Mark as done

Description Submission view

Available from: Tuesday, 14 October 2025, 9:15 AM
Due date: Tuesday, 14 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-Day2

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: Partition List by Predicate

Description:

Create a Python function named partition_list that takes two arguments:

1. *lst*: A list of elements of any type.
2. *predicate*: A function that takes a single argument from the list and returns True or False.

The function should return a tuple containing two new lists: (truthy, falsy).

- *truthy*: Contains all elements from the original lst for which the predicate function returned True.
- *falsy*: Contains all elements from the original lst for which the predicate function returned False.

Both the truthy and falsy lists must maintain the original relative order of elements as they appeared in the input lst.

Function Definition:

You will define the function with the following signature:

```
def partition_list(lst: List[Any], predicate: Callable[[Any], bool]) -> Tuple[List[Any], List[Any]]:
```

- **Input parameters:**
  - `lst`: A list of elements.
  - `predicate`: A function that takes an element and returns a boolean.
- **Output parameter:** A tuple (*truthy_list*, *falsy_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 test case will consist of:
    - A line with the name of the predicate function (as a string).
    - A line with the number of elements in the lst.
    - Subsequent lines, each representing an element of the list.

?

- **Example p1_input.txt content:**

```
2
is_greater_than_5
5
1
8
3
6
10
has_char_a
4
apple
banana
grape
kiwi
```

- **Output Example:**

  Given *lst* = `[1, 8, 3, 6, 10]` and *predicate* = `is_greater_than_5`
  Your function should produce the output:
  `([8, 6, 10], [1, 3])`

Notes and Restrictions:

- Ensure the original order of elements is preserved in both the truthy and falsy lists.
- The input list can contain elements of various types (integers, strings, etc.), depending on the predicate.
- You do not need to define the predicate functions in p1.py yourself. They will be available via the *predicate_map* in the *process_input* function.

Problem 2: Function Piping

Description:

This problem involves defining and then composing simple transformation functions.

Part 1: Define Helper Functions
You need to define three simple functions:

1. *doubler(x):* A function that doubles its input value *x*.
2. *add_ten(x)*: A function that adds ten to its input value *x*.
3. *squarer(x)*: A function that squares its input value *x*.

These functions will operate on integers.

Part 2: Create a pipe Function
Write a higher-order function named pipe that takes an arbitrary number of functions as arguments (*functions). This pipe function should return a new, composed function. The composed function, when called with an initial value, should apply the input functions **from left to right**, passing the output of each function as the input to the next.

Function Definitions:

You will define the following 4 functions:

```
1. def doubler(x: int) -> int:
2. def add_ten(x: int) -> int:
3. def squarer(x: int) -> int:
4. def pipe(*functions: Callable[[Any], Any]) -> Callable[[Any], Any]:

# The processor variable will be defined by calling your pipe function like below:
# processor = pipe(doubler, add_ten, squarer, doubler, squarer, ...)
```

- **Input parameters:**
  - *doubler, add_ten, squarer: x* (an integer).

- *pipe*: **functions* (**an arbitrary number of callable functions**).

- **Output parameters:**
  - For *doubler, add_ten, squarer* functions: An integer.
  - For *pipe* function: A new callable function

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 subsequent line contains an integer value on which the processor function should be applied.

- **Example p2_input.txt content:**

```
3
2
0
-1
```

- **Example Output:**

```
196
100
64
```

Output Explanations:

Example 1:

Input Value: `2`

The processor should perform:

1. doubler(2)→2×2=4
2. add_ten(4)→4+10=14
3. squarer(14)→142=196

Your combined output for processor(2) should be: `196`

Example 2:

Input Value: `0`

The processor should perform:

1. doubler(0)→2×0=0
2. add_ten(0)→0+10=10
3. squarer(10)→102=100

Your combined output for processor(0) should be: `100`

Example 3:

Input Value: `-1`

The processor should perform:

1. doubler(−1)→2×(−1)=−2
2. add_ten(−2)→−2+10=8
3. squarer(8)→82=64

Your combined output for processor(−1) should be: `64`

Notes and Restrictions:

- All helper functions (doubler, add_ten, squarer) and the pipe function should be implemented.
- The pipe function should be generic to work with any sequence of functions, not just the three provided.
- The processor variable will be defined in the p2.py file using your pipe function.

Problem 3 - Recursive Power Calculation

Description:

Create a Python function named `power` that takes two integer arguments — `base` and `exponent` — and returns the result of raising `base` to the power of `exponent` (**base^{n}**). The function must be implemented **recursively**, without using loops (`for`, `while`) or built-in power functions such as `pow()` or the exponentiation operator `**`.

Function Definition:

You will define the function with the following signature:

```
def power(base: int, exponent: int) -> int:
```

Input parameters:

- `base`: An integer representing the base number.
- `exponent`: A non-negative integer representing the power to which the base is raised.

Output parameter:

An integer value representing `base` raised to the power of `exponent`.

Input File (p3_input.txt):

You can test your solution 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 two integers separated by a comma, representing `base` and `exponent`.

Example p3_input.txt content:

```
2
2,3
3,4
```

Expected Output Example:

For the above input, the output should be:

```
8
81
```

Notes and Restrictions:

- You must implement the solution **recursively**.
- Make only one base condition in your recursion function(i.e. exponent==0)
- Do **not** use the `pow()` function or the `**` operator.
- Assume `base` and `exponent` will always be positive integers(i.e. base > 0, exp > 0)

Problem 4: Find Prime Pair Sum

Description:

For any given even integer `n` greater than 2, create a Python function named `find_prime_pair` that finds two prime numbers, `p` and `q`, such that their sum equals `n` (i.e., `p + q = n`).

If multiple pairs of primes exist, your function must return the pair where the absolute difference between `p` and `q` (`|p - q|`) is the smallest. This is equivalent to finding the prime pair that is closest to `n/2`.

Function Definition:

```
from typing import Tuple
def find_prime_pair(n: int) -> Tuple[int, int]:
# Your code here
```

- **Parameters:**
  - `n`: An even integer greater than 2.
- **Returns:** A tuple `(p, q)` containing the two primes, where `p <= q`.

Input File (`p4_input.txt`):

- The first line is the number of test cases.
- Each subsequent line contains a single even integer `n`.

Example `p4_input.txt`:

```
2
28
74
```

Expected Output for Example:

For `n = 28`, possible pairs are (5, 23) and (11, 17). The pair (11, 17) has a smaller difference. For `n = 74`, the pair is (3, 71), (5, 69-no), ..., (31, 43), (37, 37). The pair (37,37) has the smallest difference.

```
    (11, 17)
    (37, 37)
```

Notes:

- You will likely need to write a helper function to check if a number is prime.
- You can assume that a solution will always exist for the given inputs.

Requested files

p1.py

```python
from typing import List, Any, Callable, Tuple # do not change

def partition_list(lst: List[Any], predicate: Callable[[Any], bool]) -> Tuple[List[Any], List[Any]]:
    #write code here









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

def _is_even(x):
    return x % 2 == 0

def _is_greater_than_5(x):
    return x > 5

def _has_char_a(s):
    return 'a' in s.lower() if isinstance(s, str) else False

def _is_negative(x):
    return x < 0

def _is_divisible_by_3(x):
    return x % 3 == 0

def _has_vowel(s):
    if not isinstance(s, str):
        return False
    for char in s.lower():
        if char in 'aeiou':
            return True
    return False

def _is_single_digit(x):
    return 0 <= x <= 9

def _ends_with_s(s):
    if not isinstance(s, str):
        return False
    return len(s) > 0 and s[-1].lower() == 's'

def _is_between_10_and_20(x):
    return 10 <= x <= 20

def _starts_with_capital(s):
    if not isinstance(s, str) or len(s) == 0:
        return False
    return 'A' <= s[0] <= 'Z'

def _is_multiple_of_5(x):
    return x % 5 == 0

_predicate_map = {
    "is_even": _is_even,
    "is_greater_than_5": _is_greater_than_5,
    "has_char_a": _has_char_a,
    "is_negative": _is_negative,
    "is_divisible_by_3": _is_divisible_by_3,
    "has_vowel": _has_vowel,
    "is_single_digit": _is_single_digit,
    "ends_with_s": _ends_with_s,
    "is_between_10_and_20": _is_between_10_and_20,
    "starts_with_capital": _starts_with_capital,
    "is_multiple_of_5": _is_multiple_of_5,
}
def solution(tc):
    inp_lst, inp_pred_name = tc
    predicate_func = _predicate_map[inp_pred_name]
    return partition_list(inp_lst, predicate_func)

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

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

        num_elements = int(lines[current_line_idx])
        current_line_idx += 1

        current_list = []
        for _ in range(num_elements):
            element_str = lines[current_line_idx]

            numeric_predicates = {
                "is_even", "is_greater_than_5", "is_negative", "is_divisible_by_3",
                "is_single_digit", "is_between_10_and_20", "is_multiple_of_5"
            }

            string_predicates = {
                "has_char_a", "has_vowel", "ends_with_s", "starts_with_capital"
            }

            if predicate_name_str in numeric_predicates:
                try:
```

```
                    if '.' in element_str:
                        current_list.append(float(element_str))
                    else:
                        current_list.append(int(element_str))
                except ValueError:
                    current_list.append(element_str)
            else:
                current_list.append(element_str)

            current_line_idx += 1

        input_tests.append((current_list, predicate_name_str))

    return input_tests


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

p1_input.txt

```
3
has_char_a
3
apple
banana
grape
is_greater_than_5
4
10
2
8
4
is_even
1
7
```

p2.py

```
def doubler(x):
    #write code

def add_ten(x):
    #write code

def squarer(x):
    #write code

def pipe(*functions):
    #write code




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

processor = pipe(doubler, add_ten, squarer)

def solution(tc):
    return processor(tc)

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

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

    return input_values

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

```

p2_input.txt

```
3
2
0
-1
```

p3.py

```python
def power(base: int, exponent: int) -> int:
    # write your code here below









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

def solution(inp):
    return power(*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):
        base_str, exp_str = lines[t].split(",")
        base = int(base_str.strip())
        exponent = int(exp_str.strip())
        input_tests.append([base, exponent])

    return input_tests


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

```

p3_input.txt

```
3
2,3
5,0
3,4
```

p4.py

```python
from typing import Tuple

def find_prime_pair(n: int) -> Tuple[int, int]:
    # write code here
    pass

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

def solution(tc):
    return find_prime_pair(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):
        n = int(lines[i])
        input_tests.append(n)

    return input_tests

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

p4_input.txt

```
2
28
74
```

VPL