61A Lecture 3

Friday, September 5

Announcements

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high – you must make substantial progress

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high — you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high – you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
- Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high – you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
- Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  - 3 points, similar in format to homework, but graded for correctness

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high – you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
- Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  - 3 points, similar in format to homework, but graded for correctness
  - If you score 0/3, you will need to talk to the course staff or be dropped

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high – you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
- Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  - 3 points, similar in format to homework, but graded for correctness
  - If you score 0/3, you will need to talk to the course staff or be dropped
  - Open-computer: You can use the Python interpreter, watch course videos, etc.

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high — you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
- Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  - 3 points, similar in format to homework, but graded for correctness
  - If you score 0/3, you will need to talk to the course staff or be dropped
  - Open-computer: You can use the Python interpreter, watch course videos, etc.
  - Closed-help: Please don't **talk to your classmates**, search for answers, etc.

Announcements

- There's plenty of room in live lecture if you want to come (but videos are still better)
- ***Please*** don't make noise outside of the previous lecture!
- Homework 1 is due next Wednesday 9/10 at 2pm (***Changed from original time!***)
  - Homework is graded on effort, but the bar is high — you must make substantial progress
  - Monday homework parties 3pm–4pm in Wozniak Lounge and 6pm–8pm in 2050 VLSB
- Take-home quiz released next Wednesday 9/10 at 3pm, due Thursday 9/12 at 11:59pm
  - 3 points, similar in format to homework, but graded for correctness
  - If you score 0/3, you will need to talk to the course staff or be dropped
  - Open-computer: You can use the Python interpreter, watch course videos, etc.
  - Closed-help: Please don't **talk to your classmates**, search for answers, etc.
- Project 1 due Wednesday 9/17 at 11:59pm.

Multiple Environments

Life Cycle of a User-Defined Function

What happens?

Def statement:

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:

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

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:

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

Def statement

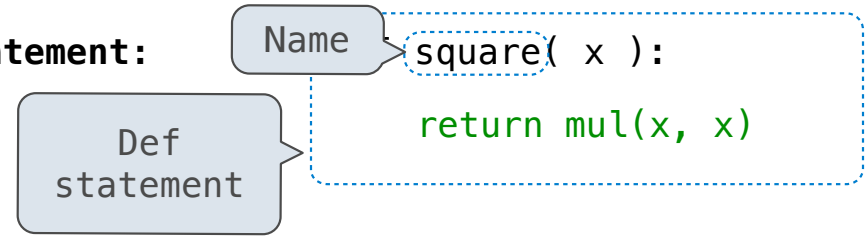
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



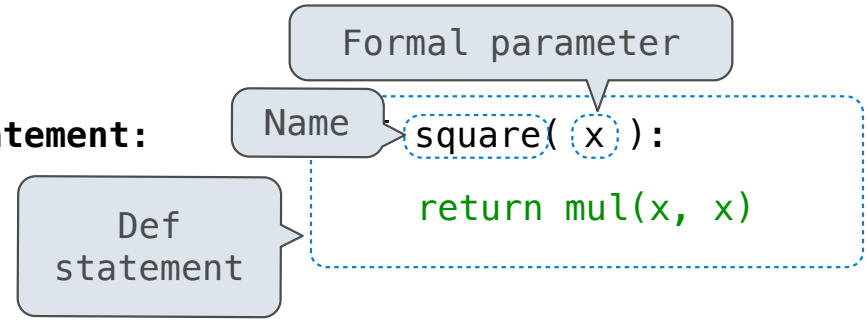
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



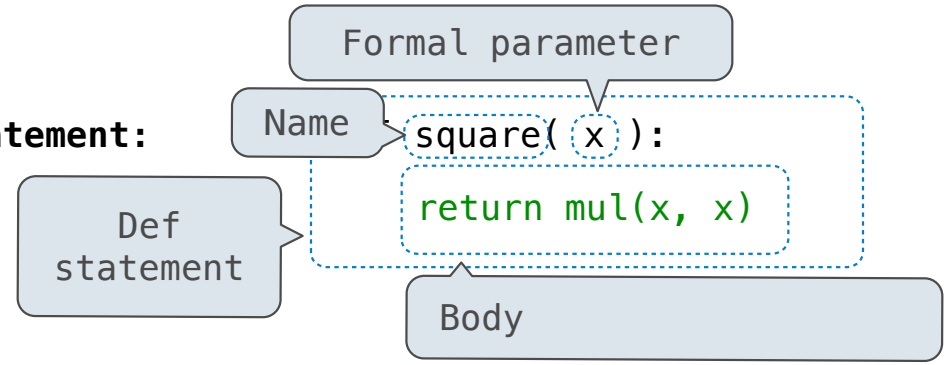
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



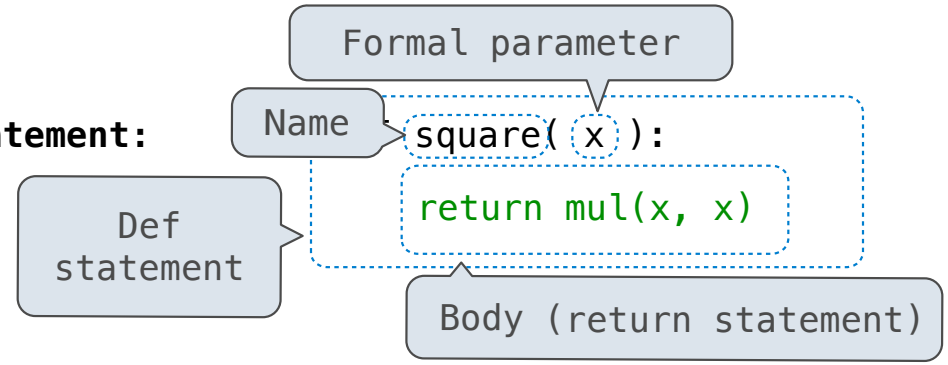
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:



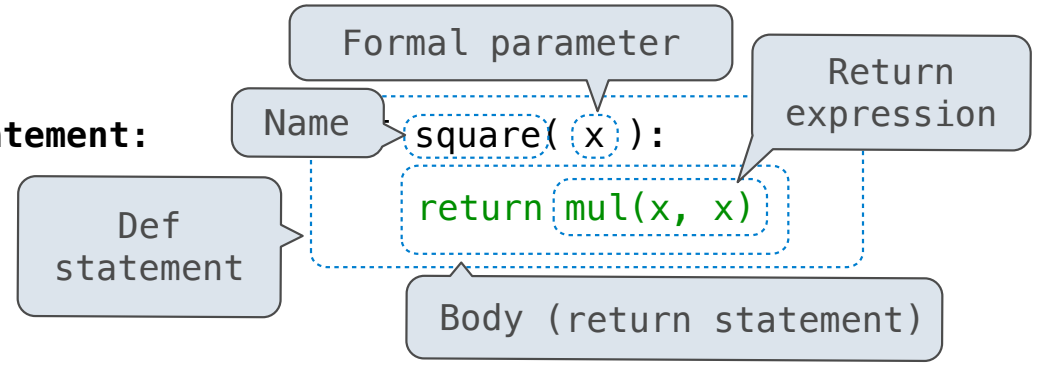
Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

What happens?

Def statement:

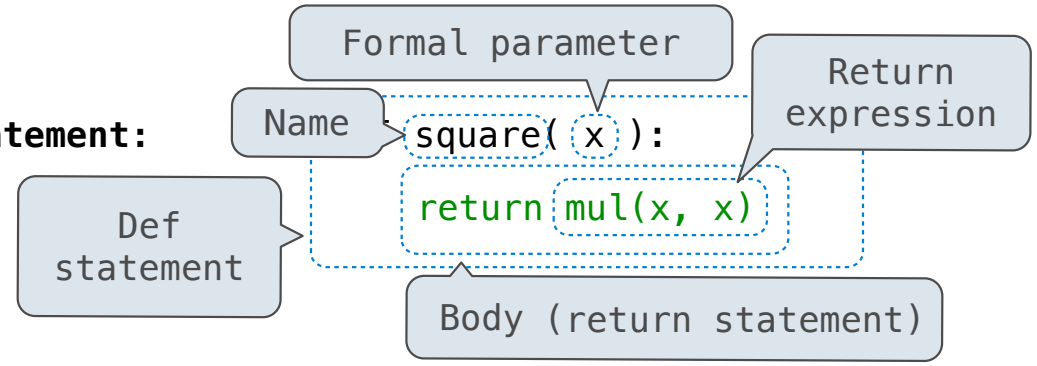


Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

Def statement:



What happens?

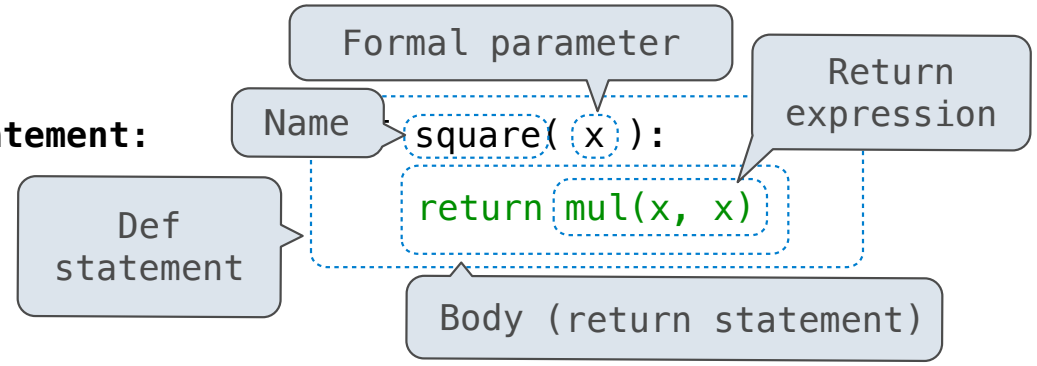
A new function is created!

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

Def statement:



What happens?

A new function is created!

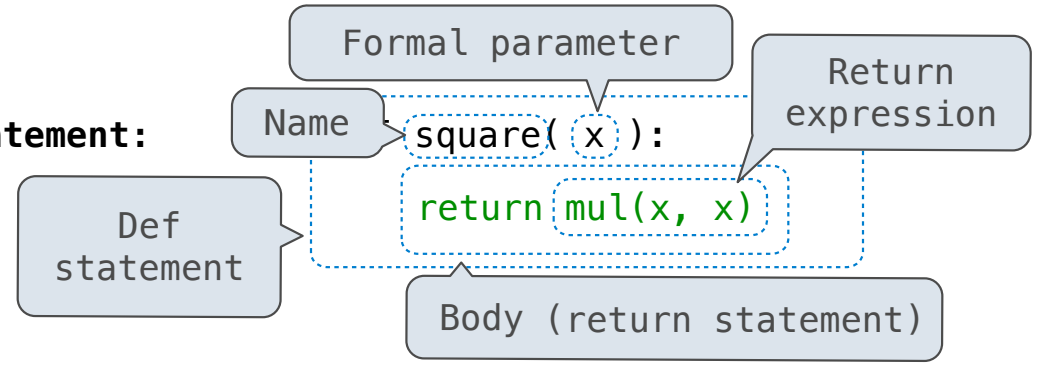
Name bound to that function in the current frame

Call expression:

Calling/Applying:

Life Cycle of a User-Defined Function

Def statement:



What happens?

A new function is created!

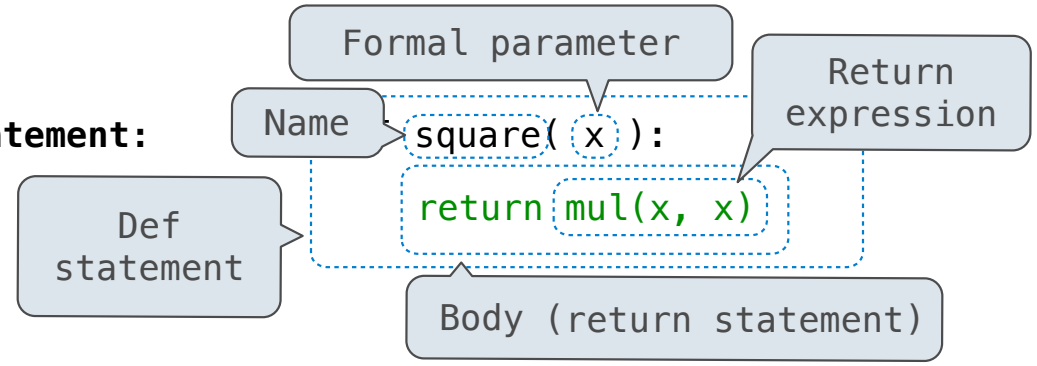
Name bound to that function in the current frame

Call expression: square(2+2)

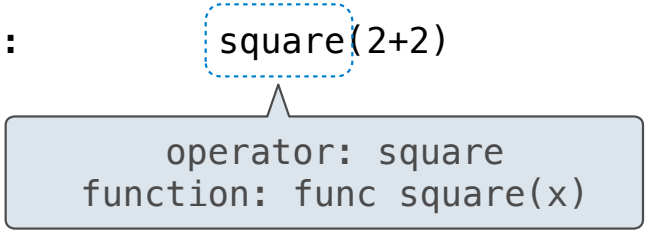
Calling/Applying:

Life Cycle of a User-Defined Function

Def statement:



Call expression:



Calling/Applying:

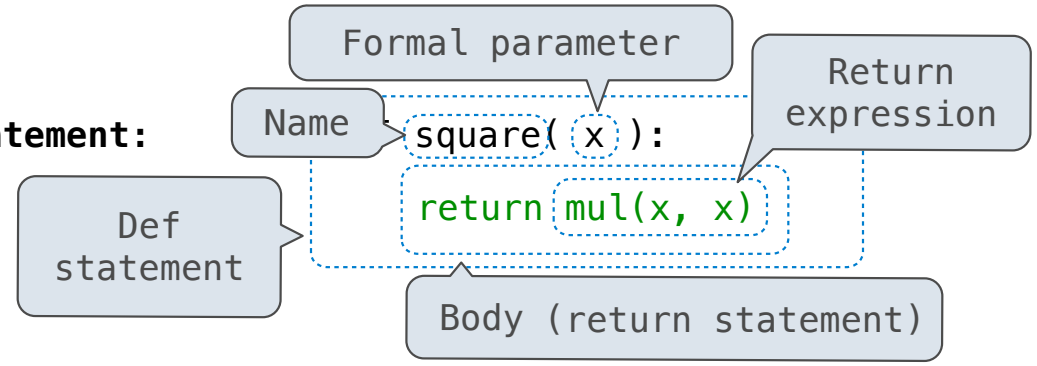
What happens?

A new function is created!

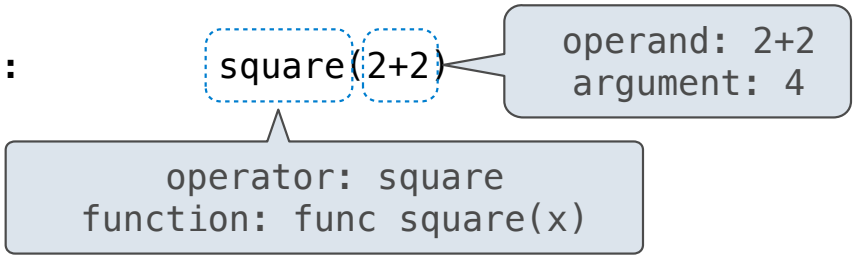
Name bound to that function in the current frame

Life Cycle of a User-Defined Function

Def statement:



Call expression:



Calling/Applying:

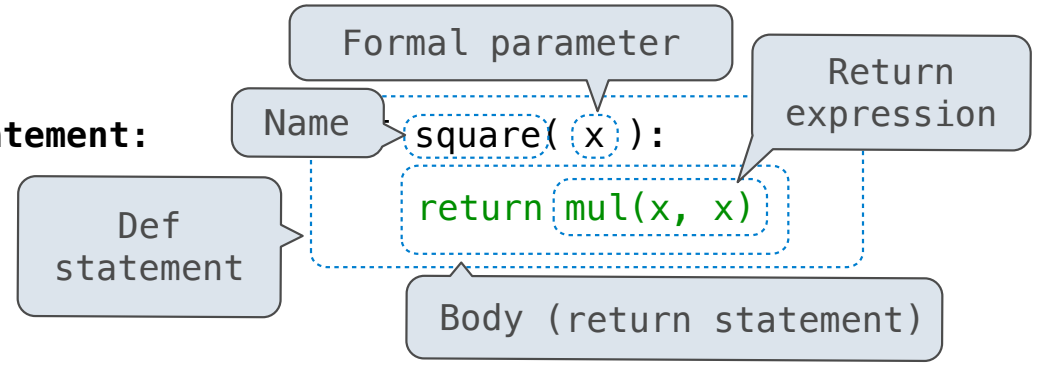
What happens?

A new function is created!

Name bound to that function in the current frame

Life Cycle of a User-Defined Function

Def statement:

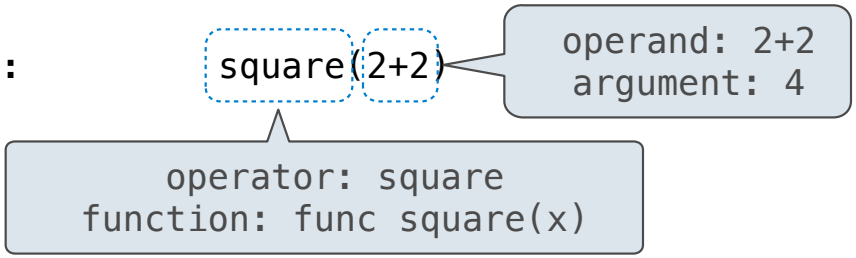


What happens?

A new function is created!

Name bound to that function in the current frame

Call expression:

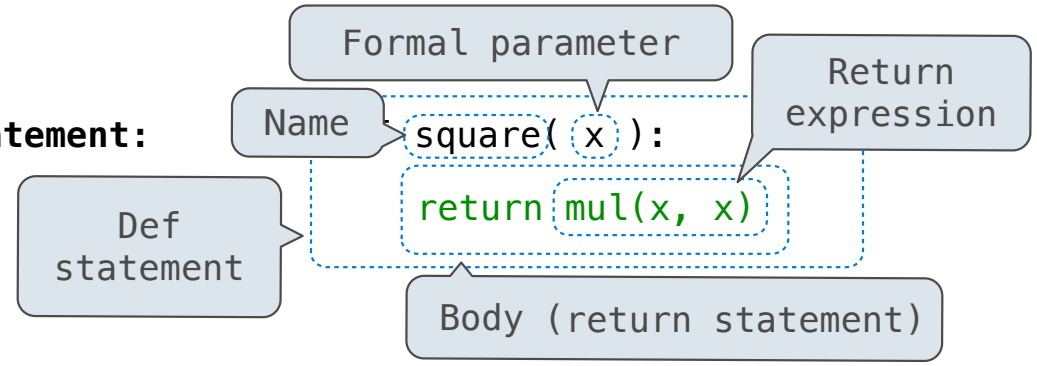


Operator & operands evaluated

Calling/Applying:

Life Cycle of a User-Defined Function

Def statement:

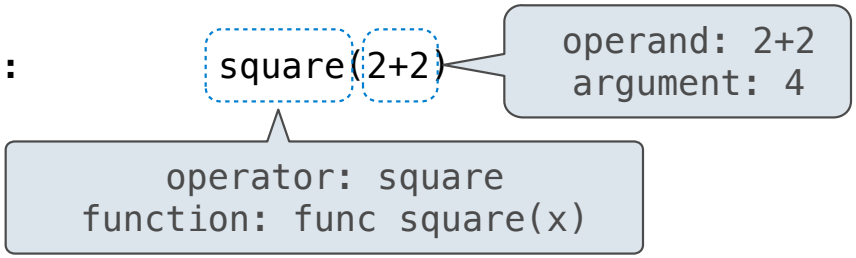


What happens?

A new function is created!

Name bound to that function in the current frame

Call expression:



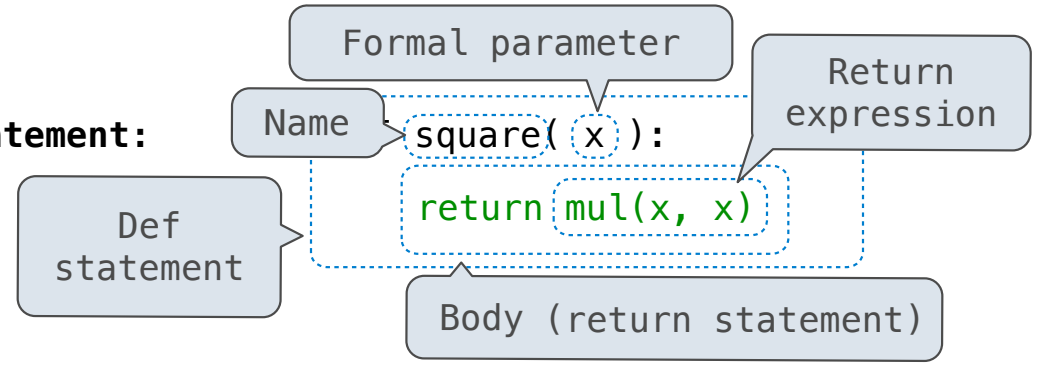
Operator & operands evaluated

Function (value of operator) called on arguments (values of operands)

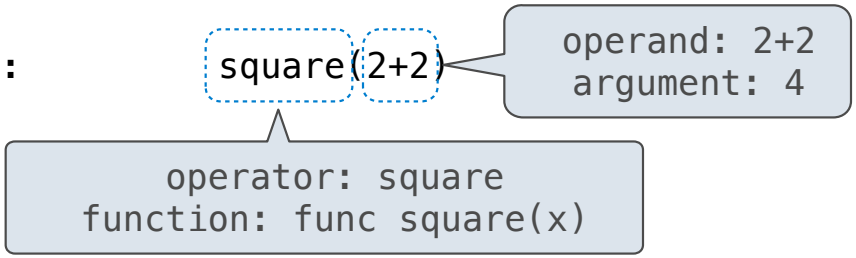
Calling/Applying:

Life Cycle of a User-Defined Function

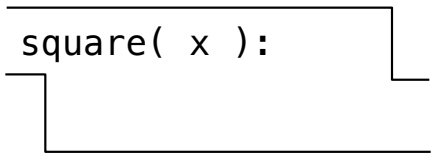
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

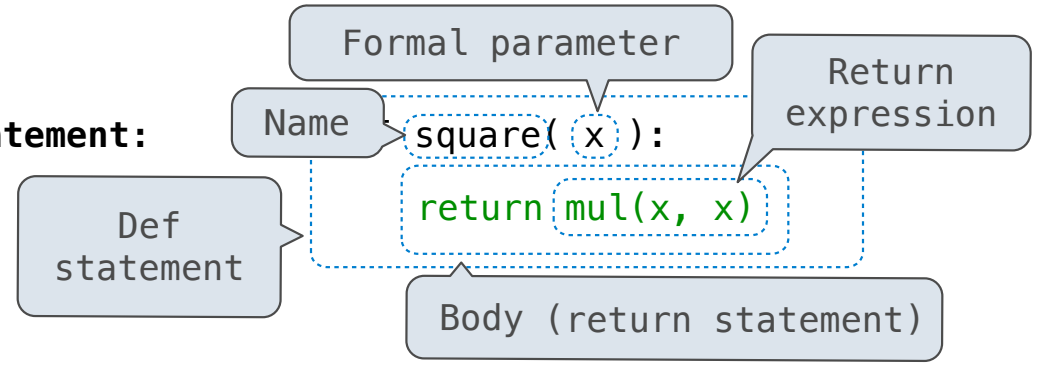
Name bound to that function in the current frame

Operator & operands evaluated

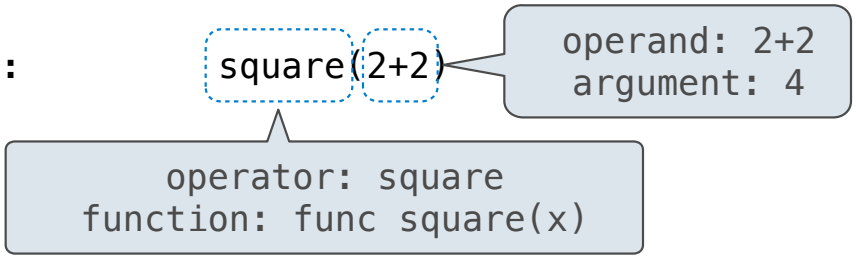
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function

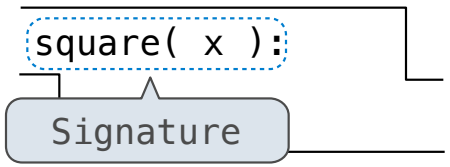
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

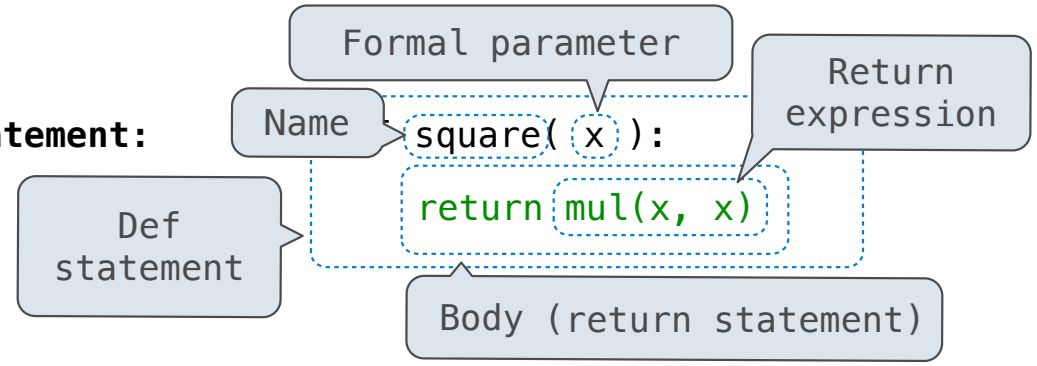
Name bound to that function in the current frame

Operator & operands evaluated

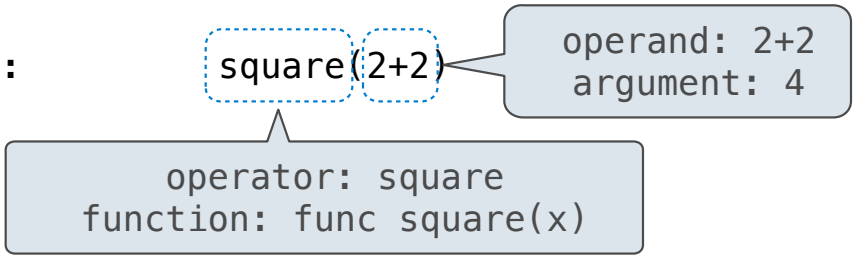
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function

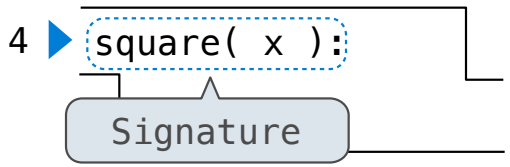
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

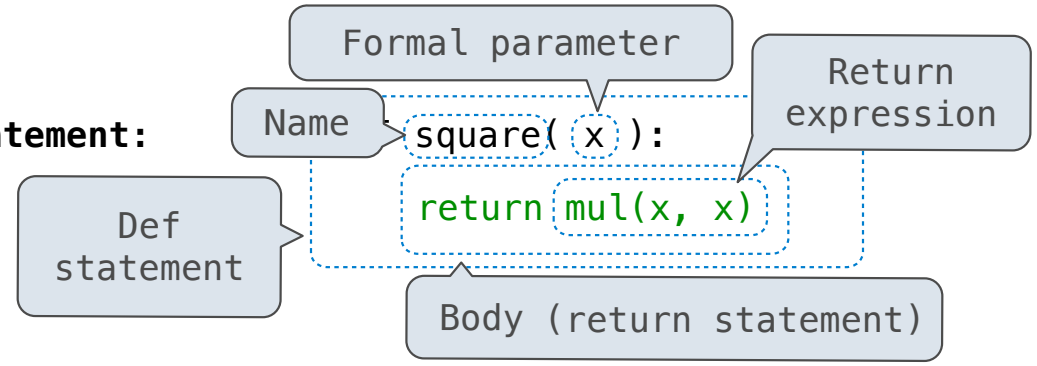
Name bound to that function in the current frame

Operator & operands evaluated

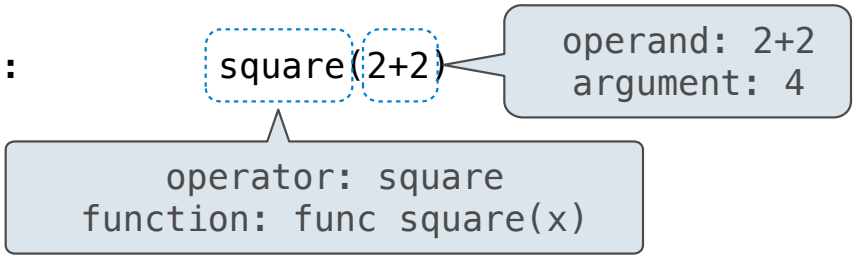
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function

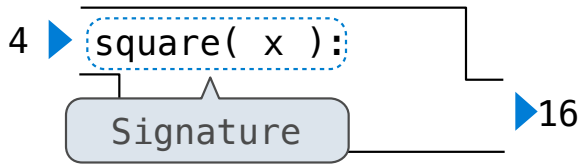
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

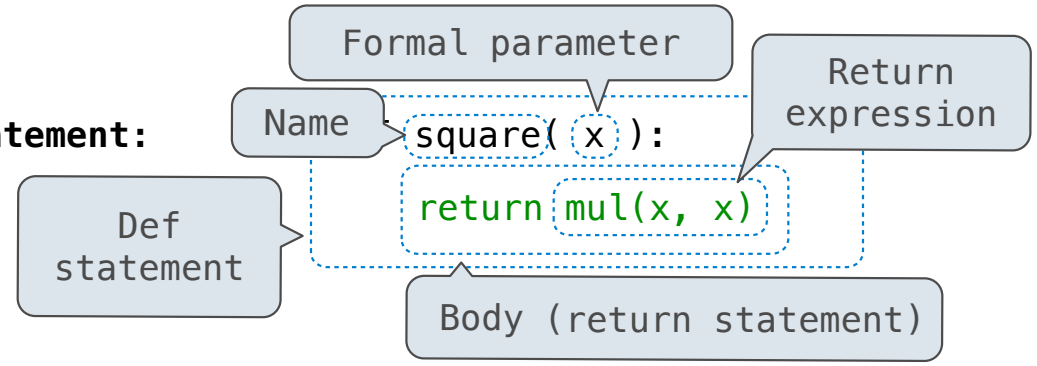
Name bound to that function in the current frame

Operator & operands evaluated

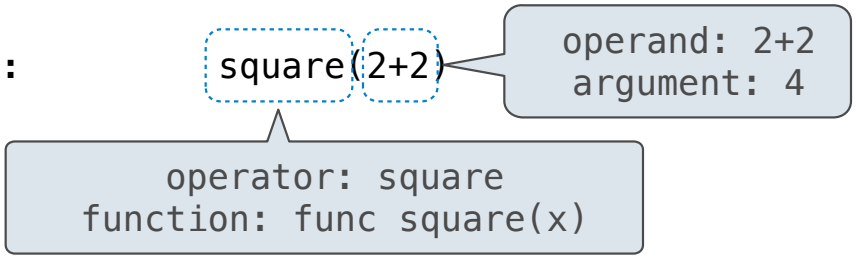
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function

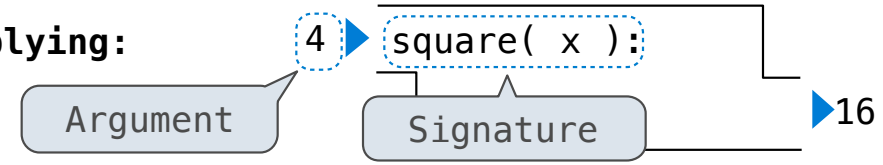
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

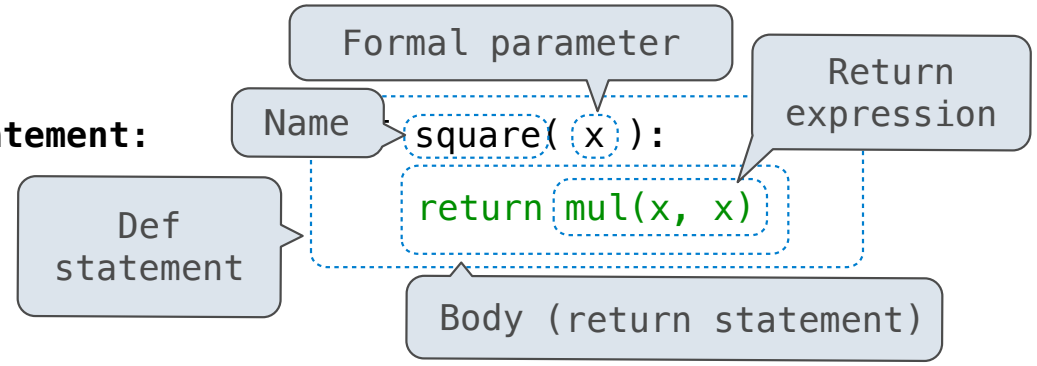
Name bound to that function in the current frame

Operator & operands evaluated

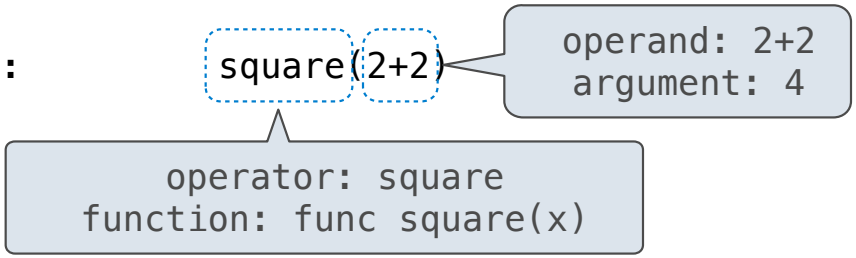
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function

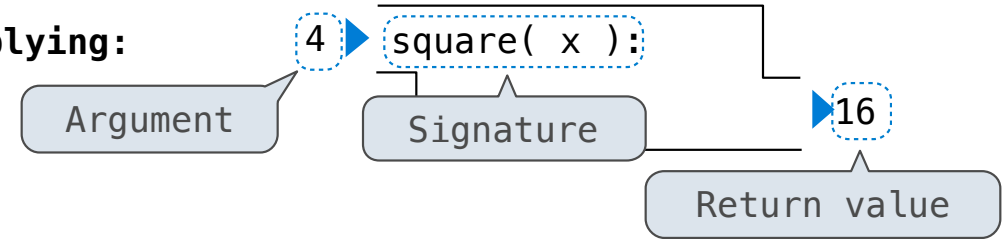
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

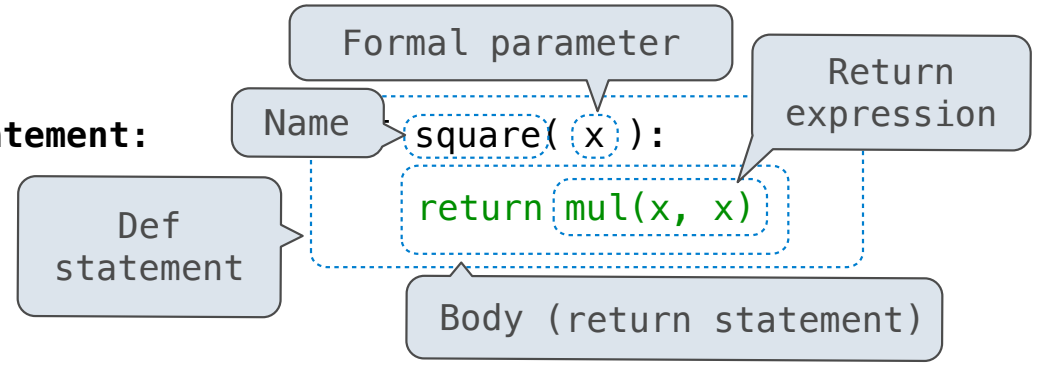
Name bound to that function in the current frame

Operator & operands evaluated

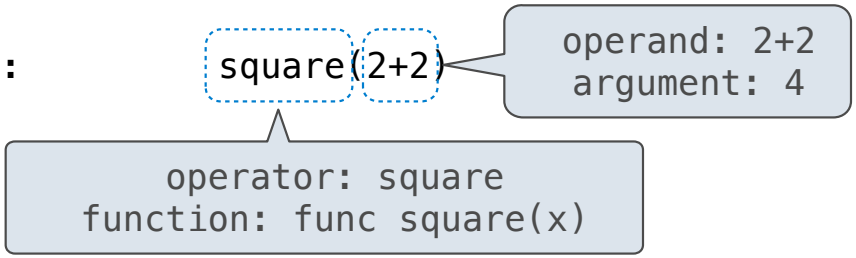
Function (value of operator) called on arguments (values of operands)

Life Cycle of a User-Defined Function

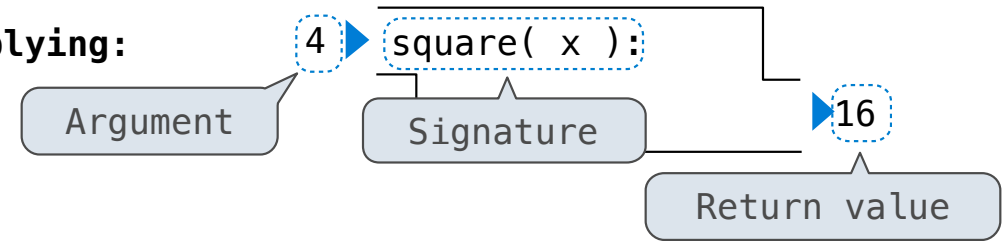
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

Name bound to that function in the current frame

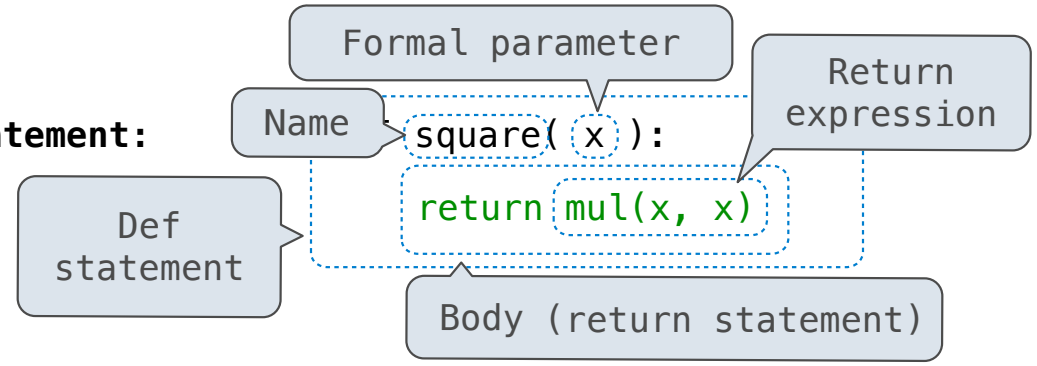
Operator & operands evaluated

Function (value of operator) called on arguments (values of operands)

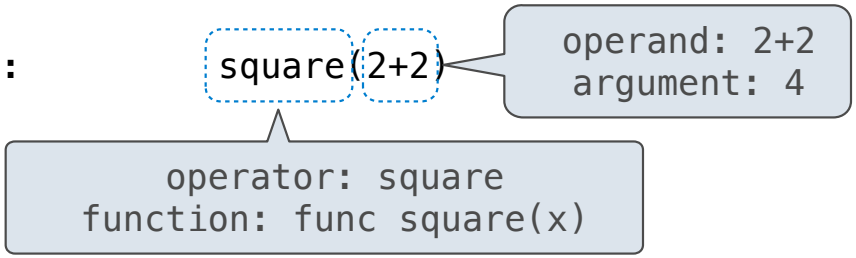
A new frame is created!

Life Cycle of a User-Defined Function

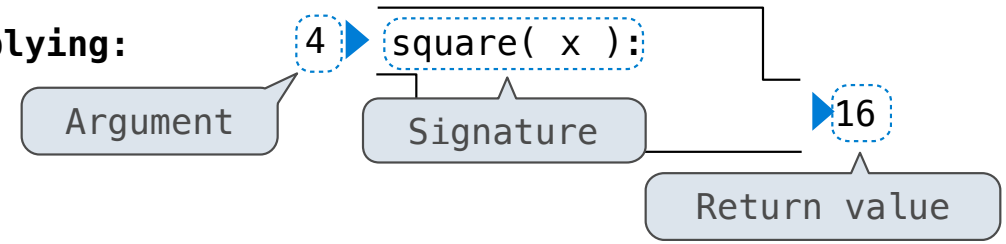
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated

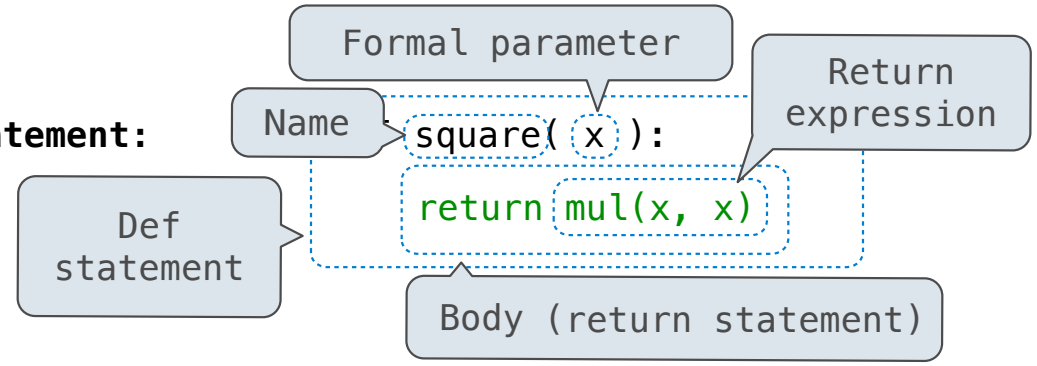
Function (value of operator) called on arguments (values of operands)

A new frame is created!

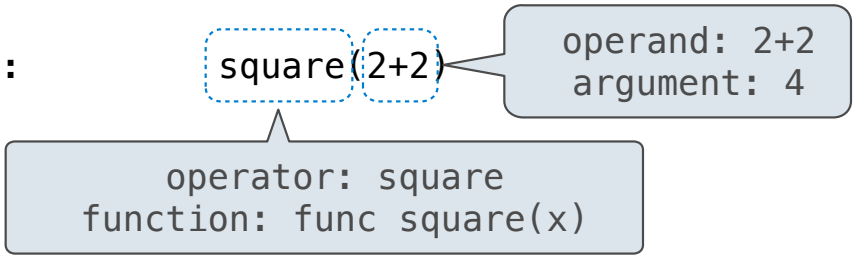
Parameters bound to arguments

Life Cycle of a User-Defined Function

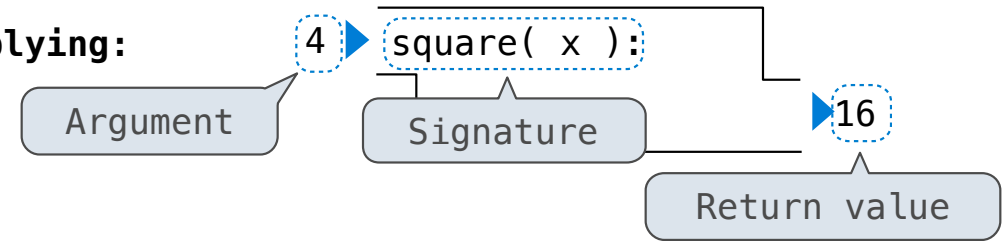
Def statement:



Call expression:



Calling/Applying:



What happens?

A new function is created!

Name bound to that function in the current frame

Operator & operands evaluated

Function (value of operator) called on arguments (values of operands)

A new frame is created!

Parameters bound to arguments

Body is executed in that new environment

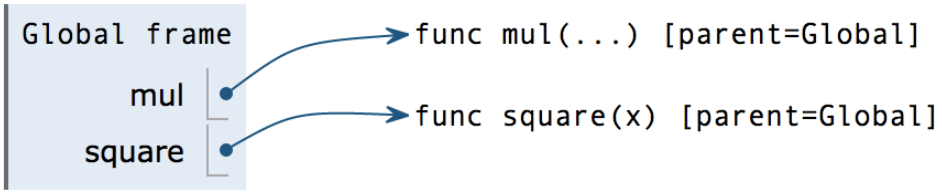
Multiple Environments in One Diagram!

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

Interactive Diagram

Multiple Environments in One Diagram!

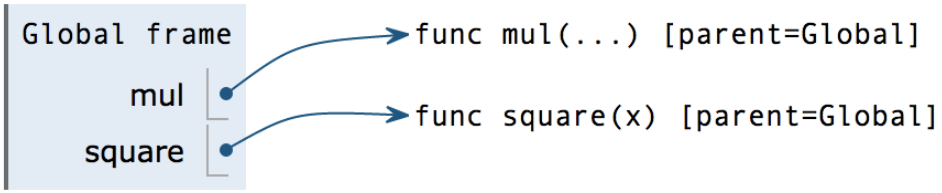
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```



Interactive Diagram

Multiple Environments in One Diagram!

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

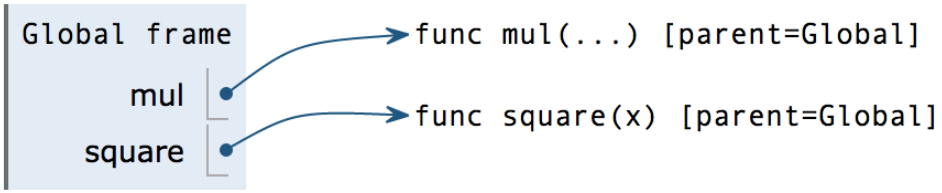


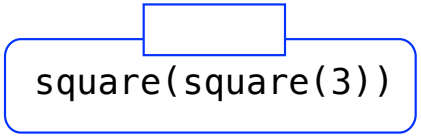

```
square(square(3))
```

Interactive Diagram

Multiple Environments in One Diagram!

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

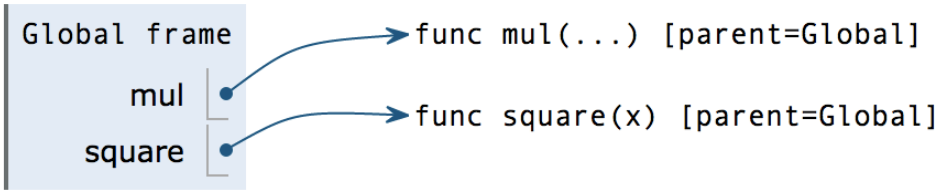


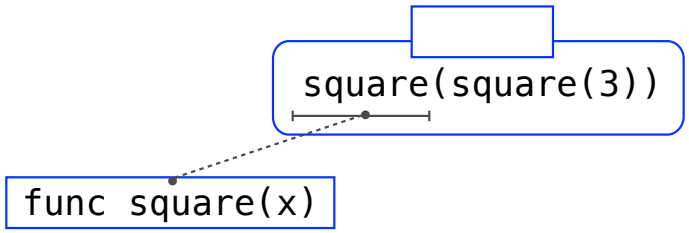


Interactive Diagram

Multiple Environments in One Diagram!

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

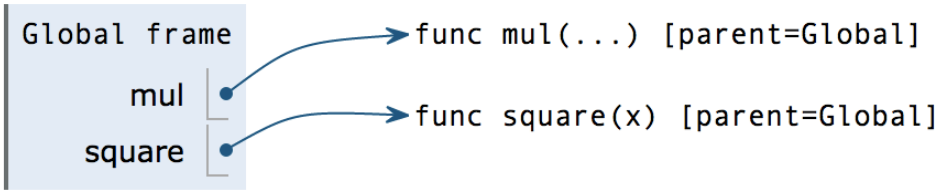


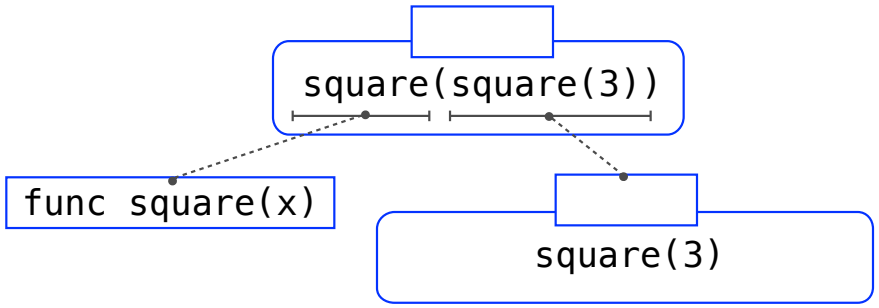


Interactive Diagram

Multiple Environments in One Diagram!

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

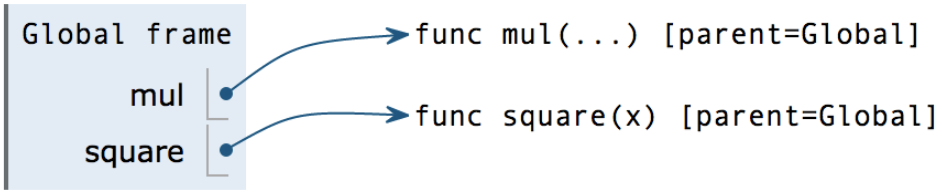


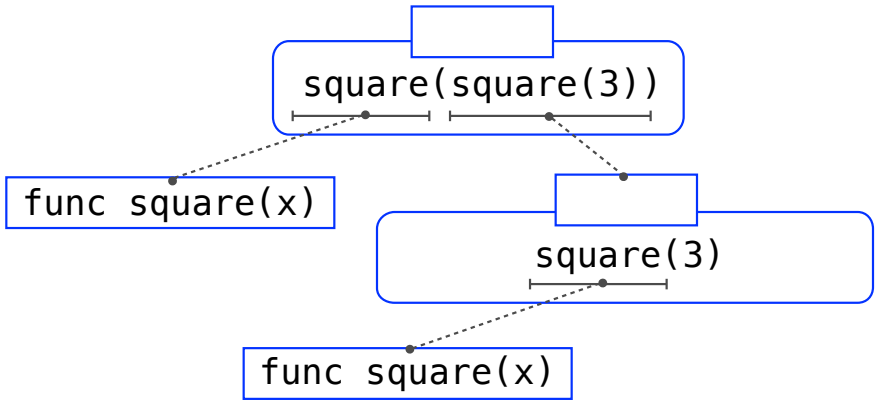


Interactive Diagram

Multiple Environments in One Diagram!

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

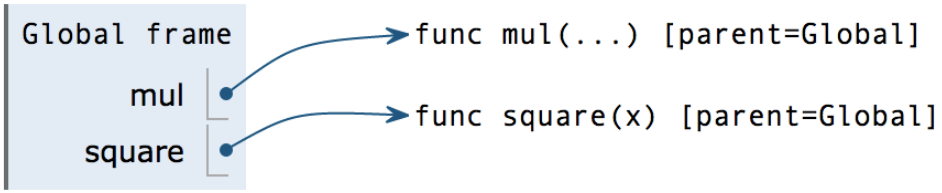


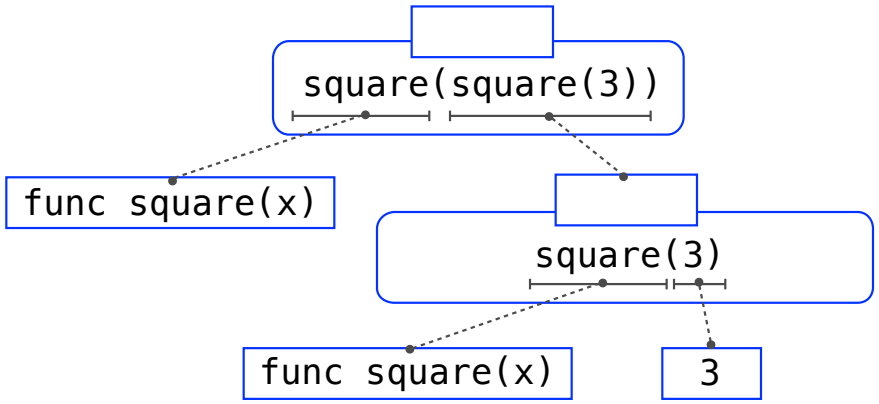


Interactive Diagram

Multiple Environments in One Diagram!

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

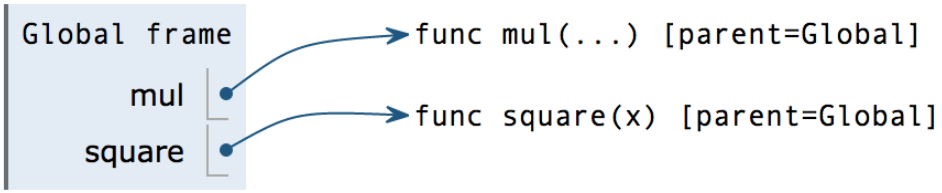


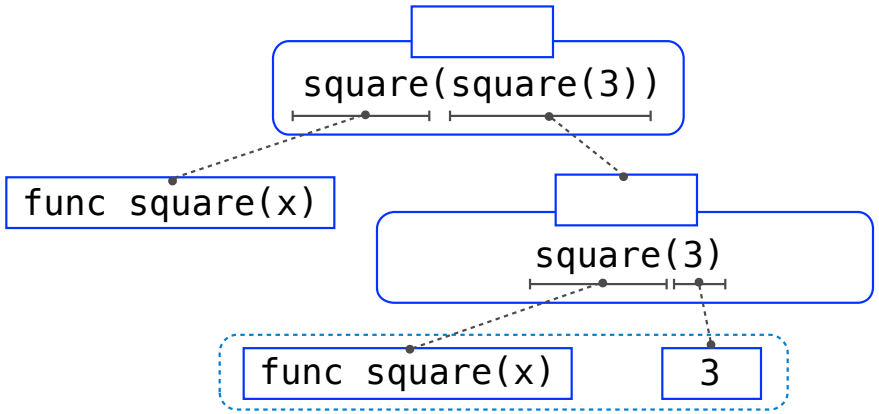


Interactive Diagram

Multiple Environments in One Diagram!

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

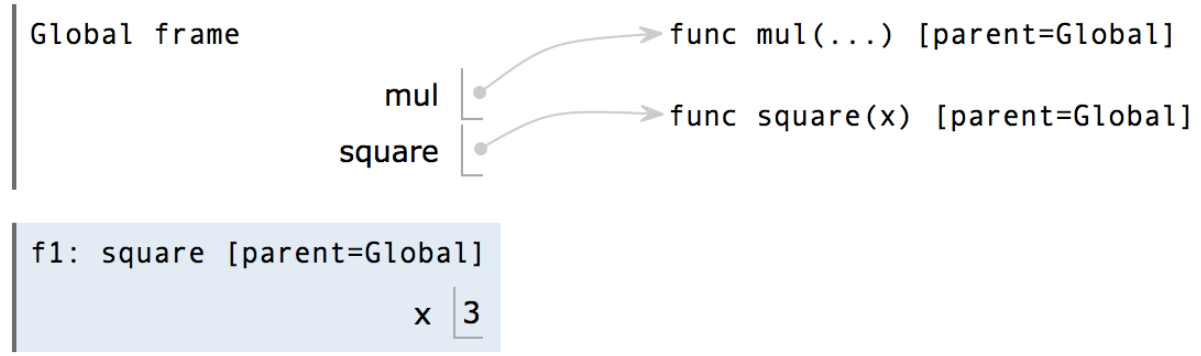


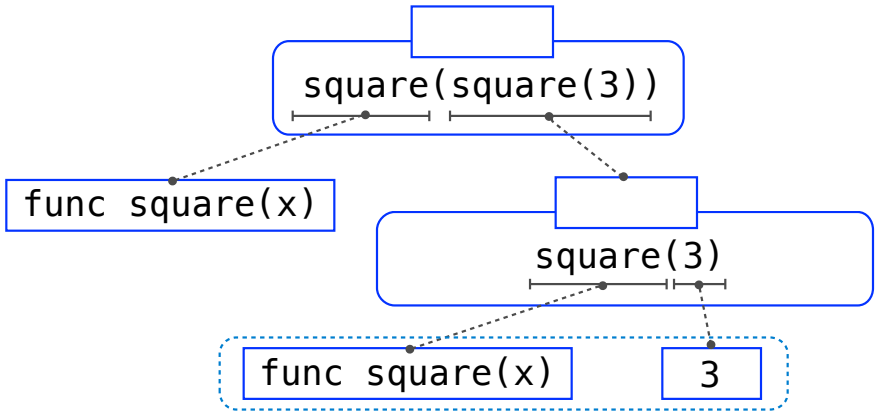


Interactive Diagram

Multiple Environments in One Diagram!

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

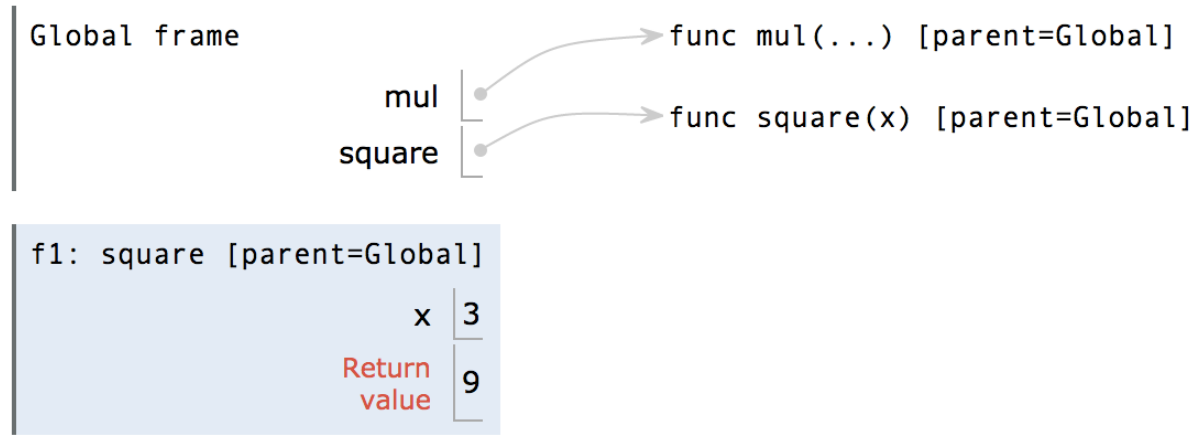


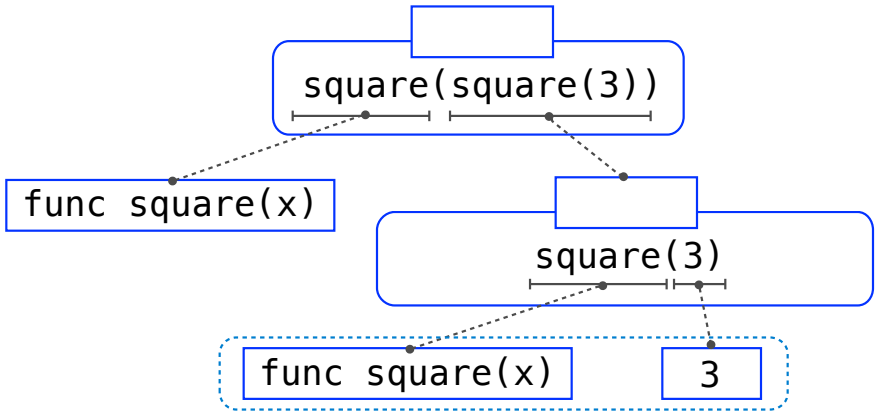


Interactive Diagram

Multiple Environments in One Diagram!

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

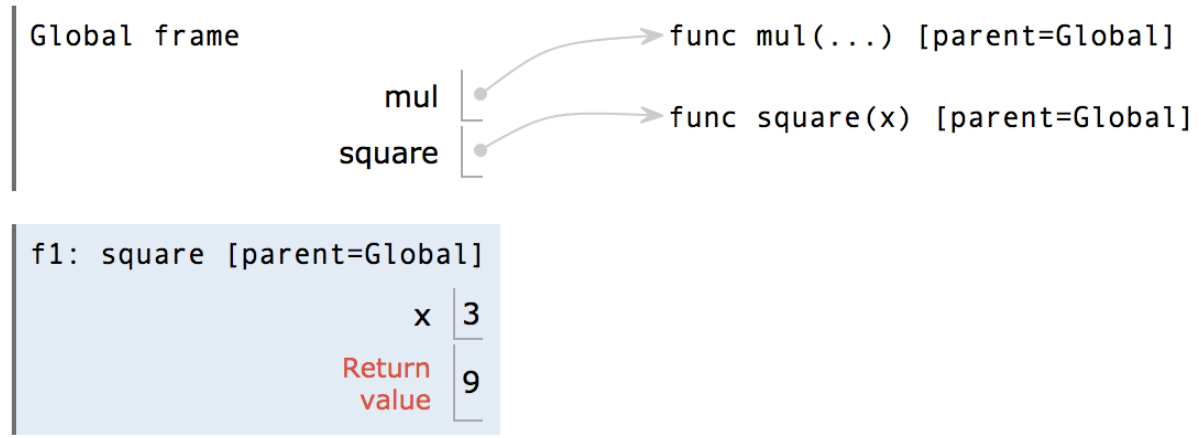


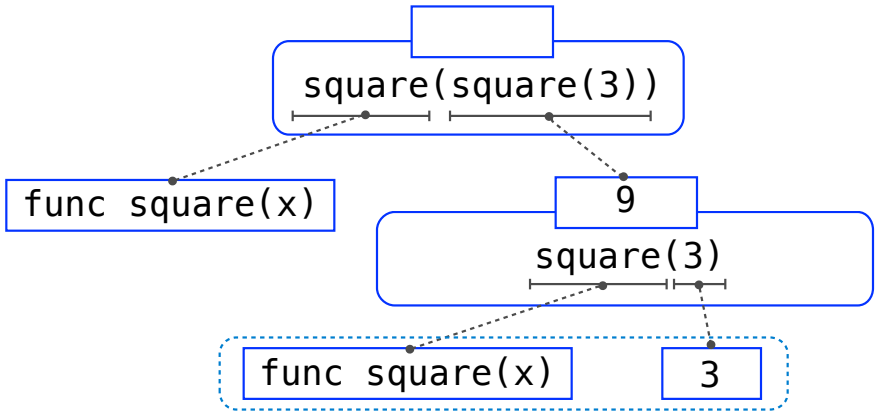


Interactive Diagram

Multiple Environments in One Diagram!

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

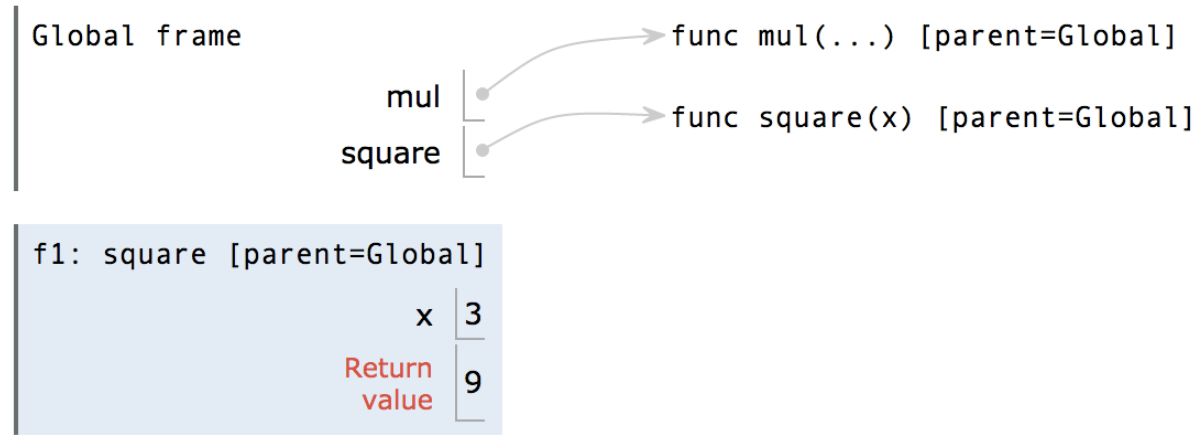


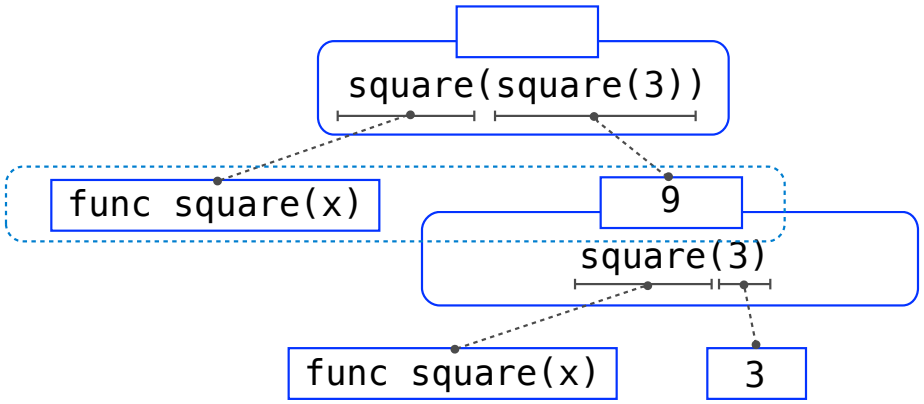


Interactive Diagram

Multiple Environments in One Diagram!

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

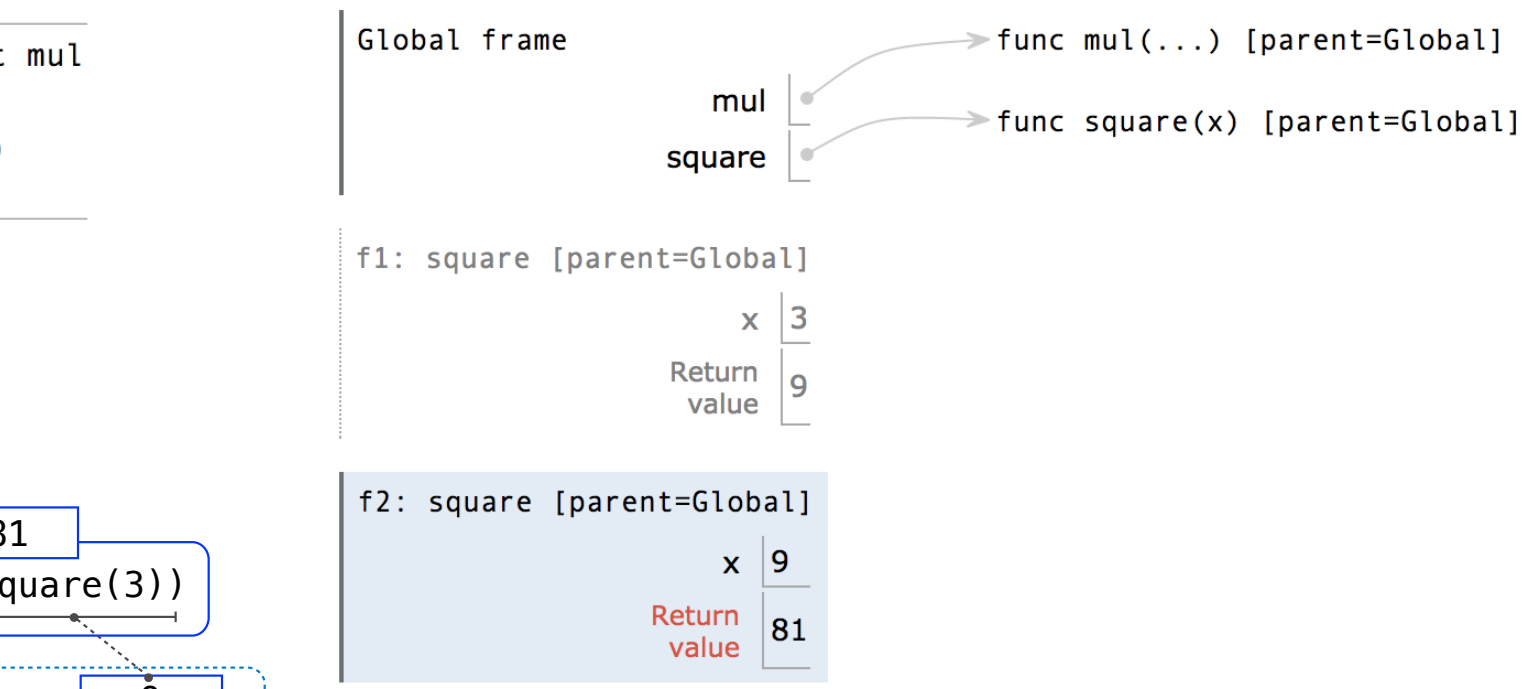


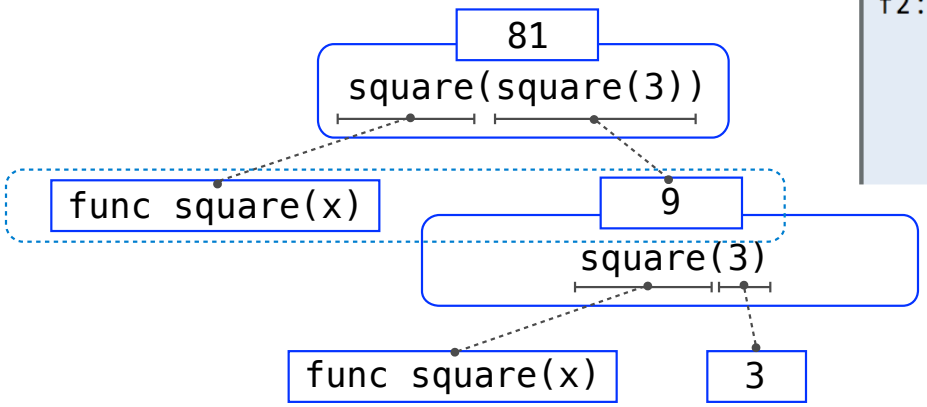


Interactive Diagram

Multiple Environments in One Diagram!

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

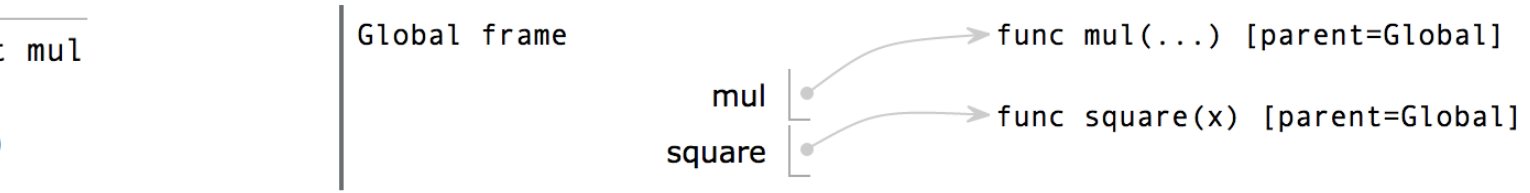


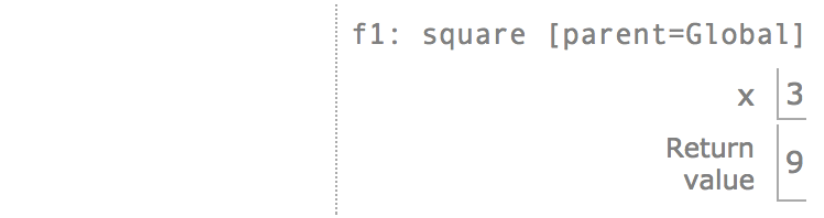


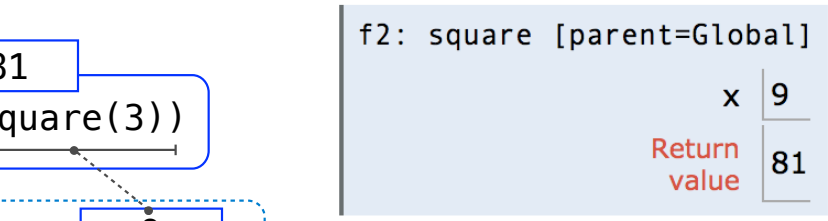
Interactive Diagram

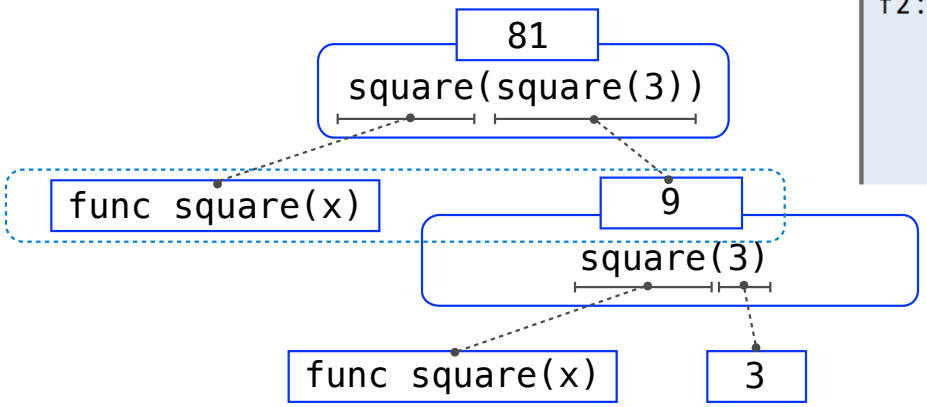
Multiple Environments in One Diagram!

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







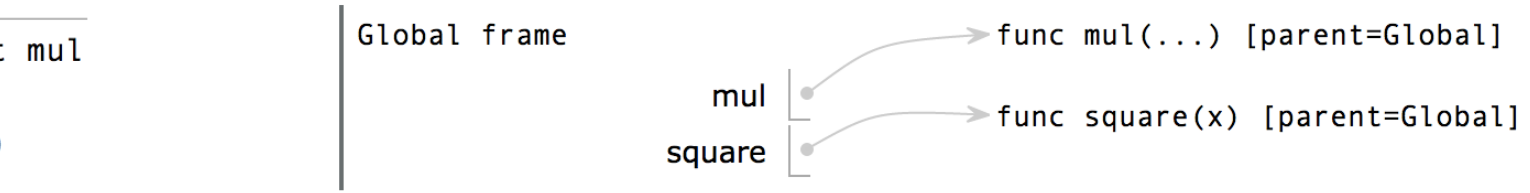


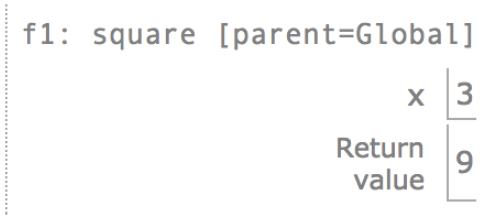
An environment is a sequence of frames.

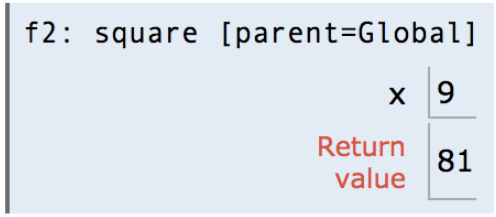
Interactive Diagram

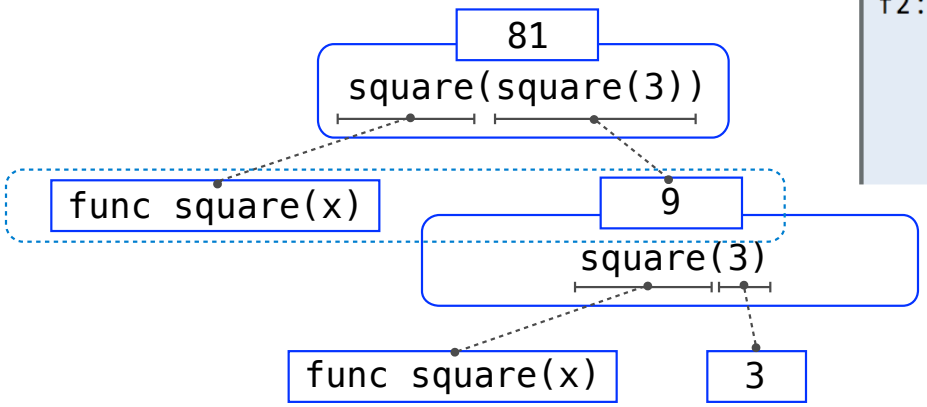
Multiple Environments in One Diagram!

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









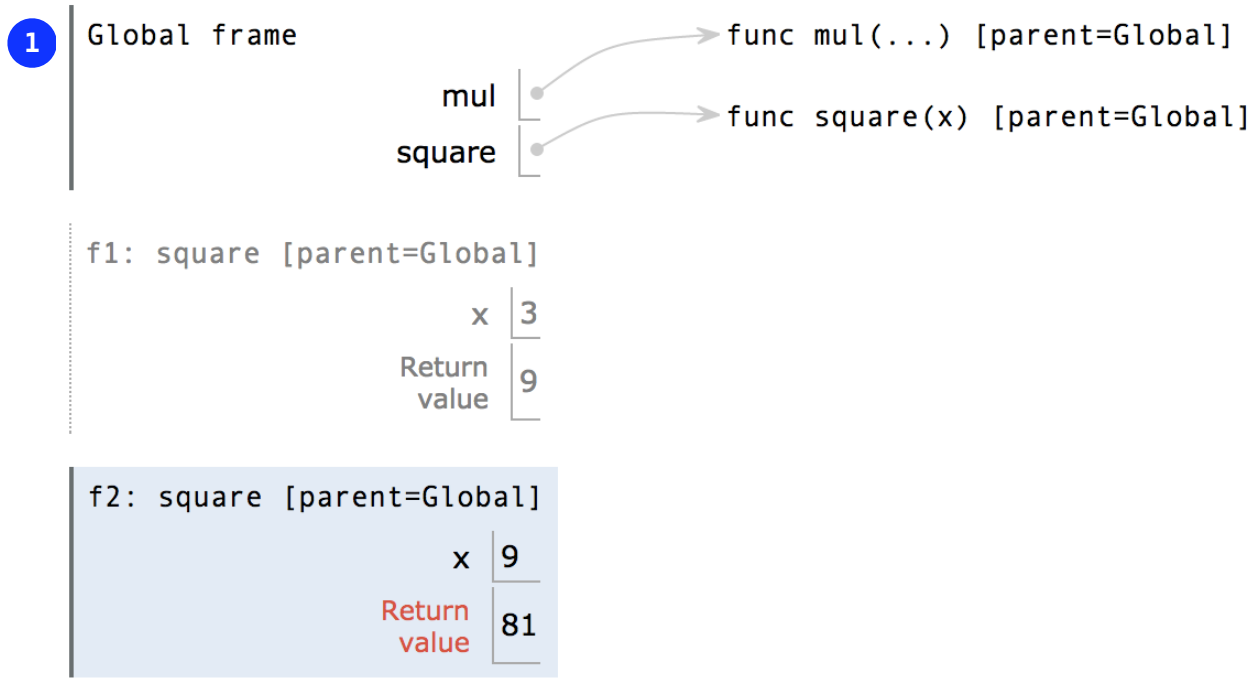
An environment is a sequence of frames.

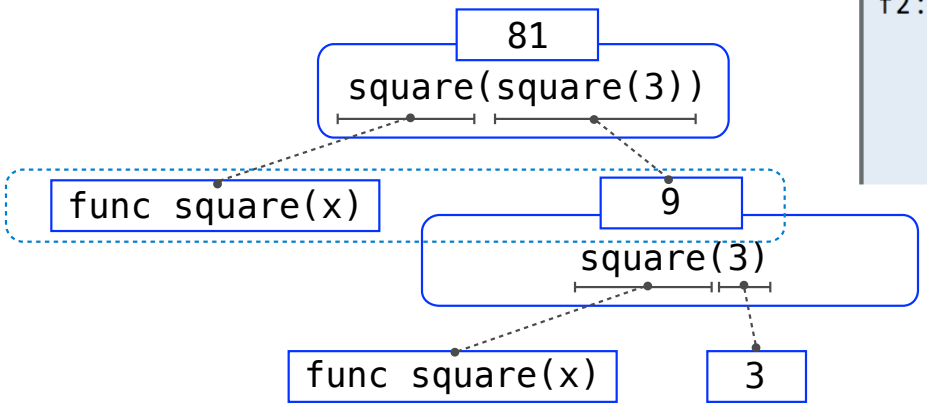
- The global frame alone
- A local, then the global frame

Interactive Diagram

Multiple Environments in One Diagram!

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





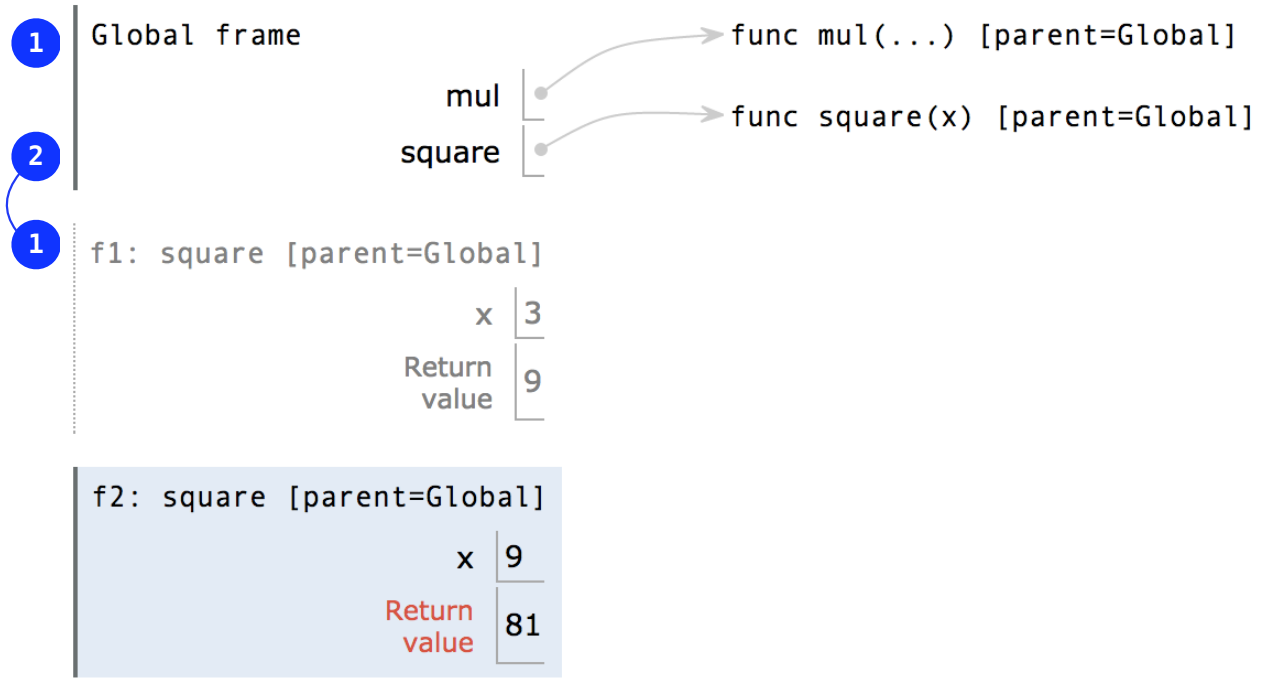
An environment is a sequence of frames.

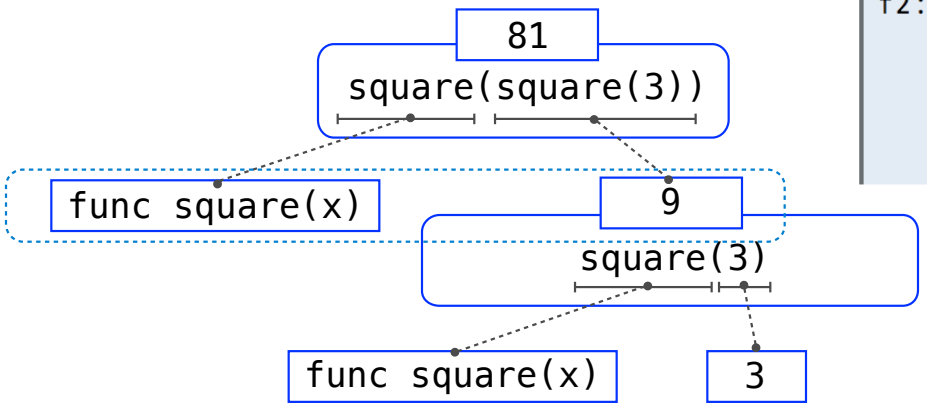
- The global frame alone
- A local, then the global frame

Interactive Diagram

Multiple Environments in One Diagram!

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





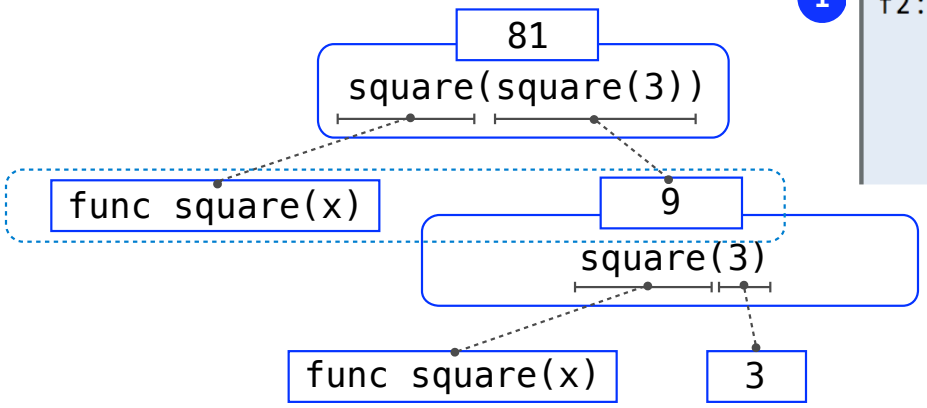
An environment is a sequence of frames.

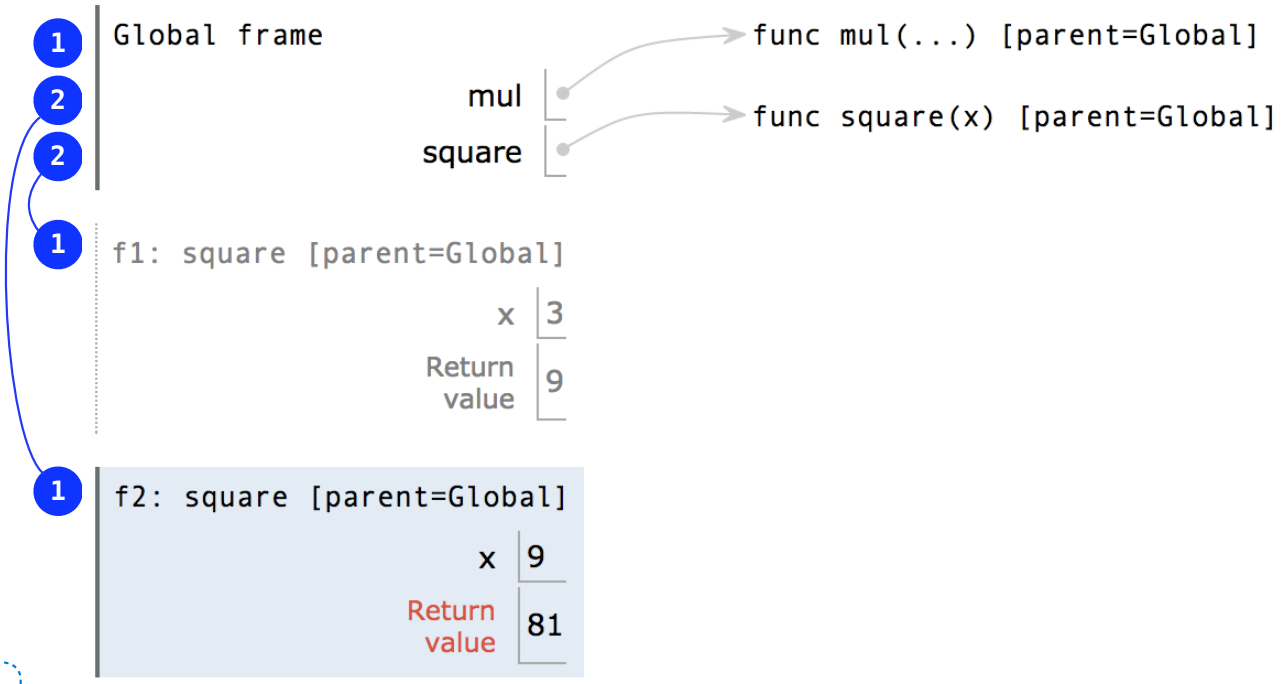
- The global frame alone
- A local, then the global frame

Interactive Diagram

Multiple Environments in One Diagram!

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





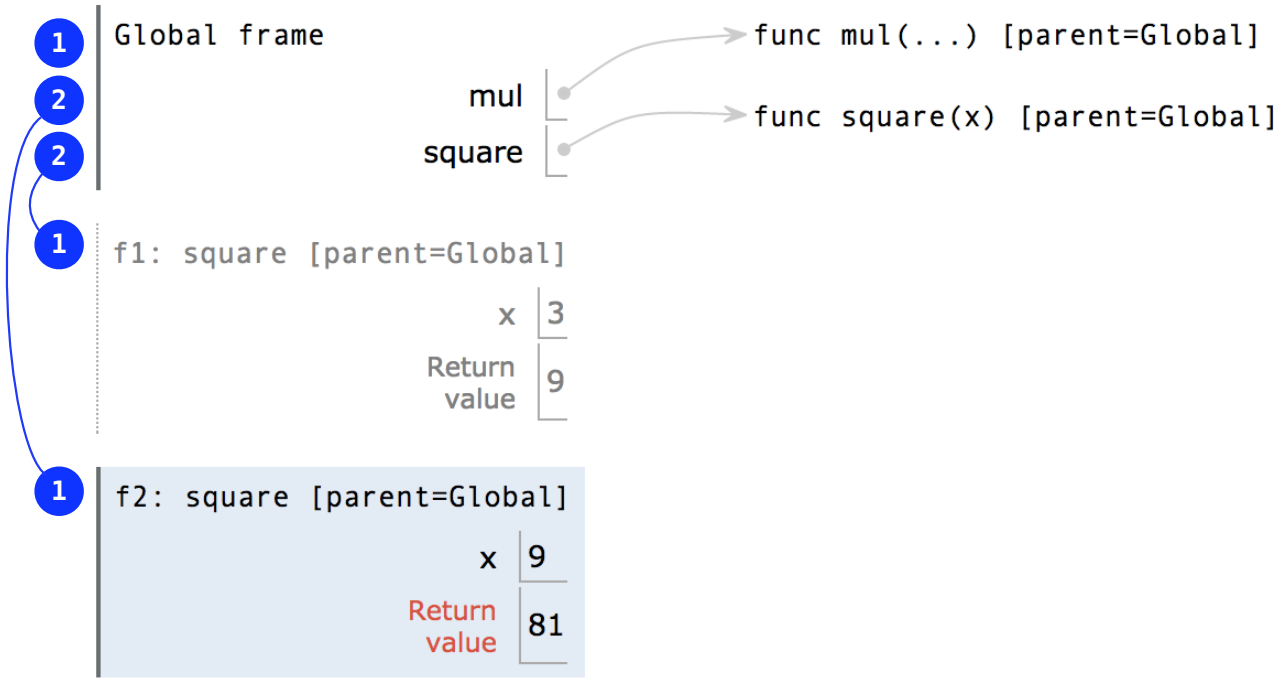
An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments

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



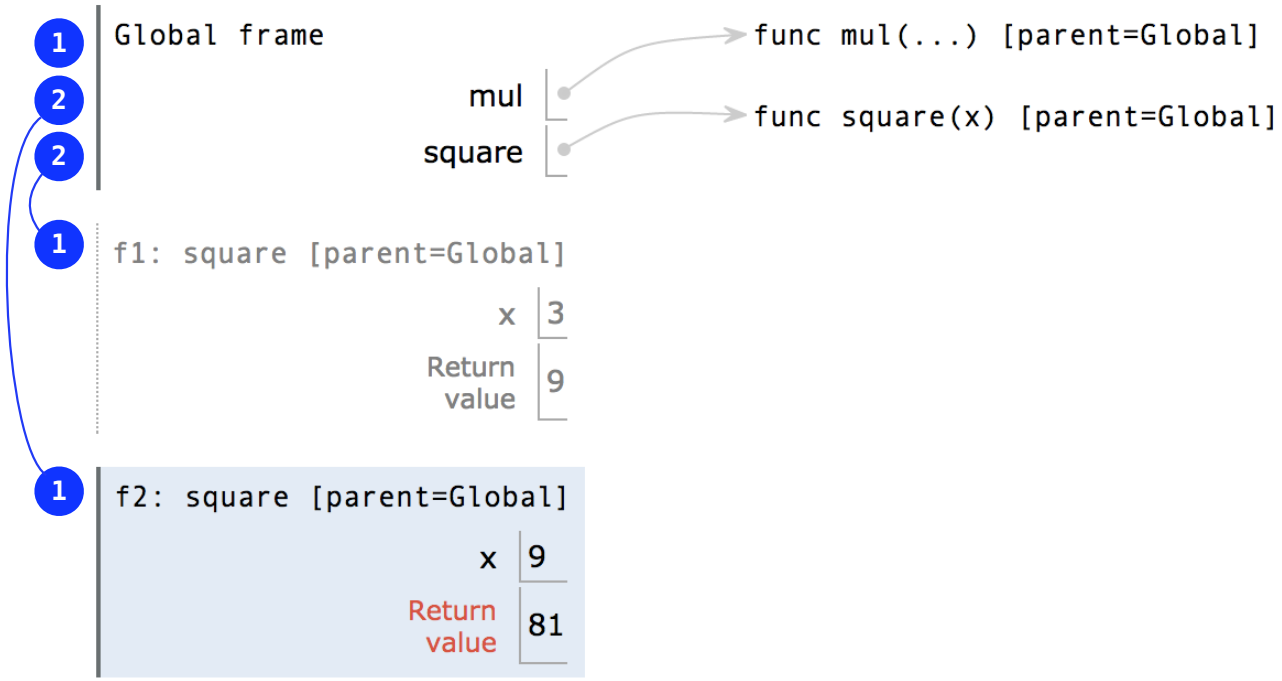
An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments

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



Every expression is evaluated in the context of an environment.

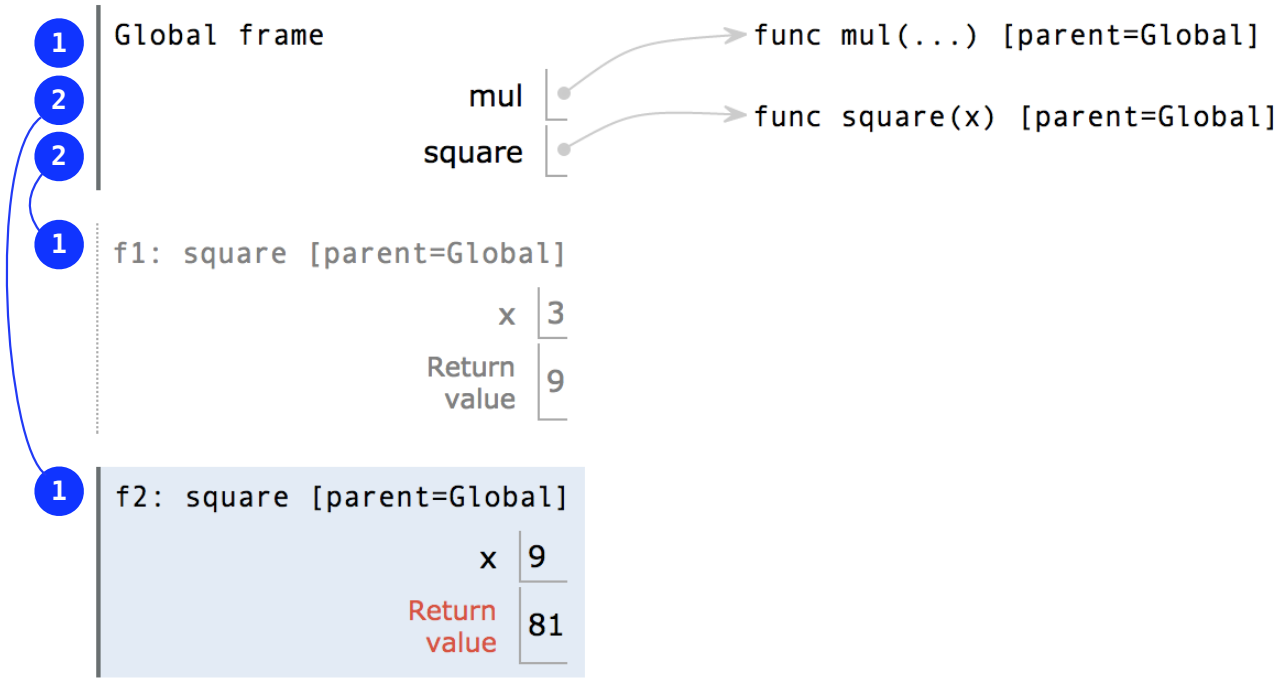
An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments

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



Every expression is evaluated in the context of an environment.

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

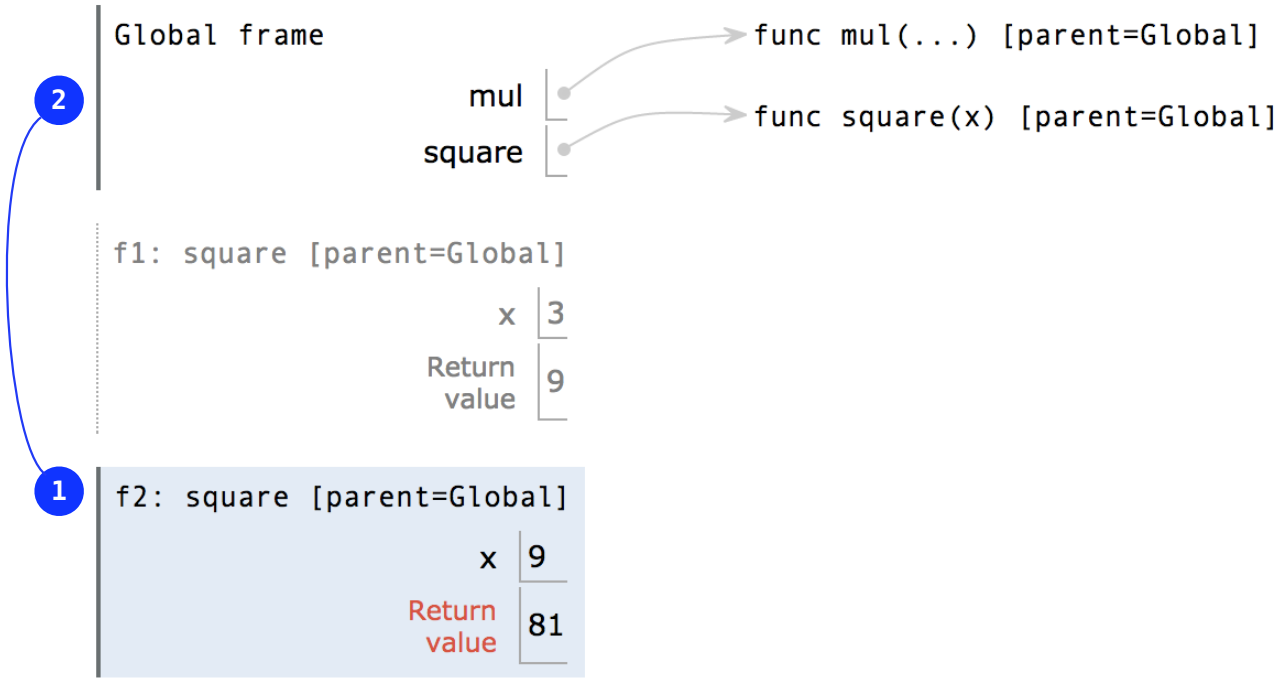
An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments

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



Every expression is evaluated in the context of an environment.

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

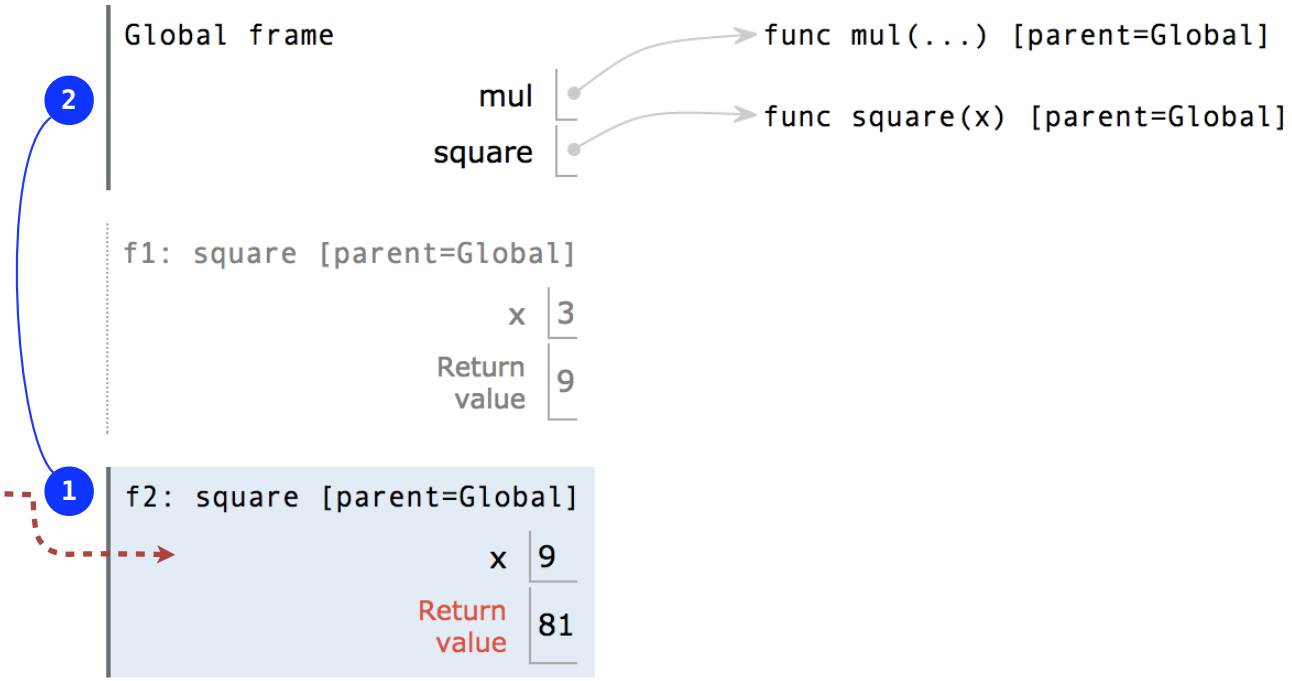
An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments

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



Every expression is evaluated in the context of an environment.

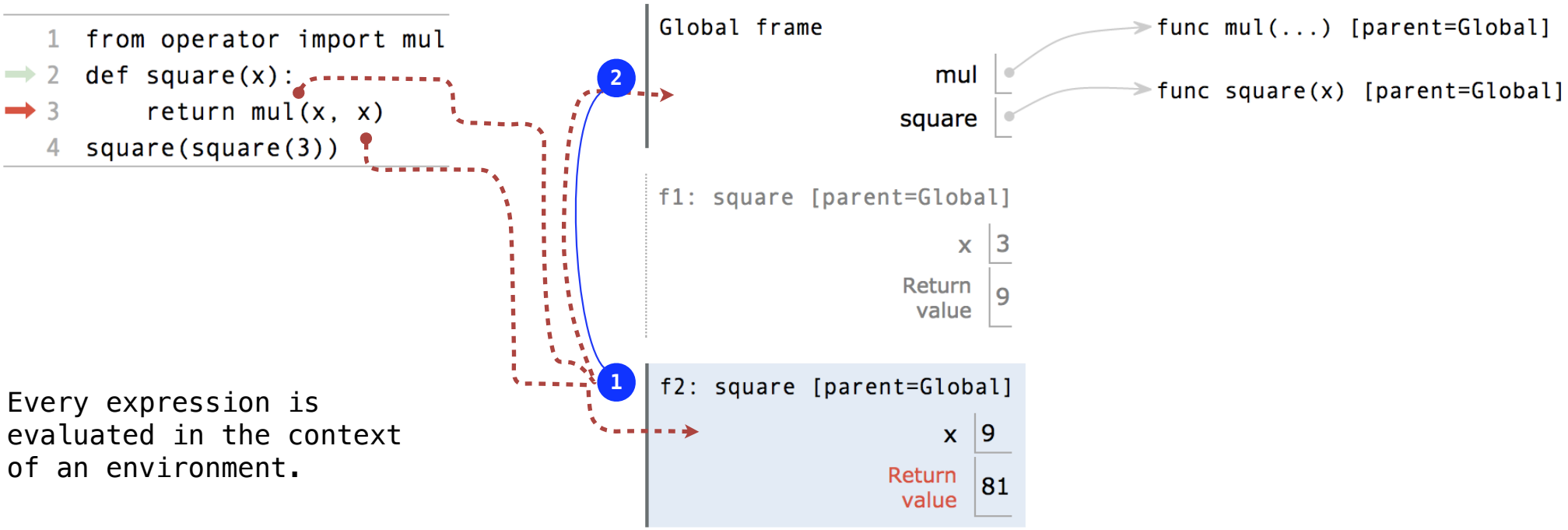
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have No Meaning Without Environments



Every expression is evaluated in the context of an environment.

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

An environment is a sequence of frames.

- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have Different Meanings in Different Environments

Every expression is evaluated in the context of an environment.

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

Interactive Diagram

Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments

Every expression is evaluated in the context of an environment.

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

Interactive Diagram

Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments

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

Every expression is evaluated in the context of an environment.

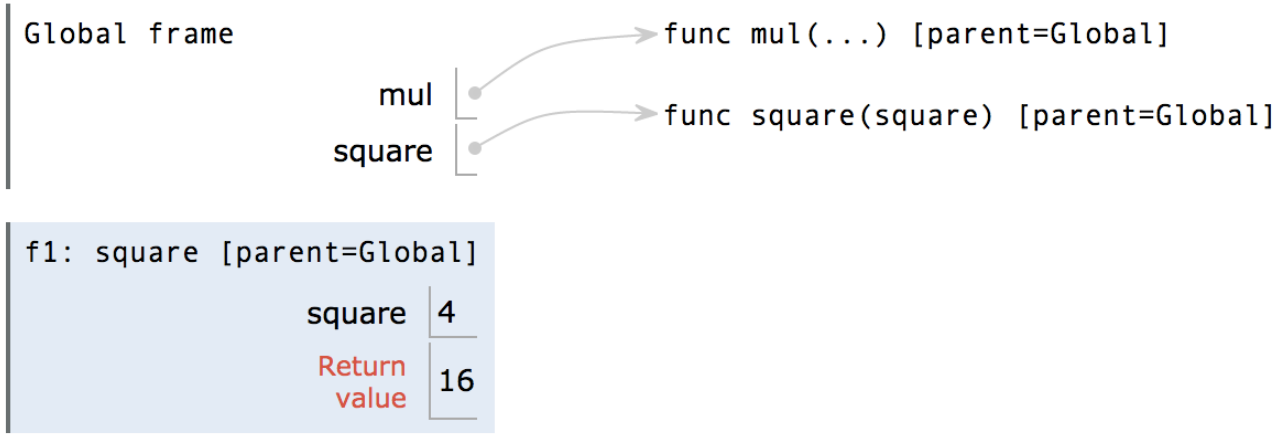
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Interactive Diagram

Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments

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



Every expression is evaluated in the context of an environment.

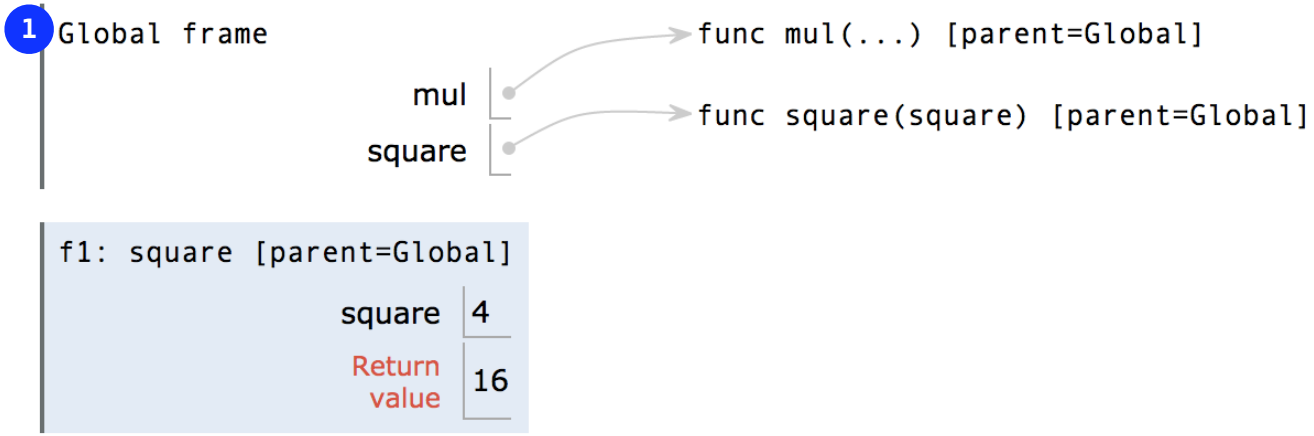
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Interactive Diagram

Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments

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



Every expression is evaluated in the context of an environment.

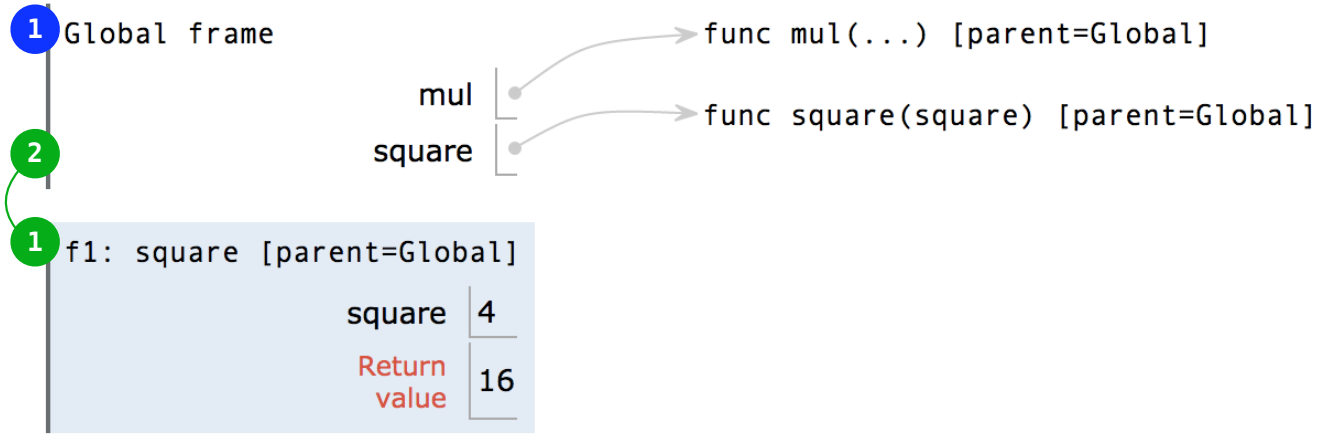
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

Interactive Diagram

Names Have Different Meanings in Different Environments

A call expression and the body of the function being called are evaluated in different environments

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



Every expression is evaluated in the context of an environment.

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

Interactive Diagram

Miscellaneous Python Features

Operators

Multiple Return Values

Docstrings

Doctests

Default Arguments

(Demo)

Conditional 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

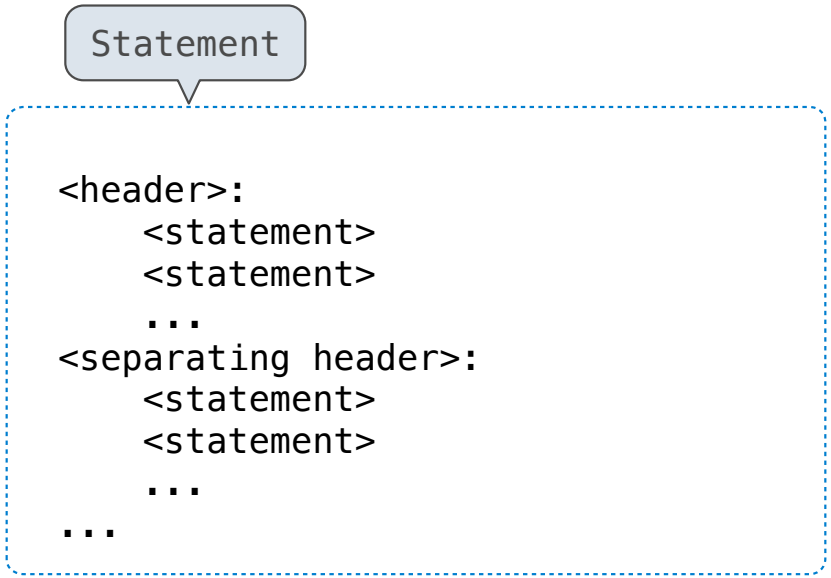
Compound statements:

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

Statements

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

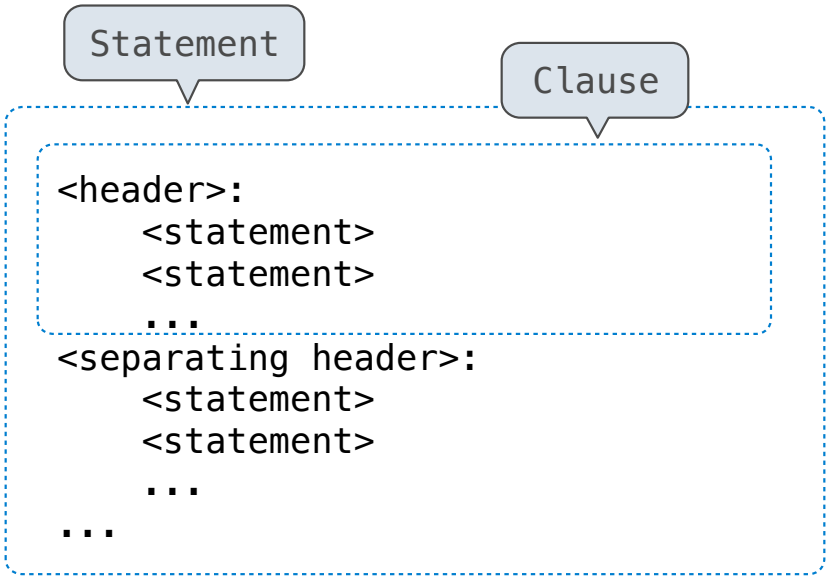
Compound statements:



Statements

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

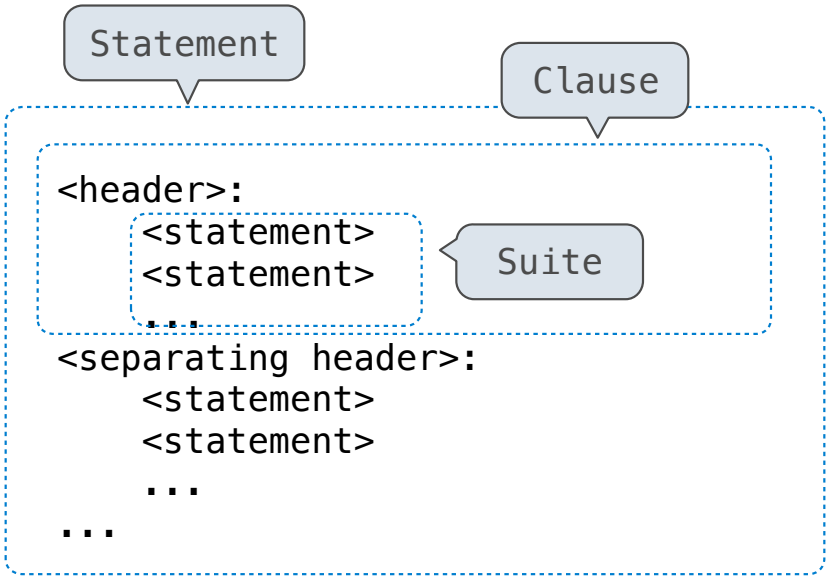
Compound statements:



Statements

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

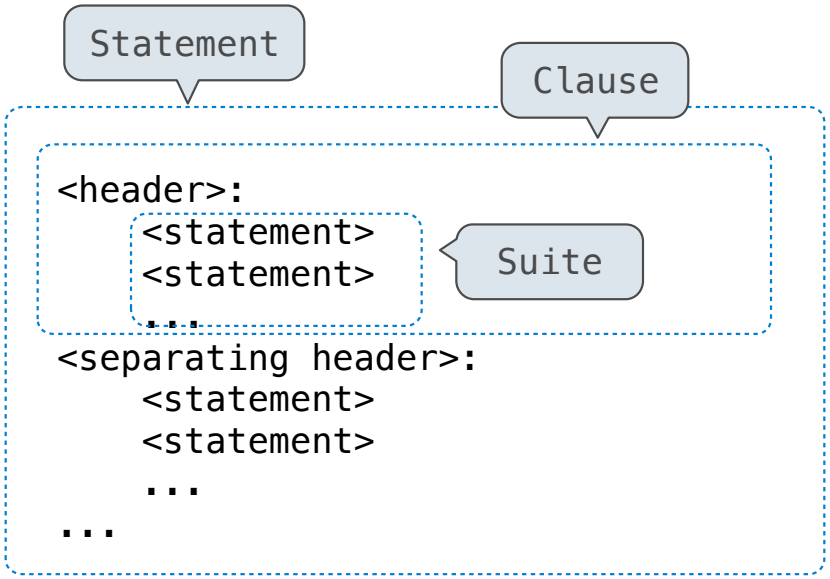
Compound statements:



Statements

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

Compound statements:

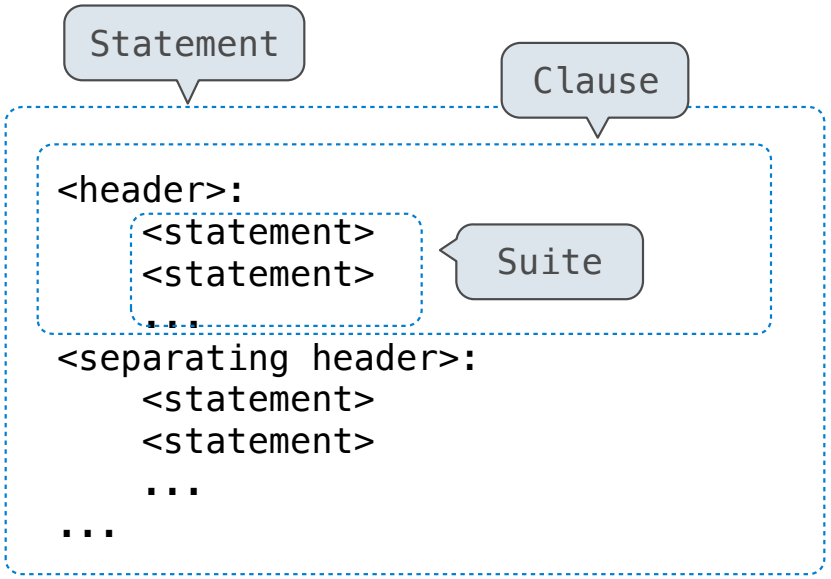


The first header determines a statement's type

Statements

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

Compound statements:



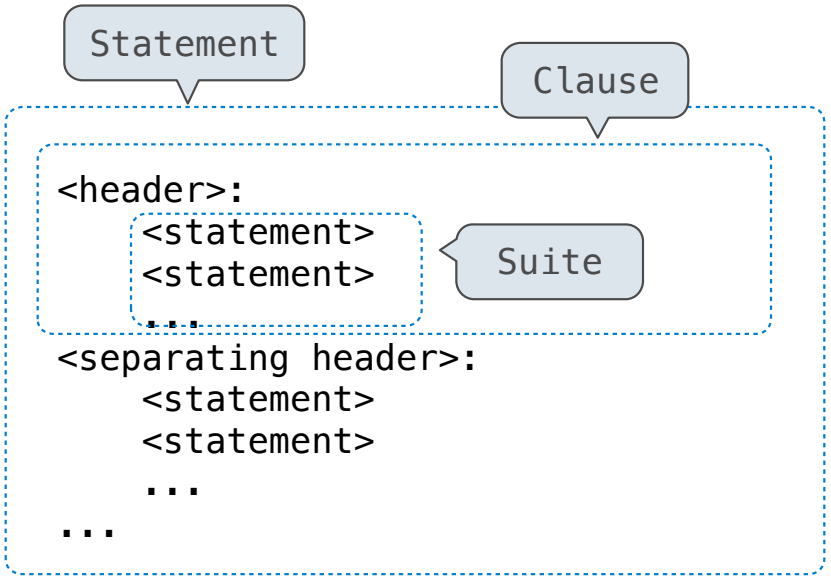
The first header determines a statement's type

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

Statements

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

Compound statements:



The first header determines a statement's type

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

def statements are compound statements

Compound Statements

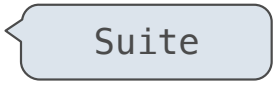
Compound statements:

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

Compound Statements

Compound statements:

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

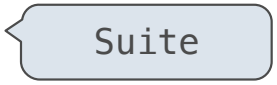


A suite is a sequence of statements

Compound Statements

Compound statements:

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

Suite

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:

- Execute the first statement
- Unless directed otherwise, execute the rest

Conditional Statements

(Demo)

Conditional Statements

(Demo)

```
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

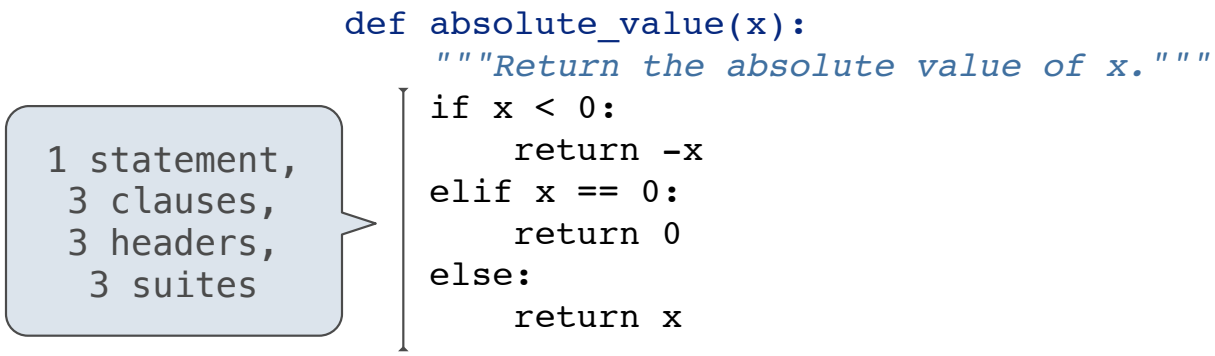
(Demo)

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

1 statement,
3 clauses,
3 headers,
3 suites

Conditional Statements

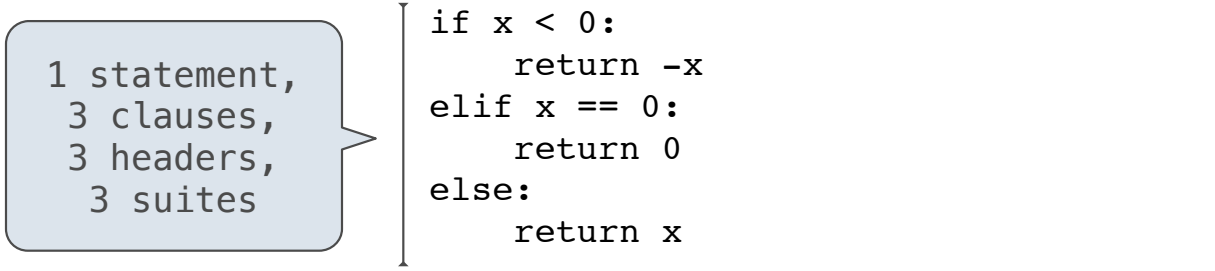
(Demo)



Execution Rule for Conditional Statements:

Conditional Statements

(Demo)




```
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:

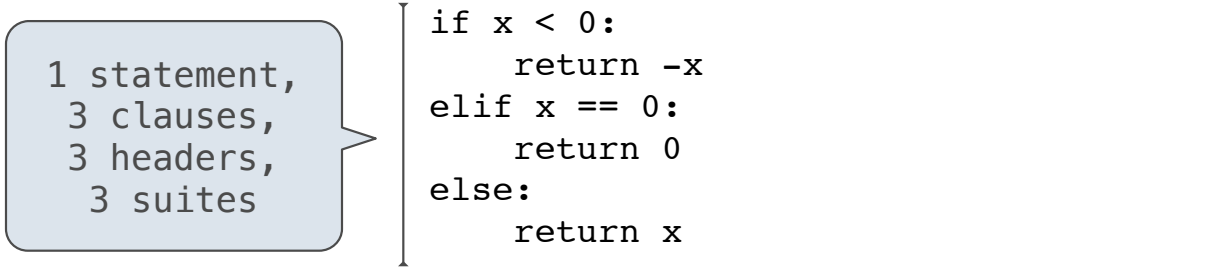
Each clause is considered in order.

1. Evaluate the header's expression.

2. If it is a true value,
   execute the suite & skip the remaining clauses.

Conditional Statements

(Demo)




```
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.

Syntax Tips:

Conditional Statements

(Demo)

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.

Syntax Tips:

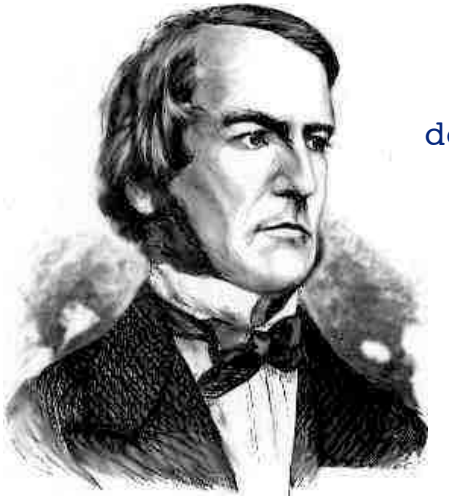
1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause,
   always at the end.

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
```

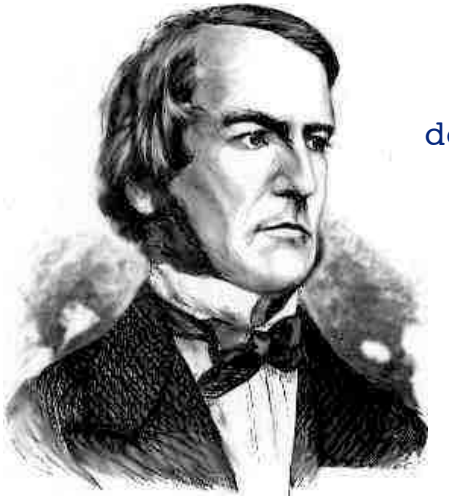
Boolean Contexts



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
```

Boolean Contexts

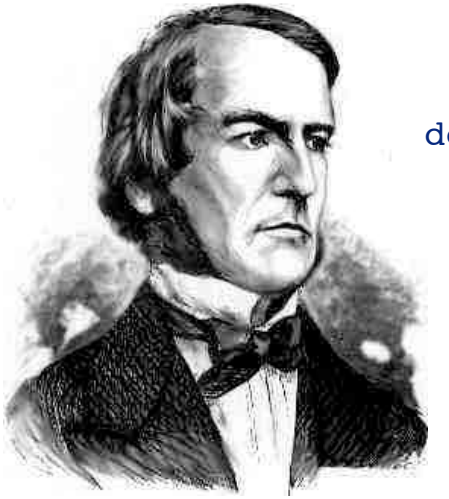


George Boole

```
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



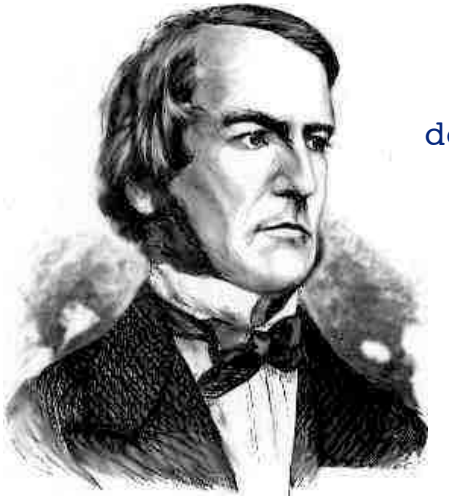
George Boole

```
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



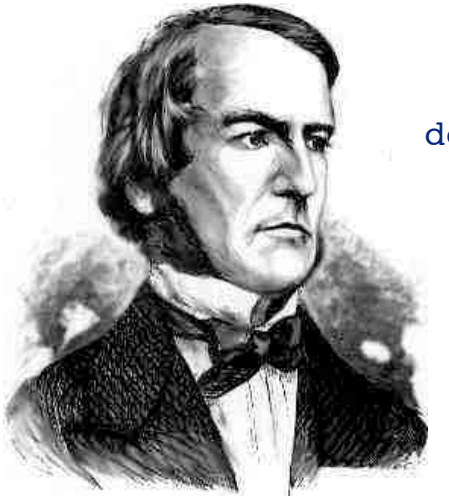
George Boole

```
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

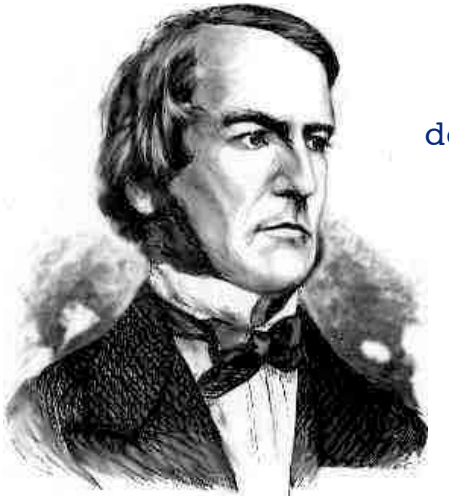
```
def absolute_value(x):
    """Return the absolute value of x."""
    if x < 0:
        return -x
    elif x == 0:
        return 0
    else:
        return x
```

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!

Reading: http://composingprograms.com/pages/15-control.html#conditional-statements

Iteration

While Statements

(Demo)

While Statements

(Demo)

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

While Statements

(Demo)

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

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.

While Statements

George Boole

(Demo)

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

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.

While Statements

George Boole

(Demo)

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

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.

While Statements

George Boole

(Demo)

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

Execution Rule for While Statements:

1. Evaluate the header's expression.

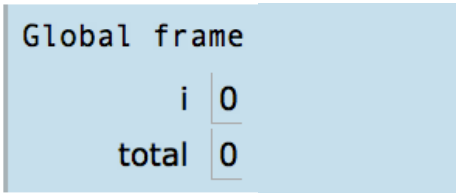
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

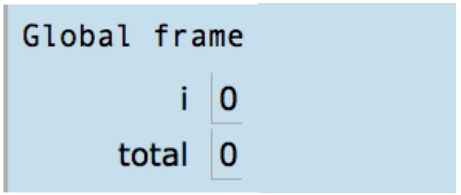
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

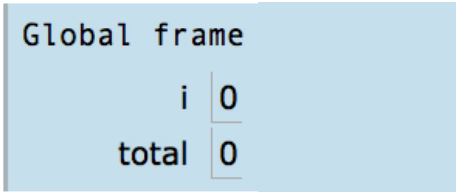
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

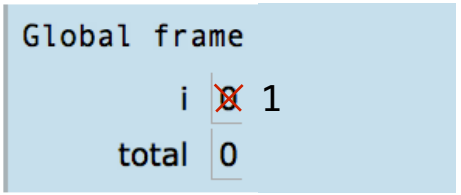
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

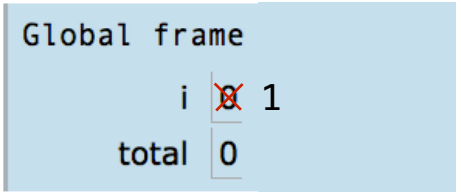
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

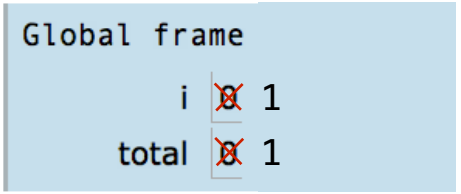
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

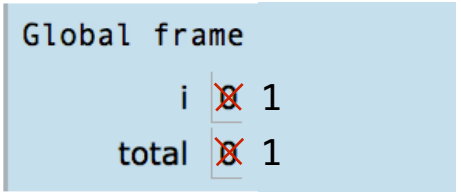
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

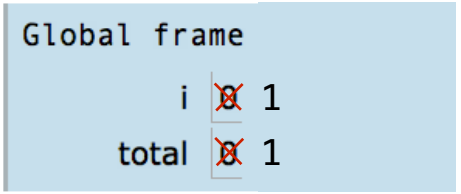
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

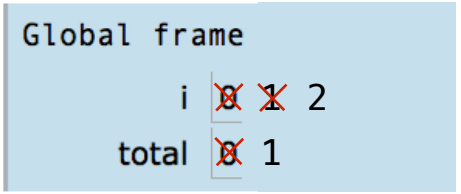
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

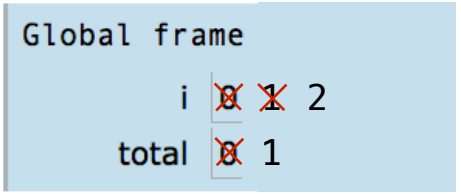
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

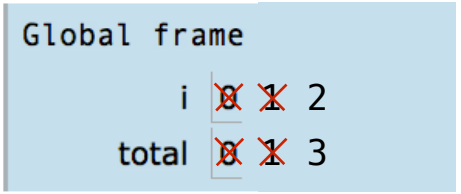
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

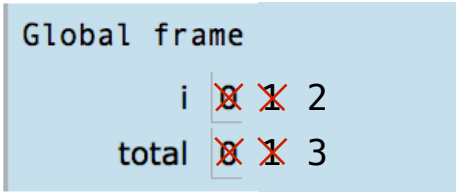
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

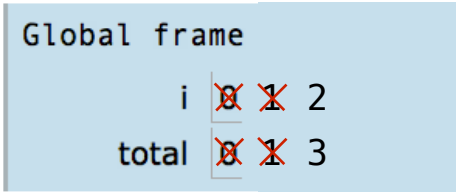
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

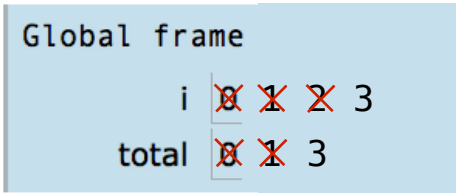
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

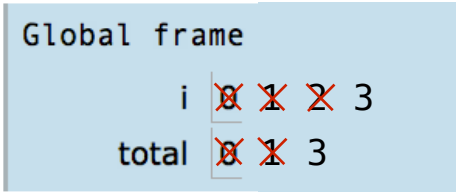
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

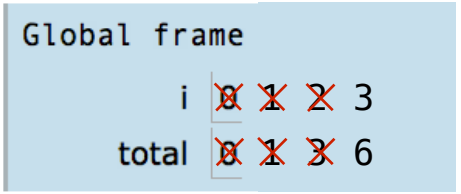
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



Execution Rule for While Statements:

1. Evaluate the header's expression.

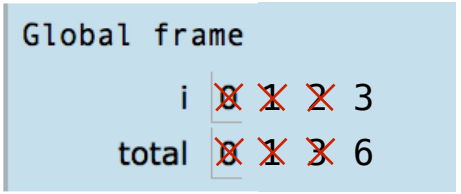
2. If it is a true value,
   execute the (whole) suite,
   then return to step 1.

While Statements

George Boole

(Demo)

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



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.