

## 61A Lecture 2

---

Wednesday, September 4, 2013

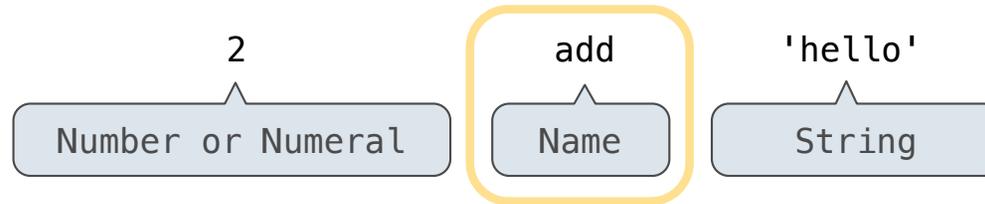
# Names, Assignment, and User-Defined Functions

(Demo)

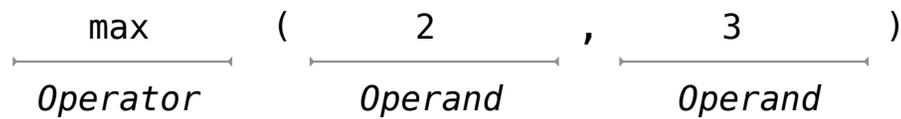
## Types of Expressions

---

**Primitive expressions:**



**Call expressions:**



An operand can also be a call expression

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

## Discussion Question 1

---

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

```
>>> f = min
```

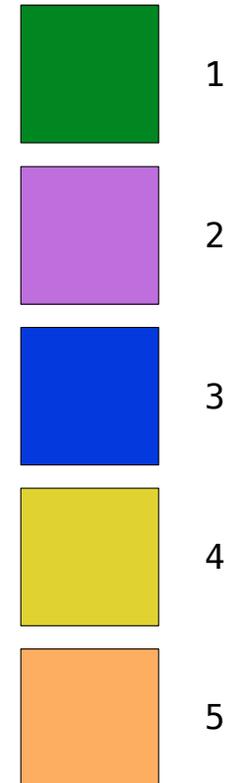
```
>>> f = max
```

```
>>> g, h = min, max
```

```
>>> max = g
```

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

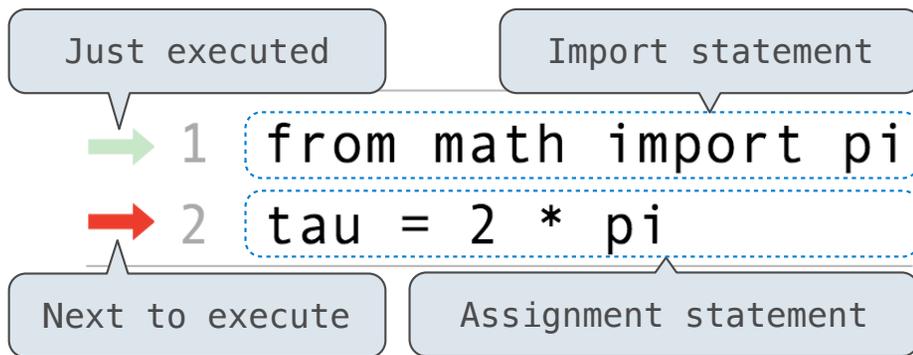
???



## Environment Diagrams

## Environment Diagrams

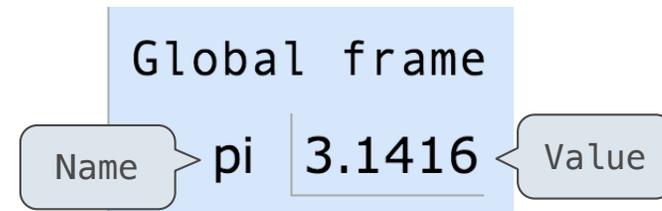
Environment diagrams visualize the interpreter's process.



### Code (left):

Statements and expressions

Arrows indicate evaluation order



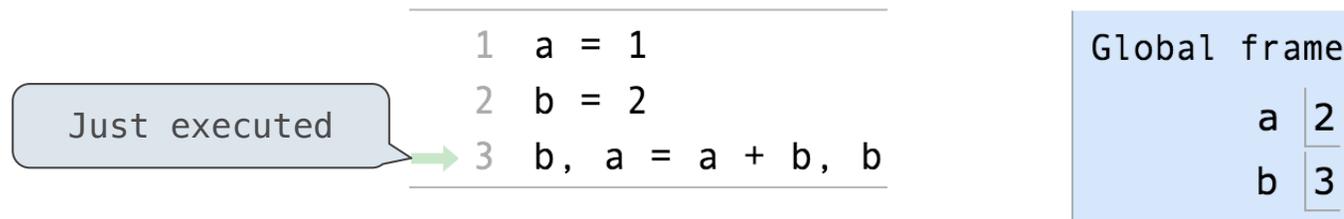
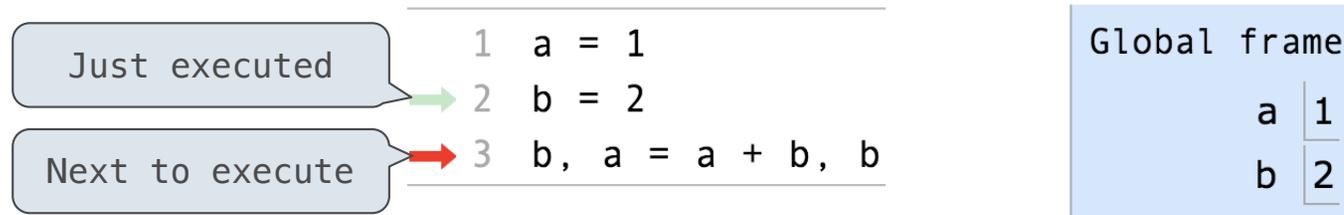
### Frames (right):

Each name is bound to a value

Within a frame, a name cannot be repeated

(Demo)

## Assignment Statements

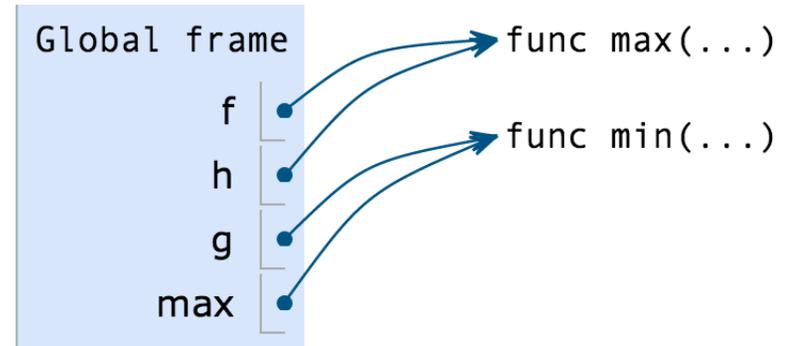
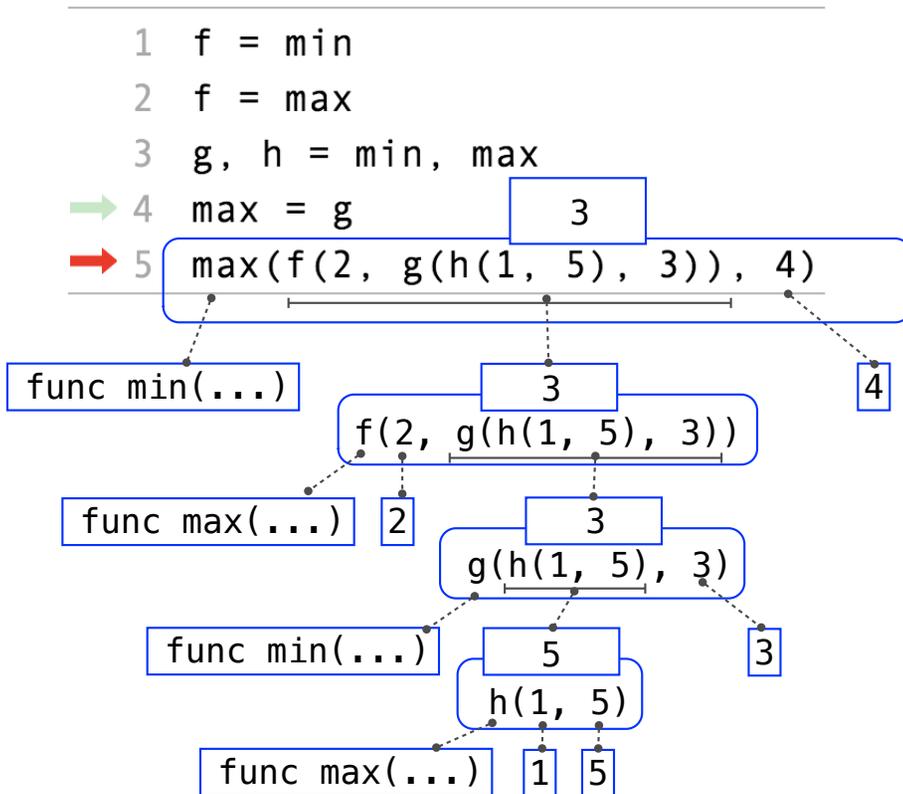


### Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.
2. Bind all names to the left of = to the resulting values in the current frame.

## Discussion Question 1 Solution

(Demo)



3

## Defining Functions

## Defining Functions

---

Assignment is a simple means of abstraction: binds names to values

Function definition is a more powerful means of abstraction: binds names to *expressions*

Function **signature** indicates how many arguments a function takes

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

Function **body** defines the computational process expressed by a function

### Execution procedure for def statements:

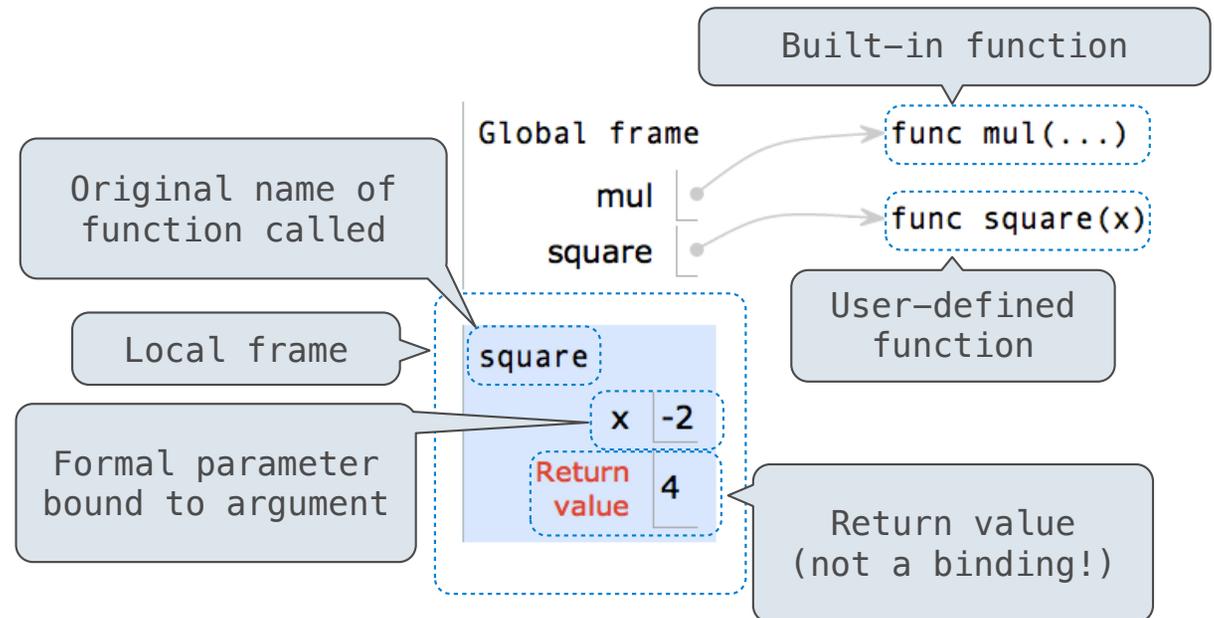
1. Create a function with signature `<name>(<formal parameters>)`
2. Set the body of that function to be everything indented after the first line
3. Bind `<name>` to that function in the current frame

## Calling User-Defined Functions

### Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a *new* environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

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



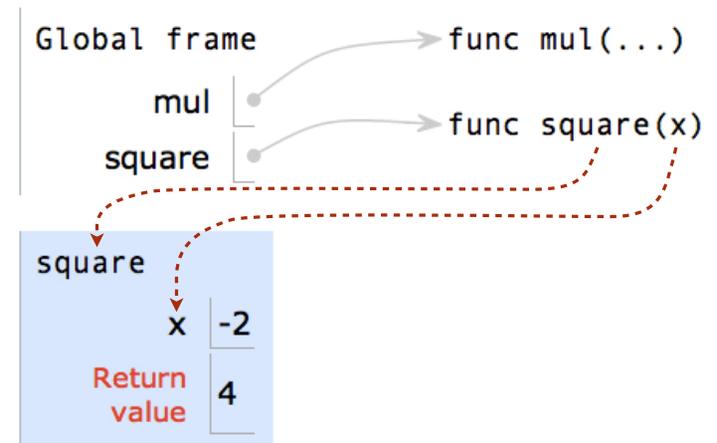
## Calling User-Defined Functions

### Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame, forming a *new* environment
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

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

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



## Looking Up Names In Environments

---

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

- The global frame alone, *or*
- A local frame, followed by the global frame.

***Most important two things I'll say all day:***

An environment is a *sequence* of frames.

A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

E.g., to look up some name in the body of the *square* function:

- Look for that name in the local frame.
- If not found, look for it in the global frame.  
(Built-in names like “max” are in the global frame too, but we don't draw them in environment diagrams.)

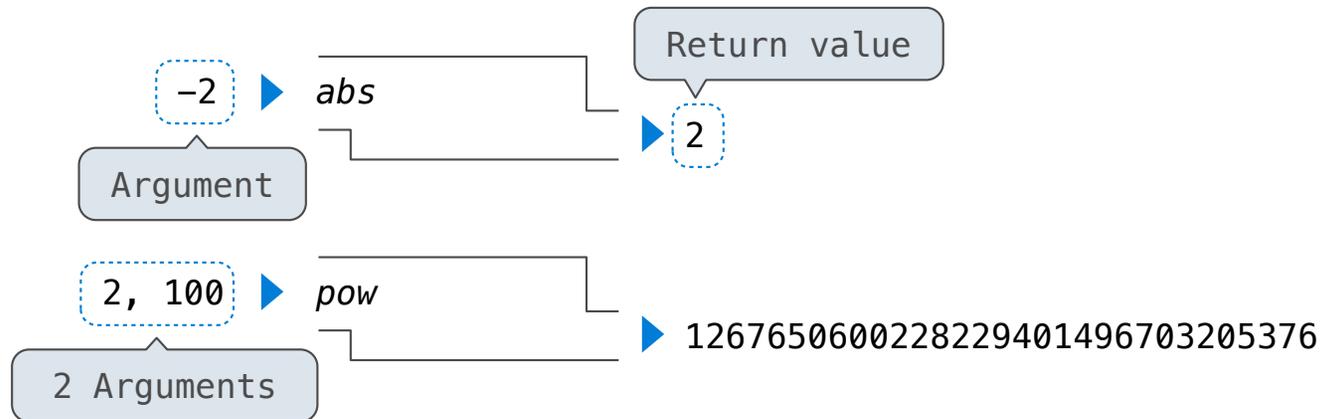
(Demo)

# The Print Function

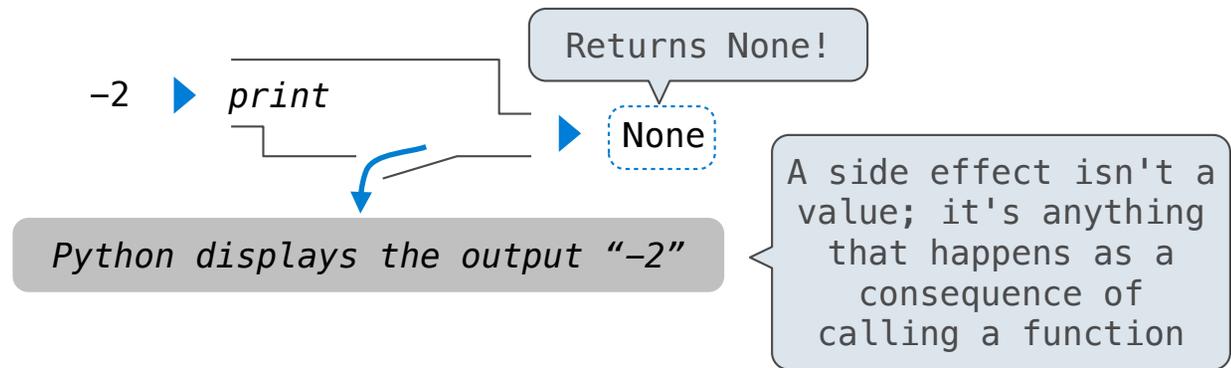
(Demo)

## Pure Functions & Non-Pure Functions

**Pure Functions**  
*just return values*

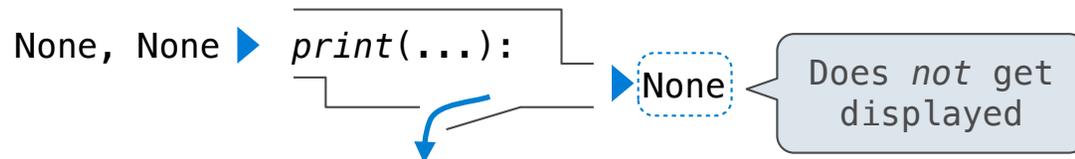


**Non-Pure Functions**  
*have side effects*



**Important:** *The interactive interpreter (>>>) displays the value of an expression, unless it is None*

## Nested Expressions with Print



display "None None"

```
>>> print(print(1), print(2))
1
2
None None
```

