EECS 1210 Introduction to Programming in Python
Spring Semester 2024
Prof. Pai H. Chou

Self-Check 9

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?
   a.
   ```
   def Double(n):
       return n + n
   ```
   lambda n: n+n

   b.
   ```
   def Bigger(a, b):
       return a if a > b else b
   ```
   lambda a, b: 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.
   lambda x: len(x)
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']
   ```
   lambda x: (len(x), x)
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.
   L = ['Apr', 'May', 'Nov', 'Mar', 'Jan', 'Feb', 'Oct','Jun', 'Jul', 'Aug', 'Sep', 'Dec']

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}

a. L.sort(key=ML) TypeError: 'list' object is not callable
b. L.sort(key=MD) TypeError: 'dict' object is not callable
c. L.sort(key=lambda x: ML[x])
   TypeError: list indices must be integers or slices, not str
d. L.sort(key=lambda x: MD[x]) Correct
e. L.sort(key=lambda x: ML.index(x)) Correct
f. L.sort(key=lambda x: MD.index(x))
   AttributeError: 'dict' object has no attribute 'index'

5. If chr(97) evaluates to 'a', then what is the value of
   list(map(chr, [97, 98, 99, 100, 101]))
   ?
   ['a', 'b', 'c', 'd', 'e']
6. What is the value of
   list(map(max, [1, 7, 2, 8], [5, 6, 3, 0]))
   ?
   [5, 7, 3, 8], because it is equivalent to
   [max(1, 5), max(7, 6), max(2, 3), max(8, 0)]
7. 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 tup les, as in
   [(1, 5), (7, 6), (2, 3), (8, 0)]
   ?
   zip([1, 7, 2, 8], [5, 6, 3, 0])
8. How to you write the equivalent **list-comprehension** version of
   list(map(max, [1, 7, 2, 8], [5, 6, 3, 0]))
   ?
   [max(*t) for t in zip([1, 7, 2, 8], [5, 6, 3, 0])]
9. 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)
   import operator

   list(map(operator.add, [1, 7, 2, 8], [5, 6, 3, 0]))

10. If you want to read and print lines from a file but skip all blank lines using the following code tem plate

```
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 n ewline character.

lambda x: x != '\n'

11. 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., w ords[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()
```

a. 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?

lambda: print(L)

It won't work if you just put print(L) without lambda because print(L) is evaluated at the time of constructing the dictionary, not at the time of lookup. print(L) returns None, so D.get('show') gives you None, but None can't be called on line 15'.

b. on line 11', what should go into the blank?
lambda: L.extends(words[1:])
because you want the dictionary lookup on line 14' to give you a function (including lambda) that can be called on line 15'.

c. on line 12', what should go into the blank? lambda: print(L.pop())

d. what does the D.get(key, altval) method do? How would it be rewritten without calling the .get() method?
D[key] if key in D else altval

e. what goes into the blank on line 14'?
lambda: print('unknown command')

f. can lines 10'-15' be rewritten without using temporary variables D and f? How?
{'show': _.. }.get(words[0], ...)()

12. 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         ...
```

a. What are the inner functions in this code fragment?
show(), push(), pop(), unknown()

b. Why would it be preferable to using inner functions in this case (hint: line 11)?
makes the lookup table D easier to maintain and update, and potentially more expressive since lambdas are limited to single-line expressions whereas inner functions can be arbitrary code.

c. How would the lookup code D.get(words[0], _____)
be written differently from the lambda version? Fill in the blank.
D.get(words[0], unknown)

13. 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
>>>
```

加上三引號

14. Does *Python Style Guide* recommend using **camel case** or **snake case** for function names?
    snake case

15. What are examples of **recursive data types** in Python? Are the following data types recursive?
    a. int
    b. list
    c. tuple
    d. dict
    e. set
    f. float
    g. bool

recursive: list, tuple, dict

Not recursive: int, set (because set is mutable but members of set are not mutable), float, bool

If someone asks, str is a debatable case, because you could argue that a substring is "contained" inside the string, and python's built-in "in" operator works on substrings; but each "member" is a character that can only be single-character strings but cannot further contain other strings.

16. What is a **recursive function**?
    a function that calls itself directly or indirectly to solve part of the problem.
17. What is a **base case** in a recursive function? Should all recursive functions have at least one base c ase? Why or why not?
    a base case is a conditional path where the function returns without making a recursive call. All recursiv e functions should have at least one base case or else it will not terminate normally.
18. If you want to count the number of integers in a list that may contain either integers or list of integ ers and other lists (of integers and other lists…),
    a. 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
```

It only counts 'integer'.

'list' will not be counted.

b. 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
```

if a list in list

it will not be counted.

c. Fill in the code below for counting recursively. You may assume types of elements are eit her 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 #type(L) == int
3       return 1
4   else:
5       n = 0
6       for i in L:
7           n += _________ #count_ints(i)
8       return n
```

d. Rewrite lines 5-8 above to eliminate the for loop and replace it with a combination of sum () and map().

return sum(map(count_ints, L))

19. Recursion can also replace a loop. Rewrite the count_int by converting the loop into a recursive ca ll 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 ____ # 1
4   if __ :  # L = []
5       return ____ # 0
6   return rec_count(____) + rec_count(____)
7       # one recursive call for current element, and
```

```
8       # 2nd recursive call for "the rest of the loop"
        # rec_count(L[0]) + rec_count(L[1:])
```

20. Explain what the following functions do in terms of what the parameters are (if any) and what the return value is.
    a. os.getcwd()
       get current working directory path (string)
    b. os.listdir(d)
       get list of names of files and directories in directory d
    c. os.path.isdir(d)
       check if d (string) is a directory

21. 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
```

    a. What does '.' mean as the default value of parameter p?
       current working directory.
    b. 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.
       not os.path.isdir(p)
    c. 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.
       os.listdir(p)
    d. How do you recursively count each path of the list so that the counts can be summed? Fill in the blank on line 6.
       map(count_files,dir_content)

22. In the recursive-find example,
    a. calling
       M = [1, 2, [3, [4, 23]
       rec_find(M, 23)
       results in the tuple value (2, 1, 1). What does it mean?
       the value 23 can be found in the list M= [1, 2, [3, [4, 23], 5, 6]] at M[2][1][1].
    b. Calling
       rec_find(43, 43)
       results in True. What does it mean?
       有在43找到43

c. What would be the result of calling
rec_find([1, 2, 3], [1, 2, 3])
? True

d. What would be the result of calling
rec_find([[1, 2, 3]], [1, 2, 3])
? (0,)

23. The source ode 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
```

a. What is the **condition of the base case** in this recursive function?

type(L) is neither list nor tuple

b. 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 lin es 3-8?
yes, line 9 can be executed even if type(L) is either list or tuple. This is because if for-loop run s to its completion without an early return on line 6 or line 8, then it falls through and executes line 9. This means val hasn't matched any member of L recursively.

c. Can line 9 compare only two ints, or can it be comparing two tuples or two lists?

d. Line 5 tests
5 [SEP] if p == True:
but why can't it be replaced with
5 [SEP] if p:
?
because if p evaluates to true even if p is not == True. The return value can also be a non-empt y tuple containing the indices of the matched items. This case catches the case returned by line 9, which is a match at the last (innermost) index, hence return (i,).

e. Line 7 tests
7 [SEP] if p != False:
but why isn't it redundant with line 5? Doesn't
p != False
imply

p == True

?

because p != False covers p == True as well as p != True but p is some non-empty values (in t
his case a tuple with indices). Since p == True on line 5 already returned on line 6, line 7 is lik
e an implicit elif, and it consist of all those p != True, p != False, and p is non-empty values. T
hat is, value returned by either line 6 or line 8 of a recursive call. By the way, the tuple (0,) has
a true value.

24. 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)
```

a. 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?

three copies:

lowest level: (L = ['F1', ['F4', 'F5', ['F8']], 'F2', 'F3', 'D3', ['F6', 'F7']], level=0)

next level: (L = ['F4', 'F5', ['F8']], level = 1)

top level: (L = ['F8'], level = 2)

b. Is line 2 ever executed when running the test case?

c. How many times total is indent_list(L) called in the test case above? How can you modif
y the code above to print your answer?

13 times. could use a global and every time the function is entered increment the global

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:

$$\text{pascal(row, col)} = \begin{cases} 1 & \text{if col = 0 or col = row} \\ ?? & \text{if 0 < col < row} \end{cases}$$

Fill in the ?? above and write this as a recursive function. You may also check the range of the par ameter 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.

2. (Difficulty: ★★★☆☆) Generalize the recursive finding function rec_find() from the recursion s lides #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 valu e (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 expressi ons, and the expected answers in a list (or tuple). Use a loop to invoke rec_find with these test val ues and use assert to check if the answer is as expected.

ANS:

```
def rec_find(L,val):
```

```
    judge=callable(val)
    if type(L) in {list, tuple}: # if look inside members of L
        for i, v in enumerate(L):
            p = rec_find(v, val) # recursively find each member
            if p == True:
                return (i,)
            if p != False:
                return (i,)+p
    if judge == False:
        return L==val
    else:
        return val(L)
```

3. (Difficulty: ★★★★☆) Rewrite the number_outline() function from the recursion slides #28 so t hat instead of concatenating the section numbers together by '.', it lets the user specify a plug-in fun ction 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
```

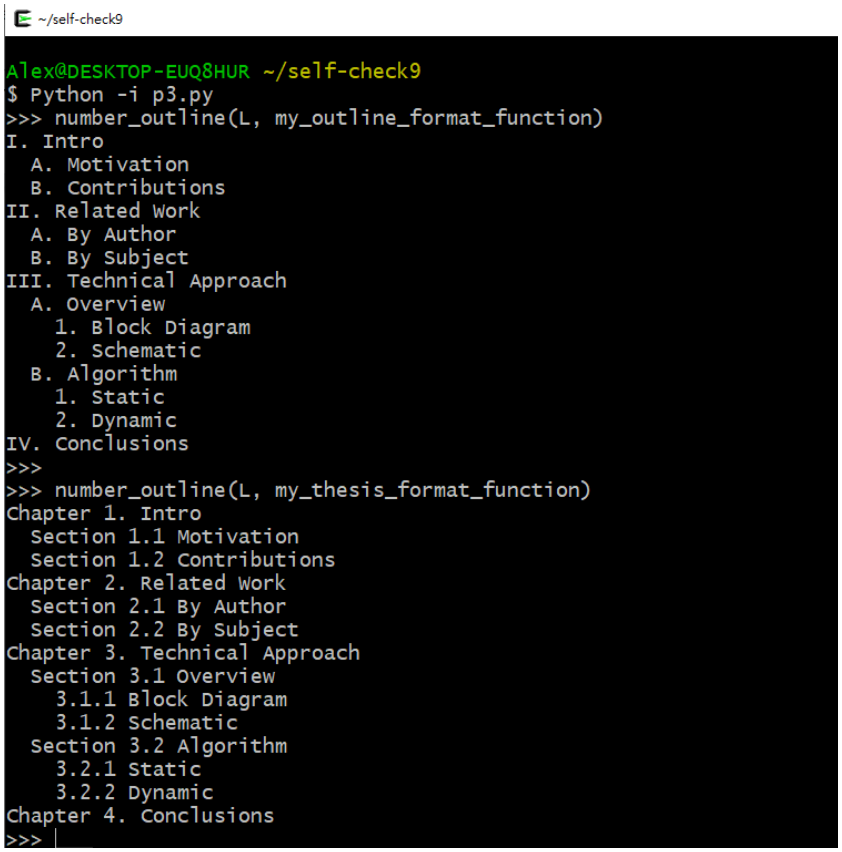
A. Overview
1. Block Diagram
2. Schematic
B. Algorithm
1. Static
2. Dynamic
IV. Conclusions

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:

a. How should the parameter list for the function be revised to accommodate the plug-in function? Should it have a default value?
b. How does your revised number_outline() function decide whether to use default formatting or to call the plug-in function for formatting?
c. 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.
d. 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

ANS:

In [1]:
```
L=['Intro',
['Motivation', 'Contributions'],
'Related Work',
['By Author', 'By Subject'],
'Technical Approach',
['Overview',
['Block Diagram', 'Schematic'],
'Algorithm',
['Static', 'Dynamic'] ],
'Conclusions']
```

In [2]:
```
def my_thesis_format_function(prefix):
    if(len(prefix)==1):
        a = 'Chapter '+ '.'.join(map(str,prefix))+'.'
    elif(len(prefix)==2):
        a = 'Section '+ '.'.join(map(str,prefix))
    else:
        a = '.'.join(map(str,prefix))
    return a
```

In [3]:
```
def my_outline_format_function(prefix):
    if(len(prefix)==1):
        a = 'I' * int(prefix[0])
        a+='.'
        if((prefix[0])==4):
            a = 'IV.'
        return(a)
    if(len(prefix)==2):
        return(chr(ord('A')+int(prefix[1])-1)+'.')
    if(len(prefix)==3):
        return(str(prefix[2])+'.')
```

In [4]:
```
def number_outline(L,key, prefix=()):
    if type(L) in {list, tuple}:
# keep prefix[-1], extend by new dimension, starting from 1
        i = 0
        for v in L:
            if type(v) not in {list, tuple}:
                i += 1
            number_outline(v, key, prefix=prefix+(i,))
# don't increment if v is a list/tuple
# otherwise, indent and join the prefix together by '.'
    else:
        s = ' ' * 4*(len(prefix)-1)
        s += key(prefix)
        s += ' ' + L
        print(s)
```