Fundamentals of Computer Science - programming exam (first partial)

Instructions

IMPORTANT - BEFORE YOU START:

1. Download the `.py` file from BBoard.
2. Edit the `.py` file in Spyder or VS-Code and solve the exercises.
3. Upload the `.py` file to BBoard.

Additional notes

- The exam is **not open book**!
- **Clean your mess** after you finish! Comments and tests/asserts are welcome, but unnecessary code should be deleted.
- **Tests are provided** for each exercise, although they are initially commented out. They are very helpful to understand the intended behavior and to check the correctness of your implementation. Make sure to uncomment and run them as you go through the exercises.

WARNING: for the exercises in this partial, your are **not allowed** to use lists' methods such as append, insert, ..., slices or comprehensions, unless explicitely stated. You are also **not allowed** to use set, dictionaries, or call functions from any library. Essentially, you will have to solve the exercises using plain conditionals and loops.

Exercise 1 - Find Position

Define a function `find_pos(l, x)` that takes as input a sorted list `l` and a number `x`, and returns the largest index `i` in `l` such that `x` is greater than or equal to `l[i]`. Return `-1` if `x` is less than any element of `l`.

You should also raise an exception if:

- `l` is not a `list`.
- `x` is not an `int` or a `float`

Moreover, the argument `x` should have default value `0`.

```
In [2]: find_pos([2,5,7], -5)
Out[2]: -1

In [3]: find_pos([2,5,7], 2)
Out[3]: 0

In [4]: find_pos([2,5,7], 5.1)
Out[4]: 1

In [5]: find_pos([-2, -1,5,7])
Out[5]: 1
```

Exercise 2 - Permutation?

HINT: for this exercise, you'll need to create a list of length `n` filled with `False` . Use this syntax:

```
  seen = [False for i in range(n)]
```

Write a function called `isperm` that takes as input a list of integers `l` . Say that the list has length `n`. The function must return `True` if the list represents a permutation of the indices `0, 1, ... , n - 1`, and `False` otherwise.

For example, `[0, 1, 2]` , `[2, 0, 1]` , and `[1,2, 0]` are all valid permutations.

On the other hand, `[1, 2, 3]` , `[0, 0, 0]` , and `[2, -1, 1]` are not valid permutations.

One way to answer this question would be to sort the input list and check if it is equal to the desired range from `0` to `n` . However, this is not how you should do it. Instead, you should use an auxiliary "occupancy list". This is a list of bools of length `n` , initialized with all values set to `False` . Call it `seen` (see the hint above) . Its `x` -th element `seen[x]` should answer to the question: "did we encounter x in the input already?". In a permutation, no element can appear more than once. Therefore, you can just scan the list `l` once, and for each element check if it is in the correct range and if we haven't seen it already, in which case you update `seen` and proceed. Otherwise the list can't be a permutation.

```
In [1]: isperm([0])
Out[1]: True

In [2]: isperm([2, 0, 1])
Out[2]: True

In [3]: isperm([5, 1, 0, 3, 2, 4])
Out[3]: True

In [4]: isperm([1, 2, 1, 0])
Out[4]: False

In [5]: isperm([1, -1]) # WE WANT False HERE, NOT AN ERROR!
```

```
Out[5]: False

In [6]: isperm([1, 2, 4, 0]) # WE WANT False HERE, NOT AN ERROR!
Out[6]: False
```

Exercise 3 - List Partitioning

IMPORTANT: For this exercise, you will need to create an empty list with the syntax `[]` and progressively `append` elements at its end. To do that, use the `append` method like in the following example:

```
r = [] # list is created empty
r.append(3) # list is now [3]
r.append(1) # list is now [3, 1]
r.append(9) # list is now [3, 1, 9]
```

(You will actually need to use `append` also to create a list of sub-lists.)

Write a function called `listpart` that takes two arguments, a list of positive integers `l` and a positive integer `k` . The function should divide `l` into chunks, such that the sum of the numbers inside each chunk does not exceed `k`.

For example, given `l = [3, 4, 2, 6, 1]` and `k=8` , the first chunk should be `[3, 4]` (beacuse its sum is 7 , and the next element is 2 which would make it go beyond 8); the second chunk should be `[2, 6]` (because its sum is 8 and the next element 1 would not fit); the third and last chunk should be `[1]` because that's all there is left. So the function should return `out=[[3, 4], [2, 6], [1]]` in that case.

Tips:

1. Have the function read `l` one element at a time. Also, build the sub-lists one at a time and put the elements of `l` inside it as you go. For any new element, evaluate whether it would fit into the sub-list that you're building or not, and thus decide whether that sub-list is over and it's time to start a new one with that element.
2. Notice that, once you're done scanning the list, you may or may not be left with a partial (i.e. non-empty) sub-list that you had been building: if so, remember to put it in the output.

Additional details:

- If the original list contains an element `x` that is larger than `k` , it cannot possibly fit: ignore it and go to the next one.
- Do not store empty sub-lists in the output.
- You don't need to perform any check on the arguments or their content.
- As usual you can't use any method or builtin that is not explicitly allowed.

Examples of the desired behavior:

```
In [1]: listpart([4, 3, 2], 7) # 4+3<=7 BUT 4+3+2>7; THEN 2<=7
Out[1]: [[4, 3], [2]]

In [2]: listpart([4, 3, 2], 5) # 4<=5 BUT 4+3>5; THEN 3+2<=5
Out[2]: [[4], [3, 2]]

In [3]: listpart([4, 3, 2], 10) # 4+3+2<=10
Out[3]: [[4, 3, 2]]
```