

Lab 2: Higher-Order Functions, Lambda Expressions

[**lab02.zip \(lab02.zip\)**](#)

Due by 11:59pm on Wednesday, September 11.

Starter Files

Download [lab02.zip \(lab02.zip\)](#).

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](#)

Short Circuiting

What do you think will happen if we type the following into Python?

```
1 / 0
```

Try it out in Python! You should see a `ZeroDivisionError`. But what about this expression?

```
True or 1 / 0
```

It evaluates to `True` because Python's `and` and `or` operators *short-circuit*. That is, they don't necessarily evaluate every operand.

Operator	Checks if:	Evaluates from left to right up to:	Example
AND	All values are true	The first false value	<code>False and 1 / 0</code> evaluates to <code>False</code>
OR	At least one value is true	The first true value	<code>True or 1 / 0</code> evaluates to <code>True</code>

Short-circuiting happens when the operator reaches an operand that allows them to make a conclusion about the expression. For example, `and` will short-circuit as soon as it reaches the first false value because it then knows that not all the values are true.

If `and` and `or` do not *short-circuit*, they just return the last value; another way to remember this is that `and` and `or` always return the last thing they evaluate, whether they short circuit or not. Keep in mind that `and` and `or` don't always return booleans when using values other than `True` and `False`.

Higher-Order Functions

Variables are names bound to values, which can be primitives like `3` or `'Hello World'`, but they can also be functions. And since functions can take arguments of any value, other functions can be passed in as arguments. This is the basis for higher-order functions.

A higher order function is a function that manipulates other functions by taking in functions as arguments, returning a function, or both. We will introduce the basics of higher order functions in this lab and will be exploring many applications of higher order functions in our next lab.

Functions as arguments

In Python, function objects are values that can be passed around. We know that one way to create functions is by using a `def` statement:

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

The above statement created a function object with the intrinsic name `square` as well as binded it to the name `square` in the current environment. Now let's try passing it as an argument.

First, let's write a function that takes in another function as an argument:

```
def scale(f, x, k):
    """ Returns the result of f(x) scaled by k. """
    return k * f(x)
```

We can now call `scale` on `square` and some other arguments:

```
>>> scale(square, 3, 2) # Double square(3)
18
>>> scale(square, 2, 5) # 5 times 2 squared
20
```

Note that in the body of the call to `scale`, the function object with the intrinsic name `square` is bound to the parameter `f`. Then, we call `square` in the body of `scale` by calling `f(x)`.

As we saw in the above section on `lambda` expressions, we can also pass `lambda` expressions into call expressions!

```
>>> scale(lambda x: x + 10, 5, 2)
30
```

In the frame for this call expression, the name `f` is bound to the function created by the `lambda` expression `lambda x: x + 10`.

Functions that return functions

Because functions are values, they are valid as return values! Here's an example:

```
def multiply_by(m):
    def multiply(n):
        return n * m
    return multiply
```

In this particular case, we defined the function `multiply` within the body of `multiply_by` and then returned it. Let's see it in action:

```
>>> multiply_by(3)
<function multiply_by.<locals>.multiply at ...>
>>> multiply(4)
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
NameError: name 'multiply' is not defined
```

A call to `multiply_by` returns a function, as expected. However, calling `multiply` errors, even though that's the name we gave the inner function. This is because the name `multiply` only exists within the frame where we evaluate the body of `multiply_by`.

So how do we actually use the inner function? Here are two ways:

```
>>> times_three = multiply_by(3) # Assign the result of the call expression to a name
>>> times_three(5) # Call the inner function with its new name
15
>>> multiply_by(3)(10) # Chain together two call expressions
30
```

The point is, because `multiply_by` returns a function, you can use its return value just like you would use any other function.

Lambda Expressions

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	<i>Expression</i> that evaluates to a value	<i>Statement</i> 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 <code>lambda</code> expression does <i>not</i> create or modify any variables.	Executing a <code>def</code> statement both creates a new function object <i>and</i> 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 <code>def</code> 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	<pre># 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)</pre>	<pre>def square(x): return x * x # A function created by a def statement # can be referred to by its intrinsic name square(3)</pre>



YouTube link (https://youtu.be/vCeNq_P3akI?list=PLx38hZJ5RLZcUPWZ1-3HYsRPgZ8OCrvqz)

Environment Diagrams

Environment Diagrams

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](https://tutor.cs61a.org) (<https://tutor.cs61a.org>) if you get stuck doing the WWP 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.

```

1 x = 10
→ 2 y = x
→ 3 x = 20
4 x, y = y + 1, x - 1

```

Global frame	
x	10
y	10

[Edit Code & Get AI Help](#)

→ line that just executed
→ next line to execute

< Prev Next >

Step 3 of 4

Visualized with pythontutor.com

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 3.6

```

1 def f(x):
2     return x + 1
3
→ 4 def g(y):
5     return y - 1
6
→ 7 def f(z):
8     return x * 2

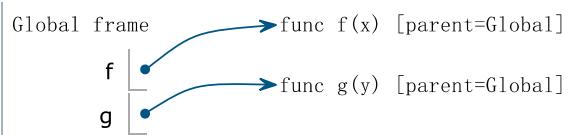
```

[Edit Code & Get AI Help](#)

→ line that just executed
→ next line to execute

Frames

Objects



< Prev Next >

Step 3 of 3

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 3.6

```

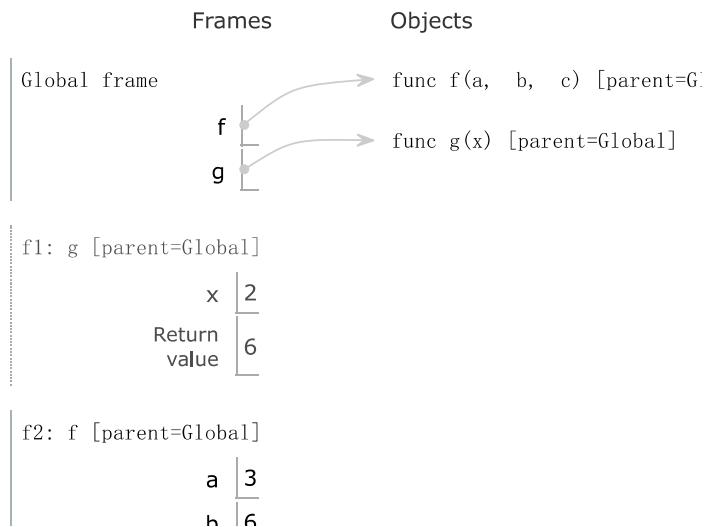
1 def f(a, b, c):
2     return a * (b + c)
3
4 def g(x):
5     return 3 * x
6
7 f(1 + 2, g(2), 6)

```

[Edit Code & Get AI Help](#)

Line that just executed
next line to execute

< Prev Next >
Step 7 of 8



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 3.6

```
1 lambda x: x * x #no binding created
2 square = lambda x: x * x
3 square(4) #calling a lambda function
```

Global frame

Frames Objects

square → function $\lambda(x)$ <line 2>

x λ <line 2>

x 4

Return value 16

◀ ▶

Edit Code & Get AI Help

line that just executed
next line to execute

< Prev Next >

Step 6 of 6

Visualized with pythontutor.com



YouTube link (<https://youtu.be/lPec2A7j2bY?list=PLx38hZJ5RLZcUPWZ1-3HYsRPgZ8OCrvqz>).

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(g)(2) # 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 # 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 count_cond
    >>> 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 Assignment

If you are in a regular section of CS 61A, fill out this [lab attendance and feedback form](#) (<https://forms.gle/dHxj8gttNWRY6Ptm9>). (If you are in the mega section, you don't need to fill out the form.)

Then, submit this assignment by uploading any files you've edited **to the appropriate Gradescope assignment**. [Lab 00 \(./lab00/#submit-with-gradescope\)](#) has detailed instructions.

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 this assignment. 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
```

