



CS61A Lecture 4

Amir Kamil
UC Berkeley
January 30, 2013

Announcements



- Reminder: hw1 due tonight

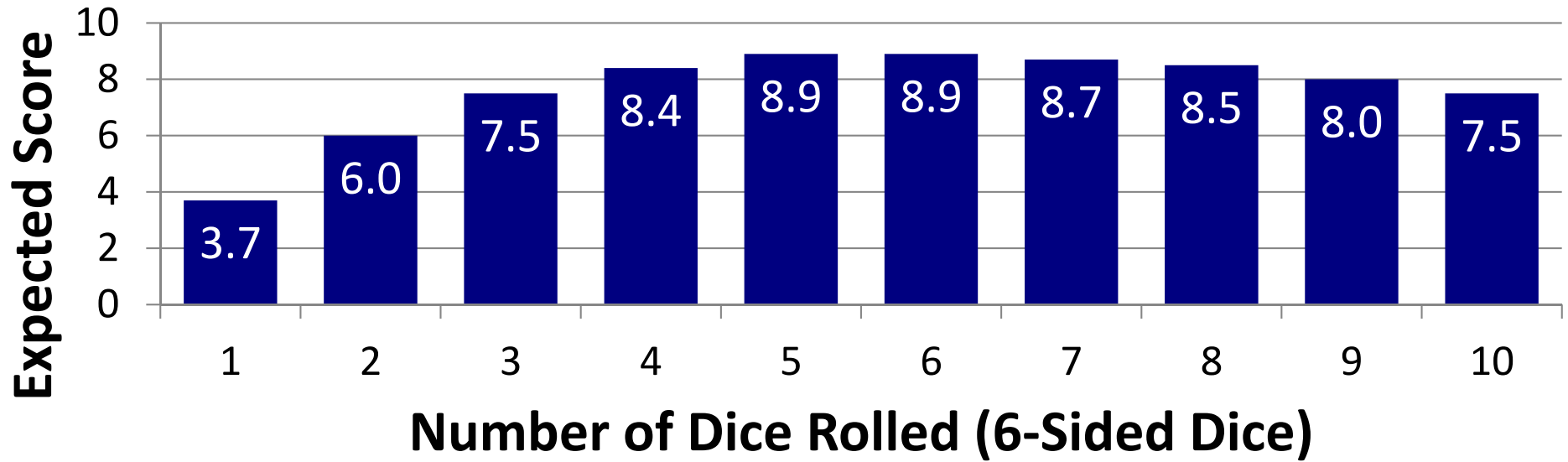
- In-class quiz on Friday
 - Covers through Wednesday's lecture
 - Bring a writing implement

- Hog project out
 - Get started early!
 - Try it out online! See the announcement on the website

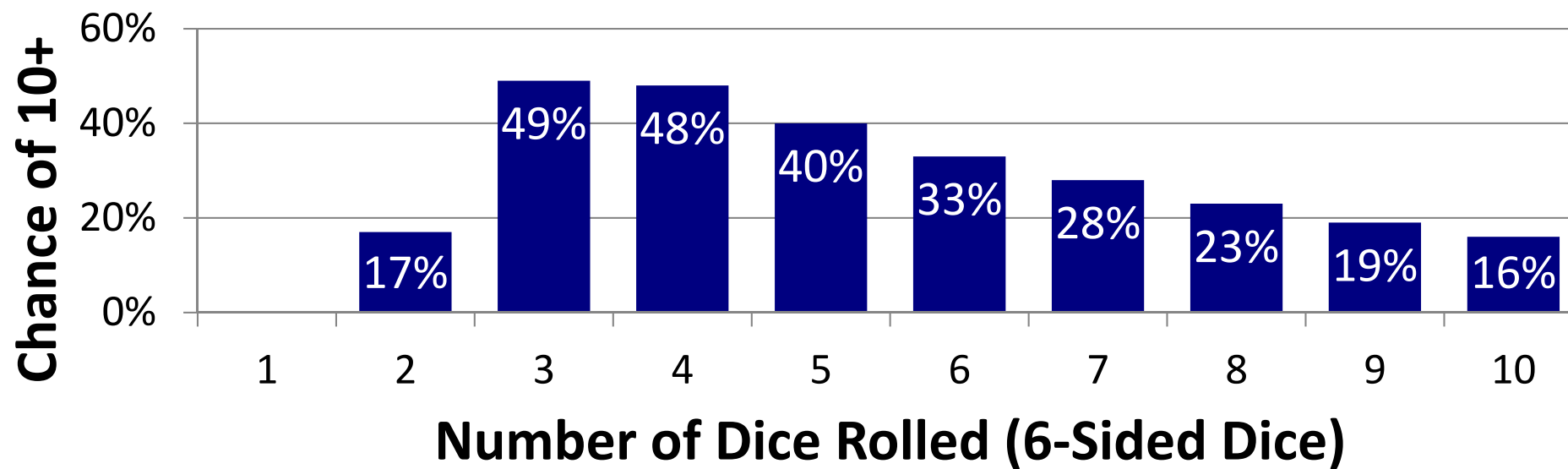
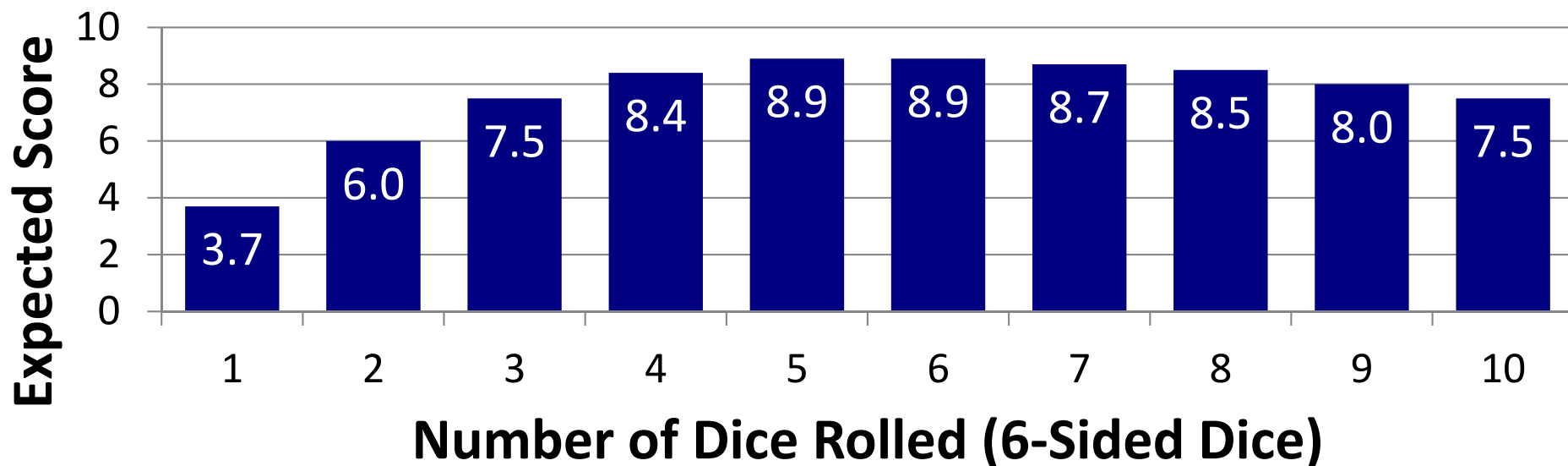
The Game of Hog



The Game of Hog



The Game of Hog



Environment Diagrams



Environment Diagrams



- Every expression is evaluated in the context of an environment

Environment Diagrams



- Every expression is evaluated in the context of an environment
- So far, the current environment is either:

Environment Diagrams



- Every expression is evaluated in the context of an environment
- So far, the current environment is either:
 - The global frame alone, or

Environment Diagrams



- 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

Environment Diagrams



- 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
- **Important properties of environments:**

Environment Diagrams



- 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
- **Important properties of environments:**
 - An environment is a sequence of frames

Environment Diagrams



- 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
- **Important properties of environments:**
 - An environment is a sequence of frames
 - The earliest frame that contains a binding for a name determines the value that the name evaluates to

Environment Diagrams



- 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
- **Important properties of environments:**
 - An environment is a sequence of frames
 - The earliest frame that contains a binding for a name determines the value that the name evaluates to
- The *scope* of a name is the region of code that has access to it

Environment of Function Application



Environment of Function Application



The environment in which a function is applied
consists of:

it

Environment of Function Application



The environment in which a function is applied consists of:

it

- A *new* local frame *each* time the function is *applied*

Environment of Function Application



The environment in which a function is applied consists of:

it

- A *new* local frame *each* time the function is *applied*
- The environment in which the function was *defined*

Environment of Function Application



The environment in which a function is applied consists of:

it

- A *new* local frame *each* time the function is *applied*
- The environment in which the function was *defined*
 - We refer to this as *lexical scoping*

Environment of Function Application



The environment in which a function is applied consists of:

it

- A *new* local frame *each* time the function is *applied*
- The environment in which the function was *defined*
 - We refer to this as *lexical scoping*
 - So far, this is just the global frame

Environment of Function Application



The environment in which a function is applied consists of:

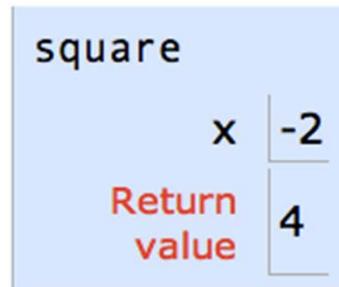
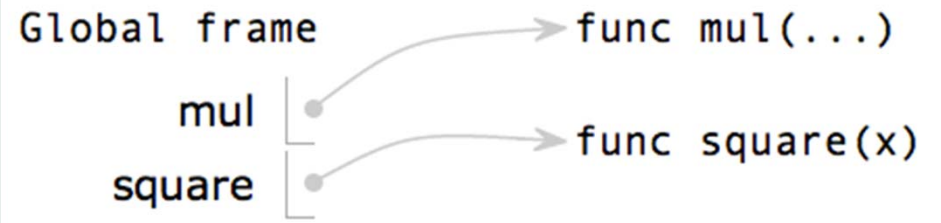
it

- A *new* local frame *each* time the function is *applied*
- The environment in which the function was *defined*
 - We refer to this as *lexical scoping*
 - So far, this is just the global frame
 - The *current* state of the environment is used, not the state when the function definition was executed

Formal Parameters



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



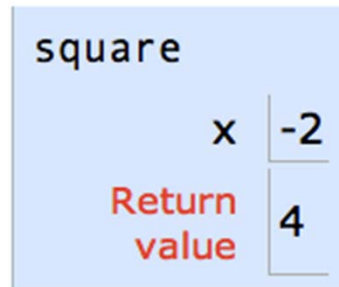
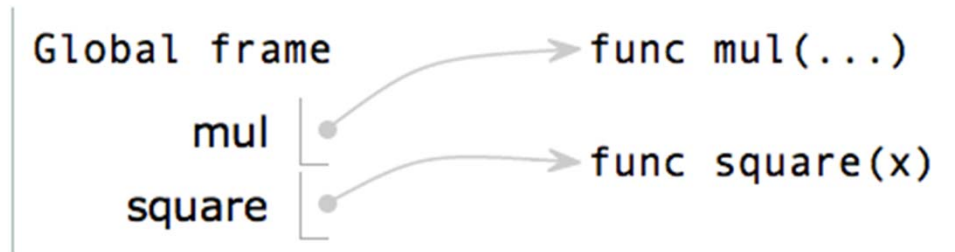
Example: <http://goo.gl/boCk0>

Formal Parameters



```
def square(x):  
    return mul(x, x)
```

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

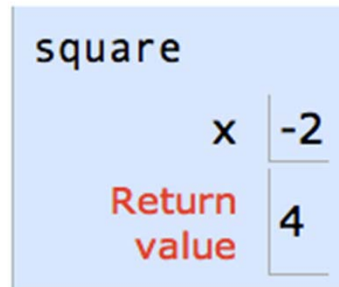
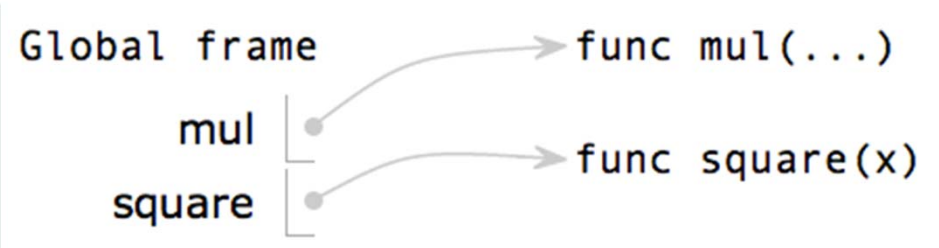


Formal Parameters



```
def square(x):  
    return mul(x, x)    VS
```

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



Formal Parameters

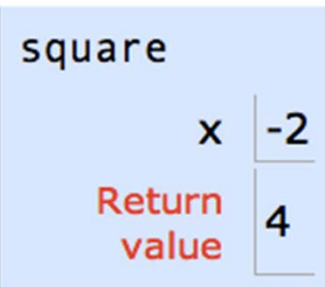
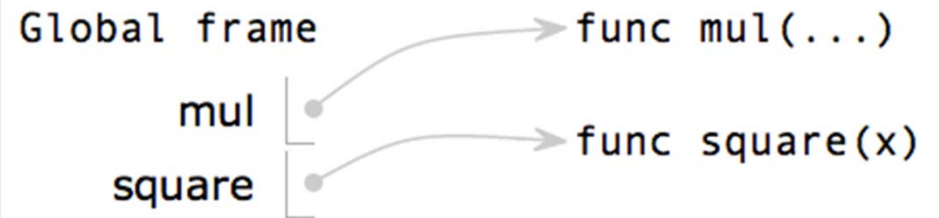


```
def square(x):  
    return mul(x, x)
```

VS

```
def square(y):  
    return mul(y, y)
```

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



Formal Parameters

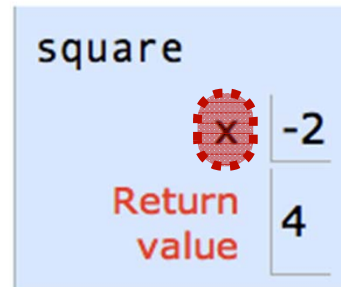
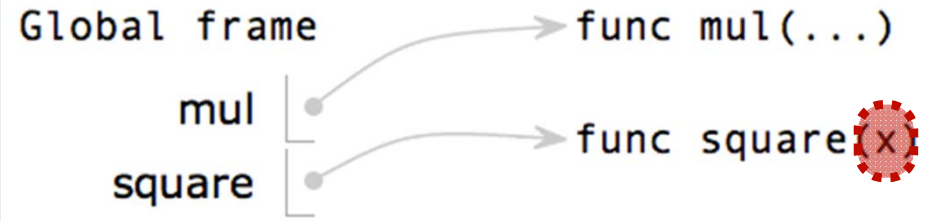


```
def square(x):  
    return mul(x, x)
```

VS

```
def square(y):  
    return mul(y, y)
```

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



Formal Parameters

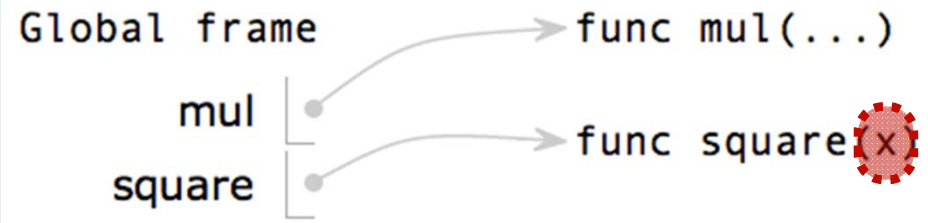


```
def square(x):  
    return mul(x, x)
```

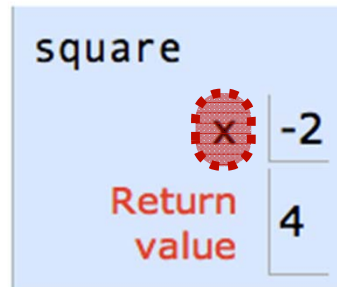
VS

```
def square(y):  
    return mul(y, y)
```

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



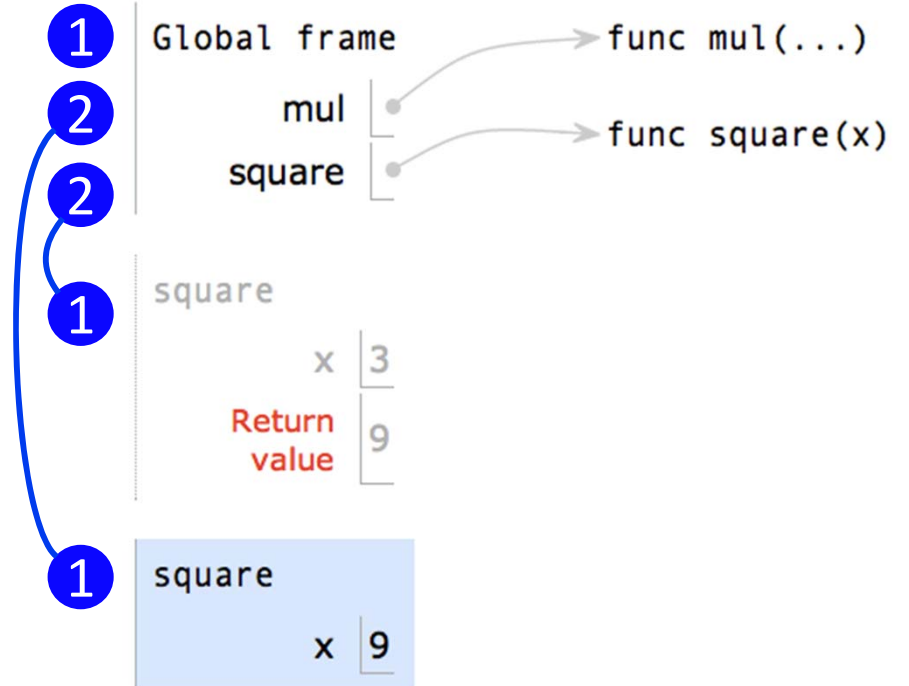
Formal parameters
have local scope



Multiple Environments in a Diagram



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



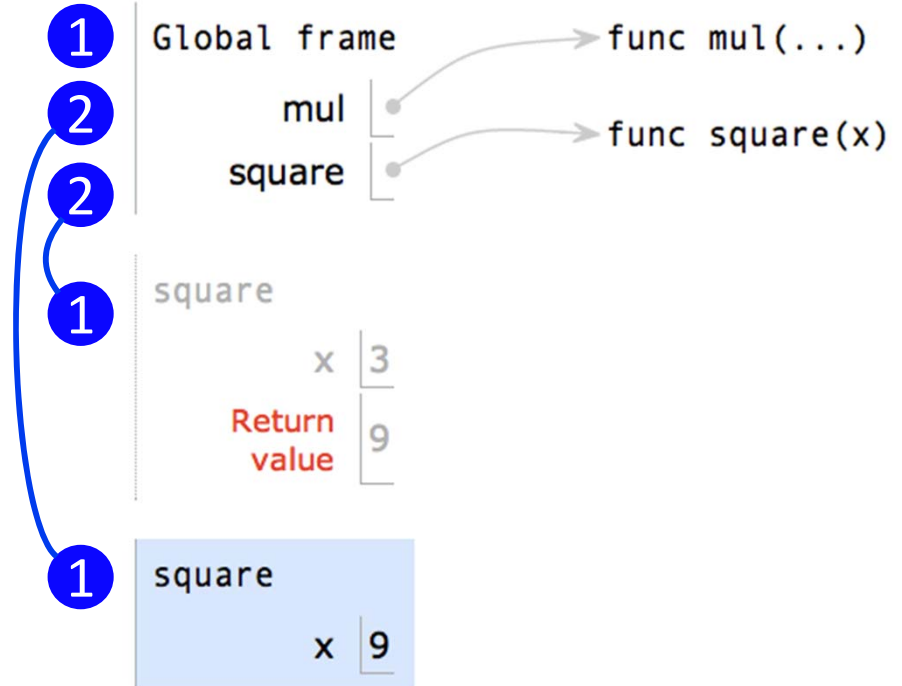
Example: <http://goo.gl/hrfnV>

Multiple Environments in a Diagram



What happens when to the local frame when a function returns?

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



Example: <http://goo.gl/hrfnV>

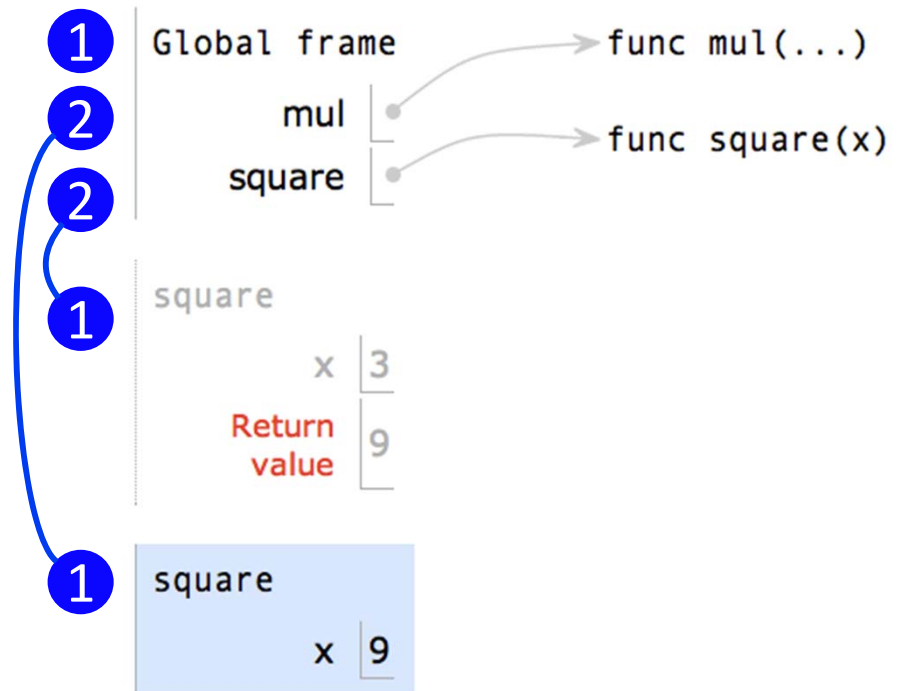
Multiple Environments in a Diagram



What happens when to the local frame when a function returns?

- It sticks around until Python realizes it is no longer needed

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



Example: <http://goo.gl/hrfnV>

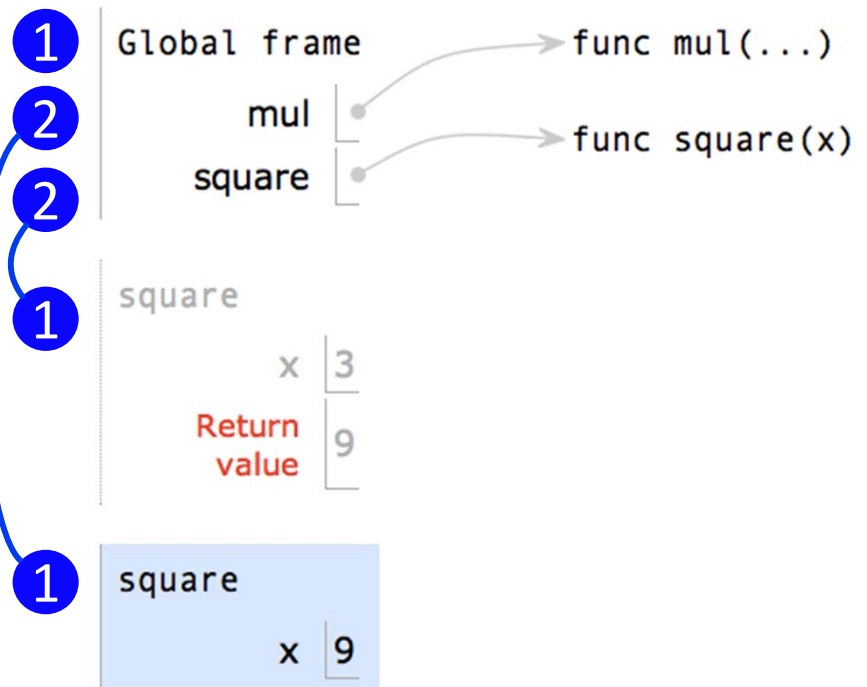
Multiple Environments in a Diagram



What happens when to the local frame when a function returns?

- It sticks around until Python realizes it is no longer needed
- We will soon see cases where it is needed after the call

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



Example: <http://goo.gl/hrfnV>

Life Cycle of a User-Defined Function



Def statement:

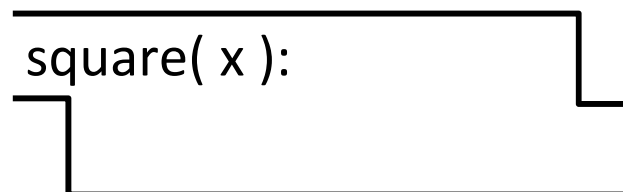
```
>>> def square(x):  
    return mul(x, x)
```

What happens?

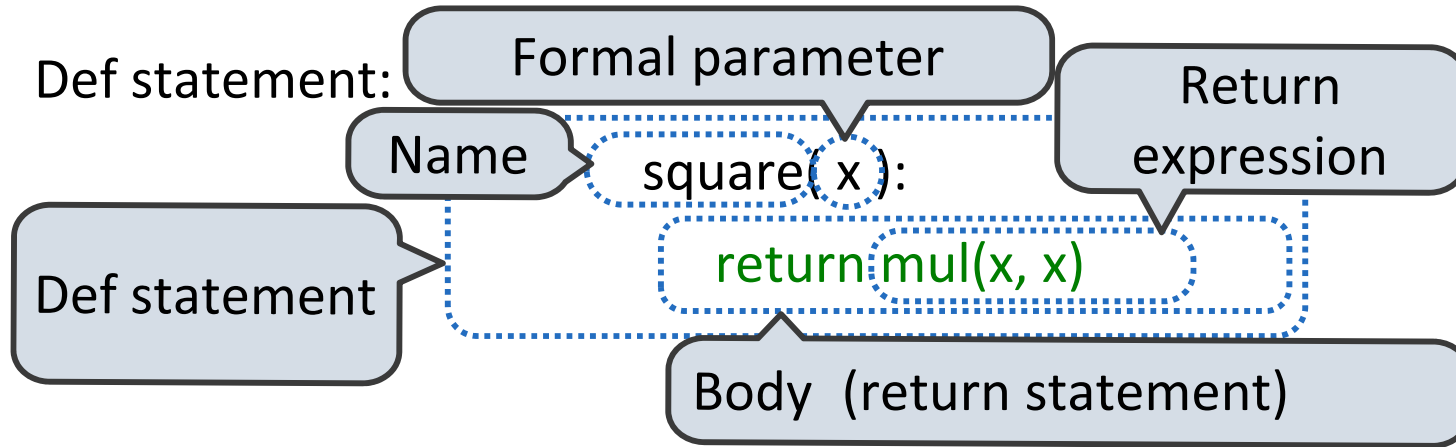
Call expression:

```
square(2+2)
```

Calling/Applying:



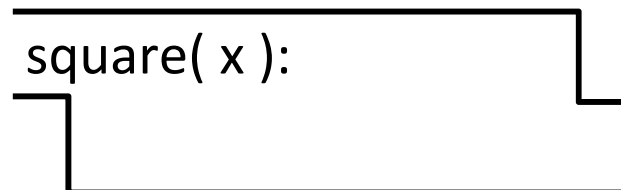
Life Cycle of a User-Defined Function



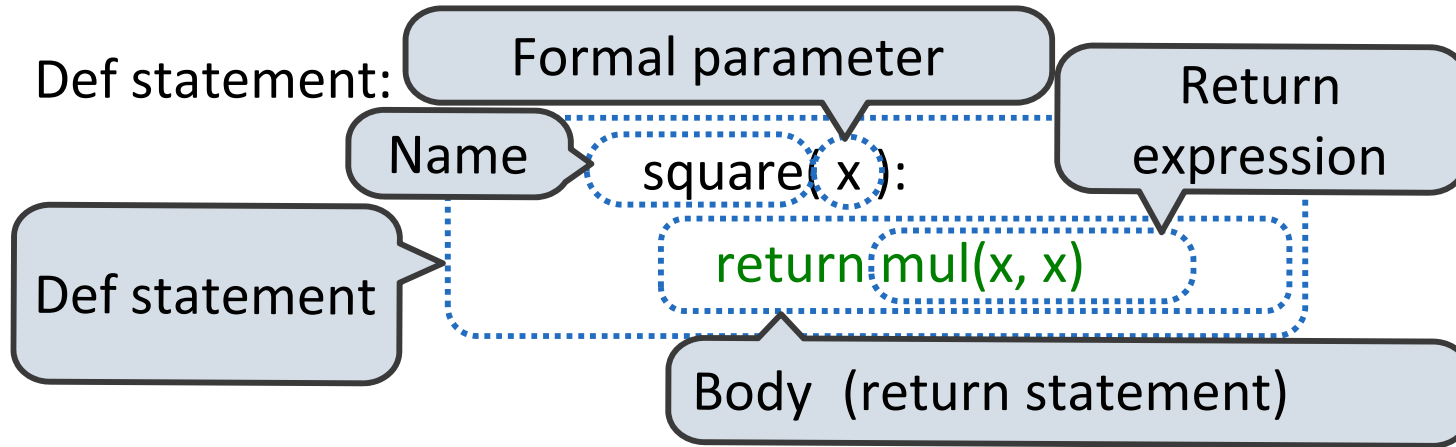
What happens?

Call expression: `square(2+2)`

Calling/Applying:



Life Cycle of a User-Defined Function

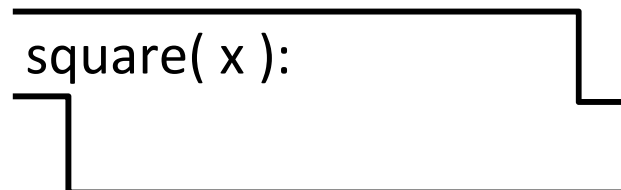


What happens?

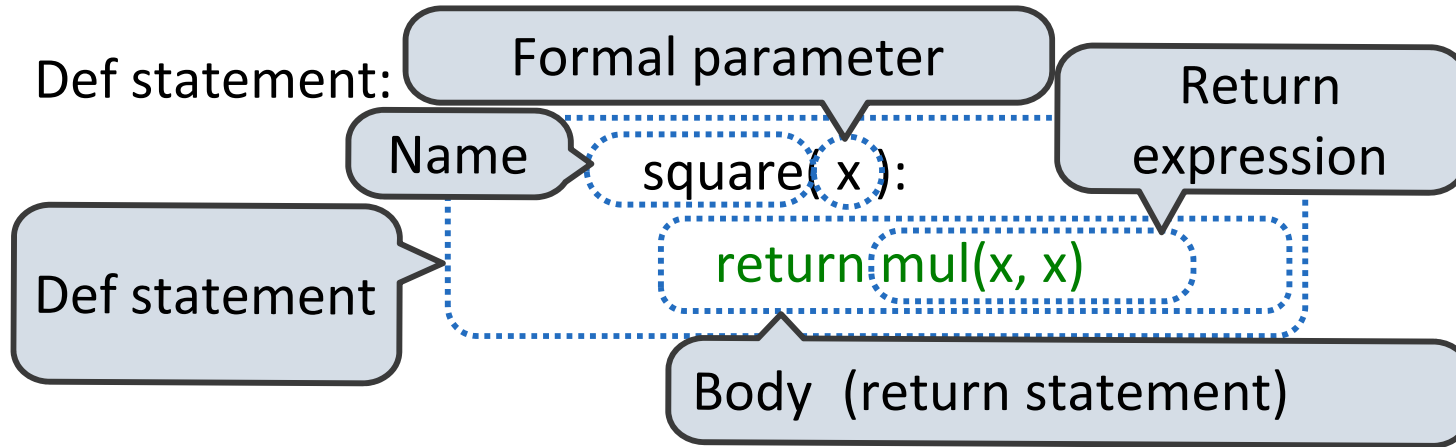
Function created

Call expression: `square(2+2)`

Calling/Applying:



Life Cycle of a User-Defined Function



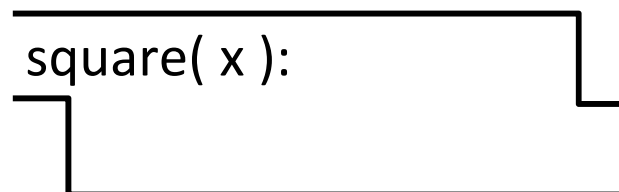
What happens?

Function created

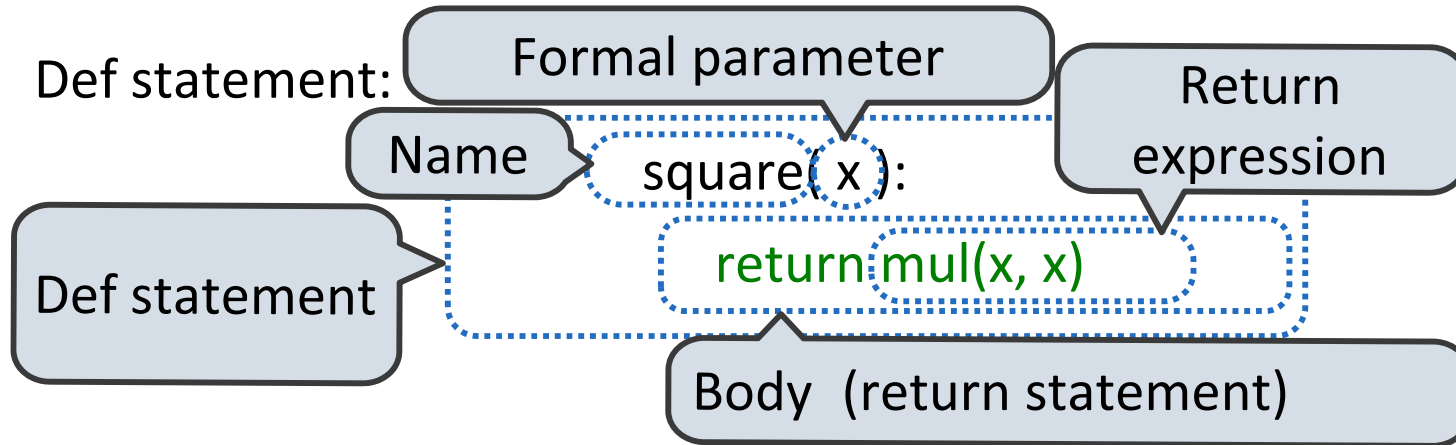
Name bound

Call expression: `square(2+2)`

Calling/Applying:



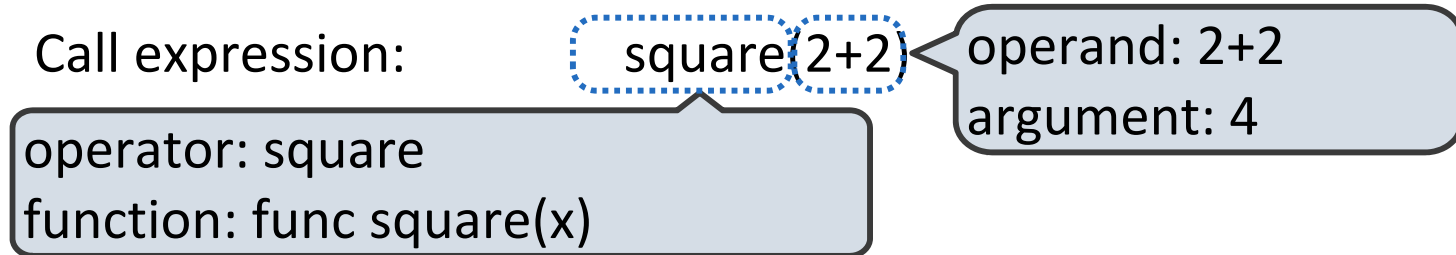
Life Cycle of a User-Defined Function



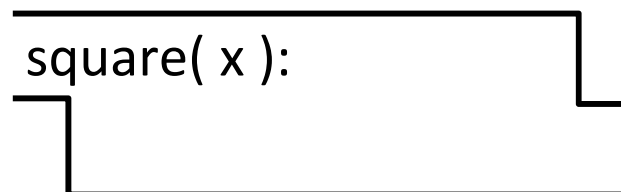
What happens?

Function created

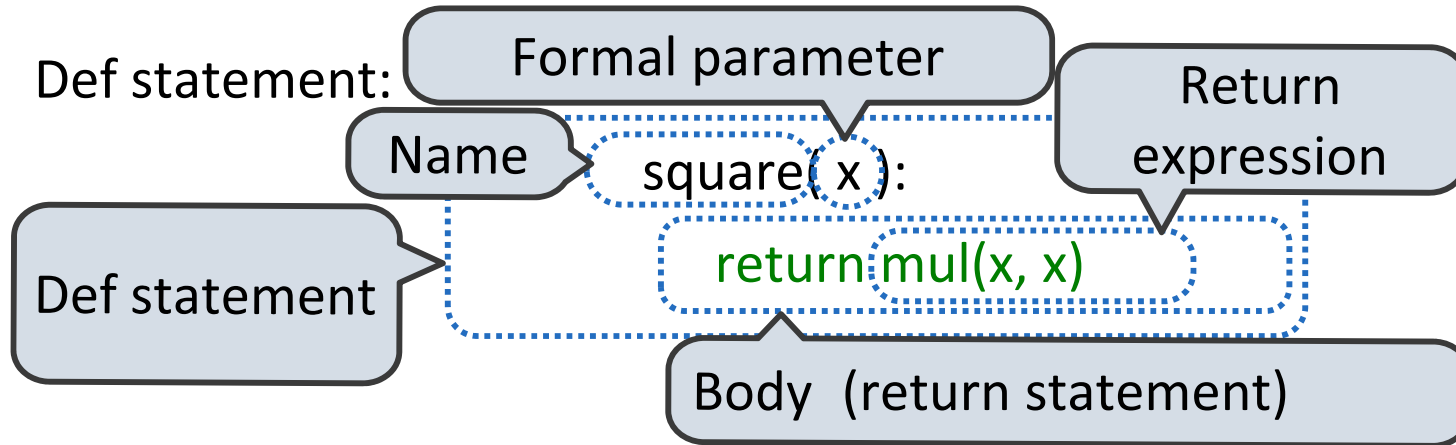
Name bound



Calling/Applying:



Life Cycle of a User-Defined Function

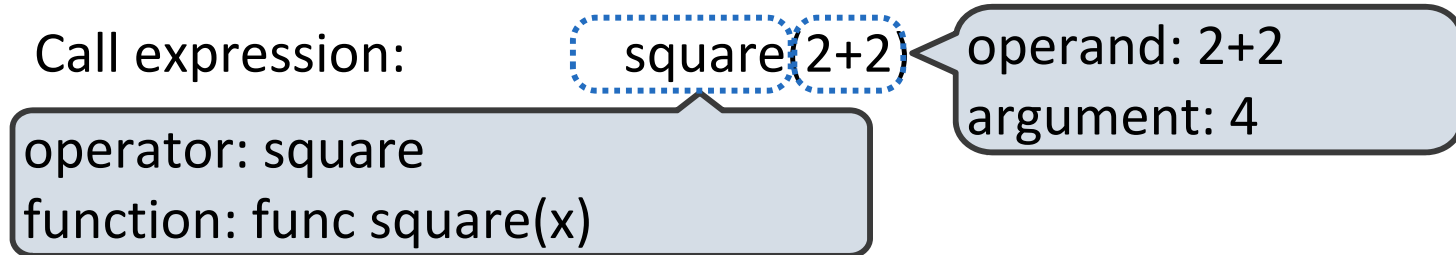


What happens?

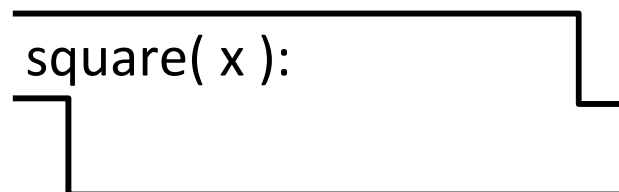
Function created

Name bound

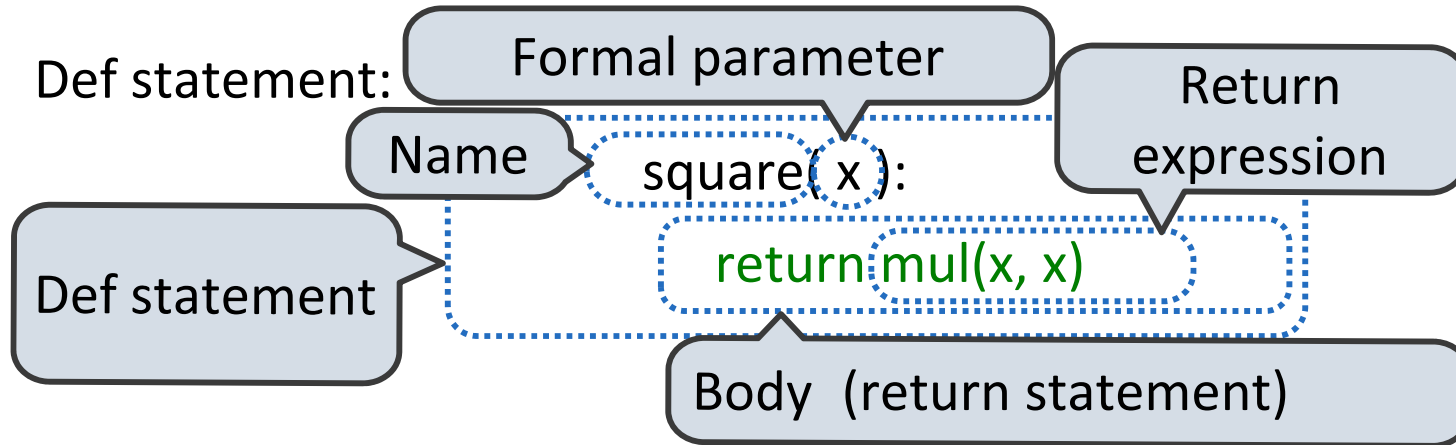
Op's evaluated



Calling/Applying:



Life Cycle of a User-Defined Function



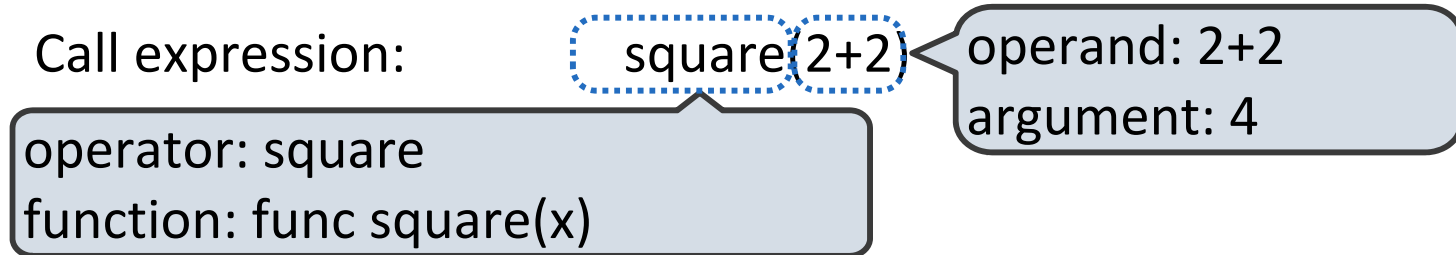
What happens?

Function created

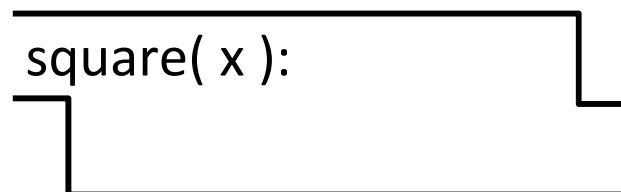
Name bound

Op's evaluated

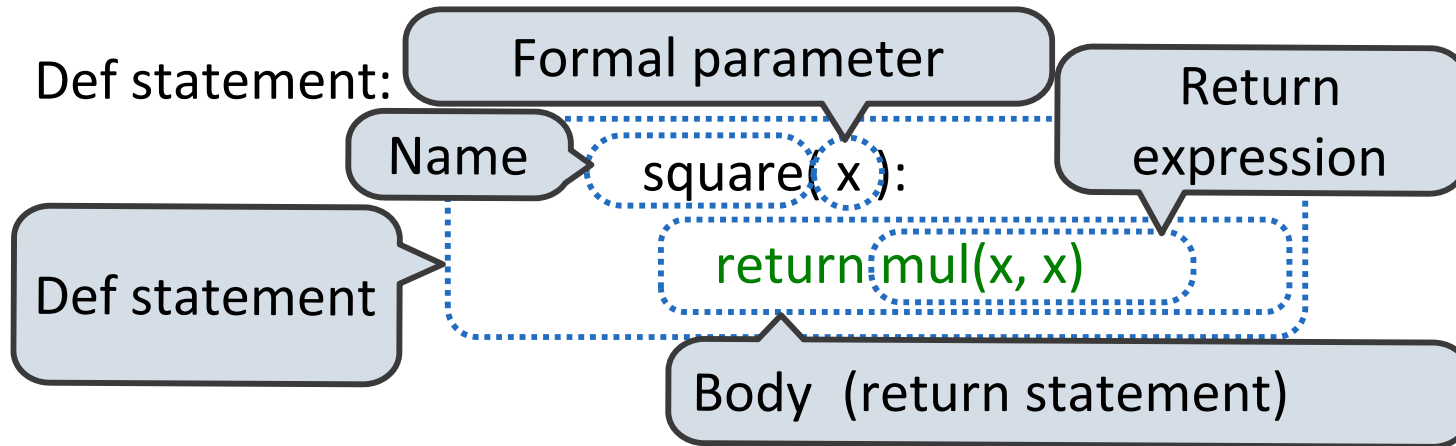
Function called with argument(s)



Calling/Applying:



Life Cycle of a User-Defined Function



What happens?

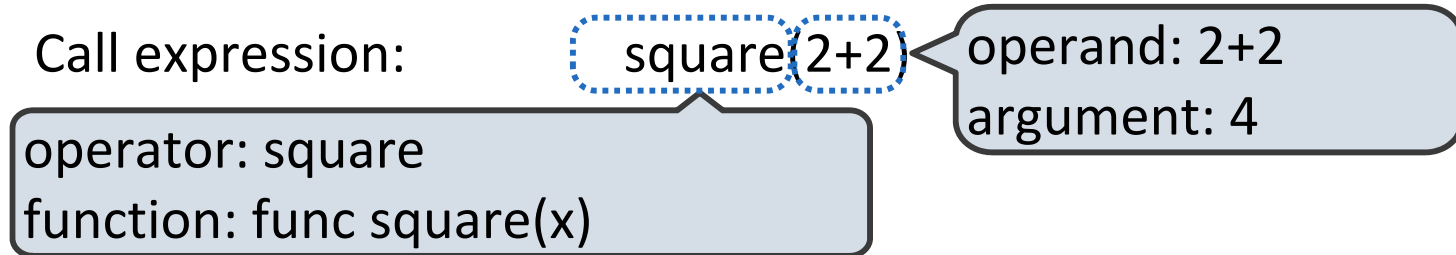
Function created

Name bound

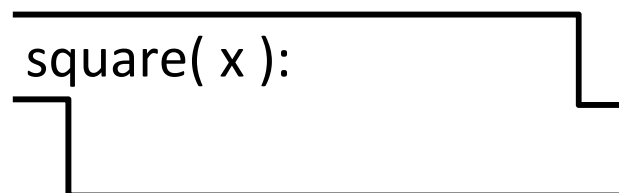
Op's evaluated

Function called with argument(s)

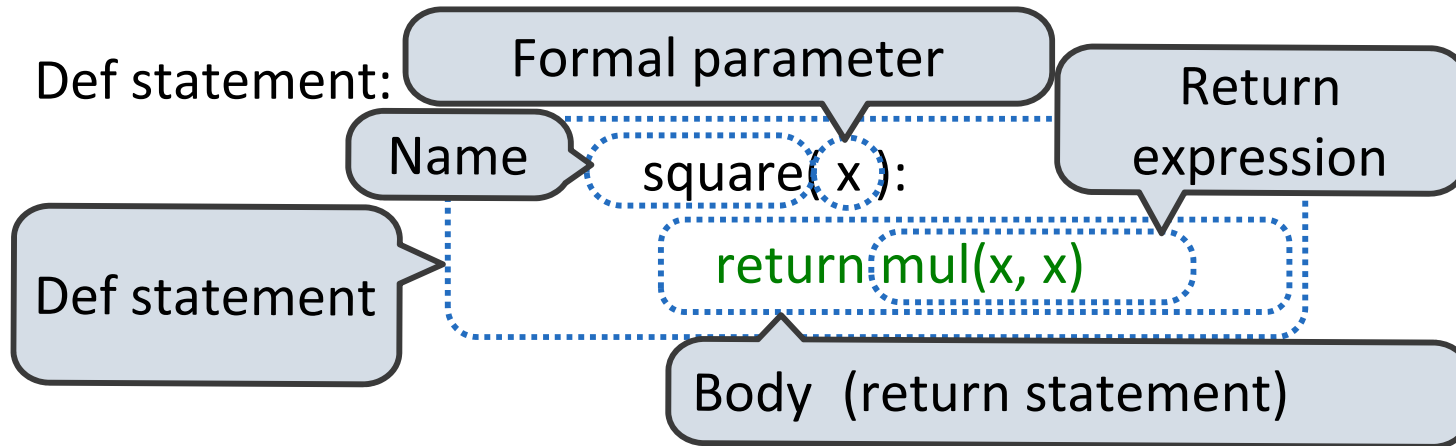
Evaluates to return value below



Calling/Applying:



Life Cycle of a User-Defined Function



What happens?

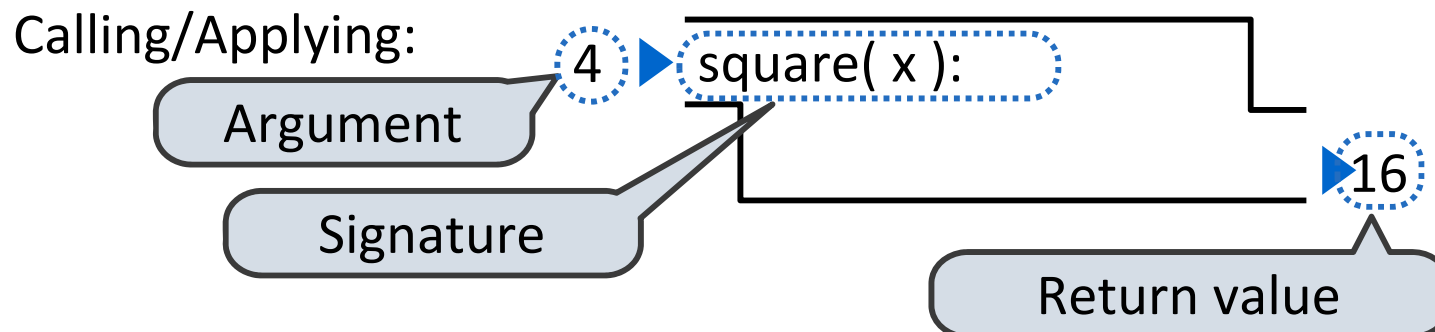
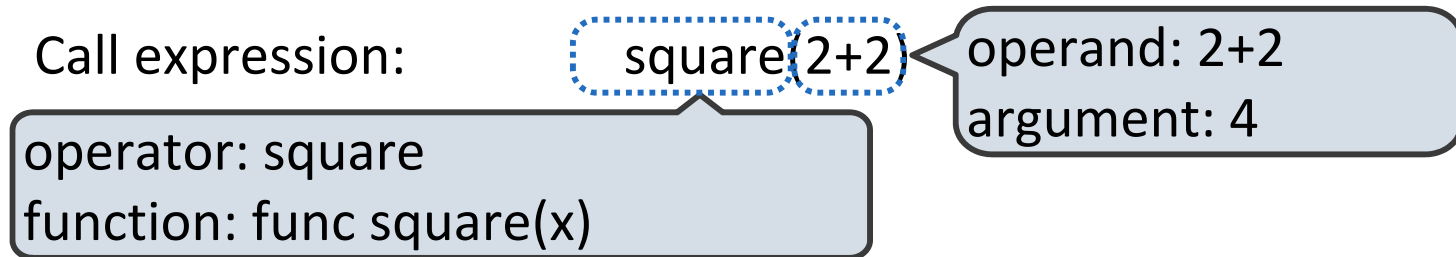
Function created

Name bound

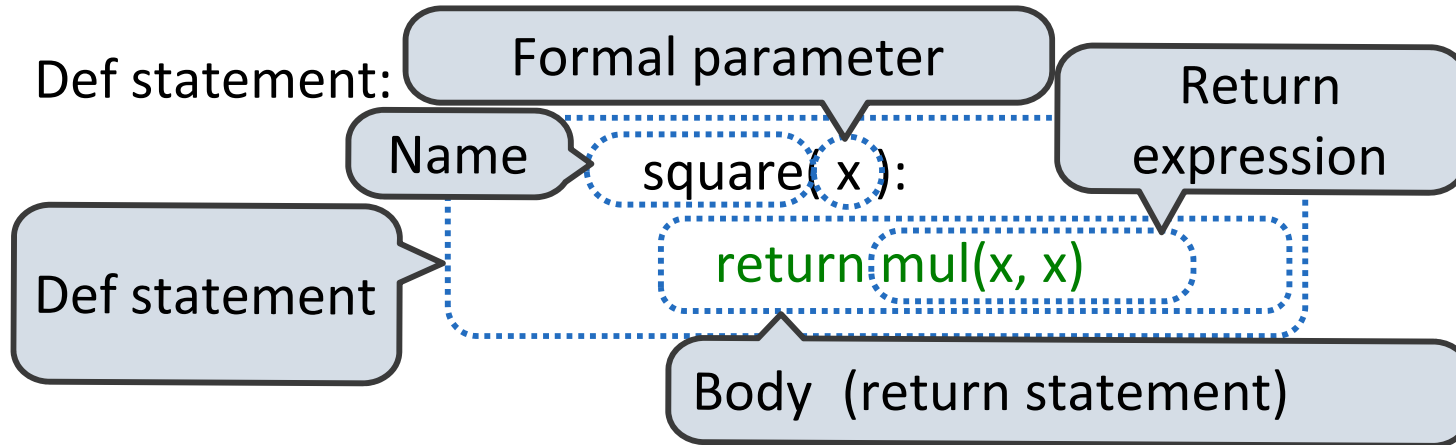
Op's evaluated

Function called with argument(s)

Evaluates to return value below



Life Cycle of a User-Defined Function



What happens?

Function created

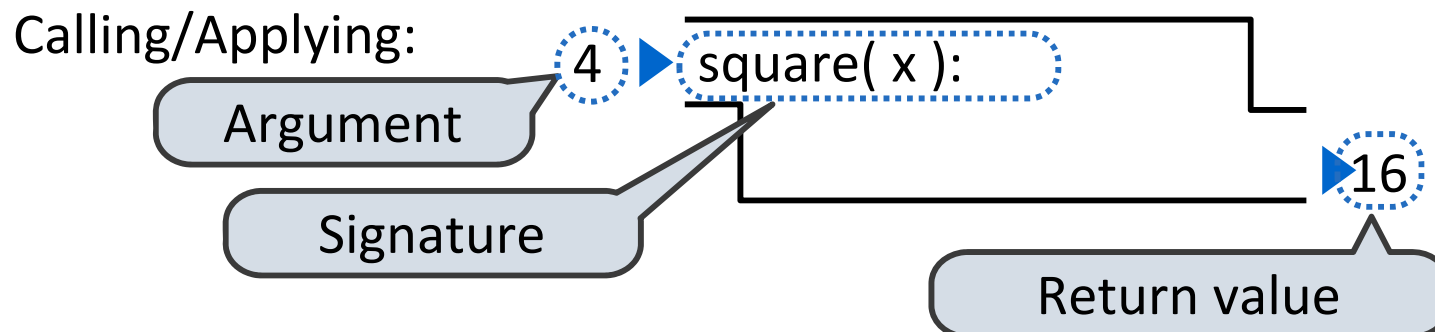
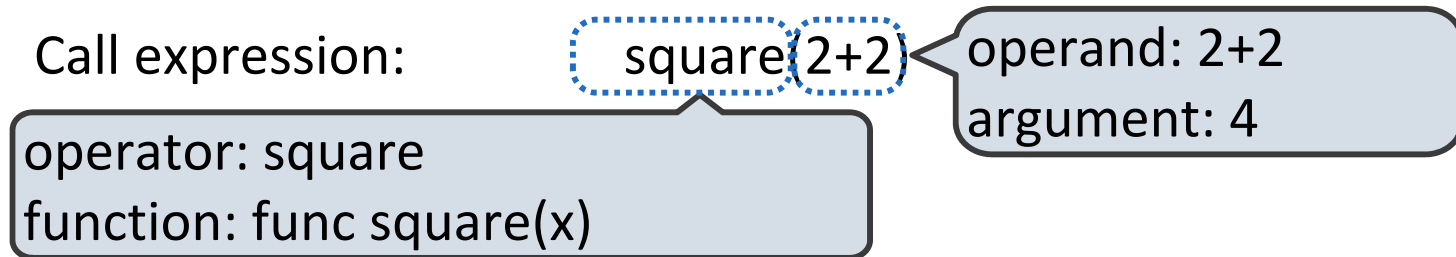
Name bound

Op's evaluated

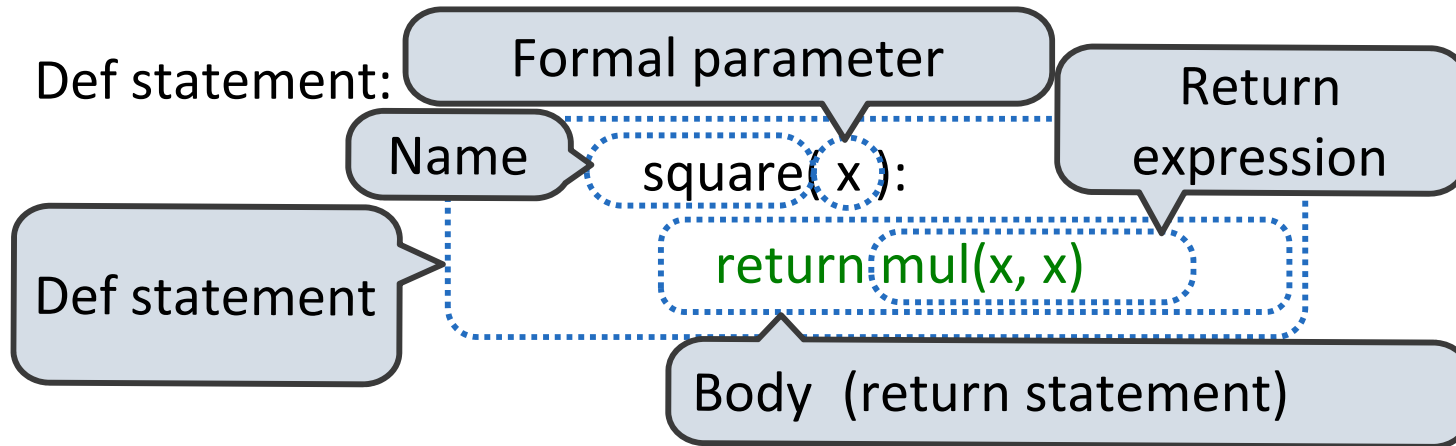
Function called with argument(s)

Evaluates to return value below

New frame!



Life Cycle of a User-Defined Function



What happens?

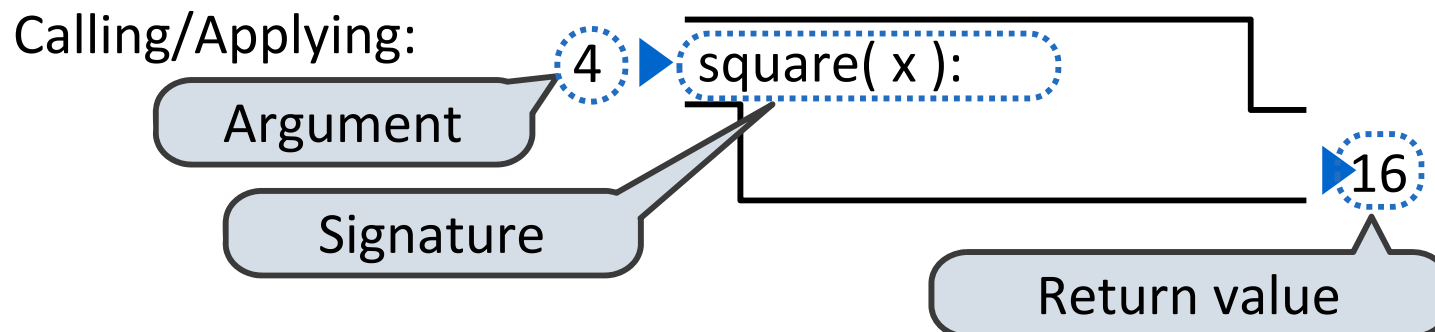
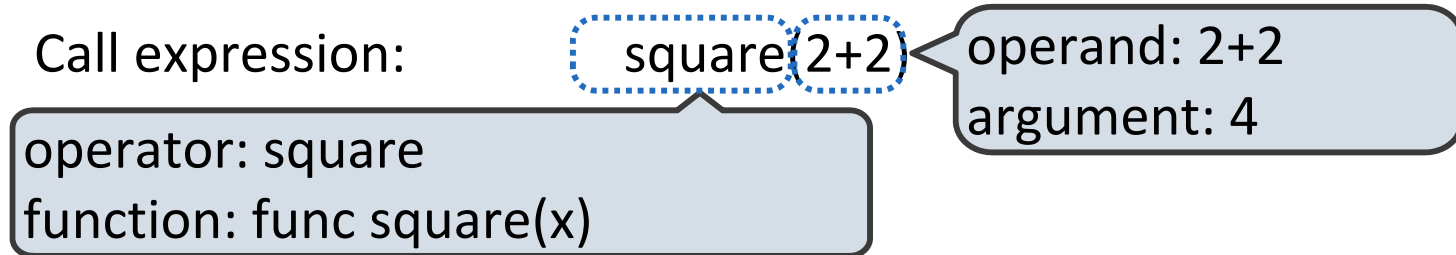
Function created

Name bound

Op's evaluated

Function called with argument(s)

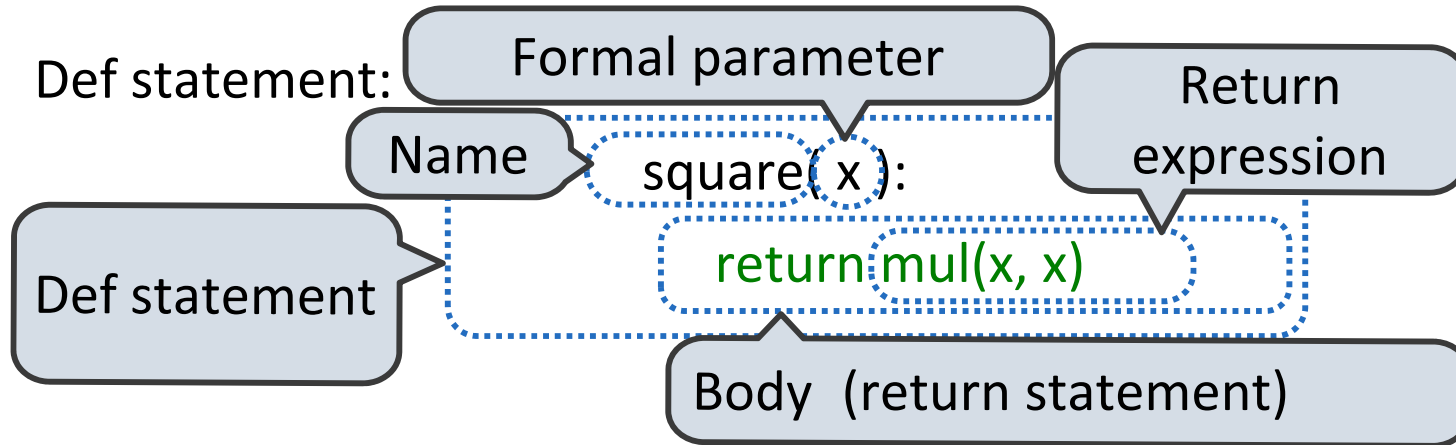
Evaluates to return value below



New frame!

Params bound

Life Cycle of a User-Defined Function



What happens?

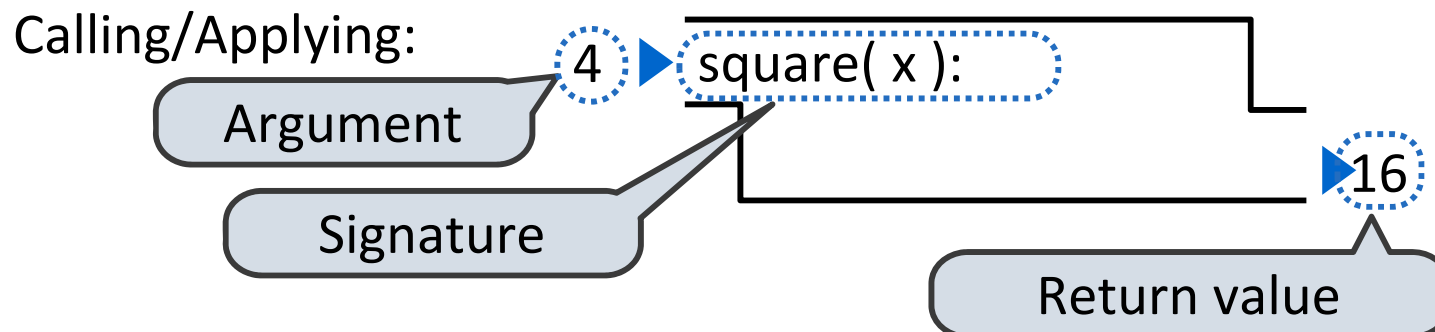
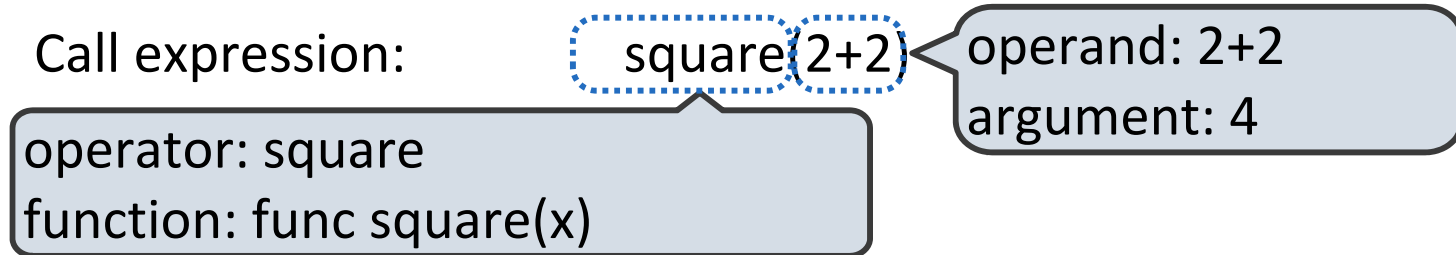
Function created

Name bound

Op's evaluated

Function called with argument(s)

Evaluates to return value below



New frame!

Params bound

Body executed

Python Feature Demonstration



Operators

Multiple Return Values

Docstrings

Doctests

Default Arguments

Statements

Statements



Statements



A *statement* is executed by the interpreter to perform an action

Statements



A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

Statements



A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment

Statements



A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment

```
radius = 10
```

Statements



A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment

```
radius = 10
```

- A function definition

A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment

```
radius = 10
```

- A function definition

```
def square(x):
```

Statements



A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment

```
radius = 10
```

- A function definition

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

Statements



A *statement* is executed by the interpreter to perform an action

Types of statements we have seen so far

- An assignment

```
radius = 10
```

- A function definition

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

- Returns, imports, assertions

Local Assignment



```
1 def percent_difference(x, y):  
→ 2     difference = abs(x-y)  
→ 3     return 100 * difference / x  
4 diff = percent_difference(40, 50)
```

Global frame

percent_difference

func percent_difference(x, y)

percent_difference

x 40

y 50

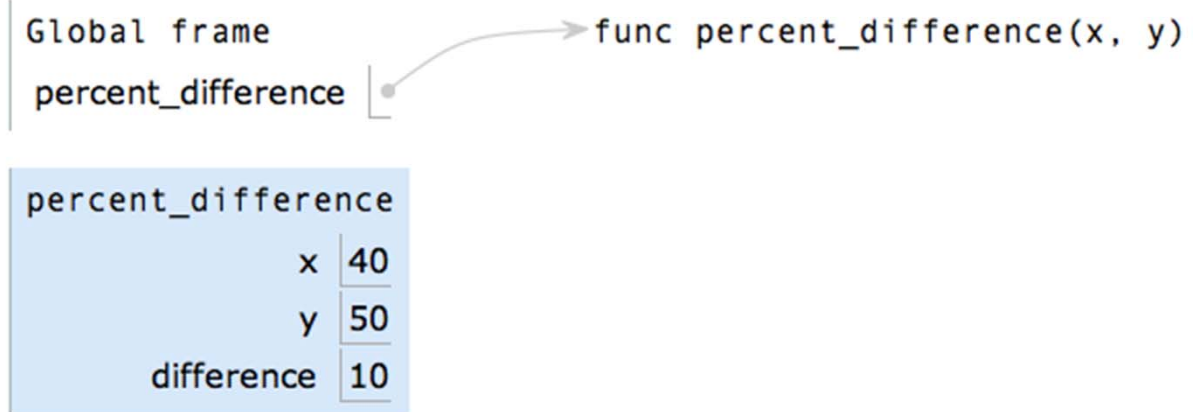
difference 10

Example: <http://goo.gl/1pyzL>

Local Assignment



```
1 def percent_difference(x, y):  
→ 2     difference = abs(x-y)  
→ 3     return 100 * difference / x  
4 diff = percent_difference(40, 50)
```



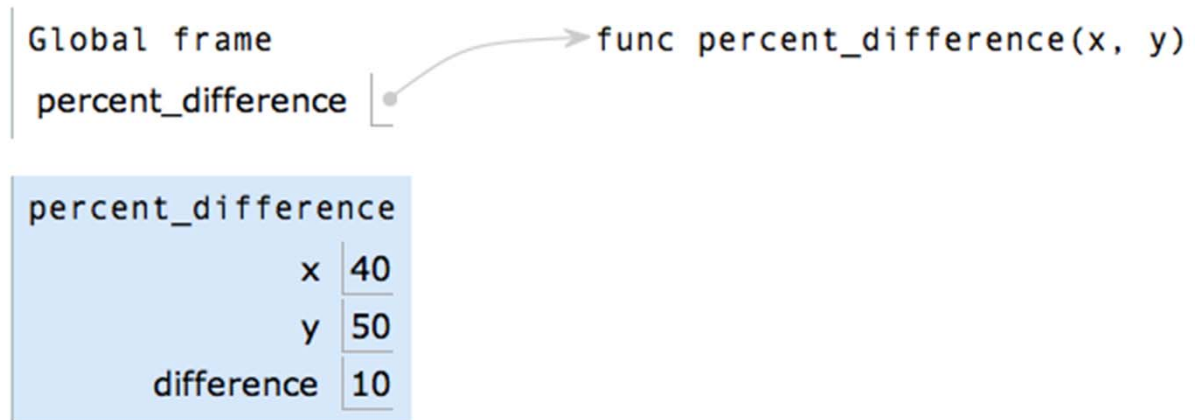
Execution rule for assignment statements:

Example: <http://goo.gl/1pyzL>

Local Assignment



```
1 def percent_difference(x, y):  
→ 2     difference = abs(x-y)  
→ 3     return 100 * difference / x  
4 diff = percent_difference(40, 50)
```



Execution rule for assignment statements:

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

Example: <http://goo.gl/1pyzL>

Compound Statements



A function definition is a *compound statement*

Compound Statements



A function definition is a *compound statement*

Compound statements:

<header>:

<statement>

<statement>

...

<separating header>:

<statement>

<statement>

...

...

Compound Statements



A function definition is a *compound statement*

Compound statements:

Statement

<header>:

<statement>

<statement>

...

<separating header>:

<statement>

<statement>

...

...

Compound Statements



A function definition is a *compound statement*

Compound statements:

Statement

Clause

<header>:
 <statement>
 <statement>
 ...

<separating header>:
 <statement>
 <statement>
 ...

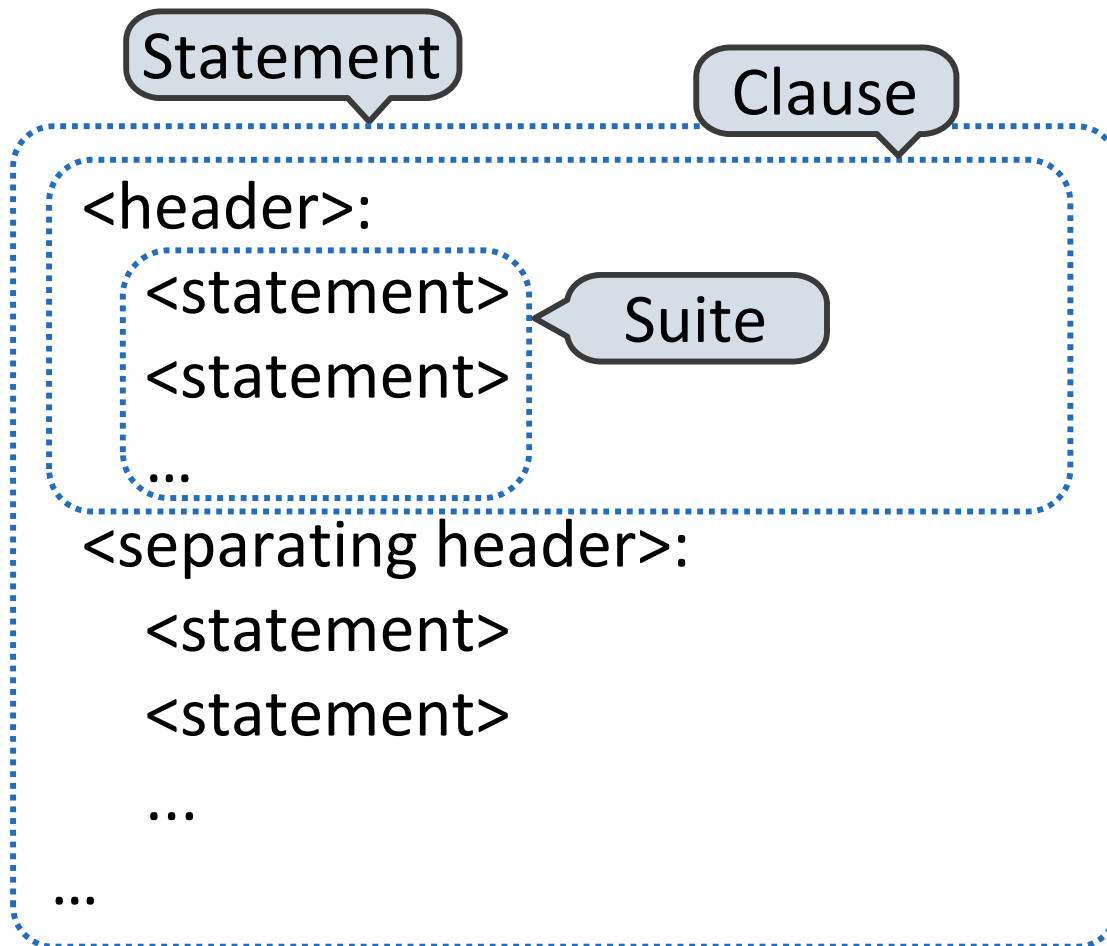
...

Compound Statements



A function definition is a *compound statement*

Compound statements:

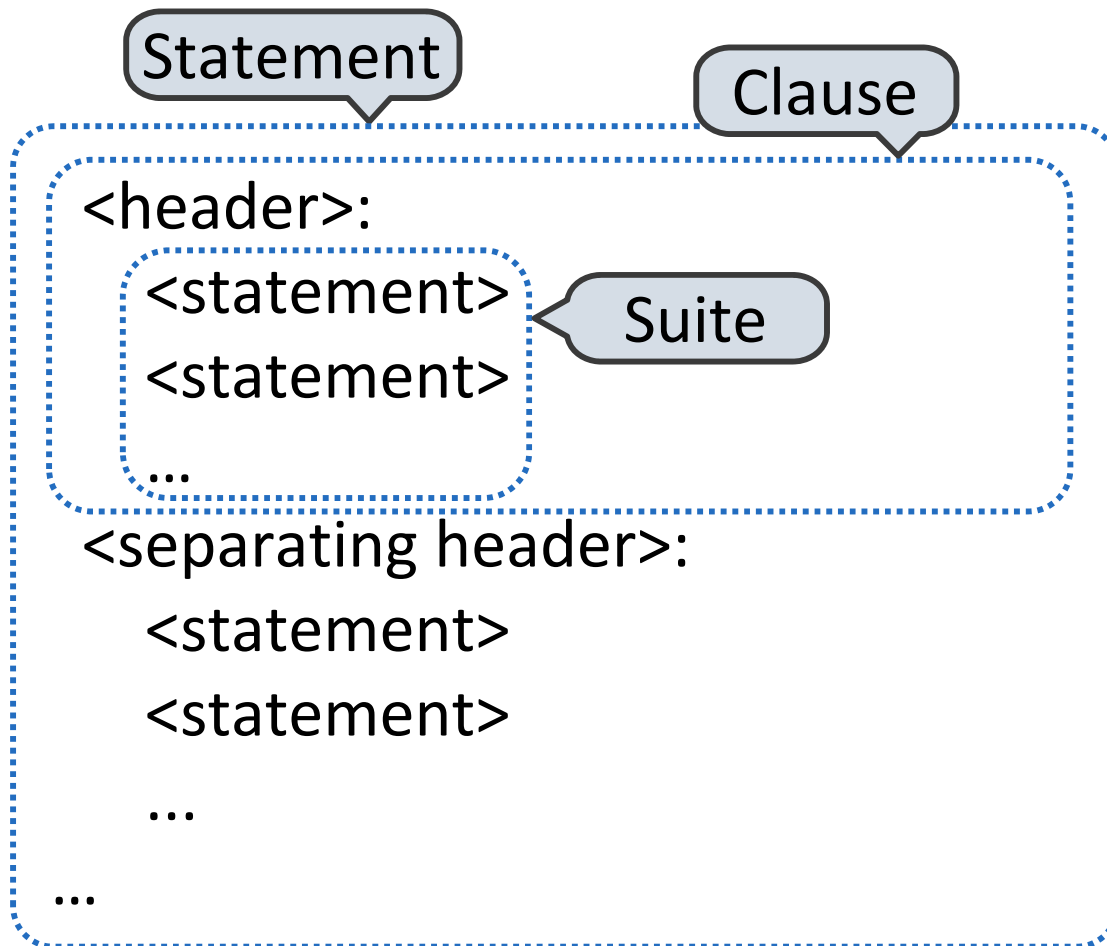


Compound Statements



A function definition is a *compound statement*

Compound statements:



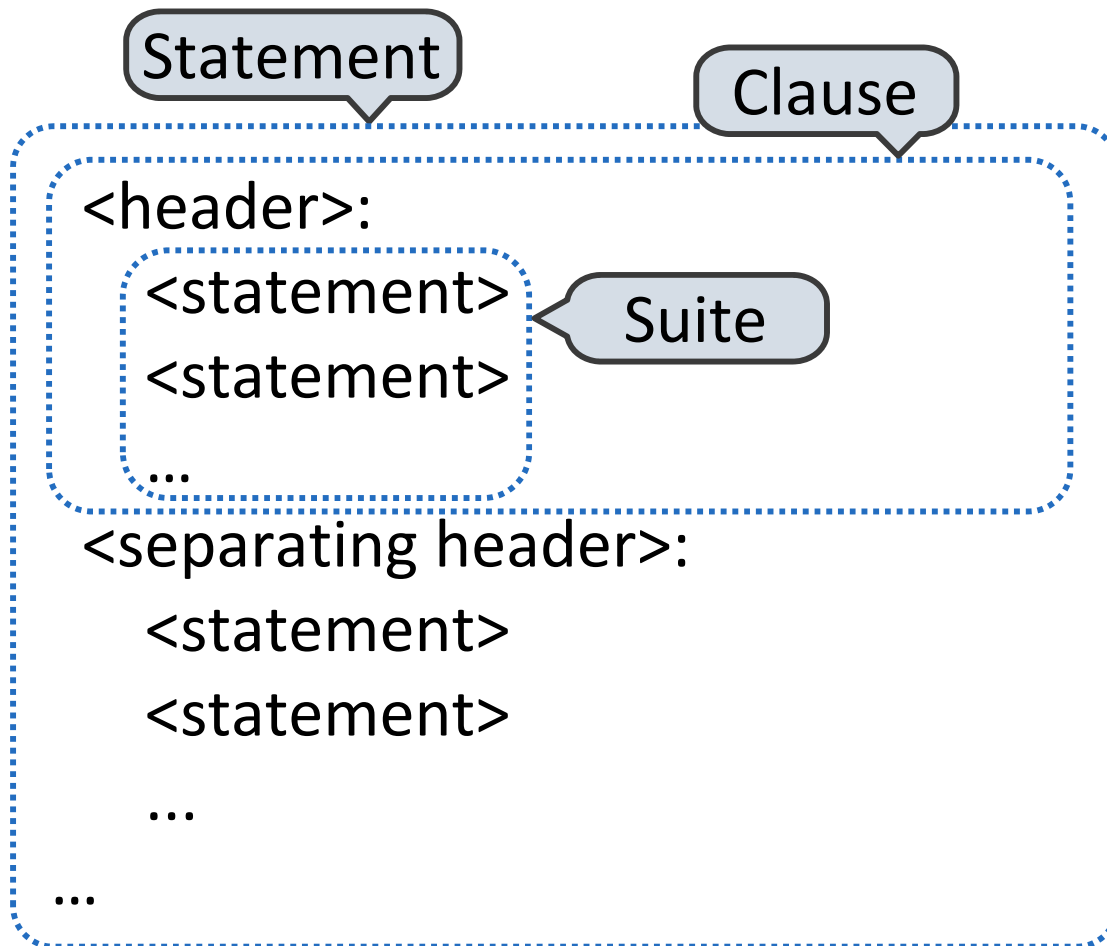
The first header determines a statement's type

Compound Statements



A function definition is a *compound statement*

Compound statements:



The first header determines a statement's type

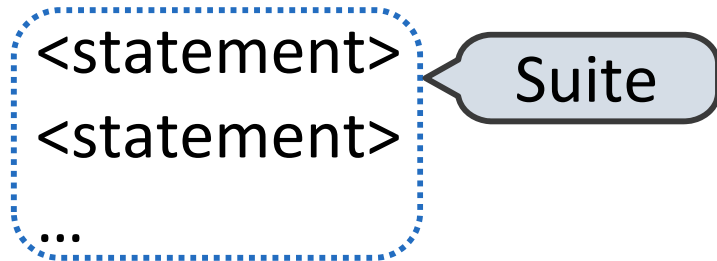
The header of a clause "controls" the suite that follows

Compound Statements



Compound statements:

<header>:



<separating header>:

<statement>

<statement>

...

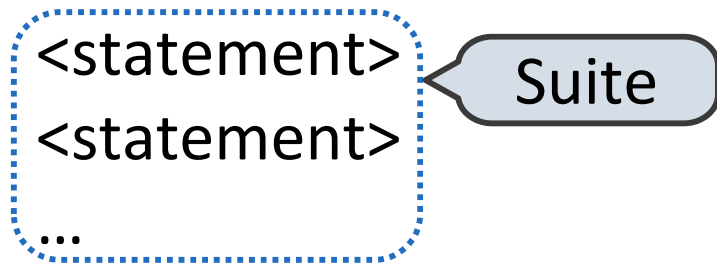
...

Compound Statements



Compound statements:

<header>:



<separating header>:

<statement>

<statement>

...

...

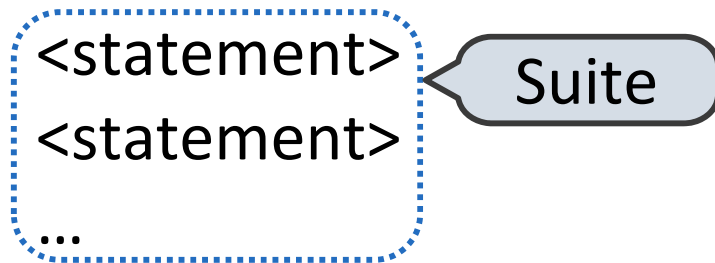
A suite is a sequence of statements

Compound Statements



Compound statements:

<header>:



<separating header>:

<statement>

<statement>

...

...

A suite is a sequence of statements

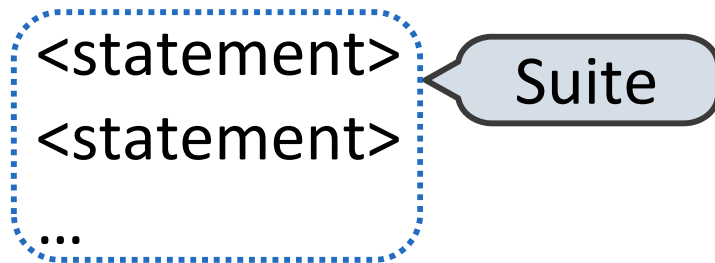
To “execute” a suite means to execute its sequence of statements, in order

Compound Statements



Compound statements:

<header>:



<separating header>:

<statement>

<statement>

...

...

A suite is a sequence of statements

To “execute” a suite means to execute its sequence of statements, in order

Execution rule for a sequence of statements:

1. Execute the first
2. Unless directed otherwise, execute the rest

Conditional Statements



```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Conditional Statements



1 statement,
3 clauses,
3 headers,
3 suites

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Conditional Statements



1 statement,
3 clauses,
3 headers,
3 suites

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Execution rule for conditional statements:

Conditional Statements



1 statement,
3 clauses,
3 headers,
3 suites

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Execution rule for conditional statements:

Each clause is considered in order.

1. Evaluate the header's expression.
2. If it is a true value,
execute the suite & skip the remaining clauses.

Boolean Contexts



```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```


Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

Boolean Contexts

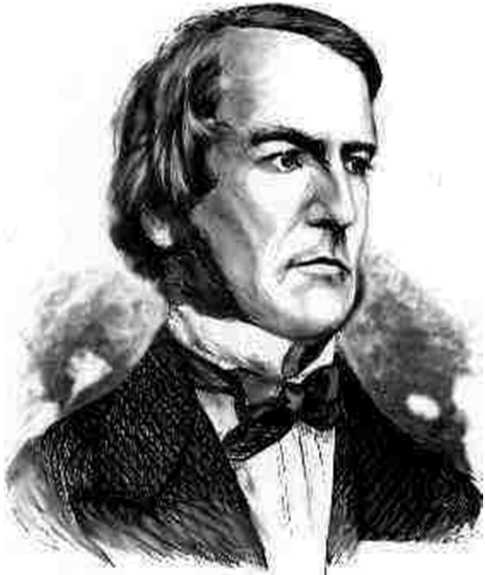


George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

False values in Python: False, 0, "", None

Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

False values in Python: False, 0, "", None (more to come)

Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

False values in Python: False, 0, "", None (more to come)

True values in Python: Anything else (True)

Boolean Contexts



George Boole

```
def absolute_value(x):  
    """Return the absolute value of x."""  
    if x > 0:  
        return x  
    elif x == 0:  
        return 0  
    else:  
        return -x
```

Two boolean contexts

False values in Python: False, 0, "", None (more to come)

True values in Python: Anything else (True)

Read Section 1.5.4!

Iteration



```
i, total = 0, 0
while i < 3:
    i = i + 1
    total = total + i
```

Global frame

i		0
total		0

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Example: <http://goo.gl/mk7Sc>

Iteration



```
▶ i, total = 0, 0
while i < 3:
    i = i + 1
    total = total + i
```

Global frame

i		0
total		0

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    i = i + 1
    total = total + i
```

Global frame

i		0
total		0

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    ▶ i = i + 1
      total = total + i
```

Global frame

i	0
total	0

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    ▶ i = i + 1
      total = total + i
```

Global frame	
i	0 1
total	0

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    ▶ i = i + 1
    ▶ total = total + i
```

Global frame		
i	0	1
total	0	

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    ▶ i = i + 1
    ▶ total = total + i
```

Global frame		
i	0	1
total	0	1

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    ▶ i = i + 1
    ▶ total = total + i
```

Global frame		
i	0	1
total	0	1

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶ while i < 3:
    ▶ i = i + 1
    ▶ total = total + i
```

Global frame		
i	0	1
total	0	1

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶ while i < 3:
▶▶▶ i = i + 1
▶▶▶ total = total + i
```

Global frame			
i	0	1	2
total	0	1	

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶ while i < 3:
▶▶▶ i = i + 1
▶▶▶ total = total + i
```

Global frame			
i	0	1	2
total	0	1	

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶ while i < 3:
▶▶▶ i = i + 1
▶▶▶ total = total + i
```

Global frame			
i	0	1	2
total	0	1	3

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶▶ while i < 3:
▶▶▶▶ i = i + 1
▶▶▶▶ total = total + i
```

Global frame			
i	0	1	2
total	0	1	3

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶▶ while i < 3:
▶▶▶▶ i = i + 1
▶▶▶▶▶ total = total + i
```

Global frame			
i	0	1	2
total	0	1	3

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶▶ while i < 3:
▶▶▶▶ i = i + 1
▶▶▶▶ total = total + i
```

Global frame				
i	0	1	2	3
total	0	1	3	

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶▶ while i < 3:
▶▶▶▶ i = i + 1
▶▶▶▶ total = total + i
```

Global frame				
i	0	1	2	3
total	0	1	3	

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶▶ while i < 3:
▶▶▶▶ i = i + 1
▶▶▶▶ total = total + i
```

Global frame				
i	0	1	2	3
total	0	1	3	6

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Iteration



```
▶ i, total = 0, 0
▶▶▶ while i < 3:
▶▶▶▶ i = i + 1
▶▶▶▶ total = total + i
```

Global frame				
i	0	1	2	3
total	0	1	3	6

Execution rule for while statements:

1. Evaluate the header's expression.
2. If it is a true value, execute the (whole) suite, then return to step 1.

Locally Defined Functions



Locally Defined Functions



Functions can be defined inside other functions

Locally Defined Functions



Functions can be defined inside other functions

```
def sum_of_squares(n):  
    """Sum of the squares of the integers 1 to n"""  
    def square(x):  
        return mul(x, x)  
    total, k = 0, 1  
    while k <= n:  
        total, k = total + square(k), k + 1  
    return total
```

Locally Defined Functions



Functions can be defined inside other functions

What happens when a def is executed?

```
def sum_of_squares(n):  
    """Sum of the squares of the integers 1 to n"""  
    def square(x):  
        return mul(x, x)  
    total, k = 0, 1  
    while k <= n:  
        total, k = total + square(k), k + 1  
    return total
```

Locally Defined Functions



Functions can be defined inside other functions

What happens when a def is executed?

1. Create a function value with the given signature and body

```
def sum_of_squares(n):  
    """Sum of the squares of the integers 1 to n"""  
    def square(x):  
        return mul(x, x)  
    total, k = 0, 1  
    while k <= n:  
        total, k = total + square(k), k + 1  
    return total
```

Locally Defined Functions



Functions can be defined inside other functions

What happens when a def is executed?

1. Create a function value with the given signature and body
2. Bind the given name to that value in the current frame

```
def sum_of_squares(n):  
    """Sum of the squares of the integers 1 to n"""  
    def square(x):  
        return mul(x, x)  
    total, k = 0, 1  
    while k <= n:  
        total, k = total + square(k), k + 1  
    return total
```


Locally Defined Functions



Functions can be defined inside other functions

What happens when a def is executed?

1. Create a function value with the given signature and body
2. Bind the given name to that value in the current frame

The name can then be used to call the function.

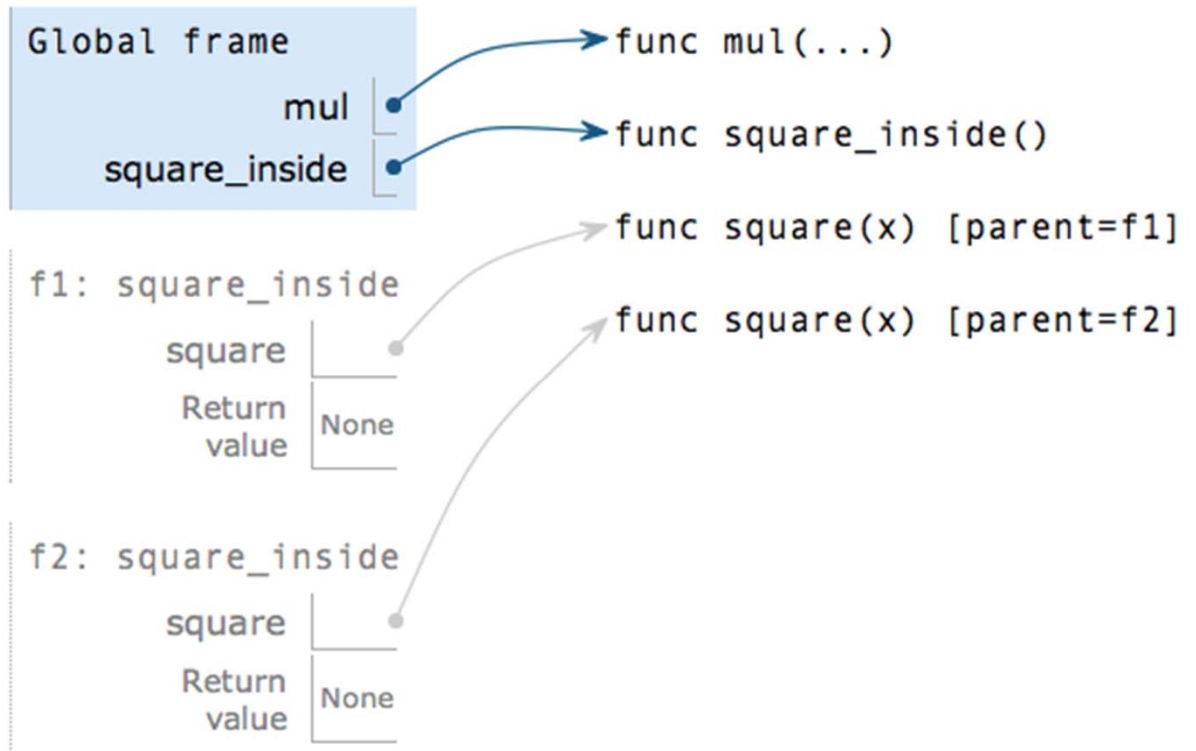
```
def sum_of_squares(n):  
    """Sum of the squares of the integers 1 to n"""  
    def square(x):  
        return mul(x, x)  
    total, k = 0, 1  
    while k <= n:  
        total, k = total + square(k), k + 1  
    return total
```

Locally Defined Functions



The inner definition is executed each time the outer function is called

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



Functions as Return Values



```
def make_adder(n):  
    """Return a function that adds n to its argument.  
  
>>> add_three = make_adder(3)  
>>> add_three(4)  
7  
"""  
def adder(k):  
    return add(n, k)  
return adder
```

Functions as Return Values



Locally defined functions can be returned

```
def make_adder(n):  
    """Return a function that adds n to its argument.  
  
>>> add_three = make_adder(3)  
>>> add_three(4)  
7  
"""  
def adder(k):  
    return add(n, k)  
return adder
```

Functions as Return Values



Locally defined functions can be returned

They have access to the frame in which they are defined

```
def make_adder(n):  
    """Return a function that adds n to its argument.  
  
>>> add_three = make_adder(3)  
>>> add_three(4)  
7  
"""  
def adder(k):  
    return add(n, k)  
return adder
```

Functions as Return Values



Locally defined functions can be returned

They have access to the frame in which they are defined

A function that returns
a function

```
def make_adder(n):  
    """Return a function that adds n to its argument.  
  
    >>> add_three = make_adder(3)  
    >>> add_three(4)  
    7  
    """  
    def adder(k):  
        return add(n, k)  
    return adder
```

Functions as Return Values



Locally defined functions can be returned

They have access to the frame in which they are defined

A function that returns
a function

```
def make_adder(n):  
    """Return a function that adds n to its argument.  
  
    >>> add_three = make_adder(3)  
    >>> add_three(4)  
    7  
    """  
    def adder(k):  
        return add(n, k)  
    return adder
```

A local
def statement

Functions as Return Values



Locally defined functions can be returned

They have access to the frame in which they are defined

A function that returns
a function

```
def make_adder(n):  
    """Return a function that adds n to its argument.  
  
>>> add_three = make_adder(3)  
>>> add_three(4)  
7  
"""  
    def adder(k):  
        return add(n, k)  
    return adder
```

The name add_three is
bound to a function

A local
def statement

Functions as Return Values



Locally defined functions can be returned

They have access to the frame in which they are defined

A function that returns
a function

```
def make_adder(n):  
    """Return a function that adds n to its argument.
```

```
>>> add_three = make_adder(3)
```

```
>>> add_three(4)
```

```
7
```

```
"""
```

```
def adder(k):  
    return add(n, k)
```

```
return adder
```

The name `add_three` is
bound to a function

A local
def statement

Can refer to names in the
enclosing function

Call Expressions as Operators



```
make_adder(1)(2)
```

```
def make_adder(n):  
    def adder(k):  
        return add(n, k)  
    return adder
```

Call Expressions as Operators



```
make_adder(1)(2)
```

```
make_adder(1) ( 2 )
```

```
def make_adder(n):  
    def adder(k):  
        return add(n, k)  
    return adder
```

Call Expressions as Operators



```
make_adder(1)(2)
```

```
make_adder(1) ( 2 )
```

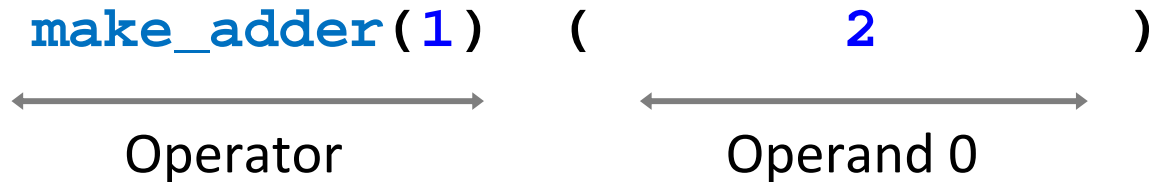
←—————→
Operator

```
def make_adder(n):  
    def adder(k):  
        return add(n, k)  
    return adder
```

Call Expressions as Operators



`make_adder(1)(2)`

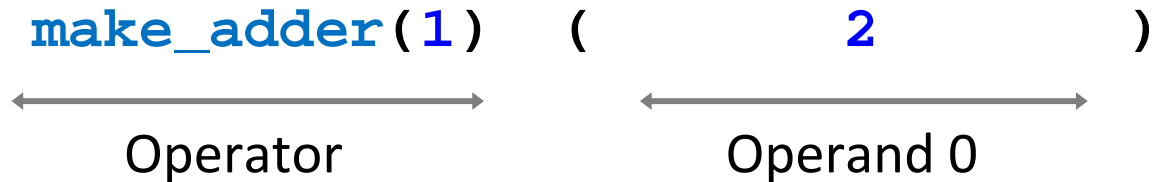


```
def make_adder(n):  
    def adder(k):  
        return add(n, k)  
    return adder
```

Call Expressions as Operators



`make_adder(1)(2)`



An expression that evaluates to a function value

```
def make_adder(n):  
    def adder(k):  
        return add(n, k)  
    return adder
```

Call Expressions as Operators



`make_adder(1)(2)`

`make_adder(1)` (`2`)
← Operator ← Operand 0

An expression that evaluates to a function value

An expression that evaluates to any value

```
def make_adder(n):  
    def adder(k):  
        return add(n, k)  
    return adder
```