Lab 2: Higher-Order Functions, Lambda Expressions **lab02.zip (lab02.zip)**

Due by 11:59pm on Wednesday, January 31.

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.

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.

Short Circuiting

Higher-Order Functions

Lambda Expressions

Environment Diagrams

Required Questions

Getting Started Videos

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: The Truth Will Prevail

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

```
python3 ok -q short-circuit -u
```

```
>>> True and 13
______

>>> False or 0
______

>>> not 10
______

>>> not None
______
```

```
>>> True and 1 / 0
______

>>> True or 1 / 0
______

>>> -1 and 1 > 0
______

>>> -1 or 5
______

>>> (1 + 1) and 1
______

>>> print(3) or ""
______
```

```
>>> def f(x):
...     if x == 0:
...         return "zero"
...     elif x > 0:
...         return "positive"
...     else:
...         return ""
>>> 0 or f(1)
______

>>> f(0) or f(-1)
______

>>> f(0) and f(-1)
______
```

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 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)
______
```

Q3: WWPD: Lambda

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 any output in the interactive 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, y: lambda: x + y  # Lambdas can return other lambdas!
>>> c = b(8, 4)
>>> c
______

>>> c()
______

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

```
>>> 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 # What did the call to print_lambda return?
______
```

Coding Practice

Q4: 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))` and `False` otherwise. You can assume the output of `g(x)` is a valid input for `f` and vice versa.

```
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          # squares x [returns 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
```

Q5: Count Cond

Consider the following implementations of `count_fives` and `count_primes` which use the `sum_digits` and `is_prime` functions from earlier assignments:

```
def count_fives(n):
    """Return the number of values i from 1 to n (including n)
    where sum_digits(n * i) is 5.
    >>> count_fives(10)  # Among 10, 20, 30, ..., 100, only 50 (10 * 5) has digit sum 5
    1
    >>> count_fives(50)  # 50 (50 * 1), 500 (50 * 10), 1400 (50 * 28), 2300 (50 * 46)
    4
    """
    i = 1
    count = 0
    while i <= n:
        if sum_digits(n * i) == 5:
            count += 1
        i += 1
    return count

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

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.

> **Note:** When we say `condition` is a predicate function, we mean that it is a function that will return `True` or `False`.

```
def sum_digits(y):
    """Return the sum of the digits of non-negative integer y."""
    total = 0
    while y > 0:
        total, y = total + y % 10, y // 10
    return total

def is_prime(n):
    """Return whether positive integer n is prime."""
    if n == 1:
        return False
    k = 2
    while k < n:
        if n % k == 0:
            return False
        k += 1
    return True

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_fives = count_cond(lambda n, i: sum_digits(n * i) == 5)
    >>> count_fives(10)   # 50 (10 * 5)
    1
    >>> count_fives(50)   # 50 (50 * 1), 500 (50 * 10), 1400 (50 * 28), 2300 (50 * 46)
    4

    >>> is_i_prime = lambda n, i: is_prime(i) # need to pass 2-argument function into cour
    >>> count_primes = count_cond(is_i_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
```

Check Your Score Locally

You can locally check your score on each question of this assignment by running

```
python3 ok --score
```

This does NOT submit the assignment! When you are satisfied with your score, submit the assignment to Gradescope to receive credit for it.

Submit

Submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment.** Lab 00 (https://cs61a.org/lab/lab00/#submit-with-gradescope) has detailed instructions.

In addition, all students who are **not** in the mega lab must complete this attendance form (https://go.cs61a.org/lab-att). Submit this form each week, whether you attend lab or missed it for a good reason. The attendance form is not required for mega section students.

Environment Diagram Practice

There is no Gradescope submission for this component.

However, we still encourage you to do this problem on paper to develop familiarity with Environment Diagrams, which might appear in an alternate form on the exam. To check your work, you can try putting the code into PythonTutor.

Q6: HOF Diagram Practice

Draw the environment diagram that results from executing the code below on paper or a whiteboard. Use tutor.cs61a.org (https://tutor.cs61a.org) to check your work.

```
n = 7

def f(x):
    n = 8
    return x + 1

def g(x):
    n = 9
    def h():
        return x + 1
    return h

def f(f, x):
    return f(x + n)

f = f(g, n)
g = (lambda y: y())(f)
```

Optional Questions

> These questions are optional. If you don't complete them, you will still receive credit for lab. They are great practice, so do them anyway!

Q7: Multiple

Write a function that takes in two numbers and returns the smallest number that is a multiple of both.

```
def multiple(a, b):
    """Return the smallest number n that is a multiple of both a and b.

    >>> multiple(3, 4)
    12
    >>> multiple(14, 21)
    42
    """
    "*** YOUR CODE HERE ***"
```

Use Ok to test your code:

```
python3 ok -q multiple
```

Q8: I Heard You Liked Functions...

Define a function `cycle` that takes in three functions `f1`, `f2`, and `f3`, as arguments. `cycle` will return another function `g` that should take in an integer argument `n` and return another function `h`. That final function `h` 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 `h` 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
```