Guerrilla Section 2: Higher Order Functions, Sequences, & Recursion/Tree Recursion

Solutions

Instructions

Form a group of 3-4. Start on Question 0. Check off with a lab assistant when everyone in your group understands how to solve Question 0. Repeat for Question 1, 2, etc. **You're not allowed to move on from a question until you check off with a tutor.** You are allowed to use any and all resources at your disposal, including the interpreter, lecture notes and slides, discussion notes, and labs. You may consult the lab assistants, **but only after you have asked everyone else in your group. The purpose of this section is to have all the students working together to learn the material. The questions defined as EXTRA are optional, make sure you finish all the non-optional ones before you attempt the extra ones.**

Higher Order Functions

Question 0

a) What do lambda expressions do? Can we write all functions as lambda expressions? In what cases are lambda expressions useful?

Lambda expressions create functions. When a lambda expression is evaluated, it produces a function. We often use lambdas to create short anonymous functions that we won't need for too long. We can't write all functions as lambda expressions because lambda functions all have to have "return" statements and they can't contain very complex multiline expressions.

b) For each of the following prompts, explain what is happening.:

```
>>> def square(x):
...  return x * x
```

Defines a function square

```
>>> square(4)
```

Calls the function square

```
>>> square
```

Evaluates to the actual function that does squaring

```
>>> square = lambda x: x * x
```

Defines a squaring function using assignment and a lambda expression

c) Determine if each of the following will error:

```
>>> 1/0
Error

>>> boom = lambda: 1/0
No error, since we don't evaluate the body of the procedure when we
define it.

>>> boom()
Error
```

Question 1

a) Express the following expressions using def statements:

i. `pow = lambda x, y: x**y`

```
def pow(x, y):
    return x**y
```

ii. `foo = lambda x: lambda y: lambda z: x + y * z`

```
def foo(x):
    def f(y):
        def g(z):
            return x + y * z
        return g
    return f
```

iii. `compose = lambda f, g: lambda x: f(g(x))`

```
def compose(f, g):
    def h(x):
        return f(g(x))
    return h
```

b) Draw Environment Diagrams for the following lines of code

```
square = lambda x: x * x
higher = lambda f: lambda y: f(f(y))
higher(square)(5)
```

Solution: https://goo.gl/LATqV9

```
a = (lambda f, a: f(a))(lambda b: b * b, 2)
```

Solution: https://goo.gl/TyriuP

Question 2:

a) Express the following expressions using lambdas:

i. `square(4)`

```
(lambda x: x * x)(4)
```

ii. `sum_of_squares(3, 4)`

```
def sum_of_squares(x, y):
    """Squares each argument and adds them together.

    >>> sum_of_squares(3, 4)     # (3 * 3) + (4 * 4)
    25
    >>> sum_of_squares(9, 5)     # (9 * 9) + (5 * 5)
    106
    """
(lambda x, y: x * x + y * y)(3, 4)
```

iii.

```
def hello_world():
  return "hello world!"
hello_world()
(lambda: "hello world!")()
```

b) What is displayed in the course of evaluating this expression?

```
cs61a = (lambda boring, difficult: difficult)((lambda tears: (tears, 2))(print(1)))
1
TypeError
```

c) **<u>EXTRA Challenge:</u>** Complete the given lambda expression so the second line evaluates to 2018. You may only use the names two_thousand, two, k, eight, and teen and parentheses in your expression (no numbers, operators, etc.). Hint: an environment diagram might be useful.

```
best_year = lambda four: lambda k: _______k(four)(four)____________
best_year(9)(lambda eight: lambda teen: 2000 + eight + teen)
```

STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!

Question 3

Write a `make_skipper`, which takes in a number `n` and outputs a function. When this function takes in a number `x`, it prints out all the numbers between 0 and `x`, skipping every `n`th number (meaning skip any value that is a multiple of `n`).

```
>>> a = make_skipper(2)
>>> a(5)
1
3
5
def make_skipper(n):


    def skipper(x):
        for i in range(x+1):
          if i % n != 0:
            print(i)
      return skipper
```

EXTRA: Question 4

Write make_alternator which takes in two functions, f and g, and outputs a function. When this function takes in a number x, it prints out all the numbers between 1 and x, applying the function f to every odd-indexed number and g to every even-indexed number before printing.

“””

```
>>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
>>> a(5)
1
6
9
8
25
>>> b = make_alternator(lambda x: x * 2, lambda x: x + 2)
>>> b(4)
2
4
6
```

6
“””

```
def make_alternator(f, g):
  def alternator(n):
    i = 1
    while i <= n:
      if i % 2 == 1:
        print(f(i))
      else:
        print(g(i))
      i += 1
  return alternator

# Alternatively
>>> def make_alternator(f, g):
...   def alternator(n):
...     for i in range(1, n+1):
...       print(f(i)) if i % 2 == 1 else print(g(i))
...   return alternator
```

STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!

Sequences

Question 0

Fill out what python would display at each step if applicable.

Note: (keep in mind list slicing creates a brand new list, does not modify existing list)

i.

```
>>> lst = [1, 2, 3, 4, 5]
>>> lst[1:3]
[2, 3]
>>> lst[0:len(lst)]   # one (not so good) way to get the whole tuple
[1, 2, 3, 4, 5]
>>> lst[-4:]          # start: 4th item from end, to the very end
[2, 3, 4, 5]
>>> lst[:3]           # ommiting the left-hand index starts from the
[1, 2, 3]               very beginning
```

```
>>> lst[3:]          # ommiting the right-hand index goes to the very
[4, 5]                 end
>>> lst[:]           # dupliates the entire tuple
[1, 2, 3, 4, 5]
```

ii. **Hint:** You can also specify the increment step-size for slicing. The notation is lst[start:end:step].

```
>>> lst[1:4:2]     # subsequence from index 1 up to index 4, but only
[2, 4]               getting every other item
>>> lst[0:4:3]     # subsequence from index 0 up to index 4, but only
[1, 4]                getting every third item
>>> lst[:4:2]
[1, 3]
>>> lst[1::2]      # subsequence from index 1 up to end, but only
[2, 4]               getting every other item
>>> lst[::2]       # subsequence from beginning up to end, but only
[1, 3, 5]            getting
every other item

>>> lst[::-1]      # get the entire tuple but in reverse
[5, 4, 3, 2, 1]

>>> lst2 = [6, 1, 0, 7]
>>> lst + lst2
[1, 2, 3, 4, 5, 6, 1, 0, 7]
>>> lst + 100
TypeError: can only concatenate list (not "int") to list
>>> lst3 = [[1], [2], [3]]
>>> lst + lst3
[1, 2, 3, 4, 5, [1], [2] , [3]]
```

Question 1

Draw the environment diagram that results from running the code below
Link: **Box and pointer diagram in Python tutor**

```
def reverse(lst):
    if len(lst) <= 1:
        return lst
    return reverse(lst[1:]) + [lst[0]]

lst = [1, [2, 3], 4]
rev = reverse(lst)
```

Question 2

Write `combine_skipper`, which takes in a function `f` and list `lst` and outputs a list. When this function takes in a list `lst`, it looks at the list in chunks of four and applies `f` to the first two elements in every set of four elements and replaces the first element with the output of the function `f` applied to the two elements as the first value and the index of the second item of the original two elements as the second value of the new two elements. `f` takes in a list and outputs a value. [Assume the length of `lst` will always be divisible by four]

```
>>> lst = [4, 7, 3, 2, 1, 8, 5, 6]
>>> f = lambda l: sum(l)
>>> lst = combine_skipper(f, lst)
>>> lst
[11, 1, 3, 2, 9, 5, 5, 6]
>>> lst2 = [4, 3, 2, 1]
>>> lst2 = combine_skipper(f, lst2)
>>> lst2
[7, 1, 2, 1]

def combine_skipper(f, lst):
    n = 0
    while n < len(lst) // 4:
        lst[4*n:4*n+2]=[f(lst[4*n:4*n+2])] + [4*n+1]
        n += 1
    return lst
```

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!

Recursion

Question 0

a) What are three things you find in every recursive function?
1. One or more Base Cases

2. Way(s) to make the problem into a smaller problem of the same type (so that it can be solved recursively).

3. One or more Recursive Cases that solve the smaller problem and then uses the solution the smaller problem to solve the original (large) problem

b) When you write a Recursive function, you seem to call it before it has been fully defined. Why doesn't this break the Python interpreter? Explain in haiku if possible.
When you define a function, Python does not evaluate the body of the function.

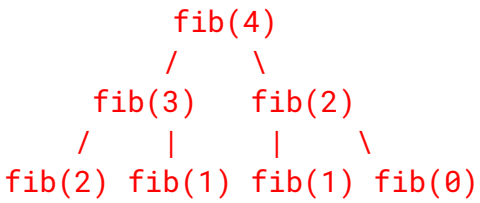
Python does not care
about a function's body
until it is called

Question 1
Hint: Domain is the type of data that a function takes in as argument. The Range is the type of data that a function returns.
E.g. the *domain* of the function `square` is numbers. The *range* is numbers.

a) Here is a Python function that computes the nth fibonnaci number. What's it's domain and range? Identify those three things from **Q0a**:

```
def fib(n): Domain is integers, Range is integers
  if n == 0: # base case
    return 0
  elif n == 1: another base case
    return 1
  else: # ONE recursive CASE with TWO recursive CALLS
    return fib(n-1) + fib(n-2) # reducing the problem
```

Write out the recursive calls made when we do `fib(4)` (this will look like an upside down tree).



```
        /    |
    fib(1)  fib(0)
```

b) What does the following `cascade2` do?

Takes in a number n and prints out n, n excluding the ones digit, n excluding the tens digit, n excluding the hundreds digit, etc, then back up to the full number

```
def cascade2(n):
    """Print a cascade of prefixes of n."""
    print(n)
    if n >= 10:
        cascade2(n//10)
        print(n)
```

What is the domain and range of `cascade2`? Identify the three things from Q0a:

```
def cascade2(n): Domain is integers, Range is None
    """Print a cascade of prefixes of n."""
    print(n) # Base case is when n < 10
    if n >= 10: # recursive case
        cascade2(n//10) # reducing the problem
        print(n)
```

c) Describe what does each of the following functions mystery and fooply do.

```
>>> def mystery(n):
...   if n == 0:
...         return 0
...   else:
...         return n + mystery(n - 1)
```

Sums integers up to n: 1 + 2 + 3 + 4 + 5 +...+ n

```
>>> def foo(n):
...   if n < 0:
...         return 0
...   return foo(n - 2) + foo(n - 1)
```

```
>>> def fooply(n):
...   if n < 0:
...         return 0
...   return foo(n) + fooply(n - 1)
```

Foo returns the nth fibonacci number
fooply returns the sum of first n fibonacci numbers

STOP!

Don't proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!

Question 2:
Fill out the function has_seven using recursion

```
def has_seven(k):
    """Returns True if at least one of the digits of k is a 7, False
otherwise.

    >>> has_seven(3)
    False
    >>> has_seven(7)
    True
    >>> has_seven(2734)
    True
    >>> has_seven(2634)
    False
    >>> has_seven(734)
    True
    >>> has_seven(7777)
    True
         """

    if k % 10 == 7:
        return True
```

```
    elif k < 10:
        return False
    else:
        return has_seven(k // 10)
```

EXTRA Question 3

Mario needs to jump over a series of Piranha plants, represented as a list of 0's and 1's. Mario only moves forward and can either step (move forward one space) or jump (move forward two spaces) from each position. How many different ways can Mario traverse a level without stepping or jumping into a Piranha plant? Assume that every level begins with a 1 (where Mario starts) and ends with a 1 (where Mario must end up).

```
def mario_number(level):
    """
    Return the number of ways that Mario can traverse the level,
    where Mario can either hop by one digit or two digits each turn.
    A level is defined as being an integer with digits where a 1 is
    something Mario can step on and 0 is something Mario cannot step
    on.
    >>> mario_number(10101)  #Hops each turn: (1, 2, 2)
    1
    >>> mario_number(11101) #Hops each turn: (1, 1, 1, 2), (2, 1, 2)
    2
    >>> mario_number(100101) #No way to traverse through level
    0
    """
    if level == 1:
        return 1
    elif level % 10 == 0:
        return 0
    else:
        return mario_number(level // 10) + mario_number((level // 10) // 10)
```

EXTRA Challenge: Question 4

Implement the combine function, which takes a non-negative integer n, a two-argument function f, and a number result. It applies f to the first digit of n and the result of combining the rest of the digits of n by repeatedly applying f (see the doctests). If n has no digits (because it is zero), combine returns result. Assume n is non negative.

```
def combine (n , f , result ):

"""
Combine the digits in n using f.
>>> combine (3, mul, 2) #mul (3, 2)
6
>>> combine (43, mul, 2) # mul (4, mul(3, 2))
24
>>> combine (6502, add, 3) # add (6, add(5, add(0, add(2, 3)))
16
>>> combine (239, pow, 0) # pow (2, pow(3, pow(9, 0)))
8
"""

if n == 0:
      return result
else:
      return ___combine ( n // 10 , f , f ( n % 10 , result ))____
```

STOP!

Don’t proceed until everyone in your group has finished and understands all exercises in this section, and you have gotten checked off!