data from a file. The goal is to analyze lines of input, extract valid words in each line, check whether the remaining words leave the line *balanced*, and return a composite structure of the balanced lines.

Notes on Testing:

Your reader function will read input from the terminal, but you will not need to type it by hand. Instead, we will redirect the data in the file `input.txt` to your program. You may modify the contents of this file to test with different input values.

Further, you may test different separators by modifying the `sep` value in your boilerplate code. Ensure that your functions handle invalid inputs, missing lines, and incorrect separators gracefully as per the given constraints.

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(line_count:int):` This function returns a parameter-less reader function, which inputs exactly `line_count` lines and returns a list of those lines. However, if that reader encounters the exception **`EOFError`** while reading the input, the program *exits* after printing the formatted message `f'Fewer than {line_count} were provided'`. Instead, if the number of lines in the input continue beyond `line_count`, without an **`EOFError`**, the program exits with the message `f'More lines than {line_count} were provided'`.

2. `tokens(s:str, sep:str, isvalid:Callable[…,bool]) -> filter`: This function has a **single** statement that returns a collection of valid words in `s` as a filter. `tokens` may define an internal helper function and call it in that single statement. Words are separated in string `s` by any repetition of the string `sep` (not 0 repetitions, of course). Assert that sep is not an empty string, i.e., `raise AssertionError` on empty sep. A valid word is a string for which `isvalid` returns True. `tokens` does not know how `isvalid` works, it just calls it.(Do not use a loop, not even in the internal function.)

3. `balanced(L: list(str))->bool` is a recursive function that takes a list `L` of strings and returns `True` if L is "balanced" and `False` otherwise.
   `L` is **balanced** if:
   1. it contains an even number of strings **AND**
   2. the sum of lengths of strings in `L1` and the sum of lengths of strings in `L2` differ by at most `1` **AND**
   3. `L1` and `L2` are also both **balanced**
      where `L1` is a sublist containing the first len(L)/2 elements of `L` and `L2` is the sublist with the second len(L)/2 elements
   4. A list with 0 or 1 element is called **balanced**.

4. `process(sep:str, num_lines:int, isValid:Callable[...,bool]): process` This function orchestrates the other functions to achieve the final goal. It first calls `make_reader` to get a reader function and then calls that reader function to get a list of `num_lines` input lines. `process` next tokenizes each line by mapping tokens to lines (asking it to use repetitions of `sep` as the separator to split the line into words, and `isValid` as the validity detector). Finally, it filters the tokenized lines, keeping only the balanced lines. `process` **returns a filter of containers** (where each container contains the valid tokens of a balanced line). *Returning a list instead of filter will be graded on 50% for this part.*

> There is **no extra credit**, but try to implement `process` using **only a single composite statement**.

Restrictions:

- Do **not** use any loops (for, while, or list comprehensions) in `tokens()` or `balanced()`.
- `tokens()` must contain **exactly one return statement** after defining its helper function.
- `balanced()` must be **implemented recursively**.
- The object returned by `process()` must be a **filter of containers** — Returning a list instead of filter will earn partial credit (50%).

?

Examples:

Example 1: Fewer Lines Than Expected

N= 5

Sep= " "

Input.txt:
```
hello world
python code
test case
```

Output:
```
Fewer than 5 were provided
Program exited as expected.
```

Explanation:
1. Reads 3 lines successfully: ["hello world", "python code", "test case"]
2. On 4th read attempt, gets EOFError
3. Prints error message and exits with SystemExit

Example 2: More Lines Than Expected

N= "5"

Sep= " "

Input.txt (7 lines, but expecting 5):
```
line one
line two
line three
line four
line five
line six
line seven
```

Output:
```
More lines than 5 were provided
Program exited as expected.
```

Explanation:
1. Successfully reads 5 lines: ["line one", "line two", "line three", "line four", "line five"]
2. After reading 5 lines, tries to read one more line to check for extras
3. Successfully reads "line six" → detects extra lines
4. Prints error message and exits with SystemExit

Example 3:

Input.txt (10 lines):
```
hello world test
bad word!
@123
data structures
algorithm design
functional code
machine learning
aa
hi
bye
```

N= 10 , **SEP**= " "

PROCESSING STEPS:

Step 1: Reading Input
- `make_reader(10)` reads exactly 10 lines

Step 2: Tokenization & Filtering with `isvalid`:
Each line split by space, then filtered through `isvalid`:

- Line 0: "hello world test" → ["hello", "world", "test"] → **ALL VALID** → ["hello", "world", "test"]
- Line 1: "bad word!" → ["bad", "word!"] → "word!" has `!` → **INVALID** → ["bad"]
- Line 2: "@123" → ["@123"] → has `@` and digits → **INVALID** → []
- Line 3: "data structures" → ["data", "structures"] → **ALL VALID** → ["data", "structures"]
- Line 4: "algorithm design" → ["algorithm", "design"] → **ALL VALID** → ["algorithm", "design"]
- Line 5: "functional code" → ["functional", "code"] → **ALL VALID** → ["functional", "code"]
- Line 6: "machine learning" → ["machine", "learning"] → **ALL VALID** → ["machine", "learning"]
- Line 7: "aa" → ["aa"] → **VALID** → ["aa"]
- Line 8: "hi" → ["hi"] → **VALID** → ["hi"]
- Line 9: "bye" → ["bye"] → **VALID** → ["bye"]

Note: This is just an example of what isvalid() is doing,it's functionality might be different,you just have to call it.

Step 3: Balanced Check:
- Line 0: ["hello", "world", "test"] → 3 tokens (odd) → **NOT balanced**
- Line 1: ["bad"] → 1 token → **BALANCED**
- Line 2: [] → 0 tokens → **BALANCED**

- Line 3: ["data", "structures"] → |4-10|=6 > 1 → **NOT balanced**
- Line 4: ["algorithm", "design"] → |9-6|=3 > 1 → **NOT balanced**
- Line 5: ["functional", "code"] → |10-4|=6 > 1 → **NOT balanced**
- Line 6: ["machine", "learning"] → |7-8|=1 ≤ 1 → **BALANCED**
- Line 7: ["aa"] → 1 token → **BALANCED**
- Line 8: ["hi"] → 1 token → **BALANCED**
- Line 9: ["bye"] → 1 token → **BALANCED**

Step 4: Final Output:
Only balanced lines shown with new indexing:

```
Processing 10 lines with separator ' ':
Line 0 tokens:    ["bad"]           ← From input line 1
bad
Line 1 tokens:    []                ← From input line 2
Line 2 tokens:    ["machine", "learning"] ← From input line 6
machine
learning
Line 3 tokens:    ["aa"]             ← From input line 7
Line 4 tokens:    ["hi"]            ← From input line 8
hi
Line 5 tokens:    ["bye"]           ← From input line 9
bye
```

Problem 2

Write a higher order function `counter(f:Callable[..., Any])`, which returns a wrapper function for given function `f`. This wrapper function maintains the count of number of invocations of `f`. The wrapper uses a variable defined in counter to maintain the number of calls. At each invocation, it returns a tuple of the count and the original return of the function `f`. The count for the first invocation is `1`.

Function Definition

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

Input Parameters:

- `f`: A callable (function) that can accept any number of arguments and return any type.

Returns:

Returns a wrapper function that maintains the count of number of invocations of `f`.

Input File (`p2_input.txt`) Format

- **First line** → number of test cases (integer).
- **Each subsequent line** → a function call in the form:
  `function_name(arguments...)`

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

- Call number 1 for is_even:
  `(1, False)`
- Call number 1 for is_positive:
  `(1, False)`
- Call number 2 for is_even:
  `(2, True)`
- and so on ...

Notes and Restrictions

- Each function maintains its own internal counter; counters are not shared across functions. Do not use global counter variables.
- All helper functions used in the input are pre-defined; they should not be implemented.

Requested files

p1.py

```python
"""Boilerplate code for Day 4 – Problem 1."""
from typing import Callable, List


def make_reader(line_count: int) -> Callable[[], list]:
    """
    Returns a parameter-less function that reads exactly `line_count` lines from standard input.

    Behavior:
        • Reads one line at a time using `input()`, stripping trailing newlines.
        • If EOFError is encountered before reading all lines, print:
              Fewer than {line_count} were provided
          and exit the program (raise SystemExit).
        • If more than `line_count` lines are provided, print:
              More lines than {line_count} were provided
          and exit the program (raise SystemExit).
    """
    pass


def tokens(s: str, sep: str, isvalid: Callable[[str], bool]) -> filter:
    """
    Returns a filter object of *valid words* from the string `s`.
    A word is considered valid if the given `isvalid` function returns True for it.

    Behavior:
        • Words are separated by one or more occurrences of `sep`.
        • Must assert that `sep` is not an empty string (raise AssertionError if it is).
        • Implement with a single `return` statement (a helper function is allowed).
        • Must NOT use any loops (including list comprehensions).

    """
    pass


def balanced(L: List[str]) -> bool:
    """
    Recursively checks whether a list `L` of strings is *balanced*.

    Definition:
        • An empty list is always balanced.
        • A non-empty list is balanced if:
            1. It has an even number of strings, AND
            2. The sums of lengths of its two halves differ by at most 1, AND
            3. Both halves are themselves balanced.

       Where:
            L1 = L[:len(L)//2]
            L2 = L[len(L)//2:]

    Constraints:
        • Must be implemented recursively.
        • Must NOT use loops.
    """
    pass


def process(sep:str, num_lines:int, isValid:Callable[...,bool]):
    """
    Performs the overall processing pipeline.

    Steps:
        1. Calls `make_reader()` to obtain a reader function that reads the input lines.
        2. Invokes that reader to get a list of lines.
        3. Tokenizes each line using `tokens(line, sep, isvalid)`.
        4. Keeps only those tokenized lines that are *balanced* using `balanced()`.
        5. Returns a *filter of list*, where each inner list represents the tokens of one line.
    """
    pass
















###############################
# Local test configuration
###############################
N = 10   # expected number of input lines
SEP = " "  # token separator








##############################################################
# Do not change below this line
##############################################################
def map2dict(M: map) -> dict:
    """Converts a map of maps into a dictionary of dictionaries for easy display."""
    D = {}
    for lno, lineM in enumerate(M):
        D[lno] = {}
        for tno, t in enumerate(lineM):
            D[lno][tno] = t
    return D


def solution(sep:str, n: int, isvalid: Callable[[str], bool]):
    """
    Calls `process()` and materializes the returned maps for readable output.
    """
    try:
        M = process(sep, n, amodule.isvalid)
        materialized = [list(line) for line in M]
        print("**********************************************")
        print(f"Processing {n} lines with separator '{sep}':")
        for lno, line_list in enumerate(materialized):
            print(f"Line {lno} tokens:")
            for t in line_list:
                print("\t", t)
        return materialized
    except SystemExit:
        print("Program exited as expected.")
    except Exception as e:
        print(f"Error in process() or dependent functions. Error: {e}")
    return None


if __name__ == "__main__":
    import amodule
    result = solution(SEP, N, amodule.isvalid)

```

p2.py

```python
"""Boilerplate Problem 2: Function Call Counter with Tuple Return"""

from typing import Callable, Any, Tuple

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

    Returns a wrapper function that:
    - Maintains a count of invocations of f
    - Returns a tuple (count, original_result) at each call
    - First invocation returns count = 1

    Args:
        f: Any callable function to wrap

    Returns:
        A wrapper function that returns (count, f(...))
    """


    # WRITE YOUR CODE BELOW
    pass

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

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


def process_input(filename: str):
    import p2_utils
    with open(filename) as f:
        lines = [line.strip() for line in f if line.strip()]

    n = int(lines[0])
    tests = lines[1:n + 1]

    funcs = {
        name: counter(getattr(p2_utils, name)) if hasattr(p2_utils, name) else None
        for name in {line.split("(", 1)[0].strip() for line in tests}
    }

    parsed = []
    for line in tests:
        name, arg = line.split("(", 1)
        arg = arg.rstrip(")").strip()
        func = funcs.get(name.strip())

        if not arg:
            parsed.append((func, None))
        elif arg[0] in "\"'":
            parsed.append((func, arg[1:-1]))
        else:
            try:
                parsed.append((func, int(arg)))
            except ValueError:
                parsed.append((func, arg))
    return parsed


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

```

p2_input.txt

```
10
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)
is_even(0)
has_char_e(apple)
```

input.txt

```
hello world test
bad word!
@123
python programming
data structures
algorithm design
functional code
machine learning
a
hi
bye
ok
hello    world     test
  python   code
data      structures
a
ab
abc
abcd
abcdefgh
hello world bad123
valid test @invalid
python code $$$
crash
code unknown
```

VPL