Mark as done

Description Submission view

Available from: Tuesday, 28 October 2025, 9:15 AM
Due date: Tuesday, 28 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 Data Transformation

Description:
Write a program by implementing four distinct functions that work together to read lines of text, transform numeric tokens within them, and rebuild them into a complex, recursively defined string format.

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**).
- Handling specific runtime errors like **EOFError** during input operations.
- Using **Higher-Order Functions** like `map()` and `filter()` to process collections of data without using loops.
- Implementing a non-trivial **Recursive** algorithm for string transformation.
- Assembling results into a nested **Data Structure** of iterators.

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. If it encounters an `EOFError` (meaning the input ends before line_count lines are read), it must exit the program with the exact message: `f"Fewer than {line_count} were provided"`. (For your convenience, you will not have to type the input at the running time. Instead, we will redirect the lines of file `input.txt` as user input.)
2. **`tokens(s:str, sep:str) -> filter`**: This function returns a collection of valid words from the string `s`. The words in `s` are separated by the delimiter `sep`. You may define a helper function internally, but the entire function should consist of a **single return statement**, which calls the helper function. A valid word is either an empty string or a string that can be converted into an integer. **Do not use loops**, even within the helper function.
3. **`convert(t:str) -> str`**: This **recursive** function transforms string `t` into a new string based on the following rule: `convert(t)` is defined as the string `convert(BLR)-LR-convert(ALR)`, where:
   - `LR` is the first longest run of a repeated character in `t`.
   - `BLR` is the part of the string *before* `LR`.
   - `ALR` is the part of the string *after* `LR`.
   - `-` is a character included in the resulting string.

   The base case is that an empty `t` converts to `"."` (a single dot character). Use the provided helper function `amodule.longest_run(t)`, which returns a tuple `(start, stop)`, the range for the longest run, i.e. LR begins at t[start] and ends at t[stop-1]. `convert` must be implemented **without using any loops**.
4. **`process(count:int, sep:str)`**: This function orchestrates the others. It uses `make_reader` to get a reader for `count` lines and then calls the reader to actual get a list of lines from the input. It then maps the `tokens` function to each line (using the provided `sep`) to get the integer tokens. Finally, it maps the `convert` function to each token (after converting it to a string) to get the transformed string. It returns a map of maps object containing the final transformed strings.

Partial Credit: If your `process` function returns a Python container (e.g., list of lists) instead of a map of maps, but the content is correct, you will receive 60% of the total marks for part 4.

?

Example:
Assume `process(2, ',')` is called and the following lines are input:

```
111233,44,abc
5,,-10
```

The first line is split by ',' into `"111233"`, `"44"`, and `"abc"`. The token `"abc"` is discarded as it's not an integer. The second line is split into `"5"`, `""`, and `"-10"`.

The valid integer tokens are then converted. The function `convert` recursively finds the first longest run (LR) in `t`, splits it into `BLR` (before), `LR` (run), and `ALR` (after), and returns `convert(BLR) - LR - convert(ALR)`. The empty string returns `'.'` (base case).

Concrete conversions (illustrative recursion trace):

- `convert("111233")`: BLR="", LR="111", ALR="233" → `".-111-((.-2-.)-33-.)"`
- `convert("44")`: LR="44" → `".-44-."`
- `convert("5")`: → `".-5-."`
- `convert("")`: → `"."`
- `convert("-10")`: → `".---((.-1-.)-0-.)"`

The parentheses only reflect recursive nesting and base cases.

The final output of `process` (when materialized as a list of lists) is:

```
[ [".-111-.-2-.-33-.", ".-44-."],
  [".-5-.", ".", ".---.-1-.-0-."] ]
```

Restrictions:

- Do not change any function signatures in `p1.py`.
- `tokens` and `convert` must not use loops (including list comprehensions).
- `tokens` must be implemented with a single `return` statement after its helper function.
- `convert` must be implemented recursively.
- You must use the provided `amodule.longest_run()` function.

Problem 2: General-Purpose Function Invocation Counter

Write a High Order Function `counter` that defines and returns a wrapper function for any function `f`, which maintains a running count of how many times `f` has been invoked. The wrapper should work for any `f` that accepts any number of positional arguments.

Each time the returned wrapper is called, it must:

1. Print the call number for that specific function. The first call is numbered 1**.**
2. Execute the original function with all its arguments normally.
3. Return the result of the original function unchanged.

Goal

Construct a function that:

- Maintains a **persistent, per-function counter**.
- Prints exactly: `Call number {xx} for {f.__name__}`, where xx is the call number.
- Calls `f` with **all its arguments**.
- Returns `f`'s result unchanged.

Function Definition

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

Input Parameters:

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

Returns:
A new wrapper function that behaves exactly like `f` but also prints the invocation count before each execution of f.

Example `p2_input.txt`:

```
5
add(3, 5)
is_even(10)
get_pi()
add(1, 2)
concat("hello", " world")
```

Expected Output Example

- `add(3, 5)` → First call to `add` → Prints: `Call number 1 for add`
- `is_even(10)` → First call to `is_even` → Prints: `Call number 1 for is_even`
- `get_pi()` → First call to `get_pi` → Prints: `Call number 1 for get_pi`
- `add(1, 2)` → Second call to `add` → Prints: `Call number 2 for add`

Notes

1. The wrapper for each distinct function maintains its own separate count for that function.
2. The wrapper must correctly handle functions with zero, one, or more arguments.

Requested files

p1.py

```python
# p1.py
"""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.

    If it encounters an EOFError while reading, it must print the exact message:
    f"Fewer than {line_count} were provided"
    and then exit the program. You can use exit() or raise SystemExit.
    """
    pass


def tokens(s: str, sep: str) -> filter:
    """
    Returns a filter object of valid "words" in string s, separated by 'sep'.

    A word is considered valid if it can be converted into an integer.
    A special case is that an empty string ("") is also considered valid and
    represents the integer 0.

    Constraints:
    - Must be implemented with a single 'return' statement (a helper function is allowed).
    - Must not use any loops.
    """
    pass


def convert(t: str) -> str:
    """
    Recursively converts a string t into another string based on its longest
    run of repeated characters.

    - Base Case: An empty string converts to a single dot ('.').
    - Recursive Step: The function finds the first longest run of repeated
      characters (LR) in t using amodule.longest_run(). It splits t into
      the part before the run (BLR) and the part after (ALR). The result is
      formatted as: convert(BLR)-LR-convert(ALR).

    Constraints:
    - Must be implemented recursively.
    - Must not use any loops.

    Example:
        convert("aaabbc") -> ".-aaa-.-bb-.-c-."
    """
    pass


def process(count: int, sep: str):
    """
    Orchestrates the reading, tokenizing, and converting of input lines.

    1. Calls make_reader() to get a reader function, then calls it to get input lines.
    2. Tokenizes each line using the 'tokens' function with the specified 'sep'.
    3. Converts each resulting token using the 'convert' function.
    4. Returns a map of maps, where each inner map contains the converted tokens for one line.

    Partial Credit: Returning a list of lists or similar container instead of a
    map of maps will be graded for 60% of the marks for this function.
    """
    pass


#### #### #### #### #### #### #### #### ####
# To test your code, change values of N and SEP:
N = 3
SEP = ","
#### #### #### #### #### #### #### #### ####

########### Do not change below this line ###########
def solution(n, sep):
    import copy
    try:
        M = process(n, sep)
        materialized = [list(line) for line in M]
        print(f"**********************************************")
        print(f"Processing {n} lines with separator '{sep}':")
        for lno, line_list in enumerate(materialized):
            print(f'Line {lno} converted 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 parsing result from process() function. Error:{e}")
    return None

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

p2.py

```python
# p2.py
"""Boilerplate for Problem 2: Function Call Counter."""

from typing import Callable, Any # Do not change

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

    This is a Higher-Order Function (HOF). It must return a NEW function
    (a "wrapper") that replaces the original function 'f'. The wrapper must
    be able to accept any number of positional arguments.

    Args:
        f (Callable): The original function to wrap.

    Returns:
        Callable: A new wrapper function that, when called:
                  1. Prints "Call number {xx} for {f.__name__}"
                  2. Executes the original function `f` with the given arguments.
                  3. Returns the result of `f` unchanged.
    """
    # todo: your code: Be sure to set and update count appropriately

    return None


############################ Do Not Change #################################
# The below code is for testing your implementation
def solution(tc):
    func, args, kwargs = tc
    if func is None:
        print("Error: counter not implemented correctly.")
        return None
    return func(*args, **kwargs)

def process_input(filename: str):
    """Parses input file and dynamically loads + decorates only required functions."""
    import p2_utils
    import ast
    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):
            # This line correctly calls the student's 'counter' function
            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:
        # Correctly handles various argument formats, including keyword arguments
        try:
            tree = ast.parse(line)
            call_node = tree.body[0].value
            func_name = call_node.func.id

            args = [ast.literal_eval(arg) for arg in call_node.args]
            kwargs = {kw.arg: ast.literal_eval(kw.value) for kw in call_node.keywords}

            func = functions.get(func_name)
            parsed_tests.append((func, args, kwargs))

        except (ValueError, SyntaxError, AttributeError) as e:
            # Fallback for simple cases if full parsing fails
            print(f"Warning: Could not parse '{line}' with AST. Falling back to simple parsing. Error: {e}")
            func_name, rest = line.split("(", 1)
            arg_str = rest.rstrip(")").strip()
            func = functions.get(func_name.strip())
            parsed_tests.append((func, [arg_str], {}))

    return parsed_tests


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

p2_input.txt

```
8
is_even(4)
add(10, 20)
is_positive(-5)
concat("hello", " ", "world")
is_even(7)
add(1, 1)
is_positive(100)
concat("test")
```

input.txt

```
111233,44,abc
5,,-10
10,20,30,abc,40
55|66|xyz|77|
8-9--11-not_a_num
100 200 300
999,
```

VPL