CSC110 Lecture 13: More with For Loops

Print this handout

Exercise 1: Looping with indexes

1. Consider the following function, which we studied last class.

```
def all_fluffy(s: str) -> bool:
    """Return whether every character in s is fluffy.

    Fluffy characters are those that appear in the word 'fluffy'.

    >>> all_fluffy('ffffffuy')
    True
    >>> all_fluffy('abcfluffy')
    False
    """
    for character in s:
        if character not in 'fluffy':
            return False

    return True
```

In the space below, rewrite the body of the function so that it uses an *index-based for loop* instead of an element-based for loop.

```
def all_fluffy(s: str) -> bool:
    """Return whether every character in s is fluffy.

    Fluffy characters are those that appear in the word 'fluffy'.

    >>> all_fluffy('ffffffuy')
    True
    >>> all_fluffy('abcfluffy')
    False
    """
```

2. Implement each of the following functions using an index-based for loop.

```
def is_sorted(lst: list[int]) -> bool:
    """Return whether lst is sorted.

    A list L is sorted when for every pair of *adjacent* elements
    x and y in L, x <= y.

    lists of length < 2 are always sorted.

    >>> is_sorted([1, 5, 7, 100])
    True
    >>> is_sorted([1, 2, 1, 2, 1])
    False
    """
```

```
def inner_product(nums1: list[float], nums2: list[float]) -> float:
    """Return the inner product of nums1 and nums2.

    The inner product of two lists is the sum of the products of the
    corresponding elements of each list:

        sum(nums1[i] * nums2[i] for i in range(0, len(nums1)))

    Preconditions:
        - len(nums1) == len(nums2)

    >>> inner_product([1.0, 2.0, 3.0], [0.5, 2.5, 0.0])
    5.5
    """
```

```
def stretch_string(s: str, stretch_factors: list[int]) -> str:
    """Return a string consisting of the characters in s, each repeated
    a given number of times.

    Each character in s is repeated n times, where n is the int at the
    corresponding index in stretch_factors.
    For example,
        - s[0] is repeated stretch_factors[0] times
        - s[1] is repeated stretch_factors[1] times
        - etc.

    Preconditions:
        - len(s) == len(stretch_factors)
        - all(factor >= 0 for factor in stretch_factors))

    >>> stretch_string('David', [2, 4, 3, 1, 11])
    'DDaaaavvvvid'
    >>> stretch_string('echo', [0, 0, 1, 5])
    'hooooo'
    """
```

Exercise 2: Nested loops

1. Implement this function:

```
def total_mice(dict_of_cats: dict[str, list[str]]) -> int:
    """Return the number of mice stored in the given cat dictionary.

    dict_of_cats is a dictionary here:
        - Each key is the name of a cat
        - Each corresponding value is a list of items that the cat owns.
          An item is a 'mouse' when it contains the string 'mouse'.
          (You can use the 'in' operator to check whether one string is
          in another.)

    >>> total_mice({'Romeo': ['mouse 1', 'my fav mouse', 'flower'],
    ...             'Juliet': ['sock', 'mouse for tonight']})
    2
    >>> total_mice({'Anya': ['chocolate', 'toy'], 'Hitsey': []})
    0

    HINT: remember that when iterating over a dictionary, the loop
    variable refers to the *key* of the dictionary. Use key lookup
    (i.e., `dict_of_cats[key]`) to access the corresponding value.
    """
```

2. Complete the following *loop accumulation table* to trace the sample function call

```
total_mice({
    'Romeo': ['mouse', 'my fav mouse', 'flower'],
    'Juliet': ['sock', 'dinner mouse']
})
```

We've started it for you to save some time.

Outer Loop Iteration	Outer Loop Variable	Inner Loop Iteration	Inner Loop Variable	Accumulator
0	N/A	N/A	N/A	
1	`'Romeo'`	0	N/A	
1	`'Romeo'`	1		
1	`'Romeo'`	2		
1	`'Romeo'`	3		
2	`'Juliet'`	0	N/A	
2	`'Juliet'`	1		
2	`'Juliet'`	2		

3. Implement this function using a nested loop.

```
import math


def max_average(lists_of_numbers: list[list[float]]) -> float:
    """Return the largest average of the given lists_of_numbers.

    Preconditions:
        - lists_of_numbers != []
        - all(numbers != [] for numbers in lists_of_numbers)

    >>> max_average([[1.0, 3.4], [3.5, 4.0, -2.5]])
    3.2
    """
    # ACCUMULATOR max_so_far: keep track of the maximum average of the lists
    # visited so far. We initialize to negative infinity so that any
    # computed average will be greater than the starting value.
    # (i.e., for all floats x, x > -math.inf)
    max_so_far = -math.inf
```

Additional exercises

1. Write a function that takes a string `s` and returns whether `s` is a palindrome. A *palindrome* is a string consists of the same sequence of characters in left-to-right order as right-to-left order. `'davad'` is a palindrome, and `'david'` is not.

   Hint: for a string `s` of length `n`, being a palindrome means:

   - `s[0] == s[n - 1]`
   - `s[1] == s[n - 2]`
   - `s[2] == s[n - 3]`
   - etc.

2. Write a function that takes two lists of integers, which have the same length and are non-empty, and returns the greatest absolute difference between the numbers at corresponding positions in the lists.

3. Write a new version of **`max_average`** that does the same thing, except it returns the *list* with the highest average rather than the highest average.

   Hint: use two accumulator variables, one to keep track of the highest average itself, and another to keep track of the list with the highest average.

4. One powerful use of nested loops is to generate all *possible pairs* of elements from a collection (or two different collection). Review this idea from Section 5.7 in the Course Notes, and then use a nested loop to implement the following function.

```
def can_pay_with_two_coins(denoms: set[int], amount: int) -> bool:
    """Return whether the given amount is the sum of two distinct numbers
    from denoms.

    >>> can_pay_with_two_coins({1, 5, 10, 25}, 35)
    True
    >>> can_pay_with_two_coins({1, 5, 10, 25}, 12)
    False
    """
```

5. Re-implement all of the functions on this worksheet using comprehensions. In some cases, you might need to define some separate helper functions.