Lab 2: Higher Order Functions

lab02.zip (lab02.zip)

Due at 11:59pm on Friday, 02/08/2019.

Starter Files

Download lab02.zip (lab02.zip). Inside the archive, you will find starter files for the questions in this lab, along with a copy of the Ok (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. Check that you have successfully submitted your code on okpy.org (https://okpy.org/).

- Questions 3-5 must be completed in order to receive credit for this lab. Starter code for question 3 is in lab02.py (lab02.py).
- Questions 6 and 7 (Environment Diagrams) are **optional**, but highly recommended. Try to work on at least one of these if you finish the required section early.
- Questions 8-10 are also **optional**. *It is recommended that you complete these problems on your own time.* Starter code for the questions are in lab02_extra.py (lab02_extra.py).

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.

Lambda Expressions

Environment Diagrams

Optional Questions

Checkoff

Q1: Print vs. Return

What would Python print? Try to figure it out before you type it into the interpreter!

```
>>> def condition_review(n):
...     if n == 4:
...         print("greetings")
...     elif n > 3:
...         return "farewell"
...
>>> condition_review(4)
______
```

Q2: Assignments

What would Python print? Try to figure it out before you type it into the interpreter!

```
>>> x = 1
>>> def f(x):
...     x += 1
...     return x
...
>>> a, b = f(x), x
>>> a
______

>>> b
______
```

Required Questions

What Would Python Display?

Q3: WWPD: Lambda the Free

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

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

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

```
>>> 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 expres
>>> print_lambda
______

>>> one_thousand = print_lambda(1000)
______

>>> one_thousand
______
```

Q4: WWPD: Higher Order Functions

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

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

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

```
>>> 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 lambda: x + y
...     else:
...         return x + y
>>> snake(10, 20)
______

>>> snake(10, 20)()
______

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

Coding Practice

Q5: 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 or refer to the textbook (http://composingprograms.com/pages/16-higher-order-functions.html#currying) if you're not sure what this means.

Your solution to this problem should fit entirely on the return line. You can try writing it first without this restriction, but rewrite it after in one line to test your understanding of this topic.

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

Use Ok to test your code:

```
python3 ok -q lambda_curry2
```

Environment Diagram Practice

There is no submission for this component. However, we still encourage you to do these problems on paper to develop familiarity with Environment Diagrams, which **will appear on the exam**.

Q6: 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. In frame `f2`, what name is the frame 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?

Q7: Lambda the Environment Diagram

Try drawing an environment diagram for the following code and predict what Python will output.

You do not need to submit or unlock this question through Ok. Instead, you can check your work with the Online Python Tutor (http://tutor.cs61a.org), but try drawing it yourself first!

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

Optional Questions

> **Note**: The following questions are in lab02_extra.py (lab02_extra.py).

Q8: 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. You may use the `compose1` function defined below.

```
def compose1(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 = compose1(square, add_one)   # (x + 1)^2
    >>> a1(4)
    25
    >>> mul_three = lambda x: x * 3      # multiplies 3 to x
    >>> a2 = compose1(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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q composite_identity
```

Q9: 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, count = 1, 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, count = 1, 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 counts all the numbers from 1 to `n` that satisfy `condition`.

```
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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q count_cond
```

Q10: 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
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q cycle
```

CS 61A (/)

Weekly Schedule (/weekly.html)

Office Hours (/office-hours.html)

Staff (/staff.html)

Resources (/resources.html)

Studying Guide (/articles/studying.html)

Debugging Guide (/articles/debugging.html)

Composition Guide (/articles/composition.html)

Policies (/articles/about.html)

Assignments (/articles/about.html#assignments)

Exams (/articles/about.html#exams)

Grading (/articles/about.html#grading)