Mark as done

Description Submission view

Available from: Thursday, 16 October 2025, 9:15 AM
Due date: Thursday, 16 October 2025, 10:45 AM
Requested files: p1.py, p1_input.txt, p2.py, p2_input.txt, p3.py, p3_input.txt, p4.py, p4_input.txt (Download)
Type of work: Individual work

Instructions for LabTest3-Day4

This set contains 4 problems, the descriptions of which are provided below. A starter code is provided for each problem: p1.py, p2.py, p3.py, and p4.py for problems 1,2,3 and 4, respectively. Each file contains some sections that are necessary for code execution and an editable section where you are expected to write your code.

Do not change anything in the non-editable sections; otherwise, you will run into issues.

Running test cases:

You are provided a text file named pno_input.txt for each problem. You can use this file to test your code for any errors. The inputs for each problem can be provided in this file. Sample test cases are provided for all problems. You can add as many testcases you want in corresponding input files. *Usage of each input file is described in the corresponding problem description.*

Problem 1: Password Validator Factory

Description:

Create a Python function named create_password_validator that acts as a "factory" function. This function should return another function (the actual validator).

The returned validator function should take a single string argument (a password) and check if it meets the following criteria:

1. **Length:** At least 8 characters long.
2. **Uppercase:** Contains at least one uppercase letter.
3. **Lowercase:** Contains at least one lowercase letter.
4. **Digit:** Contains at least one digit.
5. **Special Character:** Contains at least one special character from the set: !@#$%^&*

The validator function should return a tuple: (is_valid, messages).

- is_valid: A boolean (True if all criteria are met, False otherwise).
- messages: A list of strings, where each string describes a criterion that failed. If all criteria pass, this list should be empty.
    - Example messages for failed criteria: "No Uppercase", "No Lowercase", "No Digit", "No Special Char", "Too Short".

Function Definitions:

You will define the function create_password_validator with no arguments, and it will return a Callable.

```
def create_password_validator() -> Callable[[str], Tuple[bool, List[str]]]:
    # This function returns another function
```

The returned validator function will have the signature:

```
def validator_function(password: str) -> Tuple[bool, List[str]]:
```

- **Input parameter (for create_password_validator):** None.
- **Output parameter (for create_password_validator):** A validator function.
- **Input parameter (for validator function):** password (a string).
- **Output parameter (for validator function):** A tuple (is_valid, messages_list).

?

Input File (p1_input.txt):

Your solution will be tested using input provided in the p1_input.txt file. You should update this file to contain the desired test cases.

- **Input file format:**
    - The first line indicates the number of test cases.
    - Each subsequent line is a password string to be validated.
- **Example p1_input.txt content**

```
2
Password123!
NOdigits!
```

Expected Output Example:

Given password = "Password123!"
Your validator should produce the output:

```
(True, [])
```

(Order of messages in the list does not strictly matter, but consistency is good.)

Notes and Restrictions:

- The *create_password_validator* function should return a function, not execute the validation directly. The *process_input* in the backend will first call *create_password_validator* once to get the validator, then use it for all test cases.

Problem 2: Apply Operations Sequence

Description:

Write a function named apply_operations_sequence that takes an initial value and an arbitrary number of function arguments (*operations). Each operation is a function that takes one argument and returns a single value.

The apply_operations_sequence function should apply these operations to the value in the exact order they are passed as arguments. The result of each operation becomes the input for the next.

The function will be tested on simple numeric functions such as square, cube, doubling, negate, increment, etc. All the helper functions are already defined, and you should not edit the definition of these helper functions.

Function Definition:

```
def apply_operations_sequence(value: Any, *operations: Callable[[Any], Any]) -> Any:
```

- **Input parameters:**
    - value: The initial value to which the operations are applied. Can be of any type, but primarily tested with integers.
    - *operations: An arbitrary number of callable functions. Each function takes one argument and returns a single value.
- **Output parameter:** The final value after all operations have been applied in sequence.

Input File (p2_input.txt):

Your solution will be tested using input provided in the p2_input.txt file. You should update this file to contain the desired test cases.

- **Input file format:**
    - The first line indicates the number of test cases.
    - Each test case will consist of:
        - A line with the initial value (integer).
        - A line with a list of function names (strings), separated by spaces, representing the operations to apply in sequence.
- **Example p2_input.txt :**

```
2
3
increment double_it square_it
-5
negate add_ten
```

Expected Output Example 1:

Given value = 3, operations = [increment, double_it, square_it]

The sequence should perform:

1. increment(3) -> 4
2. double_it(4) -> 8
3. square_it(8) -> 64

Your function should return: `64`

Notes and Restrictions:

- The apply_operations_sequence function should be generic enough to work with any number of operations and various compatible types (though primarily integers will be used for testing).
- Ensure the operations are applied strictly in the order they are passed.
- If no operations are provided, the original value should be returned.
- No external libraries needed.

Problem 3 - Recursive Character Removal

Description:
Create a Python function named `removeChar` that takes two parameters — a string `s` and a character `c` — and returns a **new string** with **all occurrences** of character `c` removed. The function must be implemented **recursively**, without using loops (`for`, `while`) or built-in string replacement methods such as `replace().` If the character `c` is not found in `s`, the function should return the original string unchanged.

Function Definition:
`def removeChar(s: str, c: str) -> str:`

Input parameter:

- `s`: A string from which the character needs to be removed.
- `c`: A single character to remove from the string.

Output parameter:
A new string with all instances of character `c` removed.

Input File (p3_input.txt):

Your solution will be tested using input provided in the `p3_input.txt` file. You should update this file to include your desired test cases.

Input file format:
The first line indicates the number of test cases.
Each subsequent line contains a test case in the format:

```
string,character
```

Example (*p3_input.txt*):

```
3
banana,a
hello world,l
recursion,r
```

Output:

```
bnn
heo word
ecusion
```

Restrictions:

- The function **must** use recursion to process one character at a time.

- Do **not** use loops or string replacement functions (`replace()`, `filter()`, etc.).
- The function should handle both uppercase and lowercase characters correctly.
- Use only 1 base case to terminate the string.

Problem 4: Balanced Parentheses

Description:

Create a Python function named `is_balanced` that takes a string consisting only of opening `(` and closing `)` parentheses. The function should return `True` if the parentheses in the string are "balanced" and `False` otherwise.

A string of parentheses is considered balanced if:

1. For every opening parenthesis, there is a corresponding closing parenthesis.
2. A closing parenthesis never appears before its corresponding opening parenthesis (e.g., `)(` is not balanced).

Function Definition:

```
    def is_balanced(s: str) -> bool:
```

- **Parameters:**
    - `s`: A string containing only the characters `(` and `)`.
- **Returns:** `True` if the string is balanced, `False` otherwise.

Input File (`p4_input.txt`):

- The first line is the number of test cases.
- Each subsequent line is a string of parentheses to be checked.

Example `p4_input.txt`:

```
4
(()())
(()
)(
()
```

Expected Output for Example:

```
True
False
False
True
```

Notes:

- An empty string is considered balanced.

Requested files

p1.py

```python
from typing import Callable, Tuple, List

def create_password_validator() -> Callable[[str], Tuple[bool, List[str]]]:

    def validator_function(password):

    #write code here


    return validator_function




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

def solution(password_str):
    validator = create_password_validator()
    return validator(password_str)

def process_input(filename):
    lines = open(filename, 'r').readlines()
    lines = [line.strip() for line in lines]
    num_tests = int(lines[0])
    input_tests = []

    for t in range(1, num_tests + 1):
        input_tests.append(lines[t])

    return input_tests

if __name__ == "__main__":
    Input = process_input('p1_input.txt')
    for password in Input:
        print(solution(password))
```

p1_input.txt

```
3
ValidP@ss1
Short#2
AllValid!1
```

p2.py

```python
# Example helper functions - these will be available to the VPL system's process_input
def _increment(x):
    return x + 1

def _double_it(x):
    return x * 2

def _square_it(x):
    return x * x

def _negate(x):
    return -x

def _add_ten(x):
    return x + 10

def _cube_it(x):
    return x ** 3


def apply_operations_sequence(value, *operations):
    #write code here


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

# Map string names to actual helper functions
_func_map = {
    "increment": _increment,
    "double_it": _double_it,
    "square_it": _square_it,
    "negate": _negate,
    "add_ten": _add_ten,
    "cube_it": _cube_it,
}


def solution(tc):
    value, func_names = tc
    operations = []
    i = 0
    while i < len(func_names):
        operations.append(_func_map[func_names[i]])
        i += 1
    return apply_operations_sequence(value, *operations)


def process_input(filename):
    lines = open(filename, 'r').readlines()
    lines = [line.strip() for line in lines]
    num_tests = int(lines[0])
    input_tests = []
    current_line_idx = 1

    for _ in range(num_tests):
        value_str = lines[current_line_idx]
        current_line_idx += 1

        operations_str = lines[current_line_idx]
        func_names = operations_str.split()
        current_line_idx += 1

        try:
            value = int(value_str)
        except:
            value = value_str

        input_tests.append((value, func_names))

    return input_tests


if __name__ == "__main__":
    Input = process_input('p2_input.txt')
    i = 0
    while i < len(Input):
        print(solution(Input[i]))
        i += 1

```

p2_input.txt

```
2
3
increment double_it square_it
-5
negate add_ten
```

p3.py

```python
def removeChar(s: str, c: str) -> str:
    # write your code here below






















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

def solution(inp):
    return removeChar(*inp)


def process_input(filename):
    with open(filename, 'r') as file:
        lines = [line.strip() for line in file.readlines()]

    num_tests = int(lines[0])
    input_tests = []

    for t in range(1, num_tests + 1):
        s, c = lines[t].split(",")
        input_tests.append([s.strip(), c.strip()])

    return input_tests


if __name__ == "__main__":
    Input = process_input('p3_input.txt')
    for inp in Input:
        print(solution(inp))

```

p3_input.txt

```
3
banana,a
hello world,l
recursion,r
```

p4.py

```python
def is_balanced(s: str) -> bool:
    # write code here
    pass

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

def solution(tc):
    return is_balanced(tc)

def process_input(filename):
    lines = open(filename, 'r').readlines()
    lines = [line.strip() for line in lines]
    num_tests = int(lines[0])
    input_tests = []

    for i in range(1, num_tests + 1):
        input_tests.append(lines[i])

    return input_tests

if __name__ == "__main__":
    Input = process_input('p4_input.txt')
    for s_val in Input:
        print(is_balanced(s_val))
```

p4_input.txt

```
4
(()())
(()
)(
()
```

VPL