Lecture 2: Names & Functions

June 21st, 2021

Alex Kassil

Announcements

- Lab 0 released, due Friday
- Lab 1 released, due Thursday
  - Many office hours to help you get setup https://cs61a.org/office-hours/
  - Lab Party Wednesday/Thursday
  - Can be done after watching Wednesday's lecture
- Discussions start today (maybe you already had yours?)
- Vitamin 0 released latest due date Thursday 8am
  - We want you to finish them by monday night and wednesday night after lecture but before discussion the next day
  - Since first discussion is next day 8am PT, we will accept up to that time
- Homework 1 released, due Wednesday 6/30
  - We released it early, to give students who are deciding between CS61A and CS10 a chance to see the problems
  - Q1-Q4 can be done after watching Wednesday's Lecture + Lab 01
  - Q5-Q6 can be done after watching Thursday's Lecture + Lab 02
- Small group tutoring starts wednesday
  - Sign-up will be released tonight and emailed out using Ed
- Live Ed thread links.cs61a.org/ed-lec2

Programming Assignment FAQs

- Which assignments are programming assignments?
  - Labs, Homeworks, and Projects
- How many times can we test our code?
  - As many times as you want! You can keep running the autograder locally with `python3 ok`
- Are there any hidden tests?
  - No, unless explicitly noted
  - For most, maybe all programming assignments this semester, there will be no hidden tests
- When will I get my grade for these?
  - You can run `python3 ok --score` to see if you missed anything
  - We will release grades with an email announcement from Ed somepoint after the assignment is due
- When will howamidoing.cs61a.org be updated?
  - After we release the first batch of grades

Types of Expressions

Primitive Expressions:

```
2        'hello!'        add
```

numbers | strings | names

Call Expressions:

```
add ( 2 , 3 )
```

Operator | Operand | Operand

```
max(5 * min(-1, 4), add(2, 3))
```

An operand can also be a call expression

Review - Evaluating Call Expressions

1. Evaluate the operator expression
2. Evaluate each operand expression, left to right
3. Apply the value of the operator expression to the values of the operand expressions

Values

Programs manipulate values

Values represent different types of data

Integers: `2` `44` `-3`

Strings: `'hello!'` `'cs61a'`

Floats: `3.14` `4.5` `-2.0`

Booleans: `True` `False`

Expressions & Values

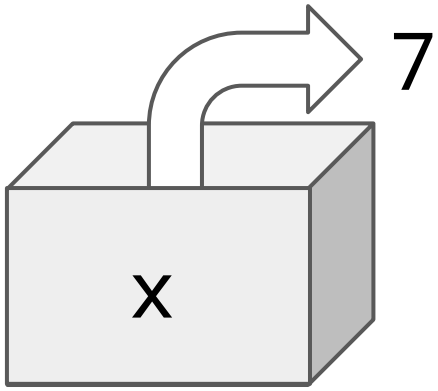
Expressions evaluate to values in one or more steps

Expression:		**Value:**
`'hello'`	→	`'hello'`
`7 / 2`	→	`3.5`
`add(1, max(2, 3))`	→	`4`

Names

Values can be assigned to **names** to make referring to them easier.

A name can only be bound to a single value.



One way to introduce a new name in a program is with an **assignment statement**.

```
x = 1 + 2 * 3 - 4 // 5
```

Name Expression

Statements affect the program, but do not evaluate to values.

Demo

Names, Assignment, and User-Defined Functions

Discussion Question 1

What is the value of the final expression in this sequence?

```
>>> f = min
>>> f = max
>>> g, h = min, max
>>> max = g
>>> max(f(2, g(h(1, 5), 3)), 4)
```

5 minute break to solve this on your own! Use a piece of paper or do it in your head

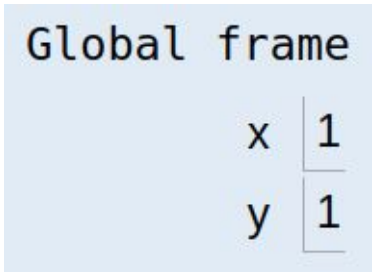
Environment Diagrams

links.cs61a.org/lec2-ex1

Visualizing Assignment

Names are bound to **values** in an environment

```
1  x = 1
2  y = x
3  x = 2
4  x, y = x + y, x
```



Code (left):

Statements and expressions
Arrows indicate evaluation order
Green = just executed, Red = next

Frames (right):

Each name is bound to a value
Within a frame, a name cannot be repeated

To execute an assignment statement:

1. **Evaluate** the expression to the right of **=**.
2. **Bind** the value of the expression to the name to the left of **=** in the current environment.

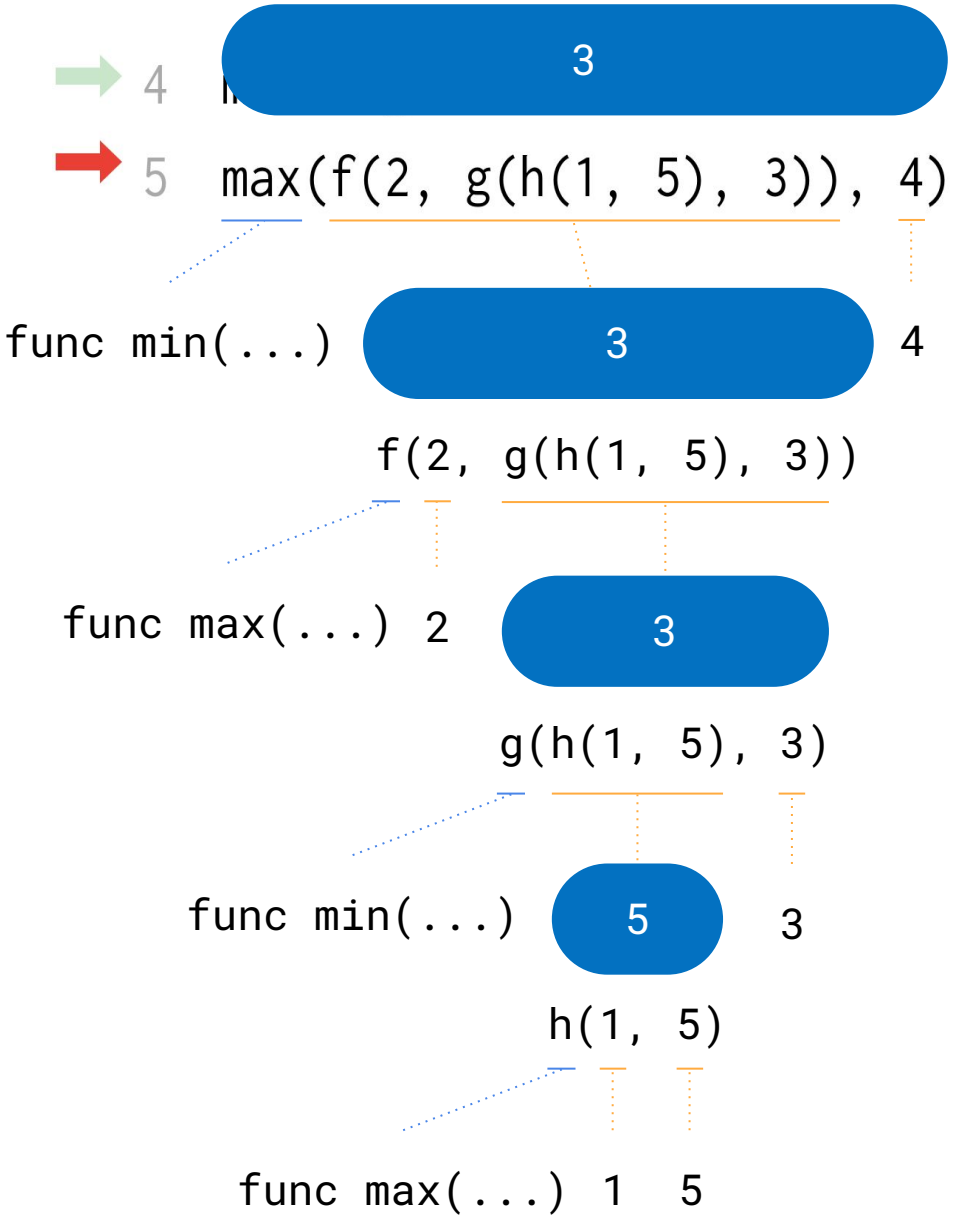
Discussion Question 1

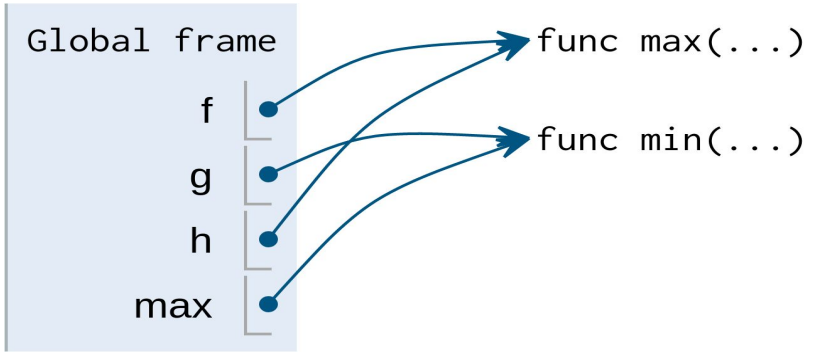
What is the value of the final expression in this sequence?

links.cs61a.org/lec2-ex2

```
>>> f = min
>>> f = max
>>> g, h = min, max
>>> max = g
>>> max(f(2, g(h(1, 5), 3)), 4)
```

Demo


4
5
3
max(f(2, g(h(1, 5), 3)), 4)
func min(...)
3
4
f(2, g(h(1, 5), 3))
func max(...) 2
3
g(h(1, 5), 3)
func min(...)
5
3
h(1, 5)
func max(...) 1 5


Global frame
f
g
h
max
func max(...)
func min(...)


3

Break

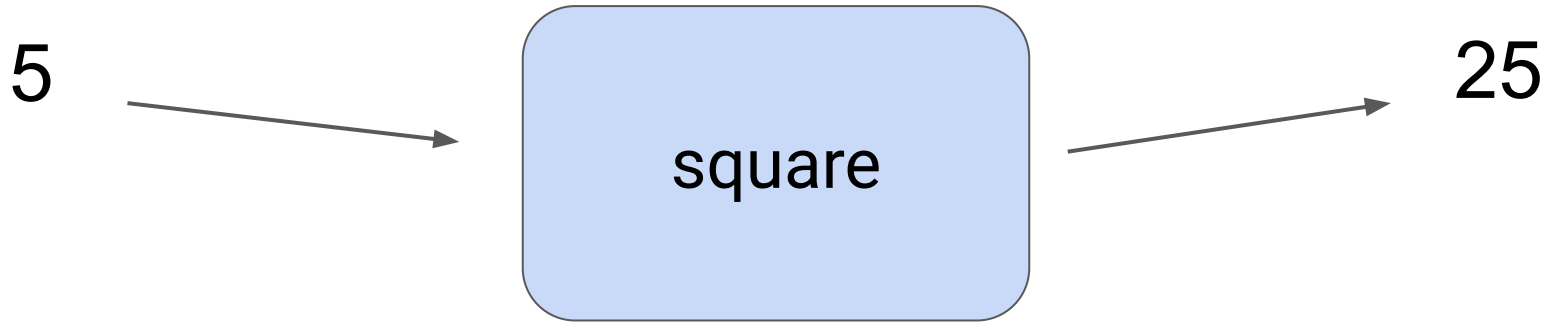
Functions

Functions

Functions allow us to abstract away entire expressions and sequences of computation

They take in some input (known as their arguments) and transform it into an output (the return value)

We can create functions using def statements. Their input is given in a function call, and their output is given by a return statement.



Defining Functions

Function **signature** indicates name and number of arguments

```
def <name>(<parameters>):
    return <return expression>
```

Function **body** defines the computation performed when the function is applied

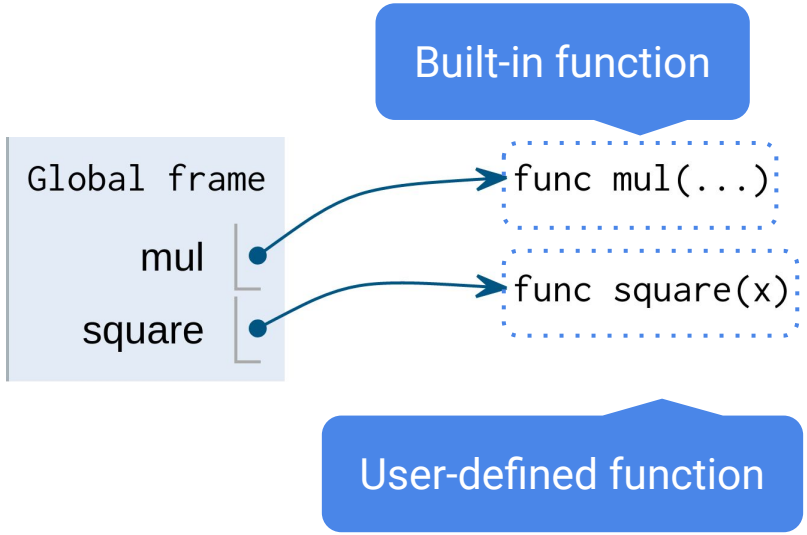
```
def square(x):
    return x * x
y = square(-2)
```

Execution rule for `def` statements

1. Create a function with signature `<name>(<parameters>)`
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame

Functions in Environment Diagrams

```
1  from operator import mul
2  def square(x):
3      return mul(x, x)
4  y = square(-2)
```



links.cs61a.org/lec2-ex3

def statements are a type of assignment that bind names to function values

Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

1. Create a new **environment frame**
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

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

square(-2)
```

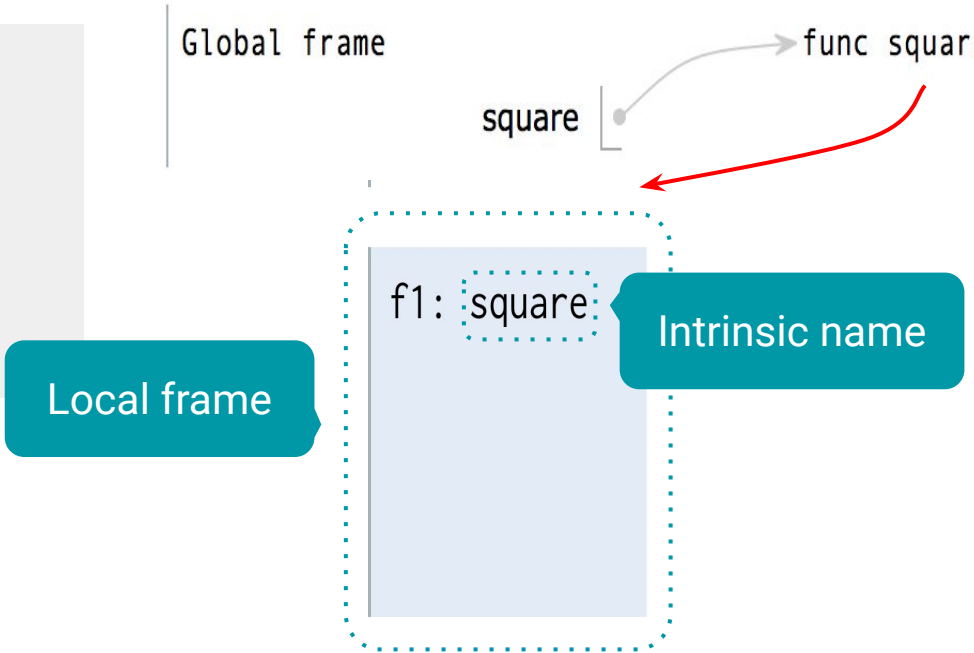
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

1. Create a new **environment frame**
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

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

square(-2)
```



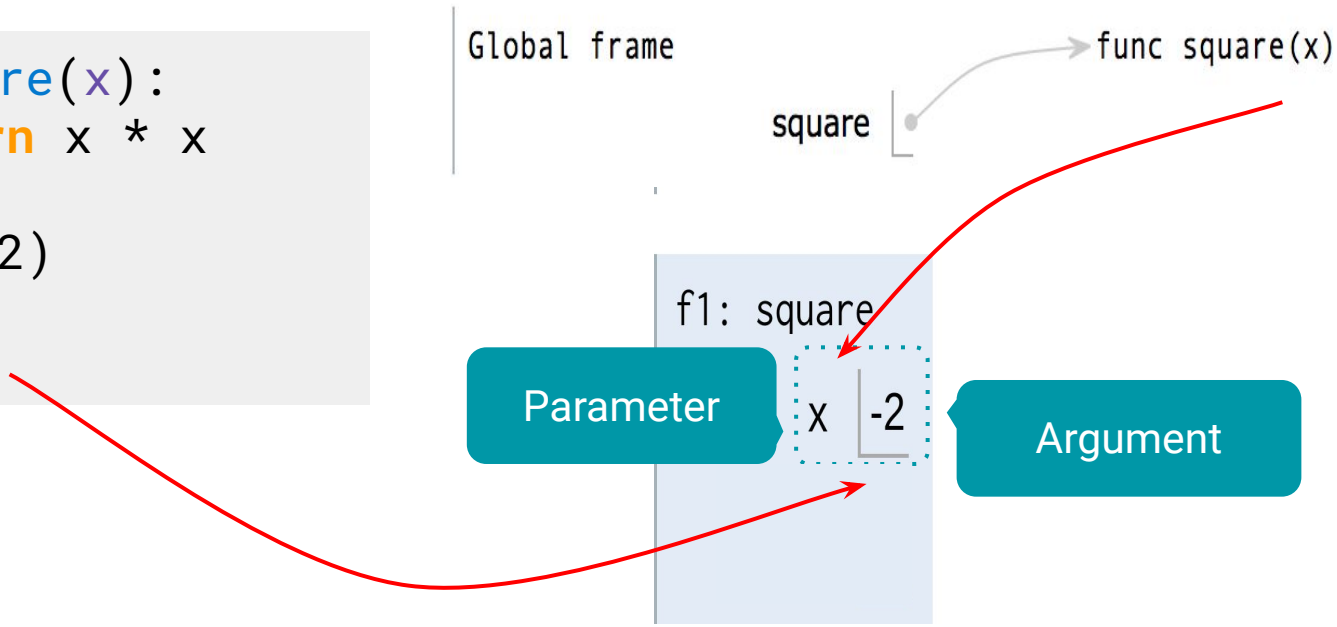
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

1. Create a new **environment frame**
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

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

square(-2)
```



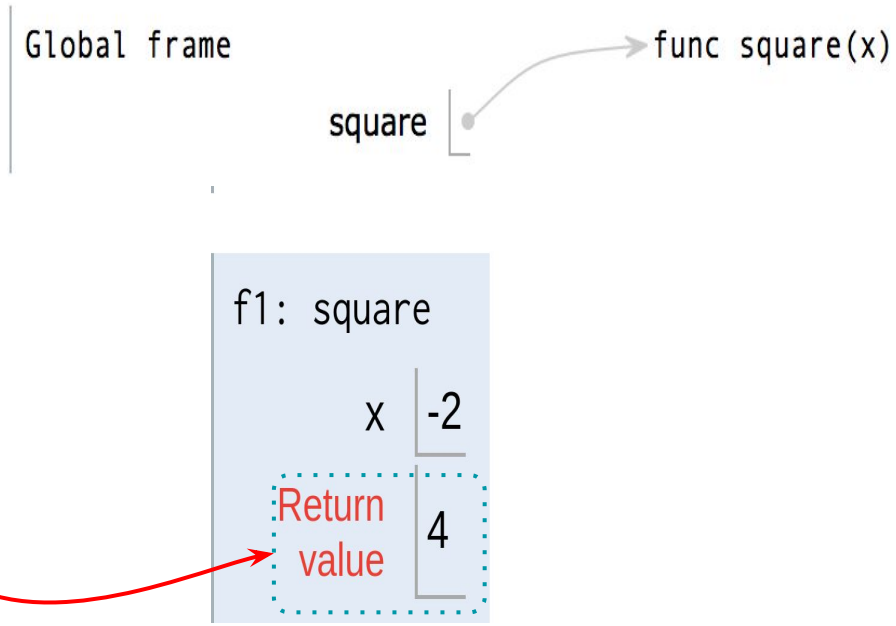
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (for now)

1. Create a new **environment frame**
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

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

square(-2)
```



Calling User-Defined Functions

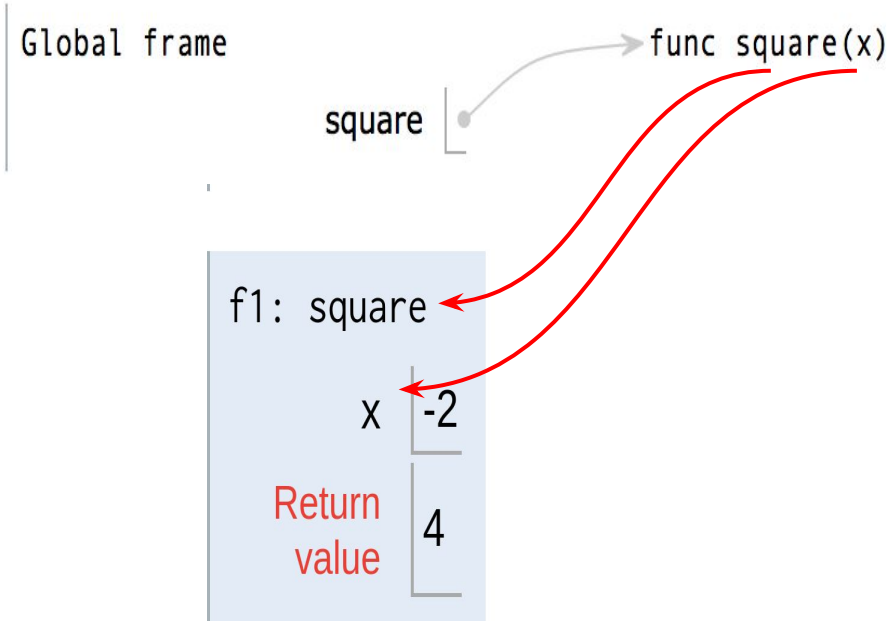
Procedure for calling/applying user-defined functions (for now)

1. Create a new **environment frame**
2. Bind the function's parameters to its arguments in that frame
3. Execute the body of the function in the new environment

A function's ***signature*** has all the information needed to create a local frame.

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

square(-2)
```



Putting it all together

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

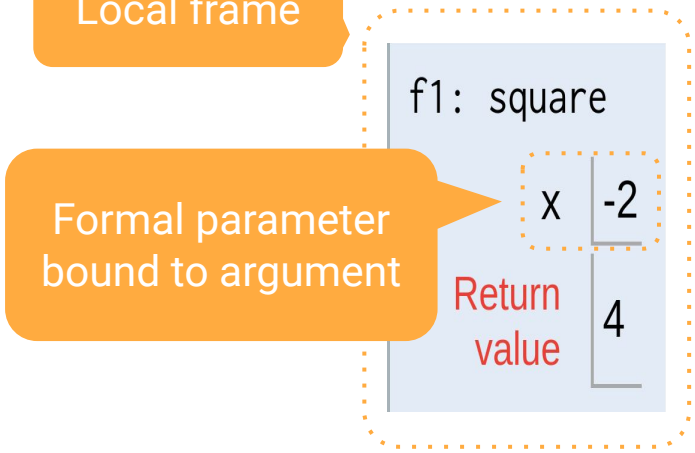
1. Evaluate the operator expression
2. Evaluate each operand expression, left to right
3. Apply the value of the operator expression to the values of the operand expressions

Operator: `square`
Function: `func square(x)`

```
square(1 - 3)
```

Operand: 1-3
Argument: -2

Local frame



Formal parameter bound to argument

Names & Environments

Demo

- Every expression is evaluated in the context of an environment.
- An **environment** is a ***sequence*** of frames
- So far, there have been two possible environments:
  - The global frame
  - A function's local frame, then the global frame

Rules for looking up names in user-defined functions (version 1)

1. Look it up in the local frame
2. If name isn't in local frame, look it up in the global frame
3. If name isn't in either frame, NameError

links.cs61a.org/lec2-ex4

Drawing Environment Diagrams

- Option 1: Python Tutor (tutor.cs61a.org)
  - Useful for quick visualization or for environment diagram questions
- Option 2: 61A Code (code.cs61a.org)
  - Includes an integrated editor/interpreter
  - Good for more complicated code or if you want to debug

Summary

- Programs consist of **statements**, or instructions for the computer, containing **expressions**, which describe computation and evaluate to **values**.
- Values can be assigned to **names** to avoid repeating computations.
- An **assignment statement** assigns the value of an expression to a name in the current **environment**.
- **Functions** encapsulate a series of statements that maps **arguments** to a **return value**.
- A **`def` statement** creates a function object with certain **parameters** and a **body** and binds it to a name in the current environment.
- A **call expression** applies the value of its **operator**, a function, to the value(s) or its **operand**(s), some arguments.