CS 1358 Introduction to Programming in Python

Spring Semester 2023

Prof. Pai H. Chou

Self-Check 9

Due Date: --

Answer the following questions to check your understanding of your material. Expect the same kind of questions to show up on your tests.

## 1. Definitions and Short Answers - functions

1. What is the equivalent **lambda expression** that computes the same as the following named function?
   1. def Double(n):  
       return n + n
   2. def Bigger(a, b):  
       return a if a > b else b
2. What lambda expression can you pass to a list's sort method's optional key **plug-in function** if you want to sort a list of strings **by string length**? For example  
   >>> L = ['an', 'apple', 'a', 'day', 'keeps', 'the', 'doctor', 'away']  
   >>> L.sort(key=lambda )  
   >>> L  
   ['a', 'an', 'day', 'the', 'away', 'apple', 'keeps', 'doctor']  
     
   which orders the strings from shortest to the longest. Fill in the yellow blank above.
3. If you want to sort a list of strings **primarily by length** and **secondarily alphabetically** (case-sensitive), what lambda would you pass to the key parameter of the list's sort method? Fill in the yellow blank below.  
   >>> L = ['a', 'glass', 'of', 'water', 'is', 'empty', 'or', 'full']  
   >>> L.sort(key=lambda )  
   >>> L  
   ['a', 'is', 'of', 'or', 'full', 'empty', 'glass', 'water']
4. Which of the following can properly sort a list of month names by month order, and why or why not? Assuming the following global symbols have been defined.  
   ML = ['Jan', 'Feb', 'Mar', 'Apr', 'May', 'Jun', 'Jul', 'Aug', 'Sep', 'Oct', 'Nov', 'Dec']  
   MD = {'Jan':1, 'Feb':2, 'Mar':3, 'Apr':4, 'May':5, 'Jun':6, 'Jul':7, 'Aug':8, 'Sep':9, 'Oct':10, 'Nov':11, 'Dec':12}

L = [ 'Apr', 'May', 'Nov', 'Mar', 'Jan', 'Feb', 'Oct','Jun', 'Jul', 'Aug', 'Sep', 'Dec']

* 1. L.sort(key=ML)
  2. L.sort(key=MD)
  3. L.sort(key=lambda x: ML[x])
  4. L.sort(key=lambda x: MD[x])
  5. L.sort(key=lambda x: ML.index(x))
  6. L.sort(key=lambda x: MD.index(x))

1. If chr(97) evaluates to 'a', then what is the value of  
   list(map(chr, [97, 98, 99, 100, 101]))  
   ?
2. What is the value of  
   list(map(max, [1, 7, 2, 8], [5, 6, 3, 0]))  
   ?
3. How do you use the built-in function zip to convert lists [1, 7, 2, 8] and [5, 6, 3, 0] into a list of tuples, as in  
   [(1, 5), (7, 6), (2, 3), (8, 0)]  
   ?
4. How to you write the equivalent **list-comprehension** version of   
   list(map(max, [1, 7, 2, 8], [5, 6, 3, 0]))  
   ?
5. Suppose you want to do  
   list(map(lambda x, y: x+y, [1, 7, 2, 8], [5, 6, 3, 0]))  
   but **replace** the lambda expression (underlined above) with **an existing function** that does the same. What can you use instead?  
   (Hint: **import** from the **operator** module)
6. If you want to read and print lines from a file but skip all blank lines using the following code template  
    1 fh = open('myfile')  
    2 for line in filter(lambda , fh.readlines()):  
    3 print(line, end='') # no need to print extra newline  
    4 fh.close()  
   What should you put as the lambda expression above? Note that a blank line consists of a single newline character.
7. In the stack interpreter example, several versions of the interpreter are given.  
   The if-elif version looks like this:  
     
    1 def StackInterpreter():  
    2 L = []  
    3 while True:  
    4 line = input('command? ')  
    5 words = line.split()  
    6 if len(words) == 0:  
    7 pass  
    8 elif words[0] == 'show':  
    9 print(L)  
   10 elif words[0] == 'push':  
   11 L.extend(words[1:])  
   19 elif words[0] == 'pop':  
   20 print(L.pop())  
   21 elif words[0] == 'quit':  
   22 break  
   23 else:  
   24 print('unknown command')  
   How can lines 8-24 be replaced with a check for quit followed by using the command word (i.e., words[0]) to look up and execute the corresponding action? That is,  
    8' if words[0] == 'quit':  
    9' break  
   10' D = {'show': \_\_\_\_,  
   11' 'push': \_\_\_\_,  
   12' 'pop': \_\_\_\_,  
   13' }  
   14' f = D.get(words[0], \_\_\_\_\_\_)  
   15' f()  
   1. on line 10' (i.e., revised line 10), what should go into the blank? Will it work if you fill in the blank on line 10' with print(L)? Why or why not?
   2. on line 11', what should go into the blank?
   3. on line 12', what should go into the blank?
   4. what does the D.get(key, altval) method do? How would it be rewritten without calling the .get() method?
   5. what goes into the blank on line 14'?
   6. can lines 10'-15' be rewritten without using temporary variables D and f? How?
8. One alternative to lambda in the lookup table above is to use **inner functions**,   
    1 def StackInterpreter():  
    2 L = []  
    3 def show(): # inner function  
    4 print(L)  
    5 def push():  
    6 L.extend(words[1:])  
    7 def pop():  
    8 print(L.pop())  
    9 def unknown():  
   10 print('unknown command')  
   11 D = {'show': show, 'push': push, 'pop': pop }  
   12 while True:  
   13 line = input('command? ')  
   14 ...  
   1. What are the inner functions in this code fragment?
   2. Why would it be preferable to using inner functions in this case (hint: line 11)?
   3. How would the lookup code D.get(words[0], \_\_\_\_\_\_)  
       be written differently from the lambda version? Fill in the blank.
9. How can you add the documentation string (docstring) to the StackInterpreter() function above so that you can do help(StackInterpreter) in interactive mode and get the help text?  
   $ python3 -i stack.py  
   >>> help(StackInterpreter)  
    This is a stack interpreter. The commands are:  
    show -- shows stack content  
    push item1 item2 item3 -- pushes item1,... as str on stack  
    pop -- pops and displays popped data  
    quit -- exit interpreter  
   END  
   >>>
10. Does *Python Style Guide* recommend using **camel case** or **snake case** for function names?
11. What are examples of **recursive data types** in Python? Are the following data types recursive?
    1. int
    2. list
    3. tuple
    4. dict
    5. set
    6. float
    7. bool

1. What is a **recursive function**?
2. What is a **base case** in a recursive function? Should all recursive functions have at least one base case? Why or why not?
3. If you want to count the number of integers in a list that may contain either integers or list of integers and other lists (of integers and other lists…),
   1. Can you use a loop such as follows? If not, for what cases will it fail?  
       1 def count\_ints(L):  
       2 n = 0  
       3 for i in L:  
       4 if type(i) == int:  
       5 n += 1  
       6 return n
   2. Can you use a loop such as follows? If not, for what cases will it fail?   
       1 def count\_ints(L):  
       2 n = 0  
       3 for i in L:  
       4 if type(i) == int:  
       5 n += 1  
       6 elif type(i) == list:  
       7 for j in i:  
       8 n += 1  
       9 return n
   3. Fill in the code below for counting recursively. You may assume types of elements are either int or list. Note that this version of the code is slightly differently from the slide.  
       1 def count\_ints(L):  
       2 if \_\_\_\_\_\_\_\_: # base case  
       3 return 1  
       4 else:  
       5 n = 0  
       6 for i in L:  
       7 n += \_\_\_\_\_\_\_\_  
       8 return n
   4. Rewrite lines 5-8 above to eliminate the for loop and replace it with a combination of sum() and map().
4. Recursion can also replace a loop. Rewrite the count\_int by converting the loop into a recursive call with its own base case (i.e., loop's terminating condition) and another recursive case for "the rest of the loop".  
    1 def rec\_count(L):  
    2 if type(L) == int: # first base case  
    3 return \_\_\_\_  
    4 if \_\_ : # 2nd base (L is list) so, what kind of list?  
    5 return \_\_\_\_  
    6 return rec\_count(\_\_\_\_) + rec\_count(\_\_\_\_)  
    7 # one recursive call for current element, and   
    8 # 2nd recursive call for "the rest of the loop"
5. Explain what the following functions do in terms of what the parameters are (if any) and what the return value is.
   1. os.getcwd()
   2. os.listdir(d)
   3. os.path.isdir(d)
6. To count files recursively, consider the following version of code  
    1 def count\_files(p = '.'):  
    2 import os  
    3 if \_\_\_\_\_: # p is the name of a file, not a directory  
    4 return 1  
    5 dir\_content = \_\_\_\_ # get list of names (files & dir)  
    6 return sum(\_\_\_\_\_\_\_) # sum recursive count of content
   1. What does '.' mean as the default value of parameter p?
   2. How do you call a function from os module to check if a path p is a **file** rather than a **directory** ("folder")? Fill in the blank on line 3.
   3. If p a **path** is a directory, how do you obtain a **list of names** (of files and directories) in p? Fill in the blank on line 5.
   4. How do you recursively count each path of the list so that the counts can be summed? Fill in the blank on line 6.
7. In the recursive-find example,
   1. calling  
      M = [1, 2, [3, [4, 23]  
      rec\_find(M, 23)  
      results in the tuple value (2, 1, 1). What does it mean?
   2. Calling  
      rec\_find(43, 43)  
      results in True. What does it mean?
   3. What would be the result of calling  
      rec\_find([1, 2, 3], [1, 2, 3])  
      ?
   4. What would be the result of calling  
      rec\_find([[1, 2, 3]], [1, 2, 3])  
      ?
8. The source code for the recursive-find function looks like this:  
    1 def rec\_find(L, val):  
    2 if type(L) in {list, tuple}: # look inside L  
    3 for i, v in enumerate(L):  
    4 p = rec\_find(v, val) # recursively find item  
    5 if p == True: # L[i] == val, so we return (i,)  
    6 return (i,)  
    7 if p != False: # L[i] recursively found val,  
    8 return (i,)+p # prepend i to its path p  
    9 return L == val # L not seq or for-loop didn't find  
   1. What is the **condition** **of the base case** in this recursive function?
   2. Is line 9 executed only if type(L) is not in {list, tuple}? Or can it be executed even if type(L) is either list or tuple? If so, describe how line 9 can still be reached after executing lines 3-8?
   3. Can line 9 compare only two ints, or can it be comparing two tuples or two lists?
   4. Line 5 tests  
       5 if p == True:  
      but why can't it be replaced with  
       5 if p:  
      ?
   5. Line 7 tests  
       7 if p != False:  
      but why isn't it redundant with line 5? Doesn't   
      p != False   
      imply  
      p == True  
      ?
9. In the code for indenting list items by their level of nesting,   
    1 def indent\_list(L, level=0):

2 if L == None:  
 3 return  
 4 if type(L) in {list, tuple}:  
 5 for child in L:  
 6 indent\_list(child, level+1)  
 7 else:  
 8 print(f'{" "\*4\*level}{L}')  
 9 if \_\_name\_\_ == '\_\_main\_\_':  
10 L = ['F1', ['F4', 'F5', ['F8']], 'F2', 'F3', \  
11 'D3', ['F6', 'F7']]  
12 indent\_list(L)

* 1. By the time 'F8' is printed in the test case, how many copies of indent\_list calls are active? What are the values of the parameters L and level?
  2. Is line 2 ever executed when running the test case?
  3. How many times total is indent\_list(L) called in the test case above? How can you modify the code above to print your answer?

## 2. Programming

1. (Difficulty: ★★☆☆☆) Write a function that computes Pascal's triangle.   
     
   Col: 0 1 2 3 4 5...   
   Row0: 1  
   Row1: 1 1  
   Row2: 1 2 1  
   Row3: 1 3 3 1  
   Row4: 1 4 6 4 1  
   Row5: 1 5 10 10 5 1  
   …  
   The function should have the following call signature:  
   def pascal(row, column):  
   The function can be defined recursively as follows:

| pascal(row, col) = { | 1 | if col = 0 or col = row |
| --- | --- | --- |
| ?? | if 0 < col < row |

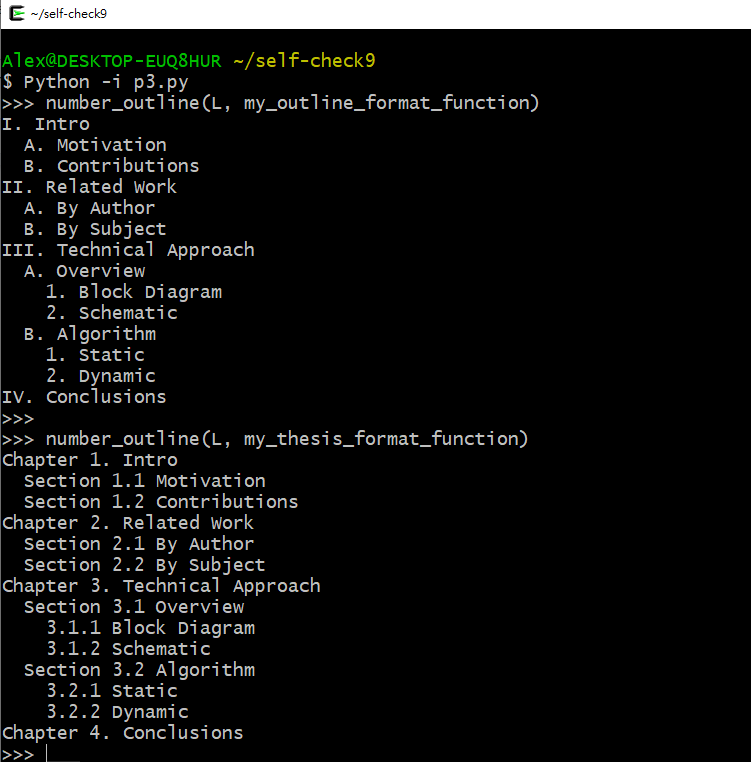
Fill in the ?? above and write this as a recursive function. You may also check the range of the parameter values. If out of range then it should raise a ValueError exception.  
  
Then, write a function print\_Pascal\_triangle(n) to print out the triangle as above. The parameter *n* indicates the number of rows.

1. (Difficulty: ★★★☆☆) Generalize the recursive finding function rec\_find() from the recursion slides #22 so that you can pass either a value to be compared for equality, or you can pass a plug-in function that defines the matching criterion. You can check if val parameter is a plug-in function by calling the built-in function  
   callable(val)  
   which returns True if val is a function object or a lambda expression, or returns False if it is a value (assumed to be int).  
   For instance, you would called the revised rec\_find as follows:  
     
   >>> L = [1, -2, [3, 4], 5]  
   >>> rec\_find(L, 3)  
   (2, 0)  
   >>> rec\_find(L, lambda x: x == 3) # alternative way  
   (2, 0)  
   >>> rec\_find(L, lambda x: x < 0) # finds -2 at L[1]  
   (1,)  
     
   You need to generate test cases both in terms of different L and val (as int values, lambda expressions, and the expected answers in a list (or tuple). Use a loop to invoke rec\_find with these test values and use assert to check if the answer is as expected.

1. (Difficulty: ★★★★☆) Rewrite the number\_outline() function from the recursion slides #28 so that instead of concatenating the section numbers together by '.', it lets the user specify a plug-in function that defines the format, or use the default formatting. For example,  
   L=['Intro',

['Motivation', 'Contributions'],  
 'Related Work',  
 ['By Author', 'By Subject'],  
 'Technical Approach',  
 ['Overview',  
 ['Block Diagram', 'Schematic'],  
 'Algorithm',  
 ['Static', 'Dynamic'] ],  
 'Conclusions']  
  
Assume you have different formatting functions defined,   
>>> number\_outline(L, my\_outline\_format\_function)  
I. Introduction  
 A. Motivation  
 B. Contributions  
II. Related Work  
 A. By Author  
 B. By Subject  
III. Technical Approach  
 A. Overview  
 1. Block Diagram  
 2. Schematic  
 B. Algorithm  
 1. Static  
 2. Dynamic  
IV. Conclusions  
  
And you can plug in another function for a different format:  
>>> number\_outline(L, my\_thesis\_format\_function)  
Chapter 1. Introduction  
 Section 1.1 Motivation  
 Section 1.2 Contributions  
Chapter 2. Related Work  
 Section 2.1 By Author  
 Section 2.2 By Subject  
Chapter 3. Technical Approach  
 Section 3.1 Overview  
 3.1.1 Block Diagram  
 3.1.2 Schematic  
 Section 3.2 Algorithm  
 3.2.1 Static  
 3.2.2 Dynamic  
Chapter 4. Conclusions  
  
Hints:

* 1. How should the parameter list for the function be revised to accommodate the plug-in function? Should it have a default value?
  2. How does your revised number\_outline() function decide whether to use default formatting or to call the plug-in function for formatting?
  3. What parameter(s) should be passed to the plug-in function? Hint: it is best if the plug-in function just returns the formatted string instead of calling print directly.
  4. What adjustments are needed when number\_outline() makes a recursive call?



def roman(number):

ROMANS = (('M', 1000),

('CM', 900),

('D', 500),

('CD', 400),

('C', 100),

('XC', 90),

('L', 50),

('XL', 40),

('X', 10),

('IX', 9),

('V', 5),

('IV', 4),

('I', 1))

result=""

for roman,value in ROMANS:

while number>=value:

number-=value

result+=roman

return result