Lab 03: Lists and Dictionaries

AUTHOR
Christian Osendorfer

Implement in Python

Unless otherwise stated, don't use any modules that implement a solution to the questions asked. Come up with more doctests and compare these with those from other students.

> **Note**
>
> For lists and dictionaries, understand the difference between `==` and `is`.

Lists

```
def max_product(s):
    """
    Let lst be a list of integers. Return the maximum product that can
    be formed using non-consecutive elements of lst.
    >>> max_product([10,3,1,9,2]) # 10 * 9
    90
    >>> max_product([5,10,5,10,5]) # 5 * 5 * 5
    125
    >>> max_product([])
    1
    """
    # YOUR CODE HERE
```

```
def remove_odd_indices(lst, odd, weight):
    """
    lst is a list, odd is a boolean. Return a new list with elements from ls
    removed at certain indices: If odd is True, remove elements at odd indic
    otherwise remove even indexed elements. The remaining elements are
    multiplied by weight.
    >>> s = [1, 2, 3, 4]
    >>> t = remove_odd_indices(s, True, 1)
    >>> s
    [1, 2, 3, 4]
    >>> t
    [1, 3]
    >>> l = [5, 6, 7, 8]
    >>> m = remove_odd_indices(l, False, 4)
    >>> m
    [24, 32]
```

```
    """
    # YOUR CODE HERE
```

```
def merge(lst1, lst2):
    """Merges two sorted lists, lst1 and lst2. The new list contains all
    elements in sorted order.
    >>> merge([1, 3, 5], [2, 4, 6])
    [1, 2, 3, 4, 5, 6]
    >>> merge([], [2, 4, 6])
    [2, 4, 6]
    >>> merge([1, 2, 3], [])
    [1, 2, 3]
    >>> merge([5, 7], [2, 4, 6])
    [2, 4, 5, 6, 7]
    """
    # YOUR CODE HERE
```

```
def alternatingSum(lst):
    """
    Return the alternating sum of the integer or float elements
    in the list lst. An alternating sum of a sequence is a sum
    where the sign alternates from positive to negative or
    vice versa.
    >>> alternatingSum([5, 3, 8, 4])
    6
    >>> alternatingSum([])
    0
    """
    # YOUR CODE HERE.
```

```
def is_anagram(s1, s2):
    """
    Decide whether a string s1 and a string s2 are anagrams.
    Use only lists in your solution.
    >>> is_anagram("", "")
    True
    >>> is_anagram("abCdabCd", "abcdabcd")
    True
    >>> is_anagram("abcdaabcd", "aabbcddcb")
    False
    """
    # YOUR CODE HERE
```

```
def rotateStringLeft(s, n)
    """
    The function takes a string s and a possibly-negative integer n.
    If n is non-negative, the function returns the string s rotated
    n places to the left. If n is negative, the function returns
    the string s rotated |n| places to the right.
    >>> rotateStringLeft('abcd',  1)
    'bcda'
```

```
    >>> rotateStringLeft('abcd', -1)
    'dabc'
    """
    # YOUR CODE HERE
```

```
def caesar_cipher(message, shift)
    """
    A Caesar Cipher is a simple cipher that works by shifting
    each letter in the given message by a certain number. For
    example, if we shift the message "We Attack At Dawn" by 1
    letter, it becomes "Xf Buubdl Bu Ebxo".

    ceasar_cipher(message, shift) shifts the given message by
    'shift' many letters. You are guaranteed that 'message' is a
    string, and that 'shift' is an integer between -25 and 25.
    Capital letters should stay capital and lowercase letters
    should stay lowercase, and non-letter characters should
    not be changed. "Z" wraps around to "A".
    >>> caesar_cipher("We Attack At Dawn", 1)
    "Xf Buubdl Bu Ebxo"
    >>> caesar_cipher("zodiac", -2)
    "xmbgya"
    """
    # YOUR CODE HERE
```

Dictionaries

```
def most_frequent(L):
    """
    Return the most frequent element in a list L, resolving
    ties arbitrarily. L only has elements of scalar types.
    >>> most_frequent([2,5,3,4,6,4,2,4,5])
    4
    >>> most_frequent([2,3,4,3,5,3,6,3,7])
    3
    >>> most_frequent([42])
    42
    >>> most_frequent([])
    None
    """
    # YOUR CODE HERE
```

```
def is_anagram(s1, s2):
    """
    Decide whether a string s1 and a string s2 are anagrams.
    Use only dictionaries in your solution.
    >>> is_anagram("", "")
    True
    >>> is_anagram("abCdabCd", "abcdabcd")
    True
    >>> is_anagram("abcdaabcd", "aabbcddcb")
```

```
    False
    """
    # YOUR CODE HERE
```

```
def most_popular_names():
    """
    Using the information in the file 'popular_names.txt' (see
    files attached to this lab), build two dictionaries, one for
    male names, and one for female names, and return the two
    dictionaries.

    Come up with two reasonable doctests!
    """
    # YOUR CODE HERE
```

```
def word_count():
    """
    Find a text file containing the complete works of
    William Shapespeare attached to this lab (see moodle course website).

    * Find the 20 most common words.
    * How many unique words are used?
    * How many words are used at least 5 times?
    * Write the 200 most common words, and their counts,
    to a file 'top200.txt'
    """
    # YOUR CODE HERE
```

List and Dictionary Comprehensions

- Using a single (possibly nested) list comprehension, compute the set of prime numbers from 0 to 99 (inclusive). Your list comprehension should return a list of lists, where the \(i\)-th list is the list of prime numbers in `[i*10, (i*10)+9]`. The result should look something like: `[[2, 3, 5, 7], [11, 13, 17, 19], ...]`. Use the known function `is_prime(x)` that returns `True` if `x` is prime and `False` otherwise.
- Use a single dictionary comprehension that maps each element of a list of items to the number of times it appears in that list, but only if it appears more than 2 times. For example, if items is: `["A", "A", "A", "B", "B", "C", "C", "C", "C", "D"]`, then the result will be: `{"A": 3, "C": 4}`.

Compute without a computer

Determine the results of the provided python code snippets by evaluating them without using a computer. If print is used, also provide the strings (order matters!) that are printed on the console.

> **Note**
>
> **Short-circuiting** happens when an operator reaches an operand that allows them to make a conclusion about the expression. For example, `and` will short-circuit as soon as it reaches the first `False` value because it then

knows that not all the values are `True`.

```
1/0
```

```
True or 1/0
```

```
False and 1/0
```

```
def ct1(L):
    a = []
    b = list()
    c = dict()
    d = c
    for item in L:
        if item in a:
            b.append(item)
        a.append(item)
    c[L[0]] = a
    c[L[1]] = b
    c[L[2]] = len(a)
    c[L[3]] = len(b)

    x = (d == c)
    c[L[4]] = x
return c

print(ct1(['n', 5, True, True, (3,5), 'n']))
```

```
import copy
def ct1(x):
    y = x
    z = copy.copy(x)
    x.append(12)
    y[0] += 3
    x = x[2:]
    print(f'x = {x}')
    x[-1] = x[1] + x[0]
    z.pop(0)
return x[-1], y[-1], z[-1]

L = [42, 30, 100]
print(f'ct = {ct1(L)}') # These are f-strings!
print(f'L = {L}')
```

```
def make2d(rows, cols):
    """
    Generate a 2-dimensional list (also known as a *matrix* in math).
    """
    # Check the slides for the '*' operation on lists!
    # Is there a difference when '*' is applied to a list of scalar
```

```
        # types or to a list of complex types (that is, objects).
        return [ [0] * rows] * cols

    my_matrix = make2d(2, 3)
    my_matrix[0][0] = 42
    print(my_matrix)
    # Explain this output.
```