Lecture 2: Names & Functions

June 22, 2022

Richard Roggenkemper

Announcements

- Discussions happened yesterday and will happen tomorrow!
- Sign up for a section if you haven't already
  - We are working on fixing some balance issues
- Lab 0 released, due Friday
- Only 1 lab assignment this week
- No Homework this week!
- Lecture recordings will be up on Bcourses first (before the website)
- We will have lecture on Friday (11am in Dwinelle 155)

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

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 date types

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

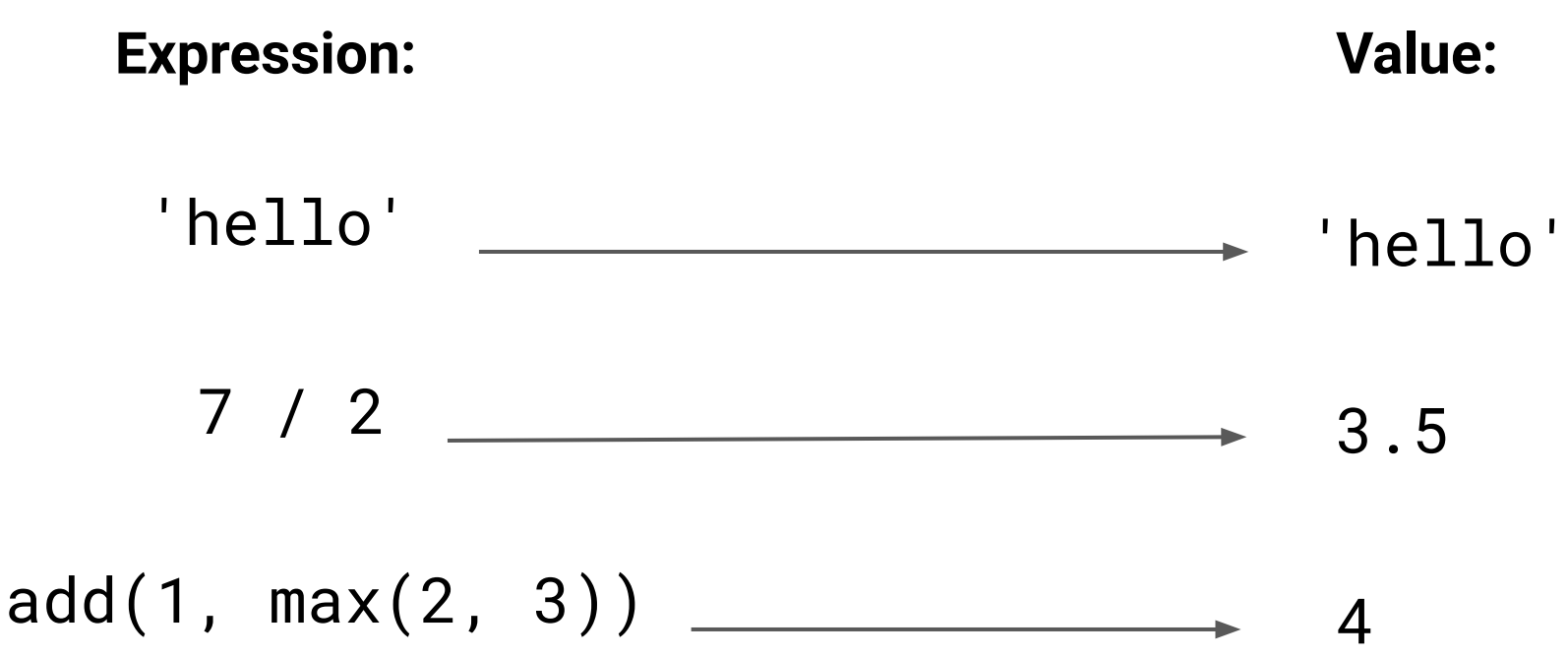
Strings: `'it's pizza time'` `'cs61a'`

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

Booleans: `True` `False`

Expressions & Values

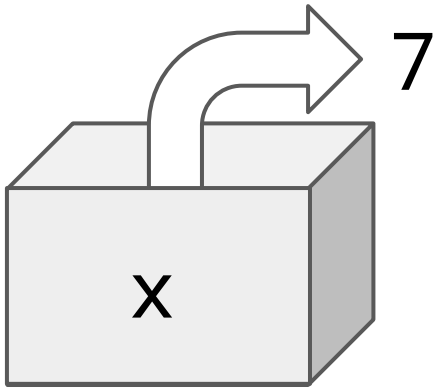
Expressions evaluate to values in one or more steps



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 Functions

Check Your Understanding

```
>>> f = min

>>> f = max

>>> g, h = min, max

>>> max = g

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

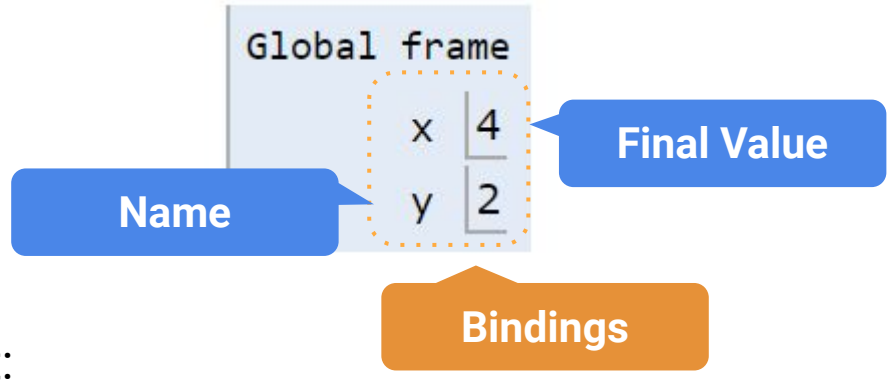
Environment Diagrams

Demo

Visualizing Assignment

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

```
1  x = 1
2  y = 2
3  x = y * 2
```



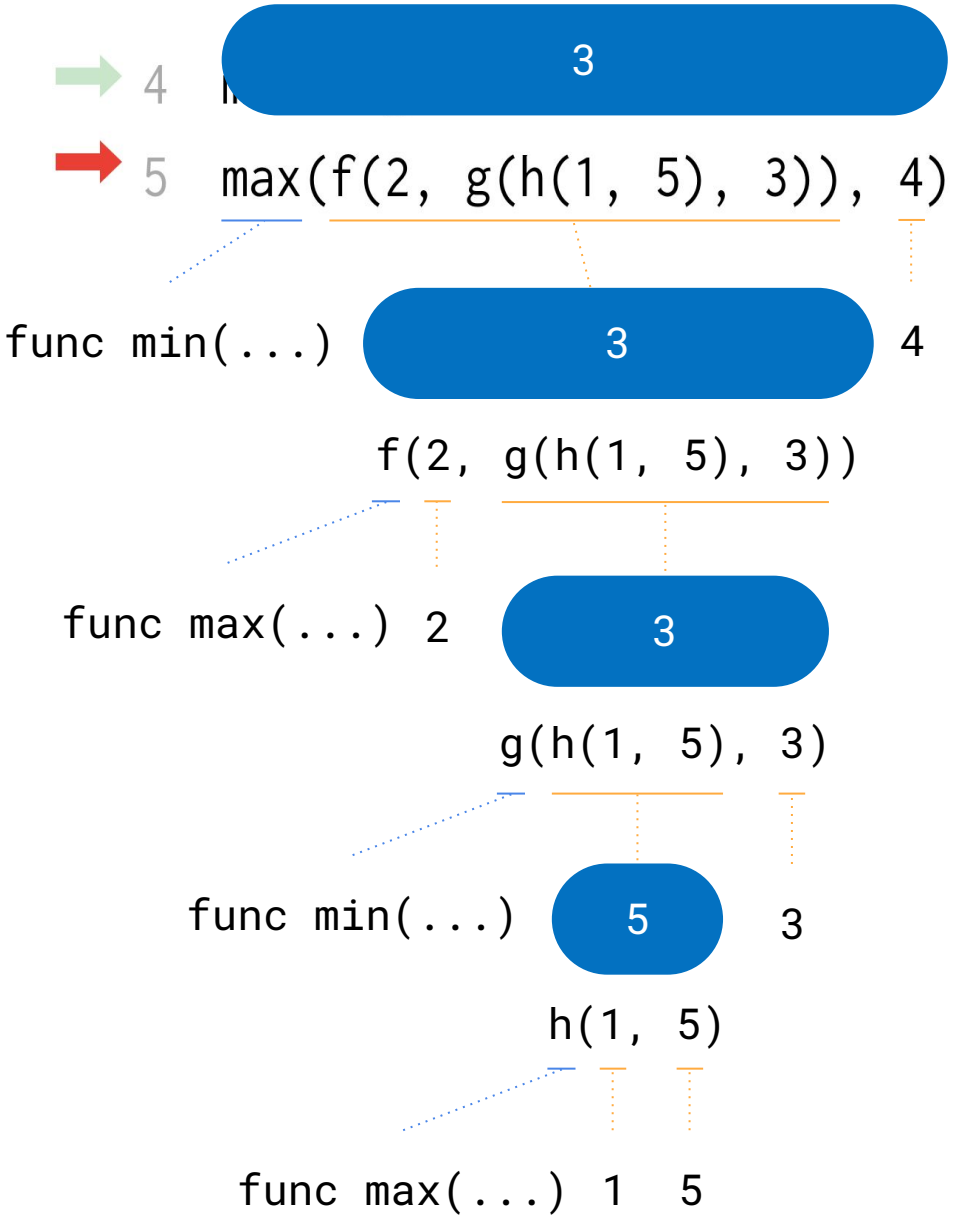
To execute an assignment statement:

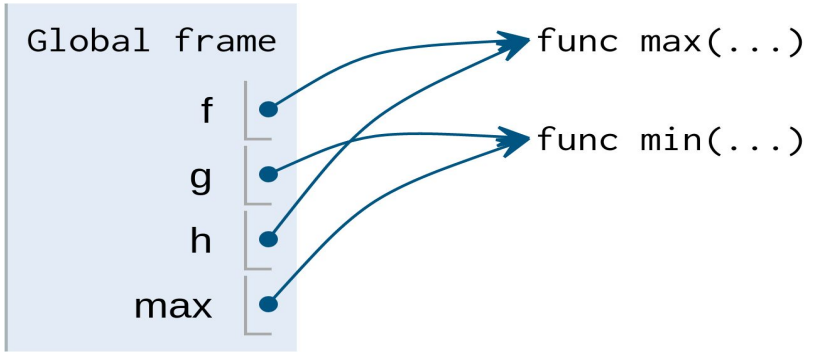
1. **Evaluate** the expression to the right of **=**.
2. **Bind** the value of the expression to the name to the left of **=** in the current environment.

Demo

Check Your Understanding Solution

```
>>> f = min

>>> f = max

>>> g, h = min, max

>>> max = g

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


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

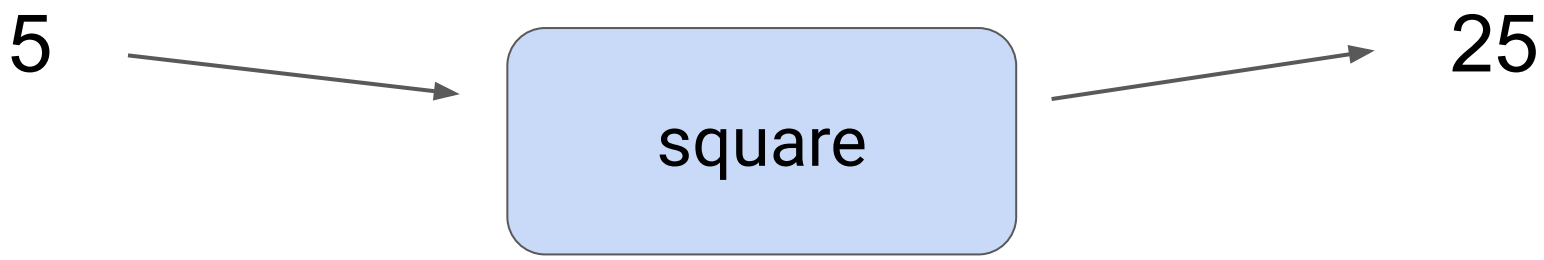
Functions

A **function** is a sequence of code that perform a particular task and can be reused easily

Using functions lets us abstract away 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.



Demo

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

Return Values

The **return** keyword returns a value to whoever calls the functions and then exits the function

Because the function is exited at the **return** , any code that comes after is not run

Forgetting a **return** is really easy to do, but forgetting it will lead to no value being returned by your function

Common Function Mistakes

```
def add(x , y):
    return sum
    sum = x + y
```

Code after the return is not being run. The final line should go before the return

The function takes in num1, num2 but the function uses x, y. Also, there is no return

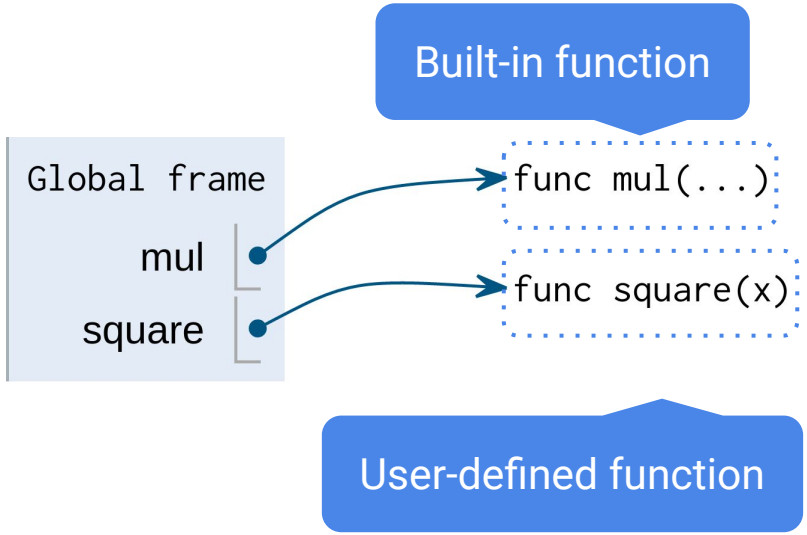
```
def add(num1 , num2):
    sum = x + y
```

```
def add():
    sum = x + y
    return sum
```

The function is using variables that don't seem to exist. They should probably be parameters for the function.

Functions in Environment Diagrams

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



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 frame

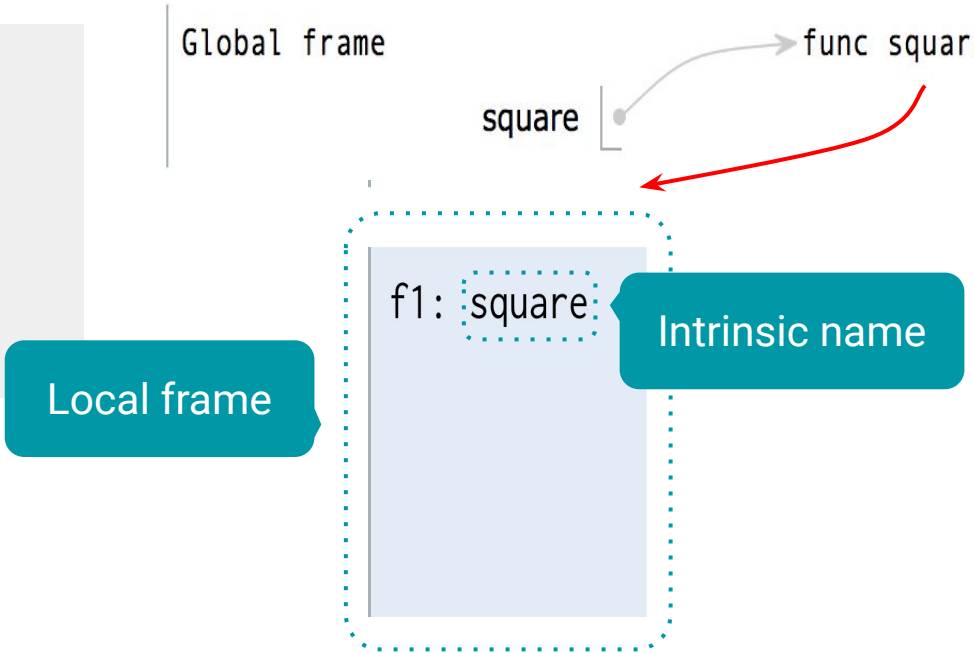
```
def square(x):
    return x * x

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)
```

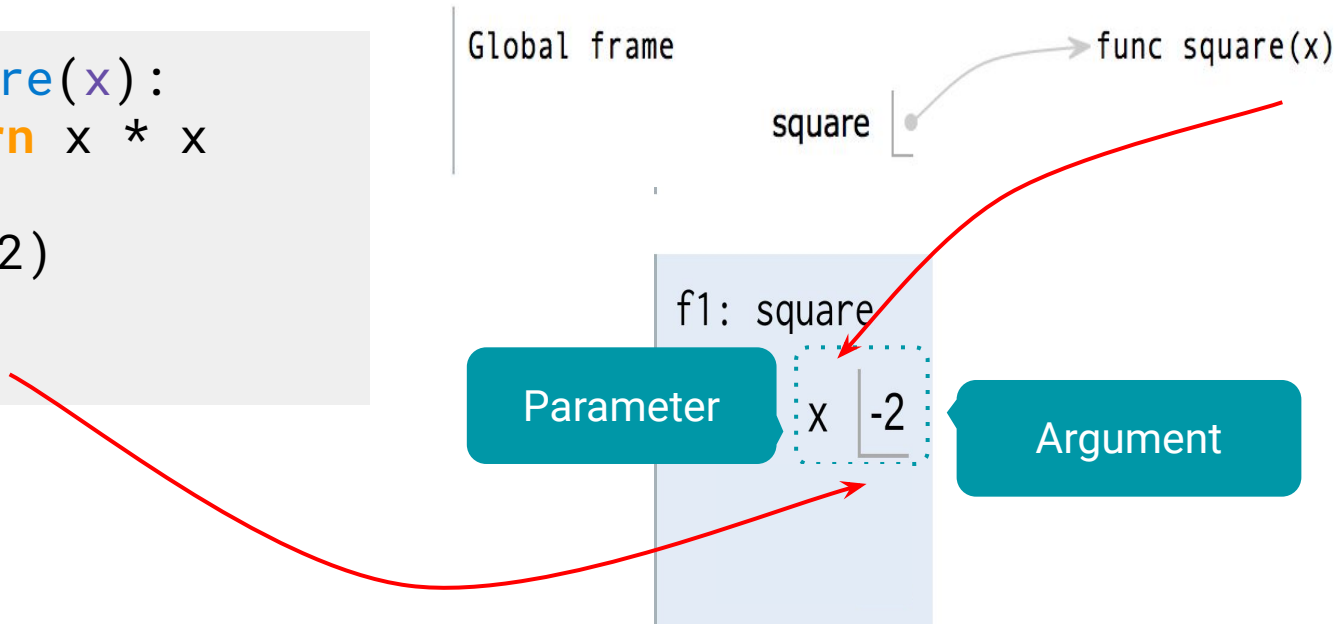


Calling User-Defined Functions

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)
```



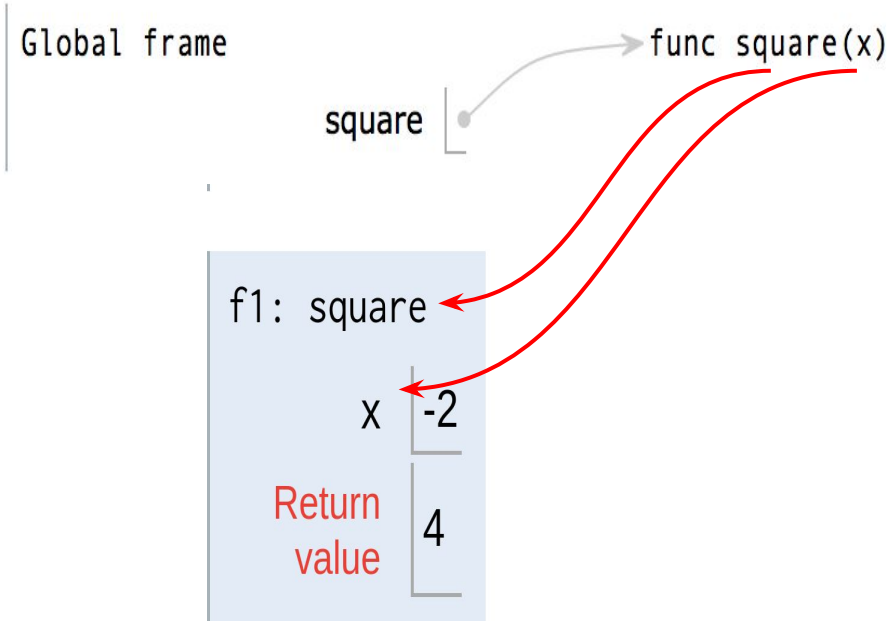
Calling User-Defined Functions

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 the information needed to create a local frame.

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

square(-2)
```

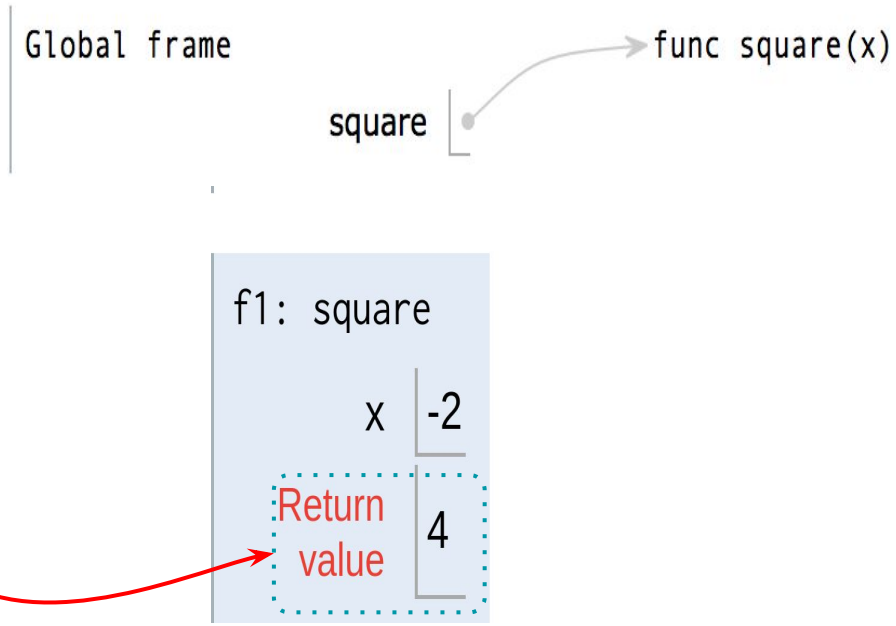


Calling User-Defined Functions

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)
```



Putting it all together

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

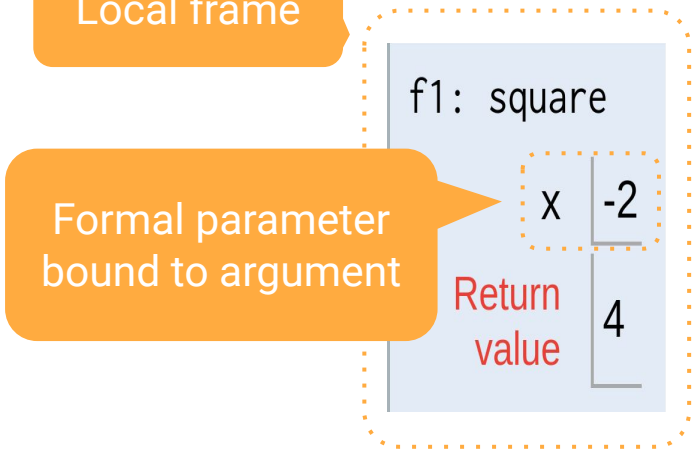
1. Evaluate the operator expression
2. Evaluate each operand expression, left to right
3. Apply the value of the operator expression to the values of the operand expressions

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

Drawing Environment Diagrams

- Python Tutor (tutor.cs61a.org)
  - Useful for quick visualization or for environment diagram questions
  - Make sure you go from our website, don't Google Python Tutor
    - It is not the same thing because we have a couple special settings

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.