Lab 2 Solutions **lab02.zip (lab02.zip)**

Solution Files

Topics

Consult this section if you need a refresher on the material for this lab. It's okay to skip directly to the questions and refer back here should you get stuck.

Required Questions

What Would Python Display?

Important: For all WWPD questions, type `Function` if you believe the answer is `<function...>`, `Error` if it errors, and `Nothing` if nothing is displayed.

Q1: WWPD: Lambda the Free

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q lambda -u
```

As a reminder, the following two lines of code will not display anything in the Python interpreter when executed:

```
>>> x = None
>>> x
```

```
>>> lambda x: x  # A lambda expression with one parameter x
------

>>> a = lambda x: x  # Assigning the lambda function to the name a
>>> a(5)
------

>>> (lambda: 3)()  # Using a lambda expression as an operator in a call exp.
------

>>> b = lambda x: lambda: x  # Lambdas can return other lambdas!
>>> c = b(88)
>>> c
------

>>> c()
------

>>> d = lambda f: f(4)  # They can have functions as arguments as well.
>>> def square(x):
...     return x * x
>>> d(square)
------
```

```
>>> x = None # remember to review the rules of WWPD given above!
>>> x
>>> lambda x: x
------
```

```
>>> z = 3
>>> e = lambda x: lambda y: lambda: x + y + z
>>> e(0)(1)()
------

>>> f = lambda z: x + z
>>> f(3)
------
```

```
>>> higher_order_lambda = lambda f: lambda x: f(x)
>>> g = lambda x: x * x
>>> higher_order_lambda(2)(g)  # Which argument belongs to which function call?
______

>>> higher_order_lambda(g)(2)
______

>>> call_thrice = lambda f: lambda x: f(f(f(x)))
>>> call_thrice(lambda y: y + 1)(0)
______

>>> print_lambda = lambda z: print(z)  # When is the return expression of a lambda express
>>> print_lambda
______

>>> one_thousand = print_lambda(1000)
______

>>> one_thousand
______
```

Q2: WWPD: Higher Order Functions

Use Ok to test your knowledge with the following "What Would Python Display?" questions:

```
python3 ok -q hof-wwpd -u
```

```
>>> def even(f):
...     def odd(x):
...         if x < 0:
...             return f(-x)
...         return f(x)
...     return odd
>>> steven = lambda x: x
>>> stewart = even(steven)
>>> stewart
------

>>> stewart(61)
------

>>> stewart(-4)
------
```

```
>>> def cake():
...     print('beets')
...     def pie():
...         print('sweets')
...         return 'cake'
...     return pie
>>> chocolate = cake()
------

>>> chocolate
------

>>> chocolate()
------

>>> more_chocolate, more_cake = chocolate(), cake
------

>>> more_chocolate
------

>>> def snake(x, y):
...     if cake == more_cake:
...         return chocolate
...     else:
...         return x + y
>>> snake(10, 20)
------

>>> snake(10, 20)()
------

>>> cake = 'cake'
>>> snake(10, 20)
------
```

Coding Practice

Q3: Lambdas and Currying

We can transform multiple-argument functions into a chain of single-argument, higher order functions by taking advantage of lambda expressions. For example, we can write a function `f(x, y)` as a different function `g(x)(y)`. This is known as **currying**. It's 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. Refer to the textbook (http://composingprograms.com/pages/16-higher-order-functions.html#currying) for more details about currying.

Your solution to this problem should fit entirely on the return line. You can try first writing a solution without the restriction, and then rewriting it into one line after.

> **If the syntax check isn't passing:** Make sure you've removed the line containing `"***YOUR CODE HERE***"` so that it doesn't get treated as part of the function for the syntax check.

```
def lambda_curry2(func):
    """
    Returns a Curried version of a two-argument function FUNC.
    >>> from operator import add, mul, mod
    >>> curried_add = lambda_curry2(add)
    >>> add_three = curried_add(3)
    >>> add_three(5)
    8
    >>> curried_mul = lambda_curry2(mul)
    >>> mul_5 = curried_mul(5)
    >>> mul_5(42)
    210
    >>> lambda_curry2(mod)(123)(10)
    3
    """
    return lambda arg1: lambda arg2: func(arg1, arg2)
```

Use Ok to test your code:

```
python3 ok -q lambda_curry2
```

To curry a two argument function, we want to only call `func` once we have received its two arguments on separate calls to our curry function. We can do so by using lambdas. The outermost lambda will receive the first argument that we will later pass into `func`, and since we haven't yet received both arguments that we need, we want this outermost lambda to return a function itself. This function (which we can implement using another lambda) will take in a single parameter, so when we call it, then we will have had both arguments that we need to call `func`.

Q4: Count van Count

Consider the following implementations of `count_factors` and `count_primes`:

```
def count_factors(n):
    """Return the number of positive factors that n has.
    >>> count_factors(6)
    4   # 1, 2, 3, 6
    >>> count_factors(4)
    3   # 1, 2, 4
    """
    i = 1
    count = 0
    while i <= n:
        if n % i == 0:
            count += 1
        i += 1
    return count

def count_primes(n):
    """Return the number of prime numbers up to and including n.
    >>> count_primes(6)
    3   # 2, 3, 5
    >>> count_primes(13)
    6   # 2, 3, 5, 7, 11, 13
    """
    i = 1
    count = 0
    while i <= n:
        if is_prime(i):
            count += 1
        i += 1
    return count

def is_prime(n):
    return count_factors(n) == 2 # only factors are 1 and n
```

The implementations look quite similar! Generalize this logic by writing a function `count_cond`, which takes in a two-argument predicate function `condition(n, i)`. `count_cond` returns a one-argument function that takes in `n`, which counts all the numbers from 1 to `n` that satisfy `condition` when called.

```
def count_cond(condition):
    """Returns a function with one parameter N that counts all the numbers from
    1 to N that satisfy the two-argument predicate function Condition, where
    the first argument for Condition is N and the second argument is the
    number from 1 to N.

    >>> count_factors = count_cond(lambda n, i: n % i == 0)
    >>> count_factors(2)   # 1, 2
    2
    >>> count_factors(4)   # 1, 2, 4
    3
    >>> count_factors(12)  # 1, 2, 3, 4, 6, 12
    6

    >>> is_prime = lambda n, i: count_factors(i) == 2
    >>> count_primes = count_cond(is_prime)
    >>> count_primes(2)    # 2
    1
    >>> count_primes(3)    # 2, 3
    2
    >>> count_primes(4)    # 2, 3
    2
    >>> count_primes(5)    # 2, 3, 5
    3
    >>> count_primes(20)   # 2, 3, 5, 7, 11, 13, 17, 19
    8
    """
    def counter(n):
        i = 1
        count = 0
        while i <= n:
            if condition(n, i):
                count += 1
            i += 1
        return count
    return counter
```

One question that might be nice to ask is: in what ways is the logic for `count_factors` and `count_primes` similar, and in what ways are they different?

The answer to the first question can tell us the logic that we want to include in our `count_cond` function, while the answer to the second question can tell us where in `count_cond` we want to be able to have the difference in behavior observed between `count_factors` and `count_primes`.

It'll be helpful to also keep in mind that we want `count_cond` to return a function that, when an argument `n` is passed in, will behave similarly to `count_factors` or `count_primes`. In other words, `count_cond` is a higher order function that returns a function, that then contains the logic common to both `count_factors` and `count_primes`.

Use Ok to test your code:

```
python3 ok -q count_cond
```

Environment Diagram Practice

There is no Ok submission for this component.

However, we still encourage you to do these problems on paper to develop familiarity with Environment Diagrams, which might appear in an alternate form on the exam.

Q5: Make Adder

Draw the environment diagram for the following code:

```
n = 9
def make_adder(n):
    return lambda k: k + n
add_ten = make_adder(n+1)
result = add_ten(n)
```

There are 3 frames total (including the Global frame). In addition, consider the following questions:

1. In the Global frame, the name `add_ten` points to a function object. What is the intrinsic name of that function object, and what frame is its parent?
2. What name is frame `f2` labeled with (`add_ten` or λ)? Which frame is the parent of `f2` ?
3. What value is the variable `result` bound to in the Global frame?

You can check your work with the Online Python Tutor (http://tutor.cs61a.org), but try drawing the environment diagram on your own first.

You can try out the environment diagram at tutor.cs61a.org (http://tutor.cs61a.org). To see the environment diagram for this question, click here (https://goo.gl/axdNj5).

1. The intrinsic name of the function object that `add_ten` points to is λ (specifically, the lambda whose parameter is `k`). The parent frame of this lambda is `f1` .
2. `f2` is labeled with the name λ. The parent frame of `f2` is `f1` , since that is where λ is defined.
3. The variable `result` is bound to 19.

Q6: Lambda the Environment Diagram

Draw the environment diagram for the following code and predict what Python will output.

You can check your work with the Online Python Tutor (http://tutor.cs61a.org), but try drawing the environment diagram on your own first.

```
a = lambda x: x * 2 + 1
def b(b, x):
    return b(x + a(x))
x = 3
x = b(a, x)
```

Submit

Make sure to submit this assignment by running:

```
python3 ok --submit
```

Optional Questions

Q7: Composite Identity Function

Write a function that takes in two single-argument functions, `f` and `g`, and returns another **function** that has a single parameter `x`. The returned function should return `True` if `f(g(x))` is equal to `g(f(x))`. You can assume the output of `g(x)` is a valid input for `f` and vice versa. Try to use the `composer` function defined below for more HOF practice.

```
def composer(f, g):
    """Return the composition function which given x, computes f(g(x)).

    >>> add_one = lambda x: x + 1        # adds one to x
    >>> square = lambda x: x**2
    >>> a1 = composer(square, add_one)   # (x + 1)^2
    >>> a1(4)
    25
    >>> mul_three = lambda x: x * 3      # multiplies 3 to x
    >>> a2 = composer(mul_three, a1)    # ((x + 1)^2) * 3
    >>> a2(4)
    75
    >>> a2(5)
    108
    """
    return lambda x: f(g(x))

def composite_identity(f, g):
    """
    Return a function with one parameter x that returns True if f(g(x)) is
    equal to g(f(x)). You can assume the result of g(x) is a valid input for f
    and vice versa.

    >>> add_one = lambda x: x + 1        # adds one to x
    >>> square = lambda x: x**2
    >>> b1 = composite_identity(square, add_one)
    >>> b1(0)                            # (0 + 1)^2 == 0^2 + 1
    True
    >>> b1(4)                            # (4 + 1)^2 != 4^2 + 1
    False
    """
    def identity(x):
        return composer(f, g)(x) == composer(g, f)(x)
    return identity

    # Alternative solution
    return lambda x: f(g(x)) == g(f(x))
```

Solution using `composer`:

Calling `composer` will return us a function that takes in a single parameter `x`.

Here, the order in which we pass in the two functions `f` and `g` from `composite_identity` matters. `composer` will give us a function that first calls the second argument to `composer` on the input `x` (let's call this return value to be `y`), and we will then call the first argument to `composer` on this return value (aka on `y`), which is what we finally return.

We want to compare the results of `f(g(x))` with `g(f(x))`, so we will want to call `composer` and then pass in (as a separate argument) `x` to these composed functions in order to get a value to actually compare them.

Solution not using `composer`:

We can also directly call `f(g(x))` and `g(f(x))` instead of calling `composer`, and then compare the results of these two function calls.

Use Ok to test your code:

```
python3 ok -q composite_identity
```

Q8: I Heard You Liked Functions...

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 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
    """
    def ret_fn(n):
        def ret(x):
            i = 0
            while i < n:
                if i % 3 == 0:
                    x = f1(x)
                elif i % 3 == 1:
                    x = f2(x)
                else:
                    x = f3(x)
                i += 1
            return x
        return ret
    return ret_fn

    # Alternative solution
    def ret_fn(n):
        def ret(x):
            if n == 0:
```

```
            return x
        return cycle(f2, f3, f1)(n - 1)(f1(x))
    return ret
return ret_fn
```

There are three main pieces of information we need in order to calculate the value that we want to return.

1. The three functions that we will be cycling through, so `f1`, `f2`, `f3`.
2. The number of function applications we need, namely `n`. When `n` is 0, we want our function to behave like the identity function (i.e. return the input without applying any of our three functions to it).
3. The input that we start off with, namely `x`.

The functions are the parameters passed into `cycle`. We want the return value of `cycle` to be a function `ret_fn` that takes in `n` and outputs another function `ret`. `ret` is a function that takes in `x` and then will cyclically apply the three passed in functions to the input until we have reached `n` applications. Thus, most of the logic will go inside of `ret`.

Solution:

To figure out which function we should next use in our cycle, we can use the mod operation via `%`, and loop through the function applications until we have made exactly `n` function applications to our original input `x`.

Use Ok to test your code:

```
python3 ok -q cycle
```