Mark as done

Description Submission view

Available from: Monday, 27 October 2025, 9:15 AM
Due date: Monday, 27 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 - Advanced Function and Data Handling

Description:
Write a program by implementing four distinct functions that work together to read lines of text, process the words within them, and organize the results into a complex, nested data structure.

Concept:
This problem tests your ability to integrate several advanced Python concepts. You must demonstrate your skills in:

- Creating functions that generate and return other functions (a **Closure** that can access variables of the enclosing scope).
- Using **Higher-Order Functions** like `map()` and `filter()` to process collections of data without using loops.
- Implementing **Exception Handling** to manage errors during data processing.
- Using **Recursion** to build a nested data structure from a flat string.
- Assembling results into a **Data Structure**.

Task:
Open the `p1.py` boilerplate file provided and implement the four required functions according to the specifications below. You must not change the function definitions (names and parameters), as this will cause your program to fail.

1. **`make_reader(line_count)`**: This function must define and return a new inner function. This inner function should have no parameters and, when called, must input exactly `line_count` lines, and return them as a list of strings -- one string per line. (Please note that you will not need to type the input on the terminal at execution time. Instead, we will direct the data in file `input.txt` to the keyboard.)
2. **`tokens(s:str, line_number:int=0)`**: This function takes two arguments, a string `s` and int `line_number`. Function must take string `s` (e.g., an input line), split it into words (using s.split()), and filter the resulting word-list by calling the provided `amodule.valid(w:str)->bool` function. If `amodule.valid(w)` is False, discard w. If `amodule.valid(w)` raises an `amodule.BadError` for any word w, `tokens` must handle that exception (and only that exception specifically -- **print** the exception details in the format as described in p1.py) and discard the word. `tokens` must be implemented **without using any loops** and with a **single statement** with **`return`** (you may define a helper function before it). `tokens` should return a filter object.
   If `amodule.valid(w)` raises an `amodule.BadError` for any word `w`, following message is printed:
   ```
   Bad Word {w} in line {line_number}
   ```
3. **`convert(t)`**: This **recursive** function "converts" string `t` into a nested list and returns that list. A string `t` made of a single character simply converts to the list `[t]`. Otherwise, `t` converts to a list of two lists *l1* and *l2*, where *l1* is the result of converting left sub-string `t` [0:len(t)//2] and *l2* is the result of converting the remaining right sub-string of `t`. This function must also be implemented **without using any loops**. `t` will never be an empty string.
4. **`process(count)`**: This function orchestrates the others. It uses `make_reader` to create a reader of `count` lines, and then uses it to read the lines. It then maps the function `tokens` to lines (which splits each line and retains the valid words of each line). Finally, it maps the `convert` function in a way that each retained word is converted to a recursive list. It returns this final map of maps object.

Partial Credit: If your `process` function works differently but does return Python container*(eg: dict, list)* of `count` containers, where each of the count containers contains the converted valid tokens of one line, you will receive 80% of the total marks for part 4. See the examples below.

Example:
Assume the following lines are input (e.g., see `input.txt`).

?

```
hello world test
bad word!
```

process(2) will read the two lines above. The word bad123 is invalid according to the unseen rules of amodule.valid(), so your tokens function must filter it out. The `tokens` function, when processing line 1 (which has `line_number=1`), will encounter `"word!"` (which happens to be a bad word). It will catch the `amodule.BadError` and print:

```
Bad Word word! in line 1
```

The word `"word!"` will then be filtered out.

The output of process is a map of two maps, the first of which contains three lists:

```
[[[['h'], ['e']], [['l'], [['l'], ['o']]]]]
[[[['w'], ['o']], [['r'], [['l'], ['d']]]]]
[[['t'], ['e']], [['s'], ['t']]]

The second contains the list (Note word! is filtered out):
[[['p'], [['y'], ['t']]], [['h'], [['o'], ['n']]]]
```

Partial Credit (80%) if the above is contained in a list of two lists instead, the first of which would contain the first three lists above, and the second would contain the fourth lists above.

Restrictions:

- Do not change any function signatures in `p1.py`.
- `tokens` and `convert` must not use loops. Not even in list comprehension.
- `tokens` must be implemented with a single `return` statement after its helper function.
- `convert` must be implemented recursively.
- You must use the provided `amodule.valid()` function for word validation.

Problem 2: Function Invocation Counter

Write a High Order function `counter` that wraps any single-argument function `f` and maintains a running count of how many times `f` has been invoked. Each time the wrapped function is called, it must:

1. Print the call number for that specific function in the specified format. The first call is numbered 1.
2. Execute the original function normally.
3. Return the result unchanged.

Goal

Construct a function that:

- Maintains a **persistent counter** for each wrapped function.
- Prints exactly:

```
Call number {xx} for {f.__name__}
```

where `{xx}` is count for this function, and `{f.__name__}` is the original function name.

- Calls `f(x)` with **exactly one argument**.
- Returns the original function's result unchanged.

Function Definition

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

Input Parameters:

- `f`: A callable (function) that accepts exactly one argument and returns any type.

Returns:
A wrapper function that behaves exactly like `f` but:

1. Maintains a **per-function internal counter** that tracks how many times the function has been called.
2. Prints the call number each time the function is invoked in the format: `Call number {xx} for {f.__name__}` where `{xx}` is the 1-based call count and `{f.__name__}` is the name of the original function.
3. Executes `f(x)` with the given argument and returns its result unchanged.

Description:

- A separate counter for each parameter f must be maintained; counters are not global.
- Wrapper should ensure invocation of f with **exactly one argument**.

Input File (`p2_input.txt`) Format

- **First line** → number of test cases (integer).
- **Each subsequent line** → a function call 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 → Prints:

  ```
  Call number 1 for is_even
  ```

- `is_positive(0)` → First call → Prints:

  ```
  Call number 1 for is_positive
  ```

- `is_even(10)` → Second call → Prints:

  ```
  Call number 2 for is_even
  ```

Notes

1. Each function maintains its own internal counter; counters are not global.
2. Print statements show the call number for that function only.
3. First call of any function will be 1.
4. Original function return values are preserved and returned unchanged.
5. All functions used in the input (e.g., is_even, is_positive, has_char_e) are already defined; there is no need to define them.
6. The HOF enforces that each function receives exactly one argument.

Requested files

p1.py

```python
"""Boilerplate code for problem 1."""
import amodule  # Do not change
from typing import List, Callable, Dict # Do not change


def make_reader(line_count: int)->Callable[[], list]:
    """
    Creates and returns a parameter-less function that reads exactly
    line_count lines from stdin and returns them as a list of strings.

    Args:
        line_count (int): Number of lines to read

    Returns:
        function: A parameter-less function that reads line_count lines

    Example:
        >>> reader = make_reader(3)
        >>> lines = reader()  # Returns list of 3 lines from stdin
    """

    pass


def tokens(s: str, line_number:int=0)->filter:
    """
    Returns a collection of valid words from string s.

    This function has a single statement that returns the collection.
    It defines an internal helper function that is used by this statement.
    Internal helper must ONLY handle BadError and print below message:
        f"Bad Word {w} in line {line_number}"

    Args:
        s (str): The string (line) to tokenize.
        line_number (int, optional): The line number for error messages. Defaults to 0.
    Returns:
        filter: A filter object containing the valid words.
    Constraints:
        - The return must be a SINGLE STATEMENT (can be after the helper function).
        - Must NOT use any 'for' or 'while' loops, even in the helper function.
    """

    pass


def convert(t: str) -> list:
    """
    Recursively converts a string t into a nested list of characters.
    Args:
        t (str): A string of length at least 1.
    Returns:
        list: A nested list representation of the string.
    Constraints:
        - Must be implemented recursively.
        - Must NOT use any loops.
    Example:
        convert("ab") -> [['a'], ['b']]
        convert("abcd") -> [[['a'], ['b']], [['c'], ['d']]]
    """

    pass


def process(count: int):
    """
    Reads 'count' lines, processes them, and returns a map of maps.

    First, it calls make_reader to get a reader function and calls it
    to get a list of lines.

    It then tokenizes each line by mapping the 'tokens' function to
    the lines along with their line numbers.

    Finally, it converts each token of each line into a recursive list
    and returns a map of maps (where each map represents the
    converted tokens of one line).

    Args:
        count (int): The number of lines to process.

    Returns:
        map: An iterator (a map object) of map objects.
             The outer map iterates over processed lines.
             Each inner map iterates over the converted tokens
             (recursive lists) for that line.

    Example structure:
        {
            0: {0: [['h'], ['i']], 1: [['b'], ['y'], ['e']]},
            1: {0: [['f'], ['o'], ['o']]}
        }

    Note: Returning a list of lists or maps instead of this "map of maps"
          structure (nested map iterators) will receive 80% credit.

    Challenge (no extra credit): Try implementing this in a single
    composite statement (helper functions are allowed).
    """

    pass


```

```python





#### #### #### #### #### #### #### #### ####
# To test your code, change values of N: ###
# N is number of lines to read          ####
N=5                                     ####
#### #### #### #### #### #### #### #### ####


########### Do not change below this line ###########

def solution(n):
    import copy
    M = process(n)
    MCopy = copy.deepcopy(M)
    try:
        print(f"**********************************************")
        print(f"Processing {n} lines of input:")
        for lno, lineM in enumerate(MCopy):
            print(f'Line {lno} converted tokens (one per line):')
            for t in lineM: print('\t',t)
    except Exception as e:
        print(f"Error parsing result from process() function. Got {MCopy}\nError:{e}")
    return MCopy

def process_input(filename):
    lines = open(filename, 'r').read().splitlines()
    num_tests = int(lines[0])
    input_tests = lines[1:]
    return input_tests[:num_tests]

if __name__ == "__main__":
    result = solution(N)
```

p2.py

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

from typing import Callable, Any # Do not change


def counter(f: Callable[[Any], Any]) -> Callable[[Any], 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
    return func(arg) if arg is not None else func()


def process_input(filename: str):
    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"):
```

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)
```

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