Mark as done

Description Submission view

Available from: Friday, 31 October 2025, 9:20 AM
Due date: Friday, 31 October 2025, 10:50 AM
Requested files: p1.py, p2.py, p2_input.txt, p1_input.txt (Download)
Maximum number of files: 5
Type of work: Individual work

Problem 1

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

Notes on Testing:

Your reader function will take input from the terminal, but you will not need to type the data 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.)

Task:
Go to `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()`: This function returns a parameter-less function that reads lines until end-of-file is reached. It returns a list of those lines. If there is any `\n` (there may be none, one or more) on any line, that `\n` must be respected — the input line must be split into multiple lines in the returned list. Note that end-of-file is indicated by input() getting an `EOFError` exception. You must handle it.

2. **`tokens(s:str, sep:str) -> filter`**: This function has a **single return statement** that returns a `collection` of valid words in string `s`. Words are separated in string `s` by any character present in the string `sep`. tokens may define an internal helper function and call it in that single statement. Word w is valid if `isvalid(w)` returns `True`. (**Do not use loops**, not even in the helper function.)
Hint: The helper function is allowed to be recursive. One strategy to consider: preprocess s, replacing different separator characters with a single character.

3. **`convert(t:str) -> str`**: This **recursive** function transforms string t into a new string based on the following rule: `convert(t)` is a string encoding a list of up to three string parts as in `[convert(BLR),LR,convert(ALR)]` (That is a string, not a list type, and no spaces are introduced.), where `LR` indicates the longest run of `t`, `BLR` is the part of `t` before `LR`, and `ALR` is the part of `t` after `LR`. If any of these three parts is empty, it is not included in the encoding and there is no corresponding comma either. Thus, there may be 1, 2, or 3 parts in the output string separated by commas. Empty string converts to empty string.
Use the provided helper function `amodule.longest_run(t)`, defined as `def longest_run(t:str) -> tuple[int, int]`, which returns tuple `(start, stop)` if the longest run of `t` is from `t[start]` to `t[stop-1]` (both inclusive). `convert` must be implemented **without using any loops**.

4. `process(sep:str, isvalid:Callable[...,bool])`: This function first calls `make_reader` to get a reader function for input, and then calls that reader function to get a list of input lines. `process` next tokenizes each line of that list by mapping tokens to lines, using the argument `sep` as the separator and argument `isvalid` as the validity detector. Finally, `process` converts each token of each line. `process` returns a `map of maps` (each of those `maps` representing the converted tokens of one line).

Partial Credit & challenge: If your `process` function returns a `list of lists` instead of `map of maps`, it will be graded on **50%** for part 4. There is no extra credit for this, but see if you can also write process with only a **single composite statement**.

Example:
Assume `process(" ,", amodule.isvalid)` is called and the input file contains the following single line (which includes a newline character `\n`):

```
alpha,beta excellence\nDelta,a word123 longwordistoolong
```

1. Reader Output
The `make_reader` function will process this input and return a list of two strings:

?

```
['alpha,beta excellence', 'Delta,a word123 longwordistoolong']
```

2. Tokenizer and Validator Output
The `tokens` function is then applied to each of these lines with the separator `" ,"`.

- **For the first line (`'alpha,beta excellence'`):**
  - Splitting by space and comma yields the tokens: `"alpha"`, `"beta"`, `"excellence"`.
  - Suppose `isvalid` returns `True` for all three words.
- **For the second line (`'Delta,a word123 longwordistoolong'`):**
  - Splitting yields the tokens: `"Delta"`, `"a"`, `"word123"`, `"longwordistoolong"`.
  - Suppose `isvalid` returns `False` for all four of these tokens.

The valid tokens to be converted are therefore `['alpha', 'beta', 'excellence']` from the first line and an empty list from the second.

3. Concrete Conversions (Illustrative Recursion Trace)
The `convert` function is called on each valid token.

- `convert("alpha")` → `"[a,[l,[p,[h,[a]]]]]"`
- `convert("beta")` → `"[b,[e,[t,[a]]]]"`
- **Detailed trace for `convert("excellence")`:**
  1. `longest_run('excellence')` finds `"ll"`. This means:
     - `BLR = "exce"`
     - `LR = "ll"`
     - `ALR = "ence"`
  2. The structure of the result will be `"[" + convert("exce") + ",ll," + convert("ence") + "]"`.
  3. **Recursive calls** are made to evaluate `convert("exce")` and `convert("ence")`.
  4. `convert("exce")` eventually resolves to `"[e,[x,[c,[e]]]]"`.
  5. `convert("ence")` eventually resolves to `"[e,[n,[c,[e]]]]"`.
  6. Substituting these back gives the final result.

  Final result for `convert("excellence")`: `"[[e,[x,[c,[e]]]],ll,[e,[n,[c,[e]]]]]"`

4. Final Output
The `process` function returns a map object. When materialized as a list of lists (as the testing harness does for display), the literal final output is:

```
[['[a,[l,[p,[h,[a]]]]]', '[b,[e,[t,[a]]]]', '[[e,[x,[c,[e]]]],ll,[e,[n,[c,[e]]]]]'], []]
```

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

Problem Description

Write a higher order function `logger(f:Callable[..., Any])`, which returns a wrapper function for given function `f`. This wrapper function calls `f` but also helps maintain a global dictionary called log of results of every call to any function sent to `logger. log[f.__name__]` is a list of results produced by successive invocations of function `f`, in the order these results are produced.

Function Definition

```
def logger(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:
  - Calls the original function f with the provided arguments
  - Records the result in the global log dictionary
  - Returns the original result unchanged

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:

```
6
add(2, 3)
multiply(4, 5)
add(1, 1)
is_even(5)
multiply(2, 2)
is_even(10)
```

Expected Output Example

```
Result of add(2, 3): 5
Result of multiply(4, 5): 20
Result of add(1, 1): 2
Result of is_even(5): False
Result of multiply(2, 2): 4
Result of is_even(10): True
```

Global log contents:

```
{
  'add': [5, 2],
  'multiply': [20, 4],
  'is_even': [False, True]
}
```

Notes

- The log dictionary is shared across all wrapped functions.
- Function calls are executed in the order they appear in the input file.
- Results are accumulated chronologically in the global log.
- The wrapper returns the original function's result unchanged.
- All helper functions used in the input (e.g., add, multiply, is_even) are pre-defined; they need not be implemented.
- The global log persists and grows with each function call throughout program execution
- Use `f.__name__` as the dictionary key

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() -> Callable[[], list]:
    """
    Creates and returns a parameter-less function that reads lines until EOF.

    The returned function should read all lines from input until an EOFError
    is encountered. If a line contains one or more newline characters ('\n'),
    it must be split into multiple lines. The function returns a list of all
    the resulting lines.
    """
    # Your code here
    pass


def tokens(s: str, sep: str, isvalid: Callable[..., bool]) -> filter:
    """
    Returns a filter object of valid "words" from string s.

    Words are separated by ANY character present in the `sep` string. A word is
    considered valid if the provided function `isvalid` returns `True` when
    called with the word as an argument.

    Constraints:
    - Must be implemented with a single 'return' statement (a helper function is allowed).
    - Must not use any loops (this includes for, while, and comprehensions).
    - Hint: A recursive helper function can be used to handle multiple separators
      without a loop. Consider a strategy where you recursively replace all
      separator characters in the input string with a single, chosen separator (like
      the first character of `sep`). Once the string is normalized, a simple
      `split()` will be sufficient.
    """
    # Your code here
    pass


def convert(t: str) -> str:
    """
    Recursively converts a string `t` into a new string that encodes a list
    of its parts.

    The conversion rule is as follows: `convert(t)` becomes a string formatted
    as `[convert(BLR),LR,convert(ALR)]`, where:
      - LR is the first longest run of repeated characters in `t`.
      - BLR is the part of `t` before LR.
      - ALR is the part of `t` after LR.

    Important Rules:
      - If any of the three parts (convert(BLR), LR, or convert(ALR)) is an
        empty string, it is completely omitted from the output, including its
        comma. The output might have 1, 2, or 3 parts.
      - The base case is that an empty string `t` converts to an empty string `""`.
      - You must use the provided `amodule.longest_run(t)` function.

    Example:
      - `convert('weer')` -> `'[[w],ee,[r]]'`
      - `convert('aaab')` -> `'[aaa,[b]]'` (BLR is empty)
      - `convert('bbaaa')` -> `'[[bb],aaa]'` (ALR is empty)
      - `convert('ccc')` -> `'[ccc]'` (BLR and ALR are empty)
    """
    # Your code here
    pass


def process(sep: str, isvalid: Callable[..., bool]):
    """
    Orchestrates the reading, tokenizing, and converting of input lines.

    This function is the main driver of the program. It receives a separator string
    `sep` and a validation function `isvalid`.

    Steps:
    1. Calls make_reader() to get a reader function, then calls it to get all input lines.
    2. Tokenizes each line by calling the `tokens` function with the specified `sep`
       and the `isvalid` function provided to this `process` function.
    3. Converts each valid token resulting from step 2 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 50% of the marks for this function.
    """
    # Your code here
    pass


#################################################################
# HOW TO TEST YOUR CODE LOCALLY:
#
# 1. Edit the contents of the `p1_input.txt` file with your test data.
# 2. Change the value of the SEP variable below to match your test.
# 3. Press the "Run" button.
#
#################################################################

SEP = " ,"   # We have a whitespace and , in SEP. So, separators would be " " & ","

########### Do not change below this line ###########
def solution(sep):
    import copy
    try:
        M = process(sep, amodule.isvalid)
        materialized = [list(line) for line in M]
        print(f"**********************************************")
        print(f"Processing all input 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 Exception as e:
        print(f"Error parsing result from process() function. Error:{e}")
    return None

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

p2.py

```python
"""Student implementation for Problem 2: Function Call Logger with Global Result Tracking"""

from typing import Callable, Any

# Global log dictionary - DO NOT MODIFY THIS LINE
log = {}

def logger(f: Callable[..., Any]) -> Callable[..., Any]:
    """
    A HOF that logs all function call results in a global dictionary.

    Returns a wrapper function that:
    - Maintains a global log of all function call results
    - Appends results to log[f.__name__] in chronological order
    - Returns the original function result unchanged

    Args:
        f: Any callable function to wrap

    Returns:
        A wrapper function that returns f(...) and logs the result
    """
    # TODO: Implement this function
    # You need to:
    # 1. Create a wrapper function that captures the global log
    # 2. The wrapper should:
    #    - Call the original function f with given arguments
    #    - Append the result to log[f.__name__] list (create list if needed)
    #    - Return the original result unchanged
    # 3. Use the global log dictionary defined above

    # WRITE YOUR CODE BELOW
    pass

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

def solution(tc):
    func, arg = tc
    if func is None:
        print("Error: logger not implemented correctly.")
        return None
    result = func(arg) if arg is not None else func()
    print(f"Result: {result}")
    return result


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: logger(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, rest = line.split("(", 1)
        arg_str = rest.rstrip(")").strip()
        func = funcs.get(name.strip())

        if not arg_str:
            parsed.append((func, None))
        elif "," in arg_str:
            # Handle multiple arguments
            args = []
            for arg in arg_str.split(","):
                arg = arg.strip()
                if arg[0] in "\"'":
                    args.append(arg[1:-1])
                else:
                    try:
                        args.append(int(arg))
                    except ValueError:
                        args.append(arg)
            parsed.append((func, args))
        elif arg_str[0] in "\"'":
            parsed.append((func, arg_str[1:-1]))
        else:
            try:
                parsed.append((func, int(arg_str)))
            except ValueError:
                parsed.append((func, arg_str))
    return parsed


if __name__ == "__main__":
    # Clear log at start
    log.clear()

    test_cases = process_input("p2_input.txt")

    for tc in test_cases:
        solution(tc)

    print("\nGlobal log contents:")
    print(log)
```

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)
```

p1_input.txt

```
alpha,beta excellence\nDelta,a word123 longwordistoolong,gamma
```

VPL