

# LAB 02

---

## Higher-Order Functions, Lambda Expressions

Mingxiao Wei

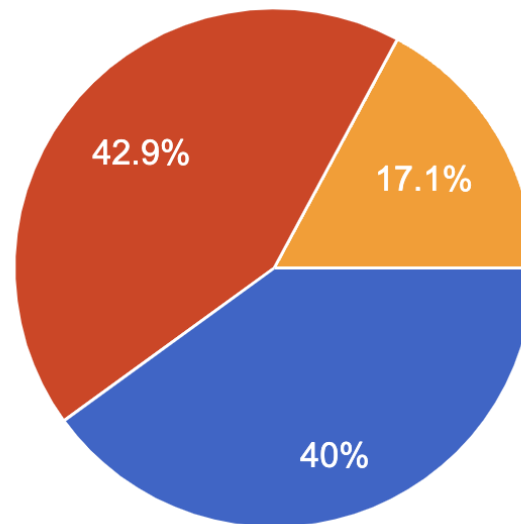
[mingxiaowei@berkeley.edu](mailto:mingxiaowei@berkeley.edu)

Jan 31, 2022

# FROM LAST TIME... 👁️👁️

Are you a morning bird or night owl or ...?


35 responses



- Morning bird 🐣
- Night owl 🦉
- Constantly awake 🍷

# LOGISTICS

---

- Lab 02 due tomorrow 02/01
- Homework 02 due Thursday 02/02
- Hog is released! 
  - You can choose to work alone or in group of 2
    - If you work with a partner, only one of you need to submit and add the other on Gradescope!
  - Checkpoint 1 due next Tue 02/07
  - The entire project due next Fri 02/10
  - Submit everything by next Thu 02/09 for one extra point!
- Sign up for [tutoring sections](#) or [CSM sections](#)
  - Not required, but a good place to go if you want more practice!

# ABOUT MIDTERM 1

---

Yes, it's coming ...

- **Default: Next Monday (02/06) 7-9pm, in-person, right-handed desk**
- If you need any type of accommodations (left-handed desk, remote exam, alternate time, extended time, etc.), please [fill out this form](#) by Thu 02/02!!
- Check out [Ed post #376](#) for more details!
- Pro Tips
  - Make sure you are (somewhat) comfortable with all the topics in scope
  - Familiarize yourself with the [study guide](#) beforehand
  - Do [past exams](#)
    - For each question, make sure you understand it before moving on to the next exam
    - Consult walkthrough videos! They are super helpful
    - Post on Ed [thread](#) / come to OH for help

# SHORT-CIRCUITING



# SHORT CIRCUITING

- Short circuit - not every operand gets evaluated
- **and**
  - evaluates from left to right until the first **FALSY** value or the last value
  - return the last thing that's evaluated
- **or**
  - evaluates from left to right until the first **TRUTHY** value or the last value
  - return the last thing that's evaluated
- If an error occurs, the execution flow is terminated immediately.

```
>>> True and 1 / 0 and 2
ZeroDivisionError
>>> True or 1 / 0 or 2
True
```

```
>>> 1 and 2 and 3
3
>>> 1 or 2 or 3
1
```

# LAMBDA EXPRESSIONS

---



# LAMBDA EXPRESSIONS

- A lambda expression evaluates to **function values** but does not bind it to a name, unless we bind it to some names using assignment statements
- Anatomy:

```
f = lambda <p1>, <p2>, ...: <returned expr>
```

This is equivalent to (though not the same as)

```
def f(<p1>, <p2>, ...):  
    return <returned expr>
```

- Similarly, the return expression of a lambda function is not evaluated until the lambda is called
- Unlike `def` statements, lambda expressions can be used as an operator or an operand to a call expression. This is because they are simply one-line expressions that **evaluate to functions**



# LAMBDA EXPRESSIONS

	<code>lambda</code>	<code>def</code>
Type	Expression that evaluates to a value	Statement that alters the environment
Results of execution	Creates an anonymous <code>lambda</code> 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	Does not create or modify any variables.	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 <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.

# LAMBDA EXPRESSIONS

---

*# 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)
```

# HIGHER ORDER FUNCTIONS

---



# HIGHER ORDER FUNCTIONS

---

- In Python, function objects are values that can be passed around
- A **higher order function (HOF)** is a function that manipulates other functions by taking in functions as arguments, returning a function, or both.

```
1  def composer(func1, func2):
2      """Return a function f, such that f(x) = func1(func2(x))."""
3      def f(x):
4          return func1(func2(x))
5      return f
```

# FUNCTIONS AS ARGUMENTS

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

- This created a function object with the intrinsic name `square` as well as binded it to the name `square` in the current environment.

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

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

- 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)`

# FUNCTIONS THAT RETURN FUNCTIONS

- Since functions are values, they are also valid as return values!

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

```
>>> 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
```

- The name `multiply` only exists within the frame where we evaluate the body of `multiply_by`

# FUNCTIONS THAT RETURN FUNCTIONS

---

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

- To use the inner function that `multiply_by(3)` returns, we can either store the function value in a variable and use this variable as a function later, or chain two call expressions

# ENVIRONMENT DIAGRAMS

---





# ASSIGNMENT STATEMENTS

---

1. Evaluate the expression on the RHS of the assignment operator (the single equal sign) from left to right
2. Bind the variable names on the LHS to the resulting values in the current frame
  - If the variable name doesn't exist in the current frame, create a new variable and binds it to the value on the RHS; otherwise the previous binding is overwritten

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

```
a, b = 1, 2
```

```
a, b = b, a
```

*# In Python, you can swap the values of multiple variables in one line*

```
a, b = b + 1, a + b
```

# DEF STATEMENTS

---

1. Draw the function object with its intrinsic name, formal parameters, and parent frame
  - **parent frame** = the frame where 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.

# DEF STATEMENTS

Python 3.6

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

[Edit this code](#)

→ line that just executed

→ next line to execute



< Prev

Next >

Step 1 of 3

Visualized with [pythontutor.com](http://pythontutor.com)

NEW: [subscribe](#) to our YouTube

[Move and hide objects](#)

Frames

Objects

# CALL EXPRESSIONS

---

1. Evaluate the operator, which should evaluate to a function value.
2. Evaluate the operands from left to right.
3. Draw a new frame, labelling it with the following:
  - A unique index (`f1`, `f2`, `f3`, ...)
  - The intrinsic name of the function, which is also the name of the function object itself
  - The parent frame (`[parent = ...]`)
4. Bind the formal parameters to the argument values obtained in step #2
5. Evaluate the body of the function in this new frame until a return value is obtained. Write down the return value in the frame.

\* If a function does not have a return value, it implicitly returns `None`

\* For built-in or imported functions like `abs` and `add`, we do not need to draw a new frame when calling them

# LAMBDA EXPRESSIONS

---

1. Draw the lambda function object and label it with  $\lambda$ , its formal parameters, and its parent frame
  - **parent frame** = the frame where the function was defined

\* **lambda** functions have no intrinsic name - in environment diagrams, they are labeled with the name **lambda** or with the lowercase Greek letter  $\lambda$

\* When multiple lambda functions are present, label the line number on which the lambda function is defined to distinguish them

\* As opposed to **def** statements, a **lambda** expression by itself does not create any new bindings in the environment

# NOW IT'S LAB TIME 🤠

---

- Get started on the lab and raise your hand whenever you need help!
- Get to know your neighbors and collaborate if you'd like!
- Slides: [go.cs61a.org/mingxiao-index](https://go.cs61a.org/mingxiao-index)
- Leave any anonymous feedback here: [go.cs61a.org/mingxiao-anon](https://go.cs61a.org/mingxiao-anon)

