National University of Singapore
School of Computing
CS1010S: Programming Methodology

Extra Practice 5 Solutions

Question 1

Using **`range`**, **`map`** and **`filter`**, define the following functions to get the following outputs. (You can use other techniques as well)

(a) Define a function **`f1`** that takes in no inputs, and returns `(6, 5, 4, 3, 2, 1, 0)`.
Solution:

```
def f1():
    return tuple(range(6, -1, -1))

# Alternate solution
def f1():
    return tuple(map(lambda x: 6-x, range(7)))
```

(b) Define a function **`f2`** that takes in a tuple as an input, and transforms it in the following manner.

```
>>> f2((1, 2, 3, 4, 5, 6, 7, 8))
(1.5, 3.5, 5.5, 7.5)
>>> f2(tuple(range(2, 17, 3)))
(5.5, 11.5)
```

Solution:

```
def f2(tup):
    return tuple(map(lambda x: x+0.5, filter(lambda x: x%2 == 1, tup)))
```

(c) Define a function **`f3`** that takes in two inputs - a tuple consisting of strings, and a word, and returns a new tuple that consists of strings in the original tuple that are an **anagram** of the word. An anagram is a word that is obtained when another word is scrambled up. (for example, "dormitory" is an anagram of "dirtyroom")
Sample Tests:

```
>>> words = ("toilet", "dirtyroom", "dirtyyroom", "ormitoryd")
>>> f3(words, "dormitory")
('dirtyroom', 'ormitoryd')
>>> f3(words, "dirtyrooom")
()
```

Solution:

```
def f3(words, w):
    return tuple(filter(lambda x: sorted(x) == sorted(w), words))
```

Question 2

You designed a robot that is supposed to move around the house and mop the floor as it does. You want it to be highly efficient such that it does not ever move in a loop (i.e. its path of motion will never intersect). Let's assume that it can only move north, south, east and west (indicated by "N", "S", "E", "W").

Using tuples only, implement a function, `check_loop` that takes in a tuple consisting of directions it moves and returns `True` if there exists a loop and `False` otherwise.

Sample Tests:

```
>>> check_loop(("N", "E", "E", "S", "W", "W", "S"))
True # there is a loop after the first 6 moves
>>> check_loop(tuple("NESESEENWNWS"))
True
```

Solution:

```
def check_loop(moves):
    visited = ((0, 0),) # store all the visited coordinates
    for move in moves:
        if move == "N":
            visited += ((visited[-1][0], visited[-1][1] + 1),)
        elif move == "E":
            visited += ((visited[-1][0] + 1, visited[-1][1]),)
        elif move == "S":
            visited += ((visited[-1][0], visited[-1][1] - 1),)
        elif move == "W":
            visited += ((visited[-1][0] - 1, visited[-1][1]),)
        if visited[-1] in visited[:-1]:
            return True
    return False
```

Question 3

In the army, there exists a chain of command such that work will be allocated from a superior to someone else directly under his chain of command until the person with the lowest rank has to do all the work. We will assume that there is strictly a one-one pairing where every superior will only have one person directly under his command, and there are multiple superiors in the army. Given a tuple that indicates this hierarchy such as

```
chain = (
    ("Civilian", "Major1"), ("Corporal1", "Recruit1"),
    ("Corporal2", "Private"), ("Major1", "Officer1"),
    ("Major2", "Officer2"), ("Officer1", "Sergeant1"),
    ("Sergeant1", "Corporal2"), ("Officer2", "CFC")
    )
```

To understand this tuple, the rank on the left in each tuple is **directly higher-ranked** than the rank on the right. There are multiple hierarchies that do not intersect.

Define a function **`taiji`** that takes in the chain of command as a tuple, and a rank, and returns the lowest ranking person under his command (directly or indirectly) so that he knows who the work will eventually go to. **Sample Tests:**

```
>>> taiji(chain, "Civilian")
'Private'
>>> taiji(chain, "Major2")
'CFC'
>>> taiji(chain, "CFC")
'CFC'
```

(a) Use recursion to solve it.
Solution:

```
def taiji(chain, rank):
    if rank not in tuple(map(lambda x: x[0], chain)):
        return rank
    for hi, lo in chain: # unpacking a tuple
        if hi == rank:
            return taiji(chain, lo)
```

(b) Use iteration to solve it.
Solution:

```
def taiji(chain, rank):
    while rank in tuple(map(lambda x: x[0], chain)):
        for hi, lo in chain:
            if hi == rank:
                rank = lo
                break
    return rank

# Alternate solution
def taiji(chain, rank):
    found = True # boolean flag
    while found:
        found = False
        for hi, lo in chain:
            if hi == rank:
                rank = lo
                found = True
                break
    return rank
```

Question 4

(CS1010S AY20/21 Sem 1 Mock Midterm)

Flappy Bird is a mobile game, developed by Vietnam-based developer Dong Nguyen and published by .GEARS Studios, a small, independent game developer also based in Vietnam. The game has a side-scrolling format and the player controls a bird, attempting to fly between the gaps of vertical green pipes.

In this problem, you will solve some problems based loosely on the game.

Suppose that the bird is originally at ground level (`height = 0`) and it needs to fly through a series of gaps at various heights. In each step, the bird can move either up one level, down one level, or stay at the same level.

The function **`can_clear(gaps)`** takes in the tuple representing the heights of each gap and returns **`True`** if the bird can successfully clear pipes, or **`False`** otherwise.

Sample Execution:

```
>>> can_clear((1, 2, 1, 1))
True
>>> can_clear((2, 1, 1))
False
>>> can_clear((1, 0, 1, 1, 0))
True
>>> can_clear((5,))
False
>>> can_clear((0, 1, 1, 2, 3, 4, 5))
True
>>> can_clear(())
True
```

(a) Provide an iterative implementation of **`can_clear`**.
Solution:

```
def can_clear(gaps):
    curr_height = 0
    for height in gaps:
        if abs(height - curr_height) > 1:
            return False
        curr_height = height
    return True
```

(b) What is the order of growth in time and in space for the function you wrote in Part (a)? Briefly explain your answer.
Solution:
Time: O(n), where n is the number of gaps in the tuple, as the for-loop will make a single pass through the tuple, checking each element for O(1) time each.

Space: O(1), constant space since there is a fixed number of variables and each variable only stores an integer.

(c) Provide a recursive implementation of **`can_clear`**.
Solution:

```
def can_clear(gaps):
    if len(gaps) == 0:
        return True
    elif len(gaps) == 1:
        return abs(gaps) <= 1
```

```
    elif abs(gaps[-1] - gaps[-2]) <= 1:
        return can_clear(gaps[:-1])
    else:
        return False
```

(d) What is the order of growth in time and in space for the function you wrote in Part (c)? Briefly explain your answer.
Solution:
*Time: O(n**2), where n is the length of the tuple, since there is a total of n recursive calls, and each call to can_clear takes another O(n).*

*Space: O(n**2), there is a total of n recursive calls, and each call will take up space on the stack due to a new tuple slice being created.*

(e) Flying is not easy. Flying up costs 2 units of energy per level. Staying at the same level costs 1 unit of energy. Luckily, flying down costs no energy per level.

Implement the function **`energy_cost(gaps)`** that takes in a tuple of gap heights, and returns the total number of units of energy that flying through the gaps will cost.
Sample Execution:

```
>>> energy_cost((1, 2, 3))          # costs 2+2+2 = 6
6
>>> energy_cost((4,))               # costs 8
8
>>> energy_cost((1, 4, 4, 4, 0))    # costs 2+6+1+1+0 = 10
10
```

Solution:

```
def energy_cost(gaps):
    gaps = (0,) + gaps
    cost = 0
    for i in range(len(gaps)-1):
        dh = gaps[i+1] - gaps[i]
        if dh > 0:
            cost += 2 * dh
        elif dh == 0:
            cost += 1
    return cost
```

(f) Next, suppose that the bird can still fly up or down only one level in each step, but it now has a limited amount of energy. It takes 2 units of energy to fly up by one level, takes 1 unit of energy to stay at the same level, and takes no energy to fly down by one level. The bird will fail to clear a series of gaps if it is unable to reach the gap height or it does not have enough energy to fly through the gaps.

Implement the function **`enough_energy_to_clear(energy, gaps)`** that takes in an initial amount of energy (**int**) and a tuple of gap heights, and returns **`True`** if the bird can successfully clear the pipes, or **`False`** otherwise.
Sample Execution:

```
>>> enough_energy_to_clear(6, (1,2,3))
```

```
True
>>> enough_energy_to_clear(10, (1,2,3))
True
>>> enough_energy_to_clear(5, (1,2,3))      # not enough energy
False
>>> enough_energy_to_clear(6, (1,2,3,0))
True
>>> enough_energy_to_clear(10, (1,2,3,4,4,4,3,2,1,0))
True
>>> enough_energy_to_clear(10, (1,4,4,4,0)) # gap difference too big
False
>>> enough_energy_to_clear(0, ())
True
```

Solution:

```
def enough_energy_to_clear(energy, gaps):
    return can_clear(gaps) and energy >= energy_cost(gaps)
```

Solution compiled by Russell Saerang.