Homework 3

MPCS 51042 – Python Programming

Due: October 27th 2020, 11:59 pm CT

Initial Setup

Make sure to perform a pull upstream inside your repository. This will grab the distribution code for hw3. The command is the following:

```
$ git pull upstream master
```

Style Guide

For this homework and all future homework assignments, we will follow the style guide used by the undergraduate Python course. It's located here: `https://classes.cs.uchicago.edu/archive/2020/fall/12100-1/style-guide/index.html`

Requirements for Problems 1-5

Problems 1-5 will have you implement a single function using the functional programming paradigm. The purpose of these problems is to provide you exposure to programming in a different style then normal. Specifically, these problems will expose you to how you would program if you were in a functional programming language. In Problems 1-5, you must adhere to these restrictions:

- When **accessing**, **modifying/creating**, and **iterating** through an iterable object (e.g., lists, dictionaries, sets, etc.), you are only allowed to use the `map`, `filter`, or `functools.reduce` functions. This means you cannot have syntax like the following: `lst[-1]`, `lst[0]`, `lst[1:3]`, `lst.append(...)` `len(lst)`, `[2,2] + [1,3]`, etc. You cannot use the methods, functions, subscript notation associated with these iterable objects. However, you can implement them using `map`, `filter`, or `functools.reduce` functions and adhering to the requirements specified in this section.
- **You can not use recursion or any looping structure (e.g., for loop, while loop, etc.)**.
- No use of global variables to help solve the problem.
- You cannot use any type of comprehension (i.e., list, dictionary, or set comprehension). However, you are allowed to use `list(...)` to convert an iterable object into a list.
- No importing other modules for help with the exception of the `functools` module. However, you can only use the `reduce` function inside `functools` and no other function.
- Specific problems may have additional restrictions so please read the problem.
- You may write helper functions but they must adhere to the restrictions above.
- You may break a tuple into parts by using unpack assignment syntax (e.g. `first,second = (1,2)`).

Please ask on Piazza if you are unsure about using certain functions. I will be happy to clarify anything so feel free to ask questions.

Problem 1

Implement the `satisfy` and `satisfy_all` functions:

- `satisfy`: takes in two predicate functions (`func1` and `func2`) and a list of integers (`values`). This function returns a list of tuples where each tuple contains two components. The first component is the integer from `values`, and the second component is `True` if both predicate functions (`func1` and `func2`) return `True` when given the integer; otherwise the second component is `False`.

  Sample Cases

```
In [1]: import problem1

In [2]: problem1.satisfy(lambda x: x > 10, lambda y: y < 100, [1,20,200])
Out[2]: [(1, False), (20, True), (200, False)]

In [3]: problem1.satisfy(lambda x: x > 10, lambda y: y < 100, [])
Out[3]: []

In [4]: problem1.satisfy(lambda x: x > 10, lambda y: y < 100, [2])
Out[4]: [(2, False)]
```

- `satisfy_all`: takes in a list of predicates (`funcs`) and a list of integers (`values`). This function returns a list of tuples where each tuple contains two components. The first component is an integer from `values`, and the second component is a list of the return values from calling each predicate function with the integer:

  Sample Cases

```
In [5]: import problem1

In [6]: funcs = [lambda x: x > 10, lambda y: y < 100]

In [7]: problem1.satisfy_all(funcs,[1,10,200])
Out[7]: [(1, [False, True]), (10, [False, True]), (200, [True, False])]

In [8]: problem1.satisfy_all(funcs,[])
Out[8]: []

In [9]: problem1.satisfy_all(funcs,[2])
Out[9]: [(2, [False, True])]

In [10]: problem1.satisfy_all(funcs,[20])
Out[10]: [(20, [True, True])]
```

There is a pytest test file (`hw3/problem1/test_problem1.py`) with additional tests. Make sure you pass those tests.

Problem 2

Implement a function `max_sum` that takes in a list of objects lists (`values`) and returns the total count of all the objects inside the lists within `values`.

```
In [1]: import problem2

In [2]: problem2.max_sum([[1,2,3],['a'],[3,4,5]])
Out[2]: 7

In [3]: problem2.max_sum([])
Out[3]: 0

In [4]: problem2.max_sum([[1],[2],[True]])
Out[4]: 3
```

Restriction:

- You cannot use the `len` function. However, think about reimplementing the `len` function using `functools.reduce`.

There is a pytest test file (`hw3/problem2/test_problem2.py`) with additional tests. Make sure you pass those tests.

Problem 3

Implement a function `duplicate` that takes in a list of integers (`values`) and an integer (`num`). This function returns a list of tuples where each tuple contains an integer from `values` duplicated `num + 1` times.

Sample Cases

```
In [1]: import problem3

In [2]: problem3.duplicate([1,2,3],3)
Out[2]: [(1, 1, 1, 1), (2, 2, 2, 2), (3, 3, 3, 3)]

In [3]: problem3.duplicate([],3)
Out[3]: []

In [4]: problem3.duplicate([1],0)
Out[4]: [(1,)]

In [5]: problem3.duplicate([100],1)
Out[5]: [(100, 100)]
```

Restriction:

- You cannot use the repetition operator ($*$) for sequence types. Think about writing a helper function that performs that functionality.

There is a pytest test file (`hw3/problem3/test_problem3.py`) with additional tests. Make sure you pass those tests.

Problem 4

Implement a function `list_range` takes in a range that is represent by an integer tuple (`rangeTup`), and a list of integers lists (`values`). The function returns a list that includes each integer list inside `values` if all their integers are within the range specified by `rangeTup` (inclusive on both ends). You can assume the

lower-bound is the first component and the upper-bound is the second component. Assume the lower-bound integer is always less then or equal to the upper-bound integer.

Sample Cases

```
In [2]: problem4.list_range((1,3),[[1,3,2],[9,87,1],[2,1]])
Out[2]: [[1, 3, 2], [2, 1]]

In [3]: problem4.list_range((1,3),[])
Out[3]: []

In [4]: problem4.list_range((1,300),[[1,3,2],[9,87,1],[2,1]])
Out[4]: [[1, 3, 2], [9, 87, 1], [2, 1]]
```

There is a pytest test file (`hw3/problem4/test_problem4.py`) with additional tests. Make sure you pass those tests.

Problem 5

Implement a function `group_ascend` takes in a list of integers as an argument and returns a list of integer lists. This function must group ascend (contiguous) sequences of integers in the original list into sublists in the result. Each sublist in the returned list represents an ascending sequence in the input list with the elements in the same relative order and the sublists in the same relative order (i.e., for two consecutive sublists, the elements of the second immediately followed the elements of the first in the original list).

For **only** this problem, you are allowed to use the following as many times as you wish:

- You can use list subscript notation to access a single element (e.g., `lst[0]`, `lst[-1]`, etc.).
- You are allowed to use the `+` operator for lists.
- You are allowed to use list literal syntax to create a list (e.g., `[2]` or `r = 2; lst =[r]`). You cannot not use comprehension syntax.

Separately, I will allow you to write a slicing(e.g. `lst[1:]`, or `{lst[1:2]},`etc.) syntax **one** time somewhere in your implementation. For example,

```
def foo(lst):
   new_lst = lst[1:]. # one time usage
   ....
########
def foo(lst):
   new_lst = lst[1:]
   f = new_lst[:] # This is not acceptable since you used slicing above.
```

Sample Cases

```
In [1]: import problem5

In [2]: problem5.group_ascend([])
Out[2]: []

In [3]: problem5.group_ascend([1])
```

```
Out[3]: [[1]]

In [4]: problem5.group_ascend([1,2,3])
Out[4]: [[1, 2, 3]]

In [5]: problem5.group_ascend([1,2,3,2,3])
Out[5]: [[1, 2, 3], [2, 3]]

In [6]: problem5.group_ascend([4,7,10,2,3,1,99,45,122,123,122,47,46,46,49])
Out[6]: [[4, 7, 10], [2, 3], [1, 99], [45, 122, 123], [122], [47], [46], [46, 49]]
```

There is a pytest test file (`hw3/problem5/test_problem5.py`) with additional tests. Make sure you pass those tests.

Problem 6

Write a function that mimics the Unix `grep` command. The function definition should look like:

```
def grep(pattern, lines, ignore_case=False):
    ...
```

`pattern` should be a string that represents a "pattern" that we want to find within each line. The function checks whether "pattern" is a substring in any of the provided lines. `lines` should be an iterable of strings (this could be a single string, a list of strings, or tuple). Finally, the `ignore_case` argument indicates whether we want to search for the pattern in a case-sensitive or case-insensitive manner. The matching lines should not be returned as a list. Instead, the function should yield the matched lines. If the function has yielded all then matched lines then it returns `None` for every subsequent call.

Note: You must implement grep as a generator function. This means it should generate the next value on demand and not upfront as with lists.

Here is an example of using `grep` based on the description above:

```
In [1]: import problem6

In [2]: lines = ['I went to Poland.',
   ...: ...          'He went to Spain.',
   ...: ...          'She is very happy.']

In [3]: t = problem6.grep('went', lines)

In [4]: for x in range(5):
   ...:     print(next(t))
   ...:
I went to Poland.
He went to Spain.
None
None
None

In [5]: t = problem6.grep('i', lines, ignore_case=True)
```

```
In [6]: for x in range(5):
   ...:     print(next(t))
   ...:
I went to Poland.
He went to Spain.
She is very happy.
None
None
```

Notice in the first example, only two lines matched the pattern. Thus, the third call (and all subsequent calls) to the function object (i.e., `t`) return `None`.

Place your solution in **hw3/problem6/problem6.py**