Lab 3: Lambda Expressions and Recursion

Due at 11:59pm on 02/04/2015.

Starter Files

Download lab03.zip. Inside the archive, you will find starter files for the questions in this lab, along with a copy of the OK autograder.

Submission

By the end of this lab, you should have submitted the lab with `python3 ok --submit`. You may submit more than once before the deadline; only the final submission will be graded.

- To receive credit for this lab, you must complete Questions 2, 4, 6, 7, and 8 in lab03.py and submit through OK.
- Questions 1, 3 (What Would Python Print?), and 5 (Environment Diagrams) are designed to help introduce concepts and test your understanding.
- Questions 9, 10, 11, 12, and 13 are optional extra practice (all except 11 are in lab03_extra.py). It is recommended that you complete these problems on your own time.

Table of Contents

Now we'll see where environment diagrams come in really handy: When dealing with lambda expressions in addition to other higher-order functions.

Higher Order Functions

Higher order functions are functions that take a function as an input, and/or output a function. We will be exploring many applications of higher order functions.

Question 1: What Would Python Output?

```
>>> def square(x):
...     return x*x
...
>>> def neg(f, x):
...     return -f(x)
...
>>> neg(square, 4)
______
>>> def even(f):
...     def odd(x):
...         if x < 0:
...             return f(-x)
...         return f(x)
...     return odd
...
>>> def identity(x):
...     return x
...
>>> triangle = even(identity)
>>> triangle
______

>>> triangle(61)
```

```
______

>>> triangle(-4)
______
>>> def first(x):
...     x += 8
...     def second(y):
...         print('second')
...         return x + y
...     print('first')
...     return second
...
>>> f = first(15)
______

>>> f
______

>>> f(16)
______
```

Question 2: Flight of the Bumblebee

Write a function that takes in a number `n` and returns a function that takes in a number `range` which will print all numbers from `0` to `range` (including `0` but excluding `range`) but print `Buzz!` instead for all the numbers that are divisible by `n`.

```
def make_buzzer(n):
  """ Returns a function that prints numbers in a specified
  range except those divisible by n.

  >>> i_hate_fives = make_buzzer(5)
  >>> i_hate_fives(10)
  Buzz!
  1
```

```
2
3
4
Buzz!
6
7
8
9
"""
"*** YOUR CODE HERE ***"
```

Lambdas

Lambda expressions are one-line functions that specify two things: the parameters and the return value.

```
lambda <parameters>: <return value>
```

One difference between using the def keyword and lambda expressions is that def is a statement, while lambda is anexpression. Evaluating a def statement will have a side effect; namely, it creates a new function binding in the current environment. On the other hand, evaluating a lambda expression will not change the environment unless we do something with this expression. For instance, we could assign it to a variable or pass it in as a function argument.

Question 3: What would Python print?

```
>>> a = lambda: 5
>>> a()
______

>>> a(5)
______

>>> a()()
______

>>> lambda x: x # Can we access this function?
```

```
______

>>> b = lambda: lambda x: 3
>>> b()(15)

______

>>> c = lambda x, y: x + y
>>> c(4, 5)

______

>>> d = lambda x: c(a(), b()(x))
>>> d(2)

______

>>> b = lambda: lambda x: x
>>> d(2)

______

>>> e = lambda x: lambda y: x * y
>>> e(3)

______

>>> e(3)(3)

______

>>> f = e(2)
>>> f(5)

______

>>> f(6)

______

>>> g = lambda: print(1) # When is the body of this function run?

______
```

```
>>> h = g()
______

>>> print(h)
______
```

Question 4: Make your own lambdas

For each of the following expressions, write functions f1, f2, f3, and f4 such that the evaluation of each expression succeeds, without causing an error. Be sure to use lambdas in your function definition instead of nested def statements. Each function should have a one line solution.

```
def f1():
  """
  >>> f1()
  3
  """
  "*** YOUR CODE HERE ***"

def f2():
  """
  >>> f2()()
  3
  """
  "*** YOUR CODE HERE ***"

def f3():
  """
  >>> f3()(3)
  3
  """
  "*** YOUR CODE HERE ***"

def f4():
  """
  >>> f4()()(3)()
  3
  """
  "*** YOUR CODE HERE ***"
```

Question 5: Environment Diagrams with Lambdas

Try drawing environment diagrams for the following code and predicting what Python will output.

You can check your work with the Online Python Tutor. Please try drawing it yourself first!

```
>>> # Part 1
>>> a = lambda x : x * 2 + 1
>>> def b(x):
...     return x * y
...
>>> y = 3
>>> b(y)
______

>>> def c(x):
...     y = a(x)
...     return b(x) + a(x+y)
...
>>> c(y)
______

>>> # Part 2: This one is pretty tough. A carefully drawn environment
>>> # diagram will be really useful.
>>> g = lambda x: x + 3
>>> def wow(f):
...     def boom(g):
...       return f(g)
...     return boom
...
```

```
>>> f = wow(g)
>>> f(2)
______

>>> g = lambda x: x * x
>>> f(3)
______
```

Recursion

Warm Up: Recursion Basics

A recursive function is a function that calls itself in its body, either directly or indirectly. Recursive functions have two important components:

1. Base case(s), where the function directly computes an answer without calling itself. Usually the base case deals with the simplest possible form of the problem you're trying to solve.
2. Recursive case(s), where the function calls itself with a simpler argument as part of the computation.

Let's look at the canonical example, factorial:

```
def factorial(n):
  if n == 0:
    return 1
  return n * factorial(n - 1)
```

We know by its definition that 0! is 1. So we choose n = 0 as our base case. The recursive step also follows from the definition of factorial, i.e. n! = n * (n-1)!.

The next few questions in lab will have you writing recursive functions. Here are some general tips:

- Consider how you can solve the current problem using the solution to a simpler version of the problem. Remember to trust the recursion: assume that your solution to the simpler problem works correctly without worrying about how.
- Think about what the answer would be in the simplest possible case(s). These will be your base cases - the stopping points for your recursive calls. Make sure to consider the possibility that you're missing base cases (this is a common way recursive solutions fail).

Question 6: In sum...

Write a function `sum` that takes a single argument `n` and computes the sum of all integers between 0 and `n` inclusive. Assume `n`is non-negative.

```
def sum(n):
  """Computes the sum of all integers between 1 and n, inclusive.
  Assume n is positive.

  >>> sum(1)
  1
  >>> sum(5)  # 1 + 2 + 3 + 4 + 5
  15
  """
  "*** YOUR CODE HERE ***"
```

Question 7: Misconceptions

The following examples of recursive functions show some examples of common recursion mistakes. Fix them so that they work as intended.

```
def sum_every_other_number(n):
  """Return the sum of every other natural number
  up to n, inclusive.

  >>> sum_every_other_number(8)
  20
  >>> sum_every_other_number(9)
  25
  """
  if n == 0:
    return 0
  else:
    return n + sum_every_other_number(n - 2)
def fibonacci(n):
  """Return the nth fibonacci number.
```

```
    >>> fibonacci(11)
    89
    """
    if n == 0:
        return 0
    elif n == 1:
        return 1
    else:
        fibonacci(n - 1) + fibonacci(n - 2)
```

Question 8: Hailstone

For the `hailstone` function from homework 1, you pick a positive integer `n` as the start. If `n` is even, divide it by 2. If `n` is odd, multiply it by 3 and add 1. Repeat this process until `n` is 1. Write a recursive version of hailstone that prints out the values of the sequence and returns the number of steps.

```
def hailstone(n):
    """Print out the hailstone sequence starting at n, and return the
    number of elements in the sequence.

    >>> a = hailstone(10)
    10
    5
    16
    8
    4
    2
    1
    >>> a
    7
    """
    "*** YOUR CODE HERE ***"
```

Extra Questions

Questions in this section are not required for submission. However, we encourage you to try them out on your own time for extra practice.

Question 9: I Heard You Liked Functions...

This question is extremely challenging. Use it to test if you have really mastered the material!

Define a function cycle that takes in three functions f1, f2, f3, as arguments. cycle will return another function that should take in an integer argument n and return another function. That final function should take in an argument x and cycle through applying f1, f2, and f3 to x, depending on what n was. Here's the what the final function should do to x for a few values of n:

- n = 0, return x
- n = 1, apply f1 to x, or return f1(x)
- n = 2, apply f1 to x and then f2 to the result of that, or return f2(f1(x))
- n = 3, apply f1 to x, f2 to the result of applying f1, and then f3 to the result of applying f2, or f3(f2(f1(x)))
- n = 4, start the cycle again applying f1, then f2, then f3, then f1 again, or f1(f3(f2(f1(x))))
- And so forth.

Hint: most of the work goes inside the most nested function.

```
def cycle(f1, f2, f3):
  """ Returns a function that is itself a higher order function
  >>> def add1(x):
  ...     return x + 1
  >>> def times2(x):
  ...     return x * 2
  >>> def add3(x):
  ...     return x + 3
  >>> my_cycle = cycle(add1, times2, add3)
  >>> identity = my_cycle(0)
  >>> identity(5)
  5
  >>> add_one_then_double = my_cycle(2)
```

```
    >>> add_one_then_double(1)
    4
    >>> do_all_functions = my_cycle(3)
    >>> do_all_functions(2)
    9
    >>> do_more_than_a_cycle = my_cycle(4)
    >>> do_more_than_a_cycle(2)
    10
    >>> do_two_cycles = my_cycle(6)
    >>> do_two_cycles(1)
    19
    """
    "*** YOUR CODE HERE ***"
```

Question 10: Lambdas and Currying

We can transform multiple-argument functions into a chain of single-argument, higher order functions by taking advantage of lambda expressions. This is useful when dealing with functions that take only single-argument functions. We will see some examples of these later on.

Write a function `lambda_curry2` that will curry any two argument function using lambdas. See the doctest if you're not sure what this means.

```
def lambda_curry2(func):
    """
    Returns a Curried version of a two argument function func.
    >>> from operator import add
    >>> x = lambda_curry2(add)
    >>> y = x(3)
    >>> y(5)
    8
    """
    "*** YOUR CODE HERE ***"
    return ______
```

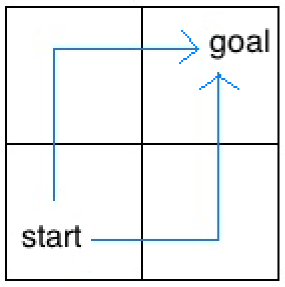
Question 11: Community

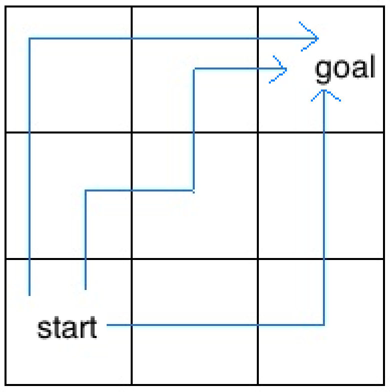
Fill in the blanks as to what Python would do here. Please try this problem first with an environment diagram, and then again without an environment diagram.

```
>>> def troy():
...     abed = 0
...     while abed < 10:
...         britta = lambda: abed
...         abed += 1
...     abed = 20
...     return britta
...
>>> jeff = troy()
>>> shirley = lambda : jeff
>>> pierce = shirley()
>>> pierce()
______
```

Question 12: Insect Combinatorics

Consider an insect in an M by N grid. The insect starts at the bottom left corner, (0, 0), and wants to end up at the top right corner, (M-1, N-1). The insect is only capable of moving right or up. Write a function paths that takes a grid length and width and returns the number of different paths the insect can take from the start to the goal. (There is a closed-form solution to this problem, but try to answer it procedurally using recursion.)





For example, the 2 by 2 grid has a total of two ways for the insect to move from the start to the goal. For the 3 by 3 grid, the insect has 6 diferent paths (only 3 are shown above).

```
def paths(m, n):
  """Return the number of paths from one corner of an
  M by N grid to the opposite corner.

  >>> paths(2, 2)
  2
  >>> paths(5, 7)
  210
  >>> paths(117, 1)
  1
  >>> paths(1, 157)
  1
  """
  "*** YOUR CODE HERE ***"
```

Question 13: GCD

The greatest common divisor of two positive integers a and b is the largest integer which evenly divides both numbers (with no remainder). Euclid, a Greek mathematician in 300 B.C., realized that the greatest common divisor of a and b is one of the following:

- the smaller value if it evenly divides the larger value, OR
- the greatest common divisor of the smaller value and the remainder of the larger value divided by the smaller value

In other words, if a is greater than b and a is not divisible by b, then

```
gcd(a, b) == gcd(b, a % b)
```

Write the gcd function recursively using Euclid's algorithm.

```
def gcd(a, b):
  """Returns the greatest common divisor of a and b.
  Should be implemented using recursion.

  >>> gcd(34, 19)
  1
```

```
>>> gcd(39, 91)
13
>>> gcd(20, 30)
10
>>> gcd(40, 40)
40
"""
"*** YOUR CODE HERE ***"
```