Intro Ex6 – One Code line Programs

Submission deadline

Thursday, December 5th, 20:55

Objectives

The purpose of this homework assignment is to exercise some advanced concepts of python programming, specifically list and dictionary comprehensions.

This assignment contains multiple tasks. Start working on it early! (Otherwise, you risk missing the assignment deadline.)

Your code must satisfy the Coding Style Guidelines posted on the website.

Note that:

1. In order to get a **perfect grade** you must solve each task using a **single line** of code (unless stated otherwise)!
    a. **We will reduce 2 points** (in the **manual grading** phase) on tasks containing solutions with more than one code line.
    b. We will also reduce points (after the 2 points reduction) for inefficient solutions.
2. Import lines DO NOT COUNT as code lines. See the following example.
3. In each task, the name of the function you write must equal the name of the task.
4. Unless explicitly stated otherwise, you don't need to check that the input is correct.
5. Try using Google to find python functions that are useful in solving this assignment. Note that some of them were not discussed in class.
6. You are **not allowed** to use lambda functions in this exercise (except for task 9).
7. Make sure that your code is readable and that functions are documented using a docstring. Making your code readable and simple is even more important when solving problems using one code line.

Here is an example:

1. Task: factorial
   Description: Given a number ***N***, return ***N*** factorial (i.e., ***N***!).
   For example, calling factorial(4) returns 24 (1*2*3*4).
   Input: An integer greater than 1.
   Return: An integer.

There are several ways to implement this task:
Option 1: (This option gets full credit!)
```
from operator import mul #this line does not count!
from functools import reduce #this line does not count as well!
def factorial(N):
    return reduce(mul, range(1,N+1), 1)
```

Option 2: (This option has more than one code line. It doesn't get full credit.)

```
def factorial(N):
    result = 1
    for num in range(1,N+1):
        result *= num
    return result
```

Option 3: (This code uses a lambda function. **As a result it gets no credit at all**)

```
from functools import reduce #this line does not count!
def factorial(N):
    return reduce(lambda x,y: x*y, [1]+list( range(1,N+1) ) )
```

All three functions produce the same output, so they will pass the auto-testers. However, **Only the first option** gets a **perfect** grade. Option 2 gets 2 points reduction in the manual grading.

Handling long lines:

If your code line is longer than 79 characters (including the 4 spaces needed for indentation), you must break it into two or more editor lines. Here is an example. The code line

```
[sum([row[j] for row in matrix]) for j in range(4)]
```

can be written as:

```
[sum([row[j] for row in matrix]) \
     for j in range(4)]
```

The second alternative is still considered one line of code. It gets full credit.

And now to the tasks themselves:

1. Task: **is_two_palindrome**
   Description: The function tests if the given string is a "two palindrome" or not. A string is a palindrome if reversing it does not change it.
   We say that a string is a "two palindrome" if:
   i) It is of even length and splitting it in the middle results with two, possibly different, palindromes of equal length (e.g., "ababdb" is a two-palindrome).
   ii) It is of odd length, and it contains two, possibly different, palindromes separated by some character (e.g., "bbb$121" is a two-palindrome).
   For example, `is_two_palindrome("aba")` and `is_two_palindrome("zwzkata")` return *True*, `is_two_palindrome("avva")` returns *False*.
   Input: String of any length. The string may include punctuation, spaces, and other symbols.
   Remarks:
   a. Upper-case and lower-case letters are different: the string "Aba" is not a palindrome!
   b. You may use at most three code lines for this task.
   Returns: Boolean.

2. Task: **uni_sort**
   Description: The function combines two unsorted lists of integers into one sorted list. The output list must be sorted in ascending order (from smaller to larger). Each distinct integer in the input lists must appear exactly once in the output list. For example, calling `uni_sort([1,2,3],[3,5,-1,5,2,7])` returns the list [-1,1,2,3,5,7].
   Input: Two lists of integers.
   Remarks:
      a. You are not allowed to use the **set** or **dictionary** containers.
      b. The two input lists may contain duplicate integers.
      c. You are allowed to use up to 3 code lines to get full credit.
   Returns: A sorted list of integers. Every integer in the two input lists must appear exactly once in the output list.

3. Task: **dot_product**
   Description: The function returns the dot product of two vectors, represented as lists. Given two vectors, $A = (a_0, a_1, \ldots, a_{n-1})$ and $B = (b_0, b_1, \ldots, b_{n-1})$, their dot product is $A \cdot B = \sum_{i=0}^{n-1} a_i \cdot b_i$. For example, calling dot_product([1,2,3] , [2,0,0]) returns **2**.
   Input: Two lists (representing vectors) of integers.
   Remarks:
      a. If the lists have different lengths, say *m* and *n*, then the dot product is defined as: $\sum_{i=1}^{\min(m,n)} a_i \cdot b_i$. For example, calling `dot_product([1,2,3], [2, 0, 0])` is the same as calling `dot_product([1, 2, 3] , [2])`.
      b. There is a solution that uses one code line. You are allowed to use up to 3 code lines to get full credit.
   Returns: An integer – the dot product of the two input vectors.

4. Task: **list_intersection**
   Description: The function gets as input two lists of integers and returns a new list sorted in ascending order. The output list must contain exactly those integers that appear in ***both*** input lists. For example, `list_intersection([1, -1, 4, 4, 5, 1] ,[-1, 1, 3, 4, 3, 4])` returns [-1,1,4].
   Input: Two (unsorted) lists of integers.
   Remarks: You may use every data-structure (e.g., set) for this task.
   Returns: A list of integers sorted in ascending order containing those integers that appear in both input lists.

5. Task: **list_difference**
   Description: The function is similar to *list_intersection*. Given two input lists of integers, the function returns a list sorted in ascending order. The output list must contain and exactly those integers that appear in just ***one*** of the input lists. For example, `list_difference([1, -1, 4, 5, 1] ,[-1, 1, 3, 4, 3, 4])` returns [3,5].
   Input: Two (unsorted) lists of integers.
   Remarks: You may use every data-structure (e.g., set) for this task.
   Returns: A list of integers sorted in ascending order containing those integers that appear in exactly one of the input lists.

6. Task: **random_string**
   Description: The function generates a random string of a given length. For example, calling random_string(4) could return "aaaa" or "avdg" or any other 4 lower-case characters string.
   Input: An integer *N* denoting the length of the output random string.
   Remarks:
      a. The output string must consist of **ascii lower-case letters** only!
   Returns: A random string of length *N*.

7. Task: **word_mapper**
   Description: The function returns a dictionary mapping from the words in the input text to their number appearances. Each word in the input text should appear only **once** in the output dictionary. Upper-case letters must be converted to lower-case letters. The order of words in the dictionary does not matter. For example, given the input text "The cat. is the cAt" a possible output is:
   {"the" : 2, "cat" : 2, "is" : 1} (Another option is: {"is" : 1 , "the" : 2, "cat" : 2}.)
   Input: A string of words separated by whitespace and/or punctuation marks.
   Remarks:
      a. You are not allowed to use the **Counter** container.
      b. The input string may contain multiple lines. For example, "aaa\na" is a text with two lines, one containing the word "aaa" and other containing the word "a". The line-feed character `\n' is considered whitespace.
      c. The order of the words in the dictionary does not matter. For example, {"a" : 1, "b" : 2} and {"b" : 2 , "a" : 1} are considered to be the same output.
      d. You will earn up to 2 bonus points for solving this task using at most 2 code lines. You will earn full credit for using up to 3 code lines.
      e. Words may contain punctuations (e.g., "I ate an… apple!"), as in the example above, these should be ignored (they count as a space character).
      f. Words may contain numbers. e.g., given the text "12 monkeys" a possible output is {12 : 1, "monkeys" : 1}.
   Returns: A dictionary containing a mapping between words (as keys) and the number of times they appear in the original input string (as value).

8. Task: **gimme_a_value**
   Description: In this task you must create a generator that gets two inputs, a function f and a starting point x_0. The generator computes a sequence of values x_1 x_2 … of the function f. For all $i \geq 2$ The i-th call to the generator returns the value $x_i = f(x_{i-1})$. The first call returns x_0 (the initial value). The second call returns $x_1 = f(x_0)$. The Third call returns $x_2 = f(x_1)$. This generator must run endlessly. (How can this happen?)
   Input: A function, *f*, and an initial value x_0.
   Remarks:
      a. You should use the keyword **yield** in your generator, and not a generator expression.
      b. Check the slides of the Tirgul if you need a reminder.
      c. You are allowed to use up to 10 code lines to get full credit.
      d. You may use recursion if it helps you, although it is not mandatory or recommended.
   Returns: The next element in the sequence generated by applying f to the previous element. The first returned value is x_0.

9. Task: **gimme_a_genexp (Not Mandatory! This is a bonus question worth 5 points)**
   Description: The input is a function *f* and an initial value x_0. The output is a generator expression that executes the same task as in **gimme_a_value(*f*, x_0)**. See the example below.
   Input: A function, *f*, and an initial value, x_0.
   Remarks:
   a. You should **not** use the keyword **yield** as you should return a generator expression.
   b. Check the slides of the Tirgul if you need a reminder.
   c. To earn bonus points, use must use only one code line! Longer code will get no bonus at all.
   d. You may use recursion if it helps you, although it is not mandatory or recommended.
   e. You may use every module python have to offer (e.g., itertools).
   f. You may use (**only in this task**) lambda functions.
   Returns: A generator expression.

Generators examples (Task 8+9):

Task 8:
Let f be:
```
def sum_range(maxNum):
   return sum( range(maxNum) )
```
and $x_0 = 5$.

The following segment of code:
```
for value in gimme_a_value(sum_range, 5):
   print(value)
   if value > 20:
      break
```
prints:
```
5
10
45
```
Explanation:
The first call to the generator returns the number 5 (our initial number), the second call returns the number 10 =sum_range(5)=(0+1+2+3+4). The third call returns 45=sum_range(10)=(0+1+2+3+…+9). (There won't be a fourth call to the generator because the for loop breaks. If a fourth call happened, it would have returned 990=sum_range(45)=(0+1+…+44).)

For those of you who chose to do task 9:
If **gimme_a_value(sum_range, 5)** is replaced by **gimme_a_genexp(sum_range, 5)**, the output should remain the same.

General remarks
1. **We remind you again – your program must behave exactly as ours.**
2. **The name of the program file must be oneliners.py**
3. **You may want to use the modules:**
   a) random
   b) string

<u>The README file:</u>
You should explain for each task which imports did you use, and most importantly **<u>why</u>**. For example, in our option 1 for the factorial example (from the beginning of the ex) the README should contain:

Task: factorial
I have used the imports mul (from module operator) and reduce (from module functools).
1) mul - Using mul(a,b) on two numbers is the same as using a*b, I have used it because we cannot use lambda functions in our exercise.
2) reduce - ….

<u>Submission and further guidelines:</u>

<u>Creating a tar file</u>

- You must create two files:
    1. oneliners.py (the solution to the assignment)
    2. README (as explained in the course guidelines)
- In order to submit the solution, create a TAR file named ex6.tar. This file must contain only the above two files! Create ex6.tar by invoking the shell command:
  tar cvf ex6.tar oneliners.py README

  (See tirgul 1 slides for details about tars.)

  We recommend that you check your TAR file at follows. Copy the TAR file to another directory. Extract its contents by executing the command: **tar xvf ex6.tar.** Verify that **oneliners.py** and **README** indeed appear in this other directory.

<u>Submitting the tar file</u>

- Submit the file **ex6.tar** via the "Upload File" link on the course homepage, under the ex6 link.
- Note that submission requires you to be registered as a student in the course and logged in.
- Shortly after you upload the file you should receive an email to your university mailbox. The email contains the automatic test results for your submission, and the pdf file that the manual grader gets.

<u>Running the tests yourself</u>

- You can run the automatic testers on your code before submitting using one of the following two procedures.
- First option: Place your tar file in an empty directory. Switch to that directory using the shell and type:
  ~intro2cs/bin/testers/ex6 ex6.tar
- Second option: Download the file **ex6testing.tar** from here. Extract the file's contents (using **tar –xjv ex6testing.tar.bz2**). Follow the instructions in the file named TESTING.