61A Lecture 3

Wednesday, August 29

Life Cycle of a User-Defined Function

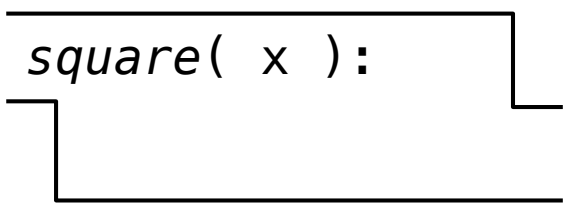
What happens?

Def statement:

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

Call expression: `square(2+2)`

Calling/Applying:



Life Cycle of a User-Defined Function

What happens?

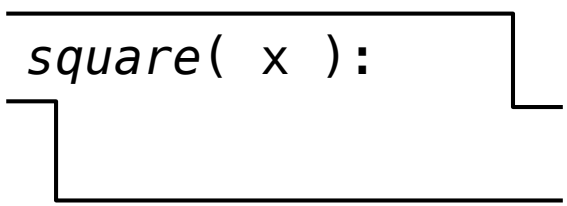
Def statement:

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

Def statement

Call expression: `square(2+2)`

Calling/Applying:



Life Cycle of a User-Defined Function

What happens?

Def statement:

Name

Def statement

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

Call expression: `square(2+2)`

Calling/Applying:

```
square( x ):
```

Life Cycle of a User-Defined Function

What happens?

Def statement:

Formal parameter

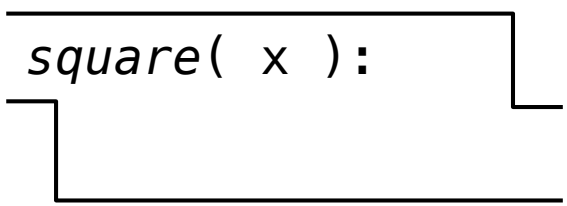
Name

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

Def statement

Call expression: `square(2+2)`

Calling/Applying:



Life Cycle of a User-Defined Function

What happens?

Def statement:

Formal parameter

Name

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

Def statement

Body

Call expression: `square(2+2)`

Calling/Applying:

```
square( x ):
```

Life Cycle of a User-Defined Function

What happens?

Def statement:

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

Formal parameter

Name

Def statement

Body (*return statement*)

Call expression: `square(2+2)`

Calling/Applying:

```
square( x ):
```

Life Cycle of a User-Defined Function

What happens?

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

Call expression: `square(2+2)`

Calling/Applying:

```
square( x ):
```

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

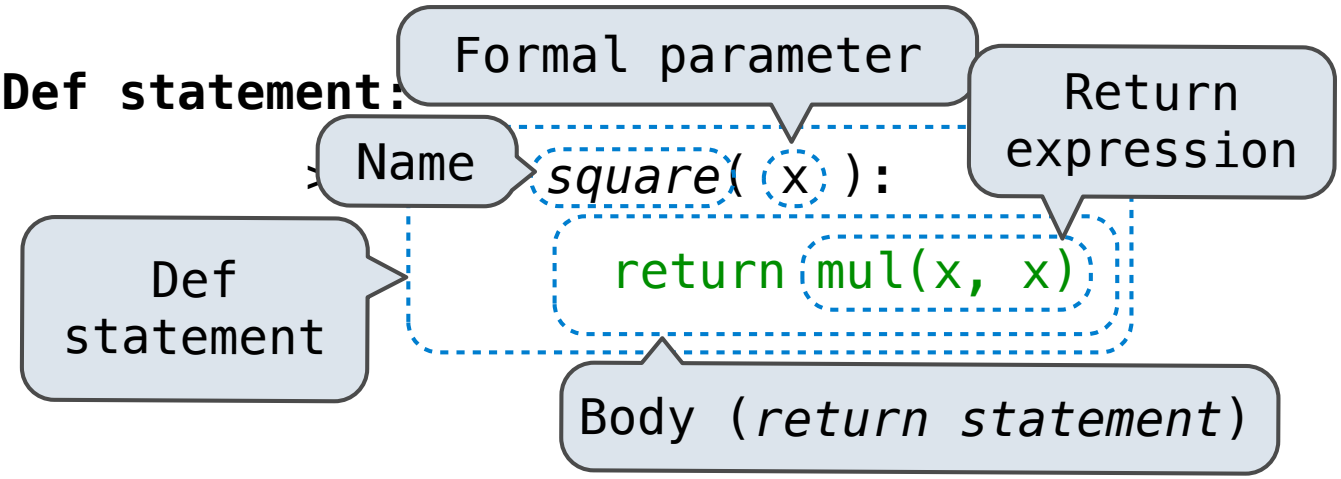
Call expression: square(2+2)

Calling/Applying:

```
square( x ):
```

Life Cycle of a User-Defined Function

Def statement:



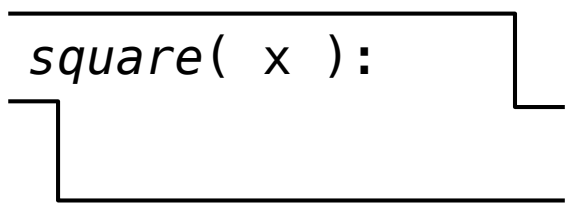
What happens?

Function created

Name bound

Call expression: square(2+2)

Calling/Applying:



Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

Calling/Applying:

square(x):

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

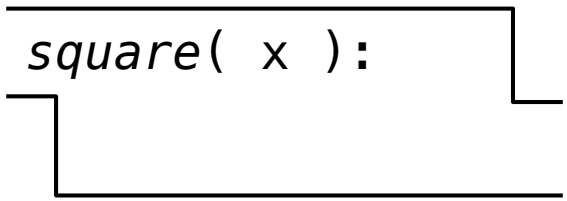
Name bound

Call expression: `square(2+2)`

operand: 2+2
argument: 4

operator: square
function: func square(x)

Calling/Applying:



Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: `square(2+2)`

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Calling/Applying:

```
square( x ):
```

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: `square(2+2)`

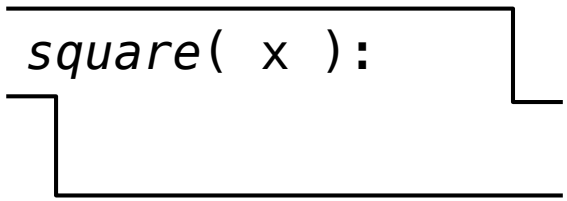
operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying:



Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying:

```
square( x ):
```

Signature

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: `square(2+2)`

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying:

4 ▶ `square( x ):`

Signature

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: `square(2+2)`

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying:

4 ▶ `square( x ):` ▶ 16

Signature

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying: 4 ▶ *square*(x): ▶ 16

Argument

Signature

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying: 4 ▶ *square*(x): ▶ 16

Argument

Signature

Return value

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying:

4 ▶ *square*(x):

▶ 16

Argument

Signature

Return value

New frame!

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying: 4 ▶ *square*(x): ▶ 16

Argument

Signature

Return value

New frame!

Params bound

Life Cycle of a User-Defined Function

Def statement:

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

Formal parameter

Name

Return expression

Def statement

Body (*return statement*)

What happens?

Function created

Name bound

Call expression: square(2+2)

operand: 2+2
argument: 4

operator: square
function: func square(x)

Op's evaluated

Function called with argument(s)

Calling/Applying:

4 ▶ *square*(x):

Argument

Signature

▶16

Return value

New frame!

Params bound

Body executed

Multiple Environments in One Diagram!

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

(Demo)

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

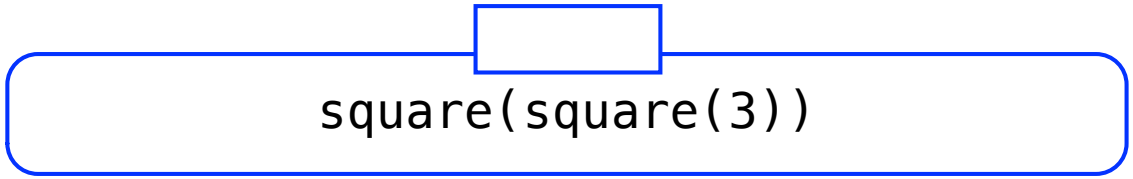
(Demo)

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

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

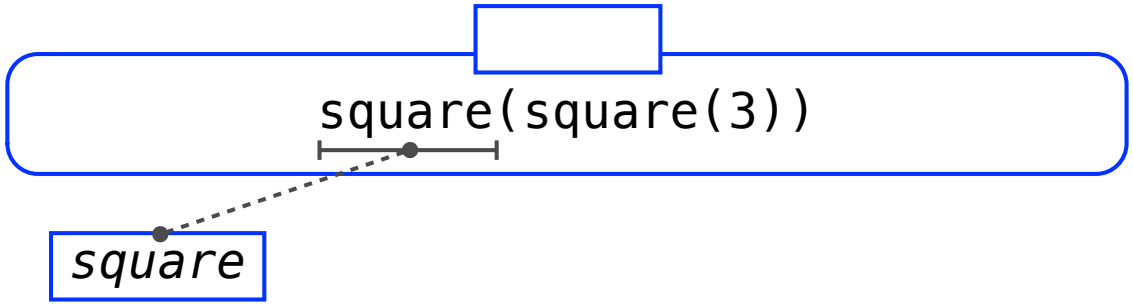
(Demo)



Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

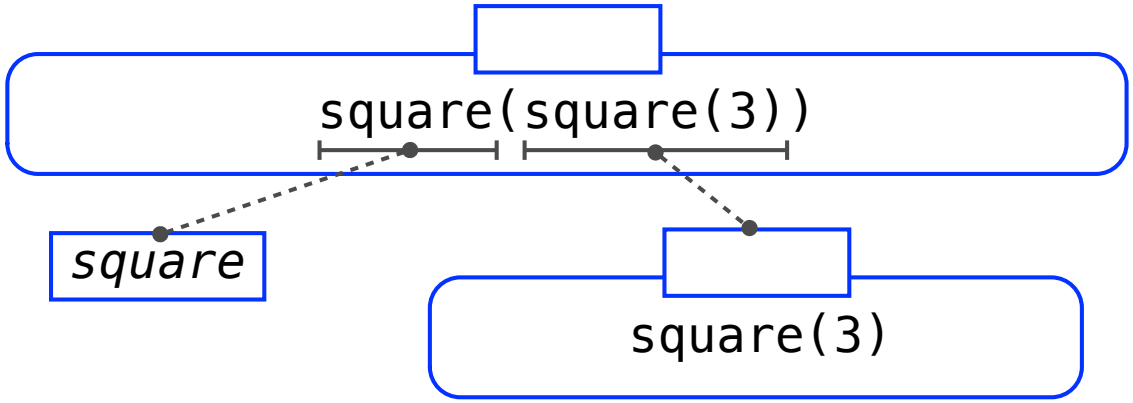
(Demo)



Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

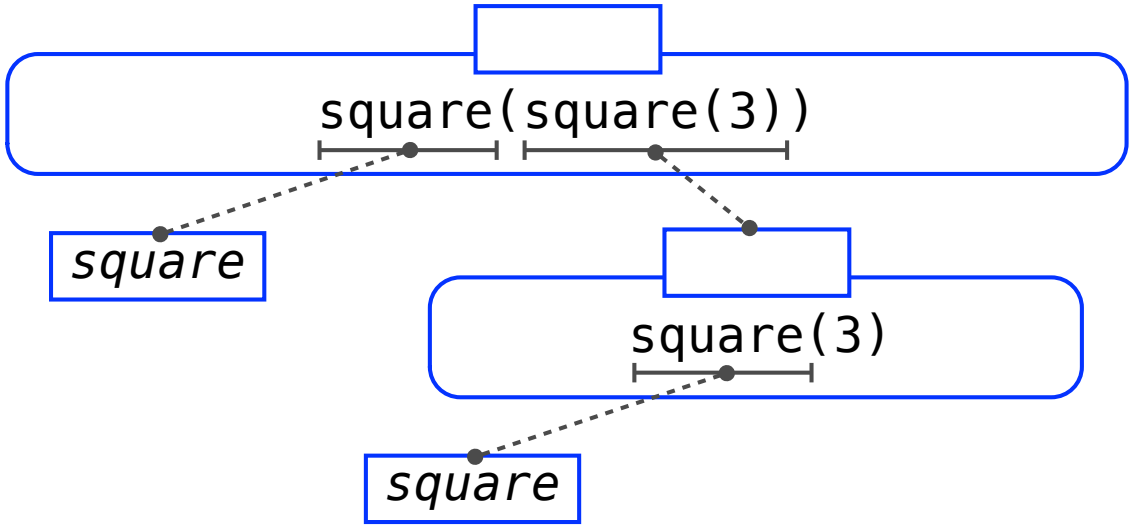
(Demo)



Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

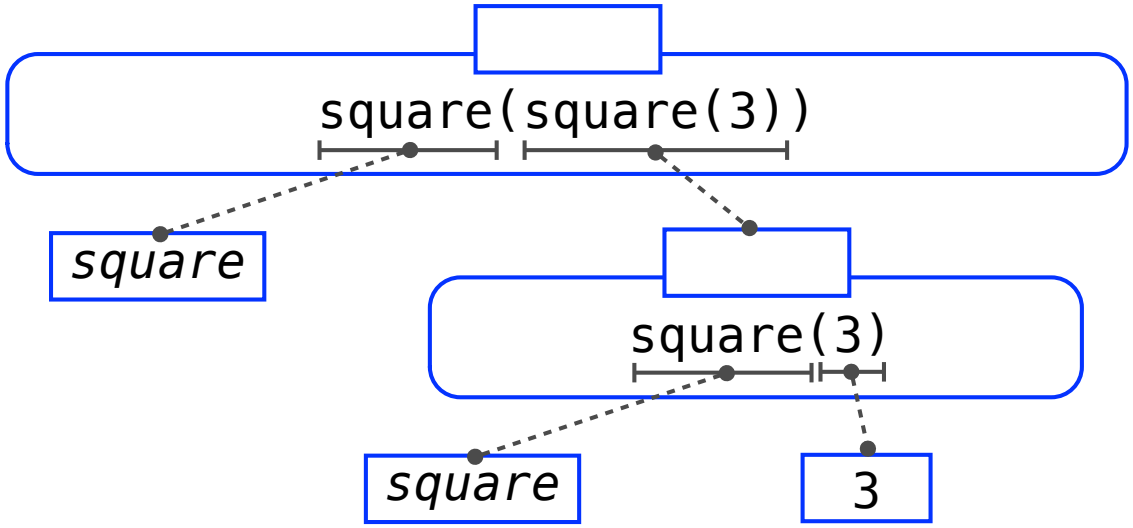
(Demo)



Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

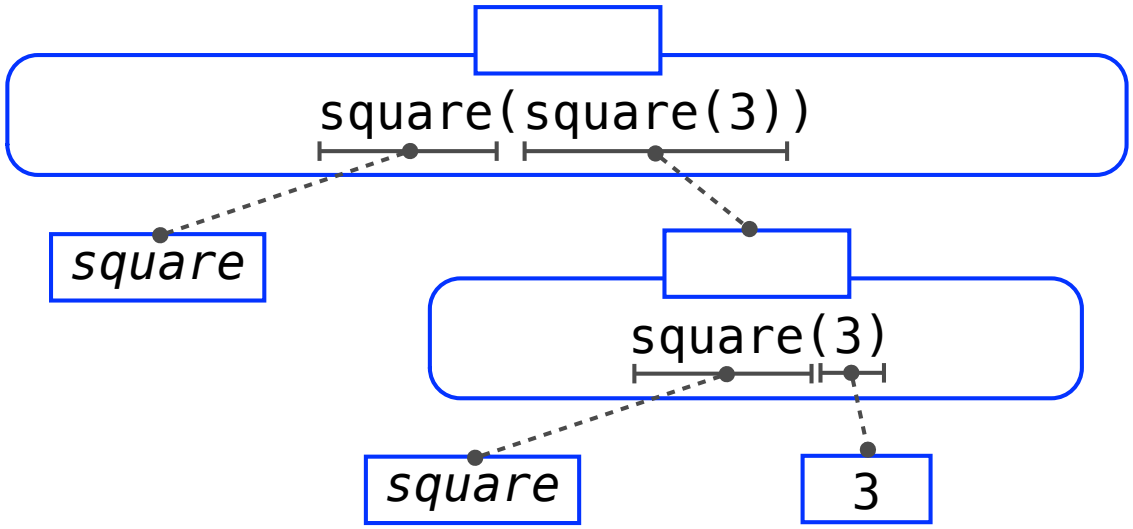
(Demo)



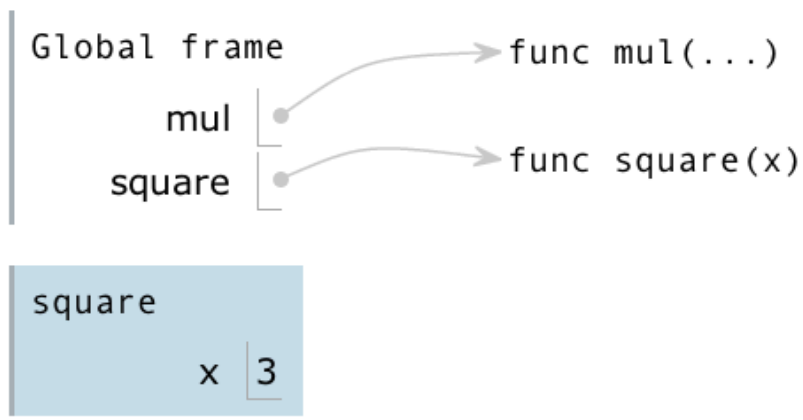
Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

(Demo)



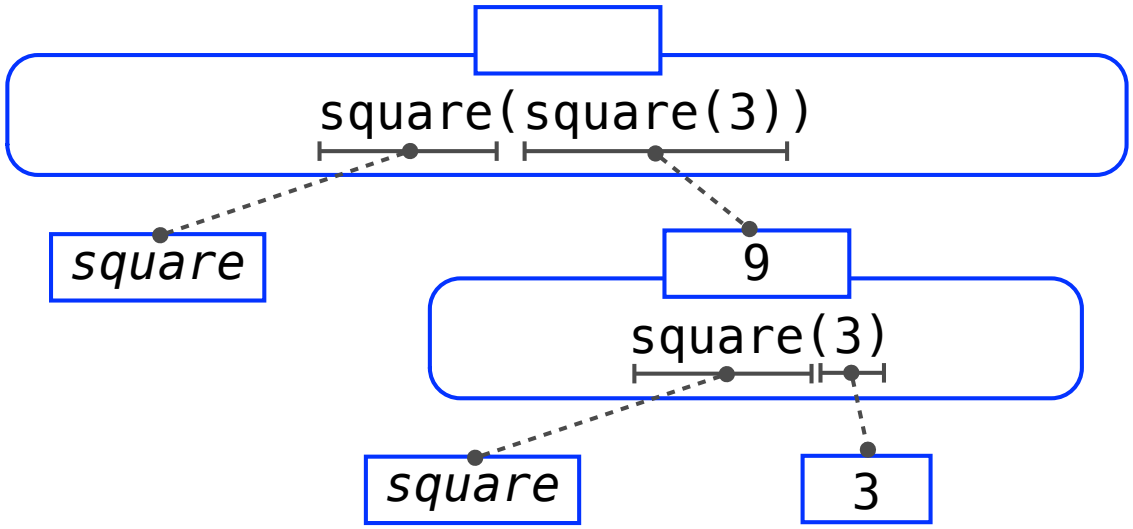

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```



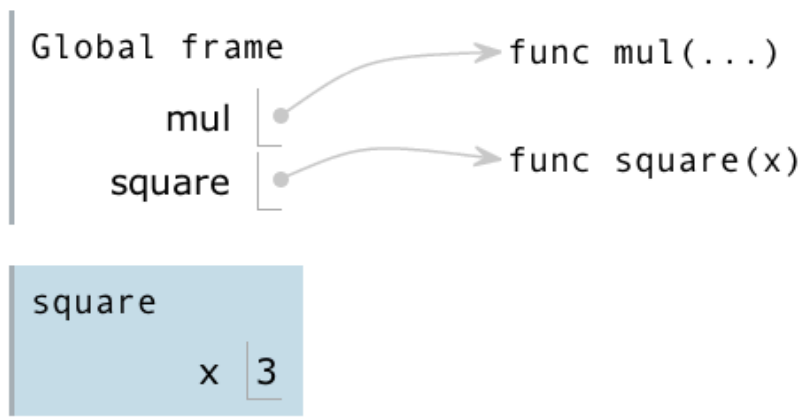
Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

(Demo)



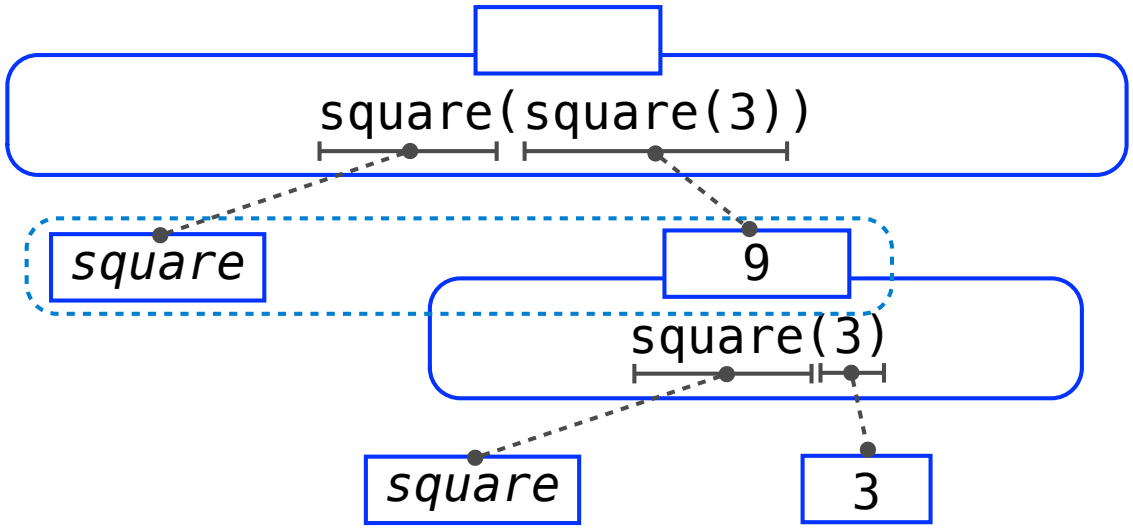

```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```



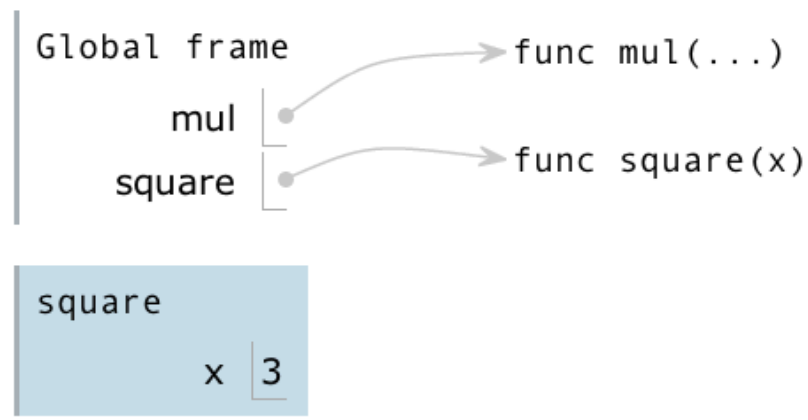
Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

(Demo)

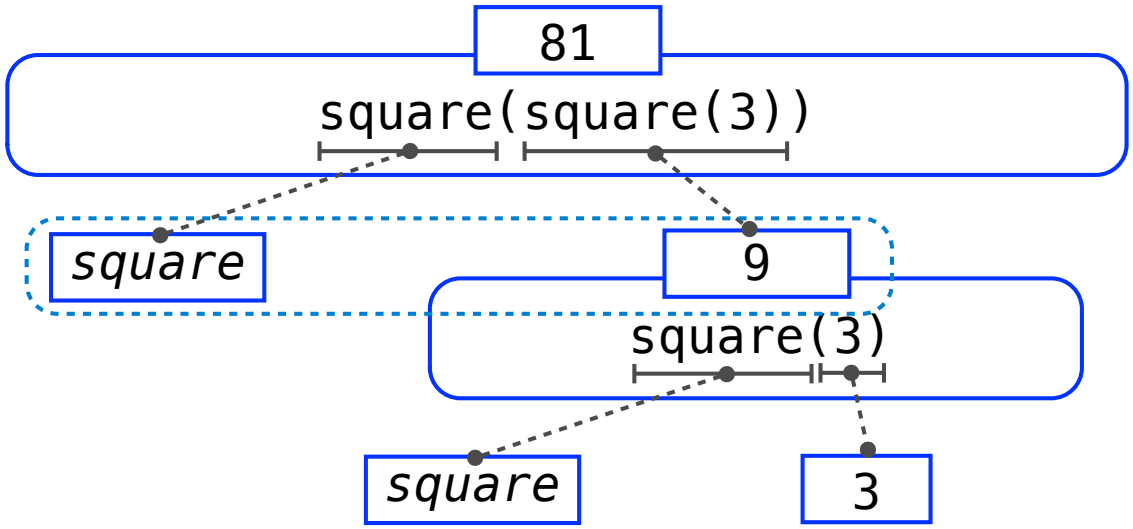



```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

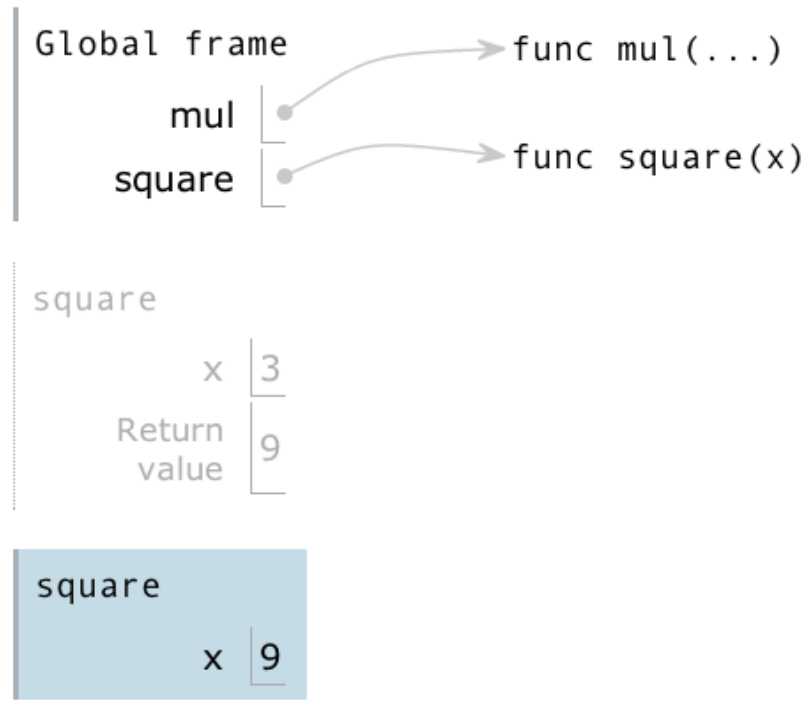


Example: http://goo.gl/668fU

Multiple Environments in One Diagram!

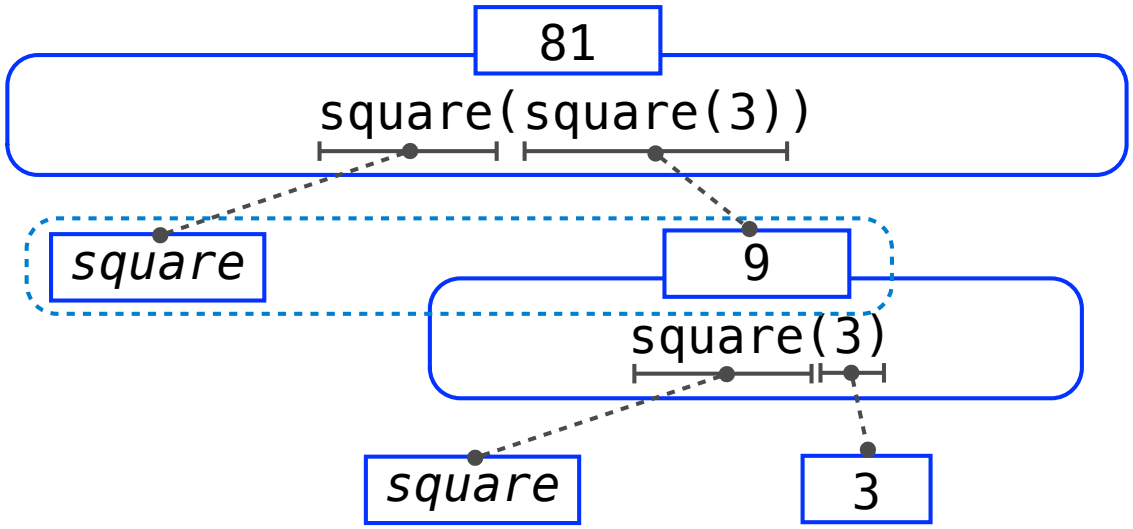



```
1  from operator import mul
2  def square(x):
3      return mul(x, x)
4  square(square(3))
```

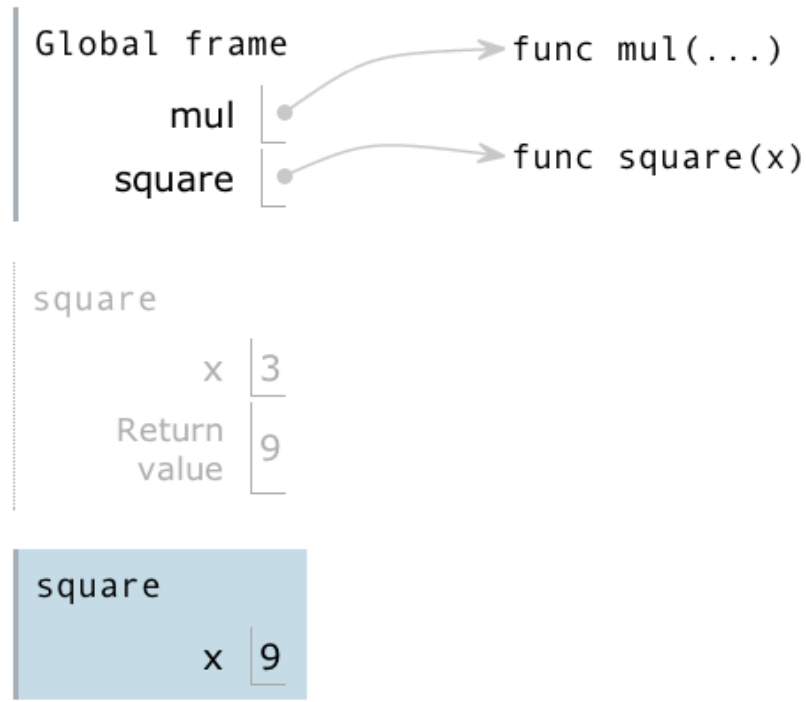


Example: http://goo.gl/668fU

Multiple Environments in One Diagram!



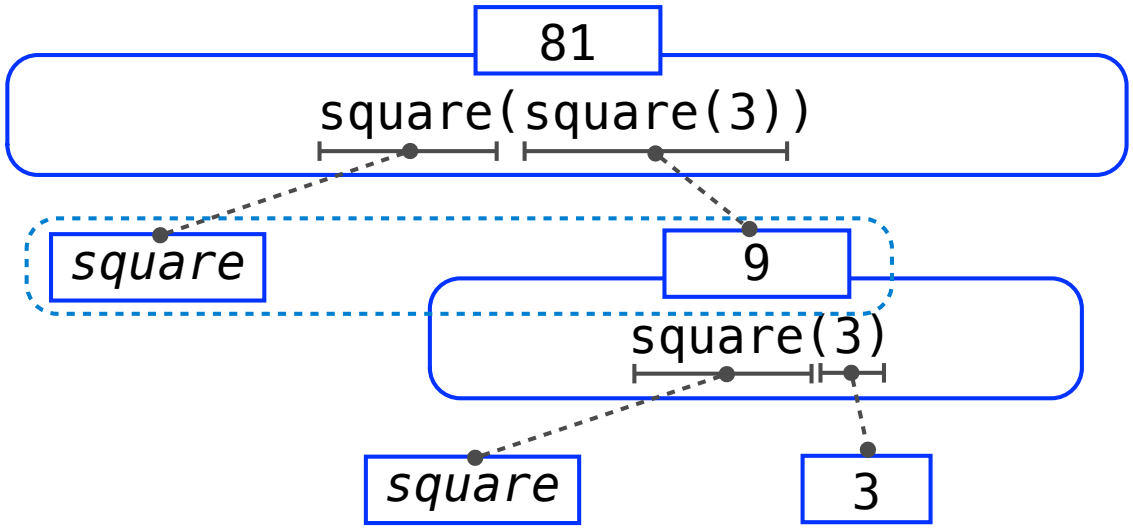

```
1  from operator import mul
2  def square(x):
3      return mul(x, x)
4  square(square(3))
```



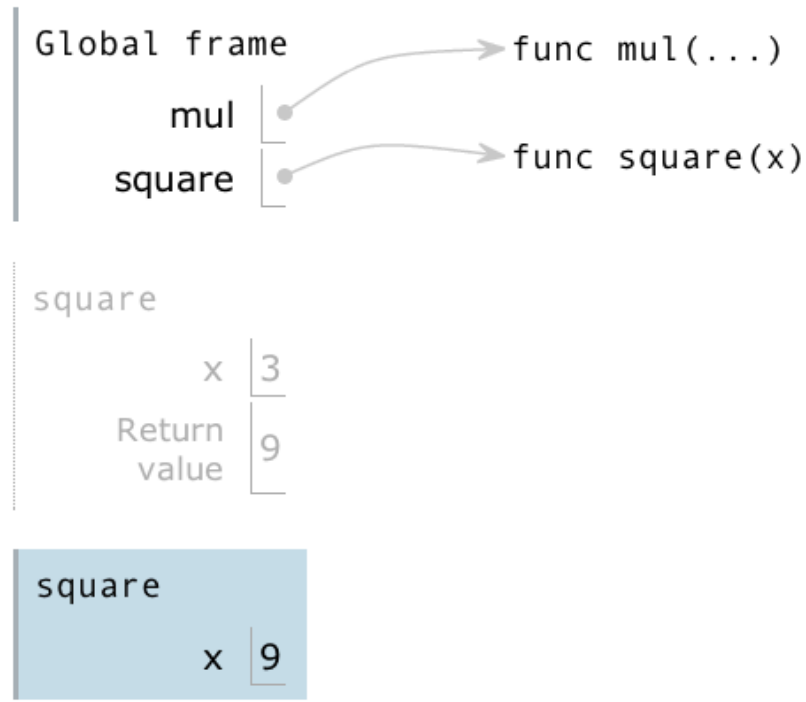
An **environment** is a **sequence** of frames.

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!




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

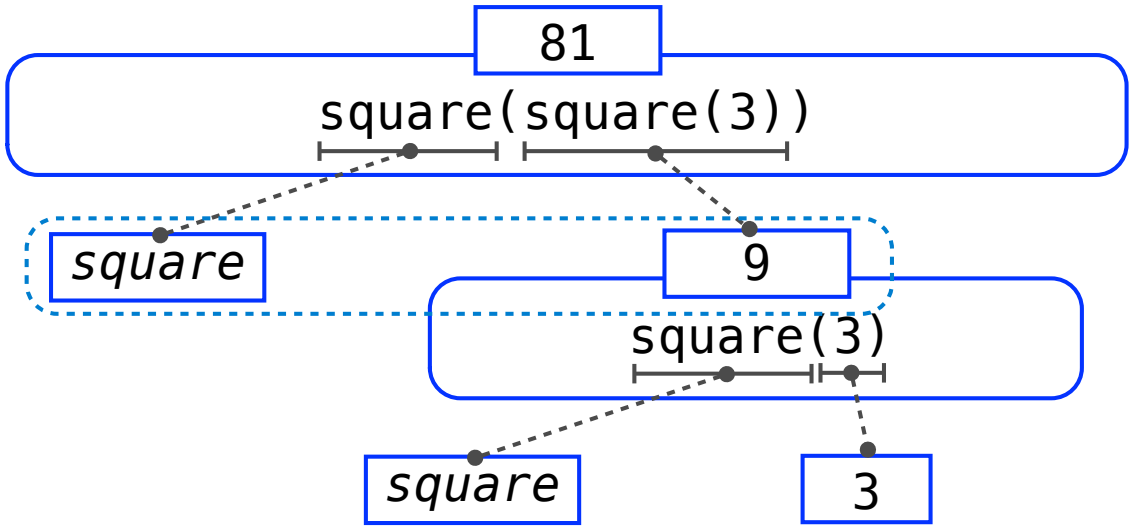


An **environment** is a **sequence** of frames.

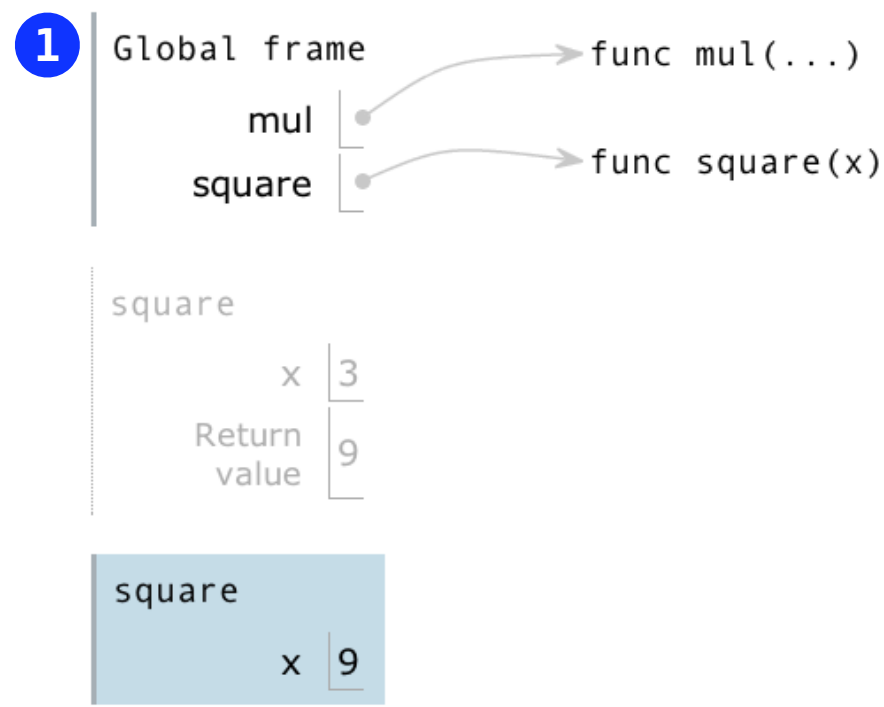
- The global frame alone
- A local, then the global frame

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!




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

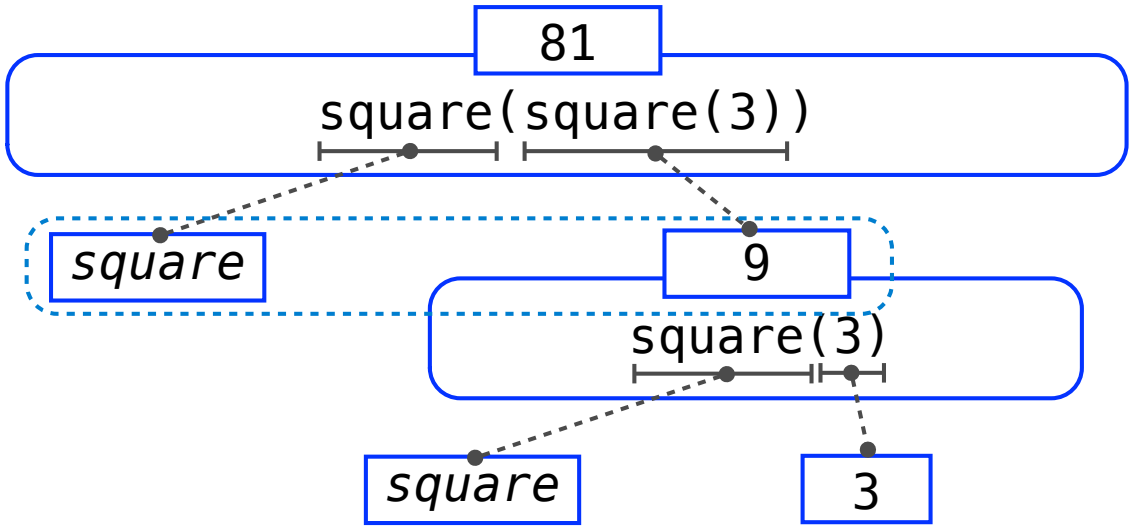


An **environment** is a **sequence** of frames.

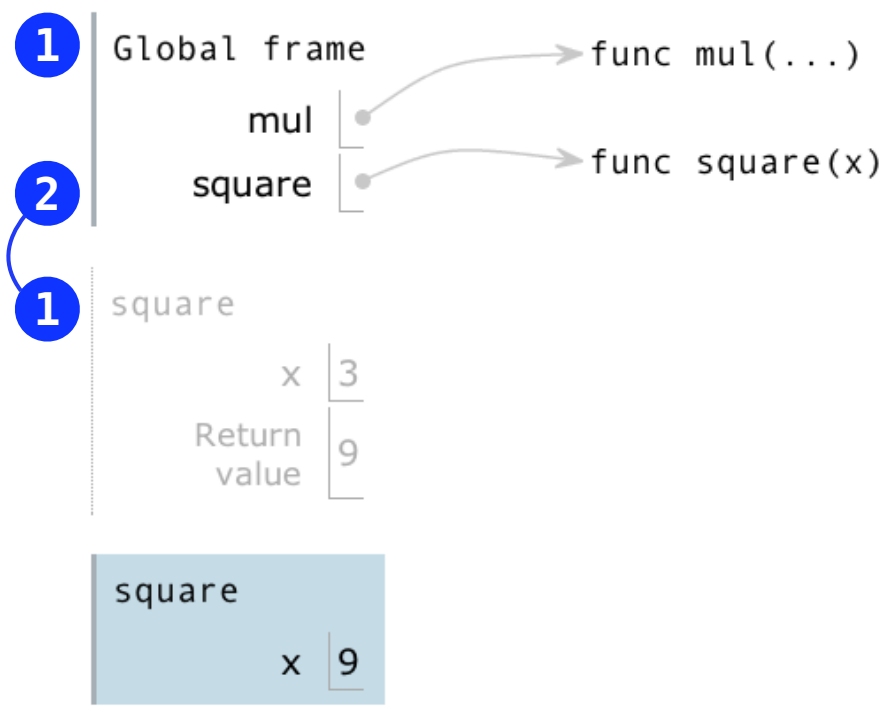
- The global frame alone
- A local, then the global frame

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!




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

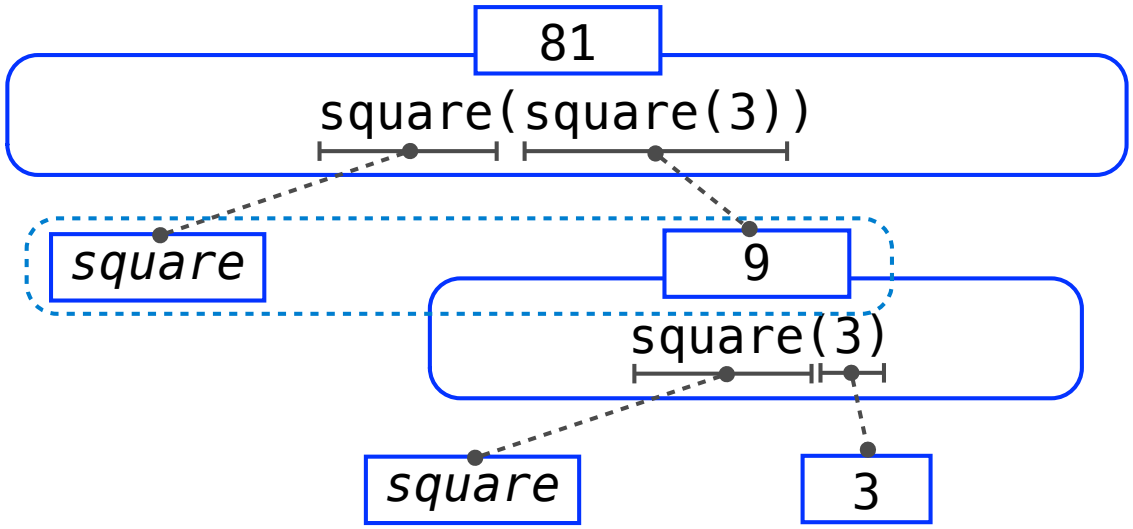


An **environment** is a **sequence** of frames.

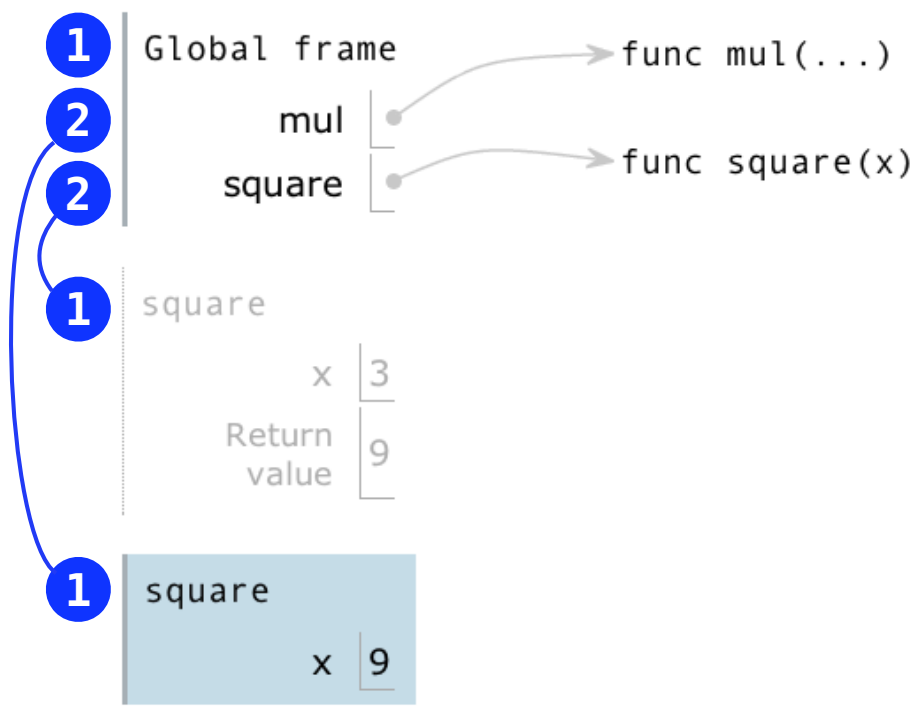
- The global frame alone
- A local, then the global frame

Example: http://goo.gl/668fU

Multiple Environments in One Diagram!




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



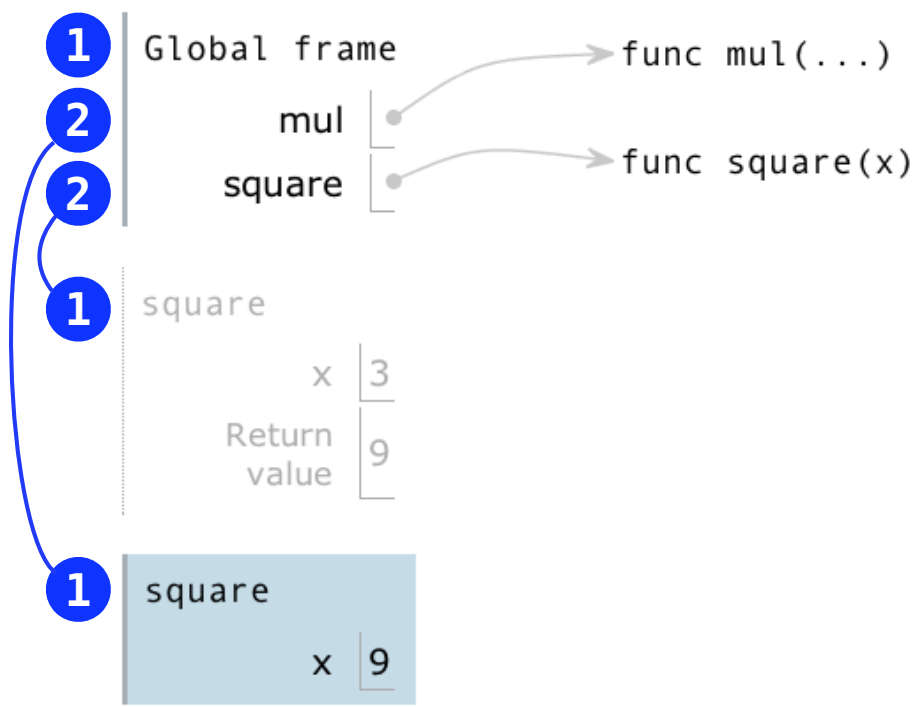
An **environment** is a **sequence** of frames.

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

Example: http://goo.gl/668fU

Names Have No Meaning Without Environments

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



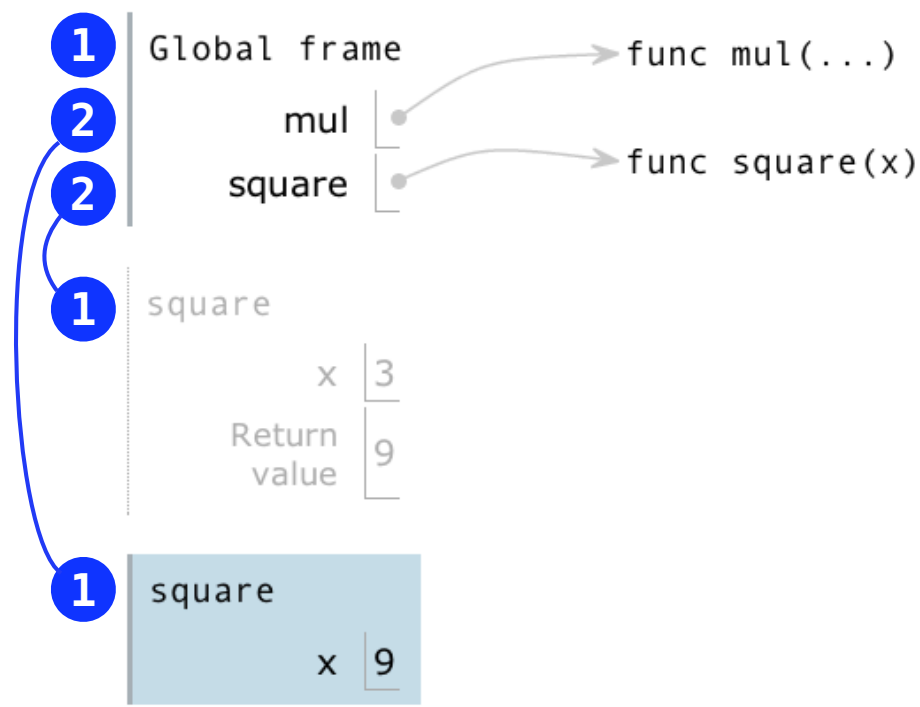
ıt:

Example: http://goo.gl/668fU

Names Have No Meaning Without Environments

Every expression is evaluated in the context of an environment.

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



ıt:

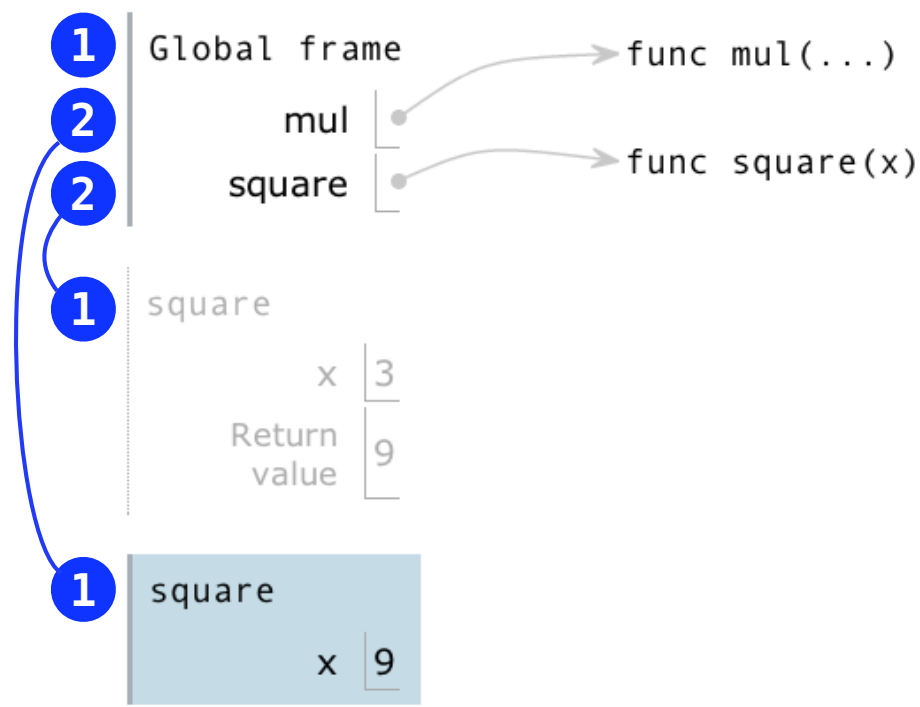
Example: http://goo.gl/668fU

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.

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



ıt:

Example: http://goo.gl/668fU

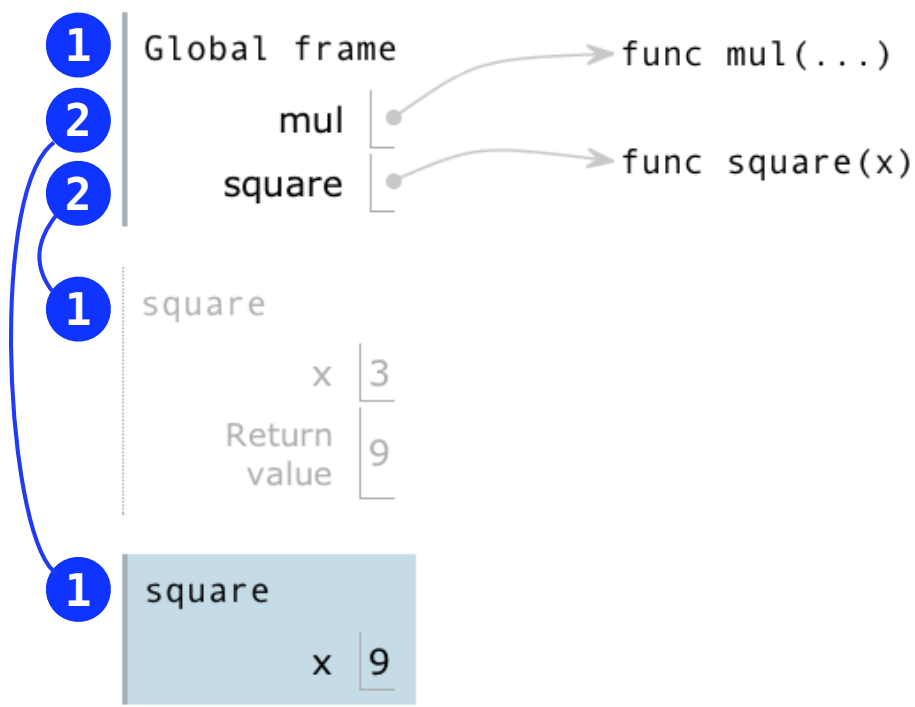
Names Have No Meaning Without Environments

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.

mul(x, x)

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



ıt:

Example: http://goo.gl/668fU

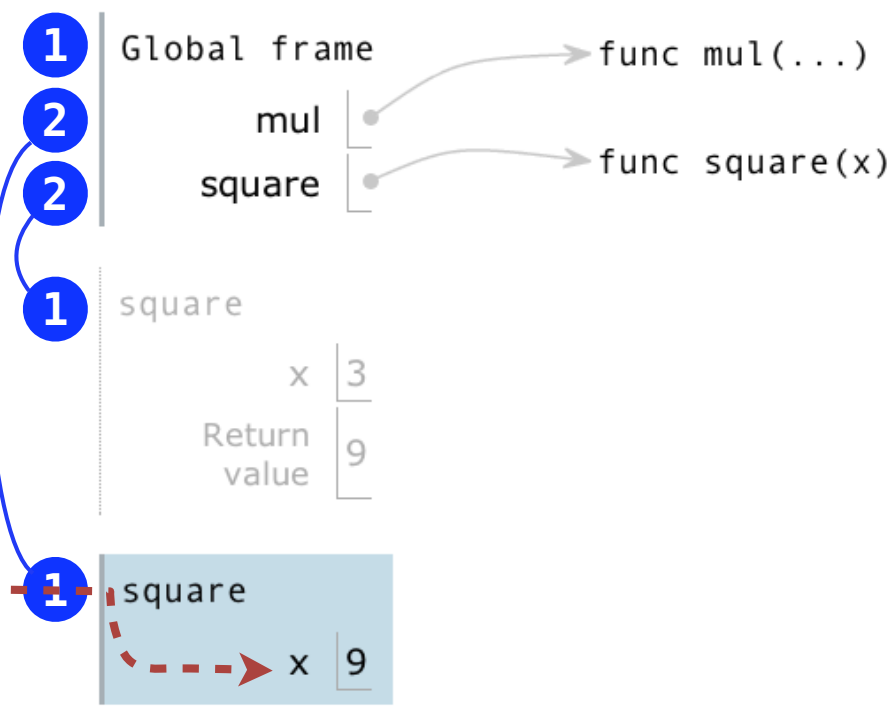
Names Have No Meaning Without Environments

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.

mul(x, x)

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



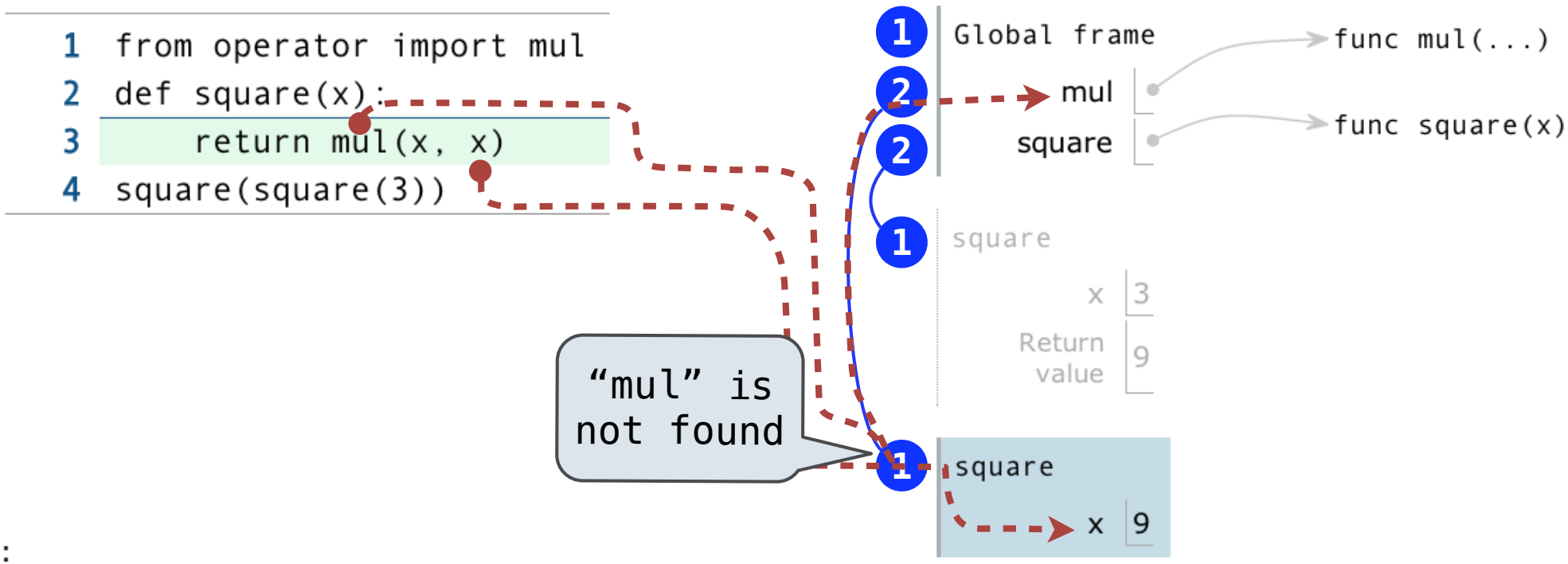
Example: http://goo.gl/668fU

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.

mul(x, x)



Example: http://goo.gl/668fU

Formal Parameters

Example: http://goo.gl/OapJa

Formal Parameters

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

Example: http://goo.gl/OapJa

Formal Parameters

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

vs

Example: http://goo.gl/OapJa

Formal Parameters

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

vs

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

Example: http://goo.gl/OapJa

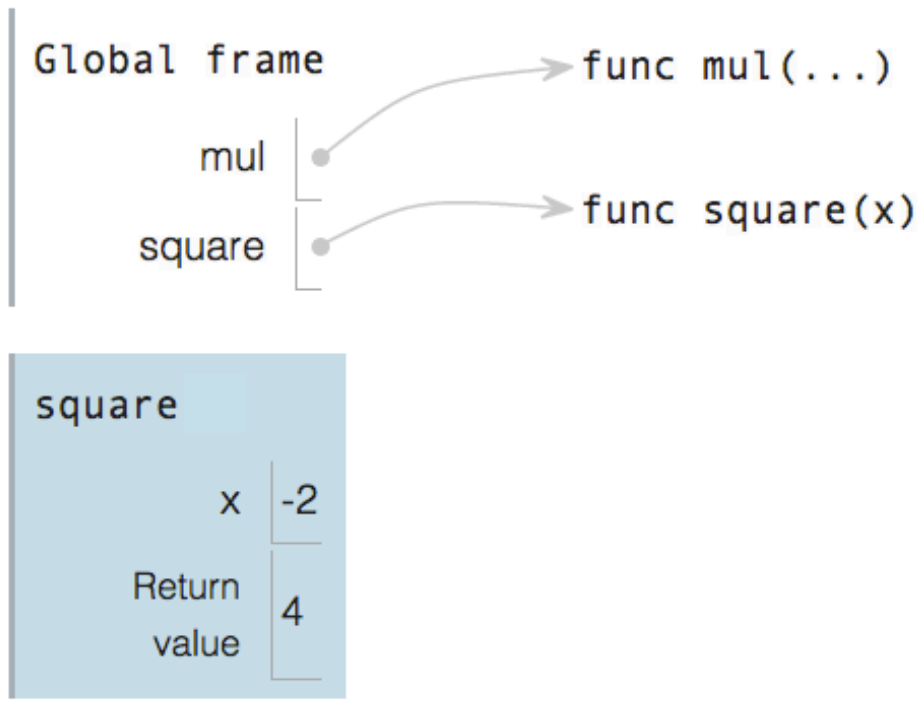
Formal Parameters

```
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)
```



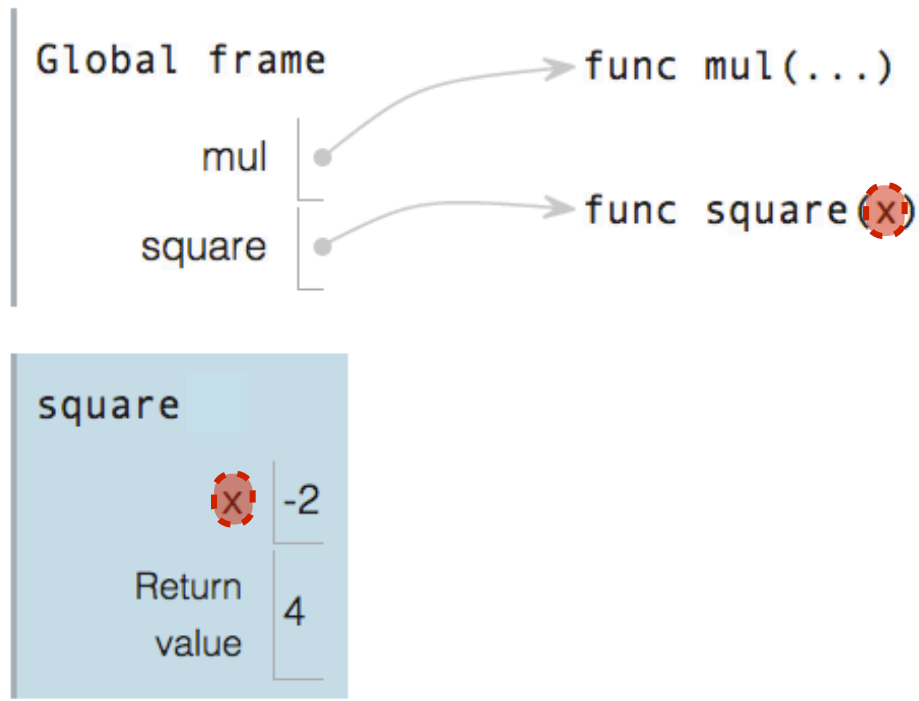
Formal Parameters

```
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)
```



Example: http://goo.gl/OapJa

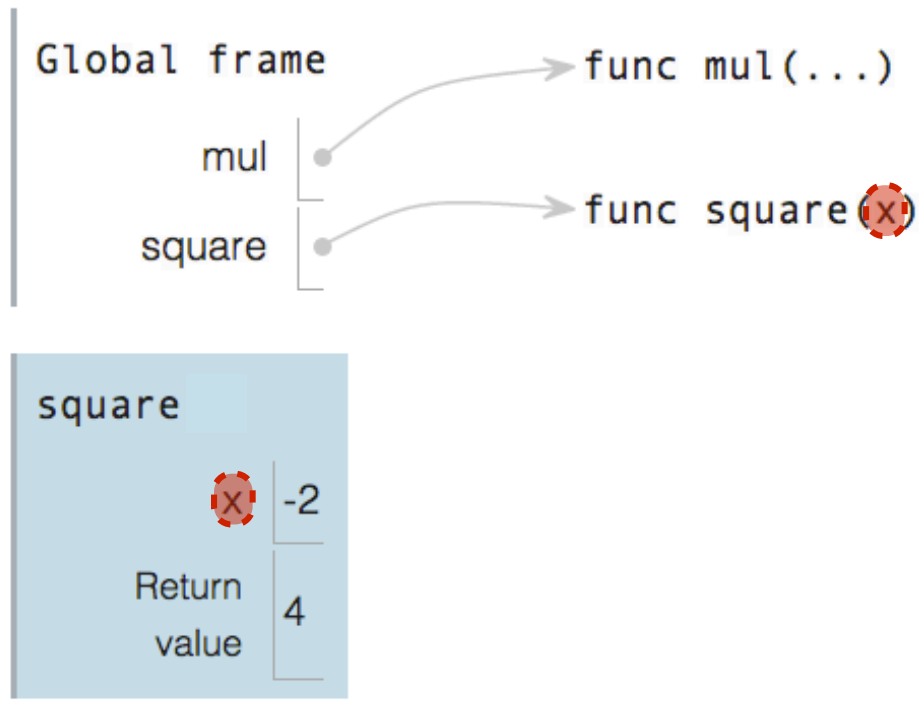
Formal Parameters

```
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

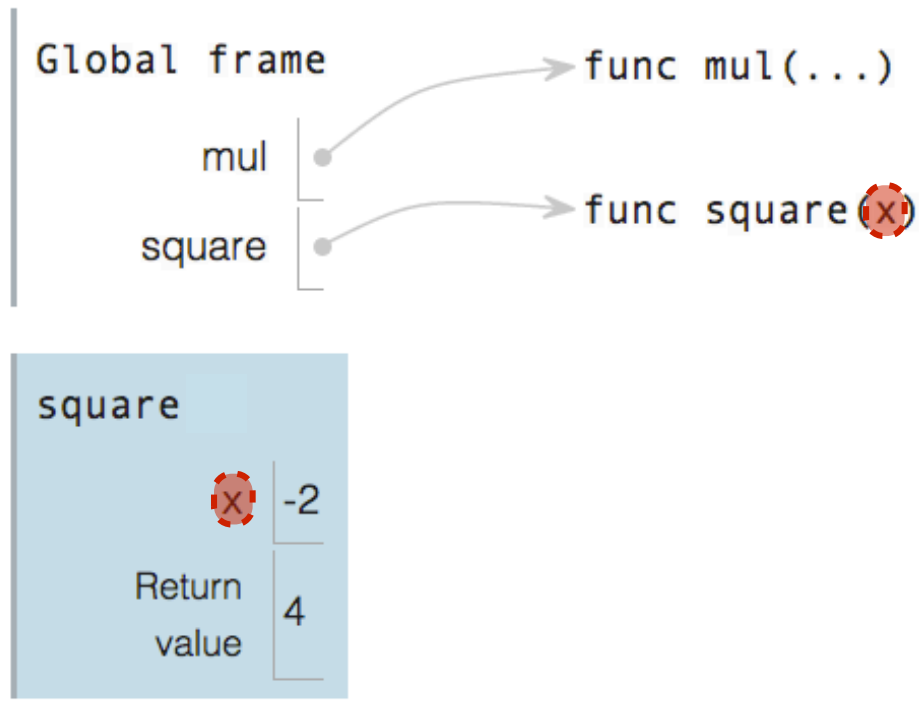
Formal Parameters

```
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

(Demo)

Example: http://goo.gl/OapJa

Python Feature Demonstration

Operators

Multiple Return Values

Docstrings

Doctests

Default Arguments

Statements

Statements

A statement

is executed by the interpret

to perform an action

Statements

A statement

is executed by the interpret

to perform an action

Compound statements:

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

Statements

A statement

is executed by the interpret

to perform an action

Compound statements:

Statement

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

Statements

A statement

is executed by the interpret

to perform an action

Compound statements:

Statement

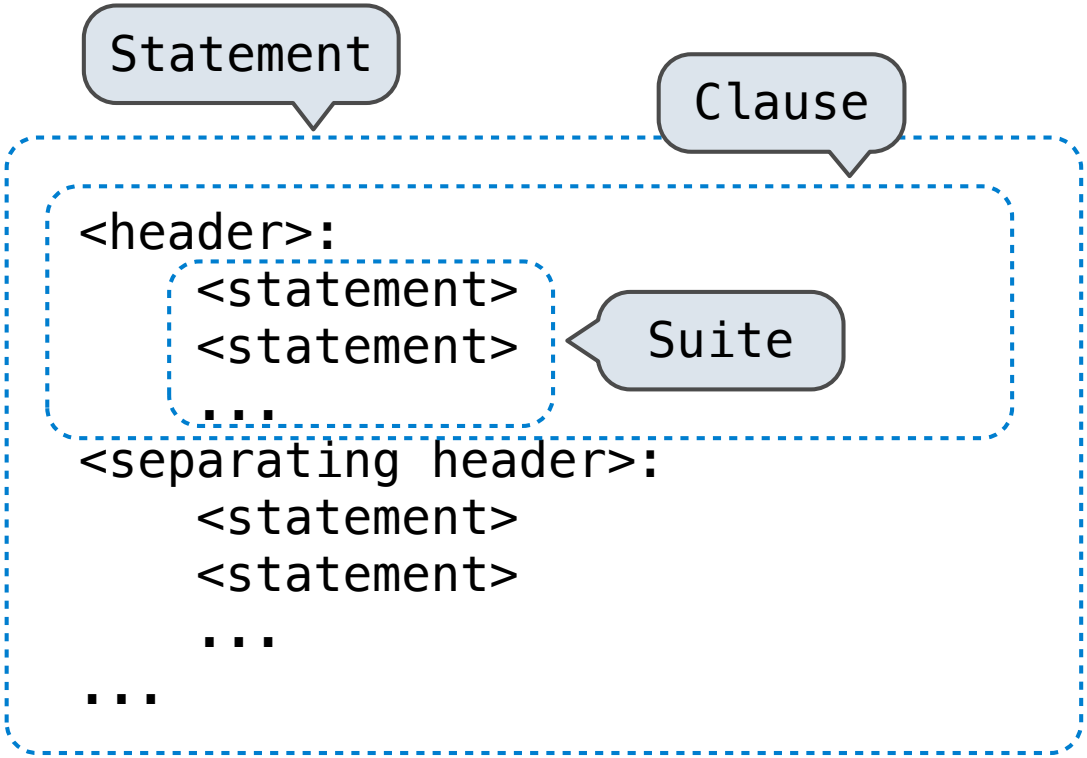
Clause

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

Statements

A statement
is executed by the interpret
to perform an action

Compound statements:



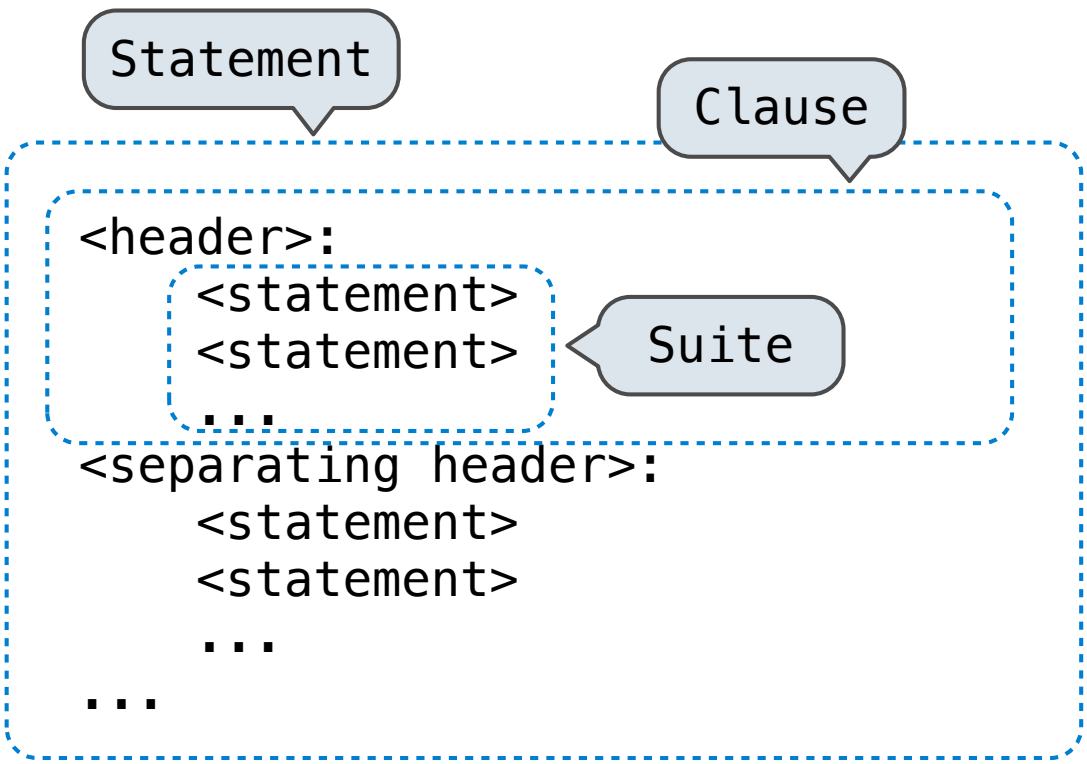
Statements

A statement

is executed by the interpret

to perform an action

Compound statements:



The first header determines a statement's type

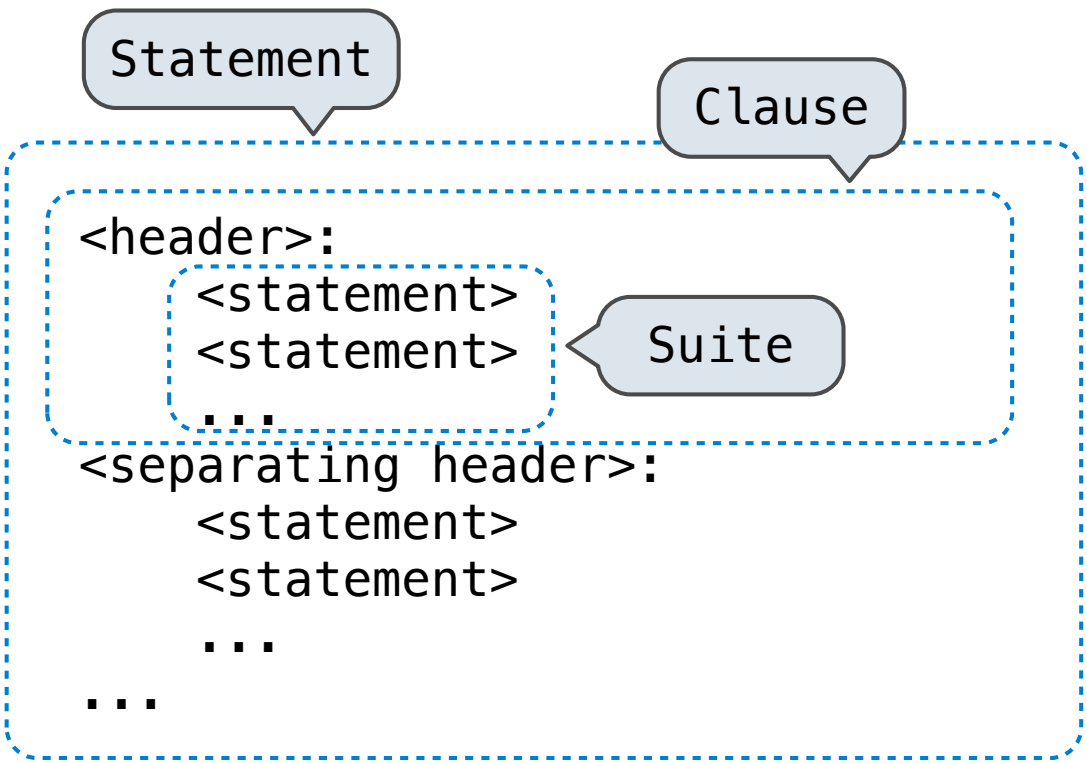
Statements

A statement

is executed by the interpret

to perform an action

Compound statements:



The first header determines a statement's type

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

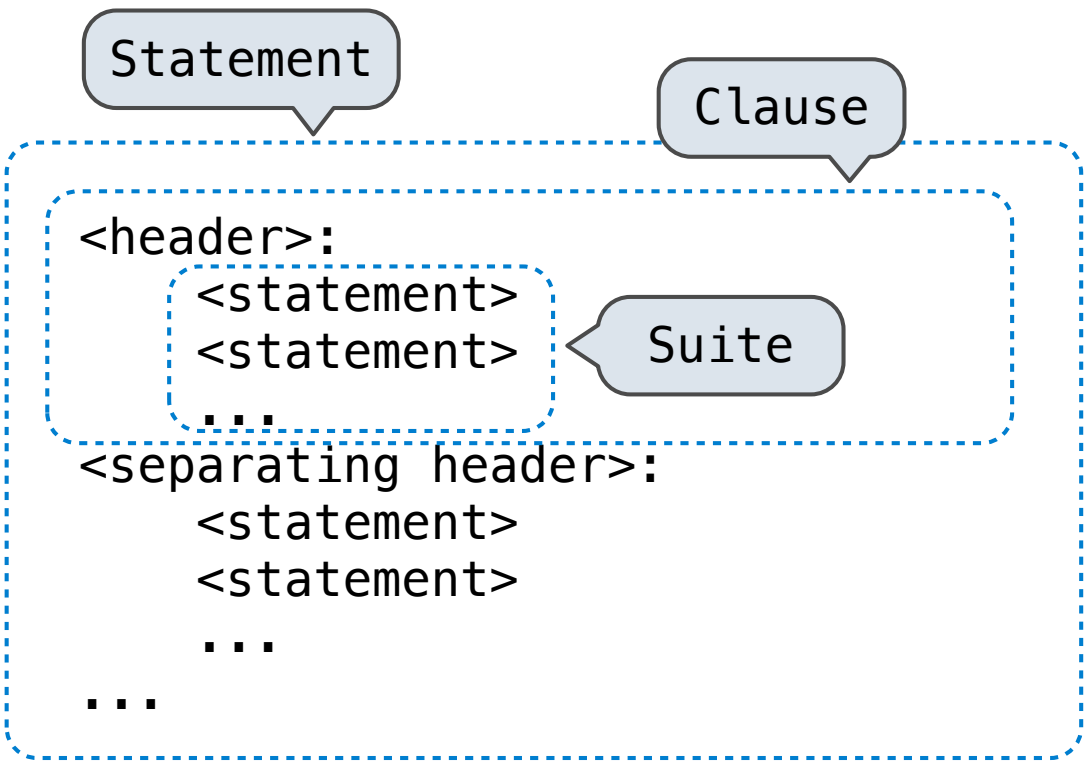
Statements

A statement

is executed by the interpret

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>
    ...
...
```

Suite

Compound Statements

Compound statements:

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

Suite

A suite is a sequence of statements

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

Compound Statements

Compound statements:

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

Suite

A suite is a sequence of statements

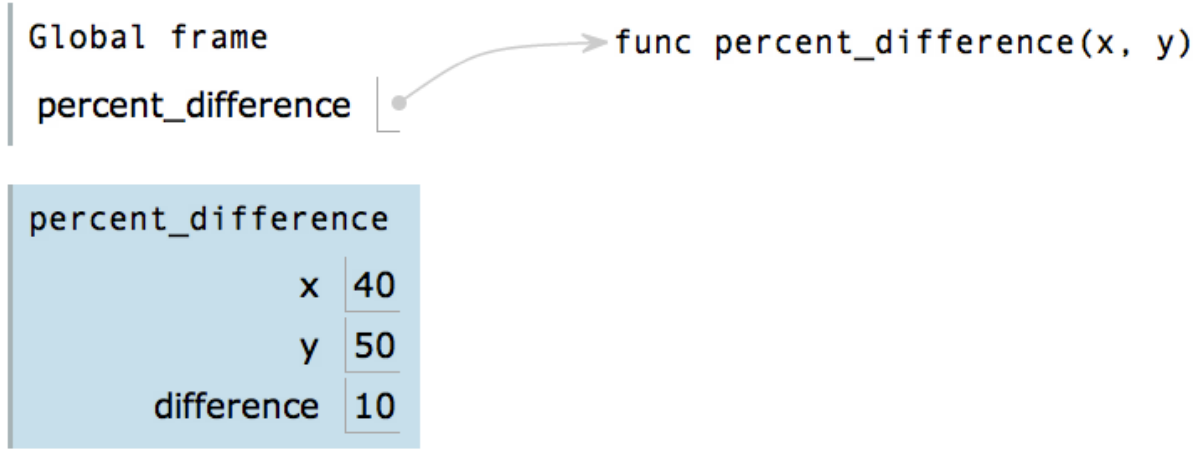
To “execute” a suite means to execute its sequence of statements, in order

Execution Rule for a sequence of statements:

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

Local Assignment

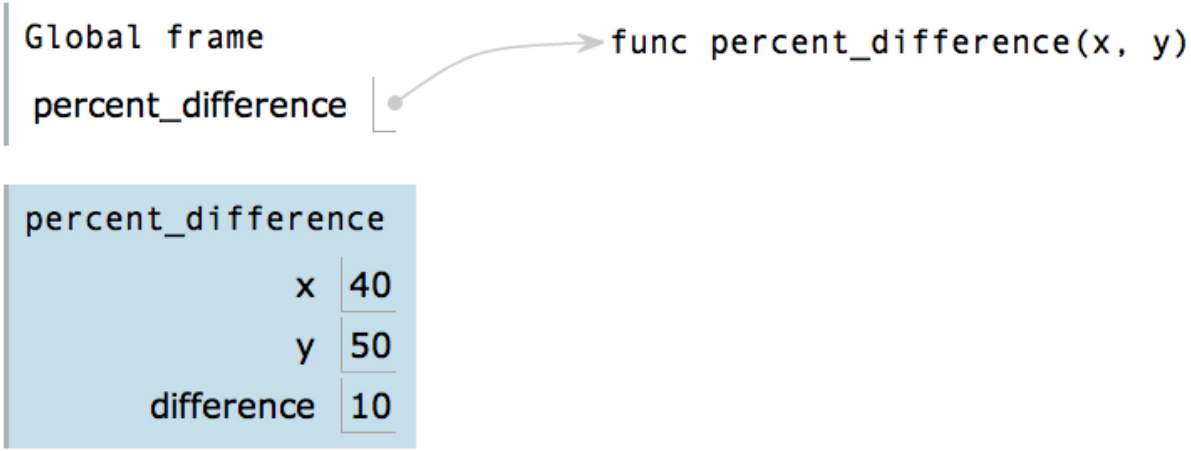
```
1  def percent_difference(x, y):
2          difference = abs(x-y)
3          return 100 * difference / x
4  diff = percent_difference(40, 50)
```



Example: http://goo.gl/wcF71

Local Assignment

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

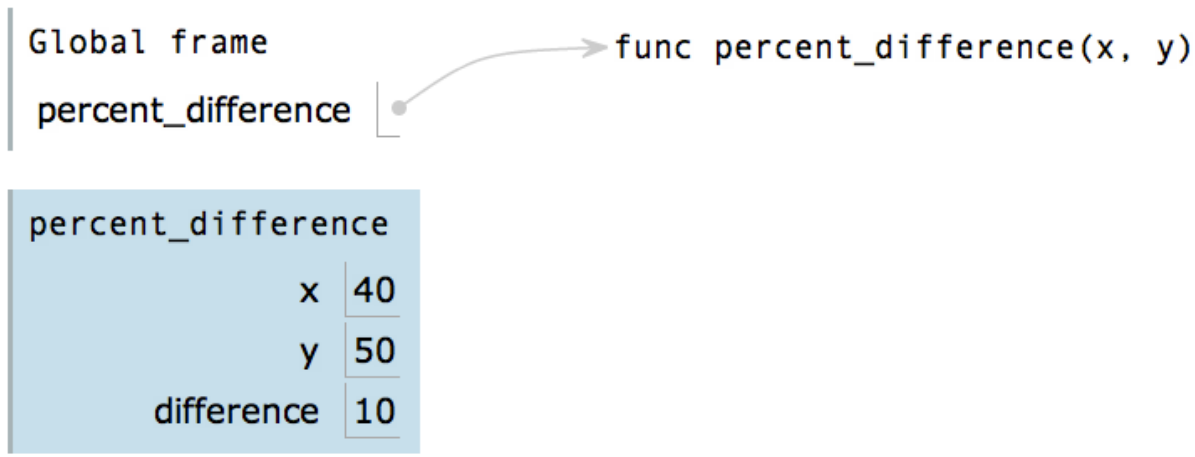


Execution rule for assignment statements:

Example: http://goo.gl/wcF71

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/wcF71

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

```
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

```
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

Execution rule for conditional statements:

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
```

1 statement,
3 clauses,
3 headers,
3 suites

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

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

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/O7y0D

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/O7y0D

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/O7y0D

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/O7y0D

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/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D

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.

Example: http://goo.gl/O7y0D