Lab 2: Higher-Order Functions, Lambda Expressions

> Adapted from cs61a of UC Berkeley.

Starter Files

Get your starter file by cloning the repository: https://github.com/JacyCui/sicp-lab02.git

```
git clone https://github.com/JacyCui/sicp-lab02.git
```

`lab01.zip` is the starter file you need, you might need to unzip the file to get the skeleton code.

```
unzip lab02.zip
```

`README.md` is the handout for this homework. `solution` is a probrab solution of the lab. However, I might not give my solution exactly when the lab is posted. You need to finish the task on your own first. If any problem occurs, please make use of the comment section.

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

Lambda expressions are expressions that evaluate to functions by specifying two things: the parameters and a return expression.

```
lambda <parameters>: <return expression>
```

While both `lambda` expressions and `def` statements create function objects, there are some notable differences. `lambda` expressions work like other expressions; much like a mathematical expression just evaluates to a number and does not alter the current environment, a `lambda` expression evaluates to a function without changing the current environment. Let's take a closer look.

	lambda	**def**
Type	*Expression* that evaluates to a value	*Statement* that alters the environment
Result of execution	Creates an anonymous lambda function with no intrinsic name.	Creates a function with an intrinsic name and binds it to that name in the current environment.
Effect on the environment	Evaluating a `lambda` expression does *not* create or modify any variables.	Executing a `def` statement both creates a new function object *and* binds it to a name in the current environment.
Usage	A `lambda` expression can be used anywhere that expects an expression, such as in an assignment statement or as the operator or operand to a call expression.	After executing a `def` statement, the created function is bound to a name. You should use this name to refer to the function anywhere that expects an expression.

Example:

- **lambda**

```
# A lambda expression by itself does not alter
# the environment
lambda x: x * x

# We can assign lambda functions to a name
# with an assignment statement
square = lambda x: x * x
square(3)

# Lambda expressions can be used as an operator
# or operand
negate = lambda f, x: -f(x)
negate(lambda x: x * x, 3)
```

- **def**

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

# A function created by a def statement
# can be referred to by its intrinsic name
square(3)
```

Environment Diagrams

> Python Tutor: https://pythontutor.com/composingprograms.html#mode=edit

Environment diagrams are one of the best learning tools for understanding `lambda` expressions and higher order functions because you're able to keep track of all the different names, function objects, and arguments to functions. We highly recommend drawing environment diagrams or using Python tutor if you get stuck doing the WWPD problems below. For examples of what environment diagrams should look like, try running some code in Python tutor. Here are the rules:

Assignment Statements

1. Evaluate the expression on the right hand side of the `=` sign.
2. If the name found on the left hand side of the `=` doesn't already exist in the current frame, write it in. If it does, erase the current binding. Bind the *value* obtained in step 1 to this name.

If there is more than one name/expression in the statement, evaluate all the expressions first from left to right before making any bindings.

> Python Tutor Demo

def Statements

1. Draw the function object with its intrinsic name, formal parameters, and parent frame. A function's parent frame is the frame in which the function was defined.
2. If the intrinsic name of the function doesn't already exist in the current frame, write it in. If it does, erase the current binding. Bind the newly created function object to this name.

> Python Tutor Demo

Call expressions

> Note: you do not have to go through this process for a built-in Python function like `max` or `print`.

1. Evaluate the operator, whose value should be a function.
2. Evaluate the operands left to right.
3. Open a new frame. Label it with the sequential frame number, the intrinsic name of the function, and its parent.
4. Bind the formal parameters of the function to the arguments whose values you found in step 2.
5. Execute the body of the function in the new environment.

> Python Tutor Demo

Lambdas

> *Note:* As we saw in the `lambda` expression section above, `lambda` functions have no intrinsic name. When drawing `lambda` functions in environment diagrams, they are labeled with the name `lambda` or with the lowercase Greek letter λ. This can get confusing when there are multiple lambda functions in an environment diagram, so you can distinguish them by numbering them or by writing the line number on which they were defined.

1. Draw the lambda function object and label it with λ, its formal parameters, and its parent frame. A function's parent frame is the frame in which the function was defined.

This is the only step. We are including this section to emphasize the fact that the difference between `lambda` expressions and `def` statements is that `lambda` expressions *do not* create any new bindings in the environment.

> Python Tutor Demo

Required Questions

What Would Python Display?

Q1: WWPD: Lambda the Free

> Use Ok to test your knowledge with the following "What Would Python Display?" questions:
>
> ```
> python3 ok -q lambda -u --local
> ```
>
> 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)
______
```

```
>>> 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 expression executed?
>>> 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 --local
> ```
>
> 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 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 for more details about currying.

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.

```
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
    """
    "*** YOUR CODE HERE ***"
    return ______
```

Use Ok to test your code:

```
python3 ok -q lambda_curry2 --local
```

Q4: Count van Count

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

```python
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 takes in `n`, which counts all the numbers from 1 to `n` that satisfy `condition` when called.

```python
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 --local
```

Environment Diagram Practice

There is no test 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 try out the environment diagram at python tutor. To see the environment diagram for this question, click here.

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

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

You do not need to test or unlock this question through Ok. Instead, you can check your work with the Online Python Tutor, 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

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 `compose1` function defined below for more HOF practice.

```
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 --local
```

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.

```python
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 --local
```

In the end, you can use doctest module to run all your doctest.

```
python3 -m doctest lab02.py
```