61A Lecture 3

Announcements

Print and None

(Demo)

None Indicates that Nothing is Returned

The special value **None** represents nothing in Python

A function that does not explicitly return a value will return **None**

Careful: **None** is *not displayed* by the interpreter as the value of an expression

```
>>> def does_not_return_square(x):
...     x * x
...
>>> does_not_return_square(4)
>>> sixteen = does_not_return_square(4)
>>> sixteen + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

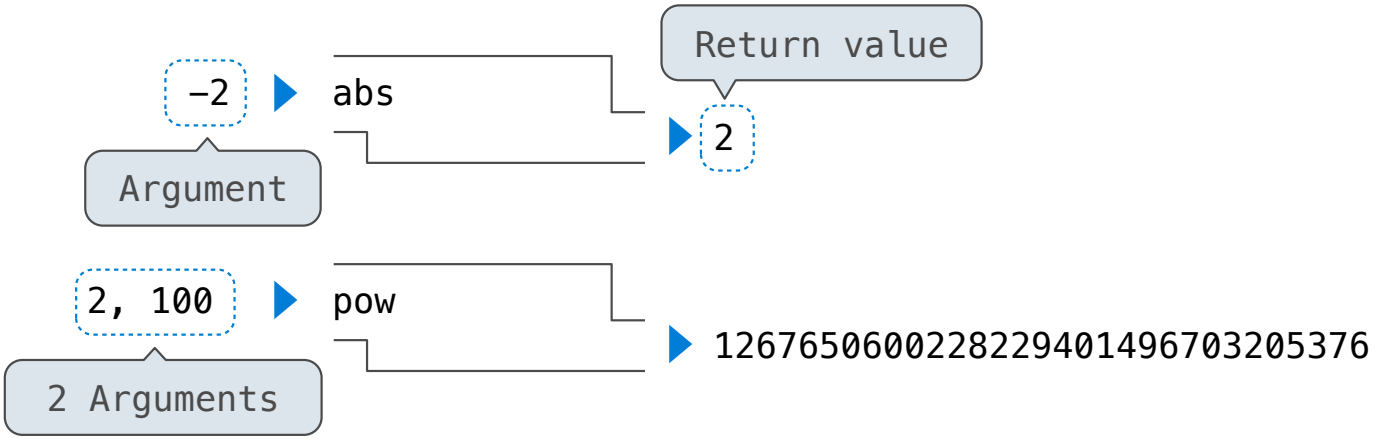
No **return**

None value is not displayed

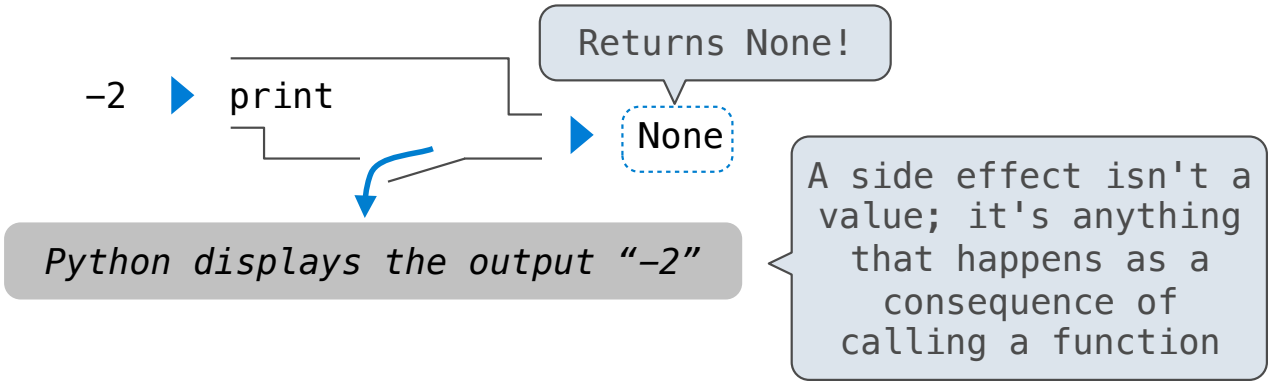
The name **sixteen** is now bound to the value **None**

Pure Functions & Non-Pure Functions

Pure Functions
just return values



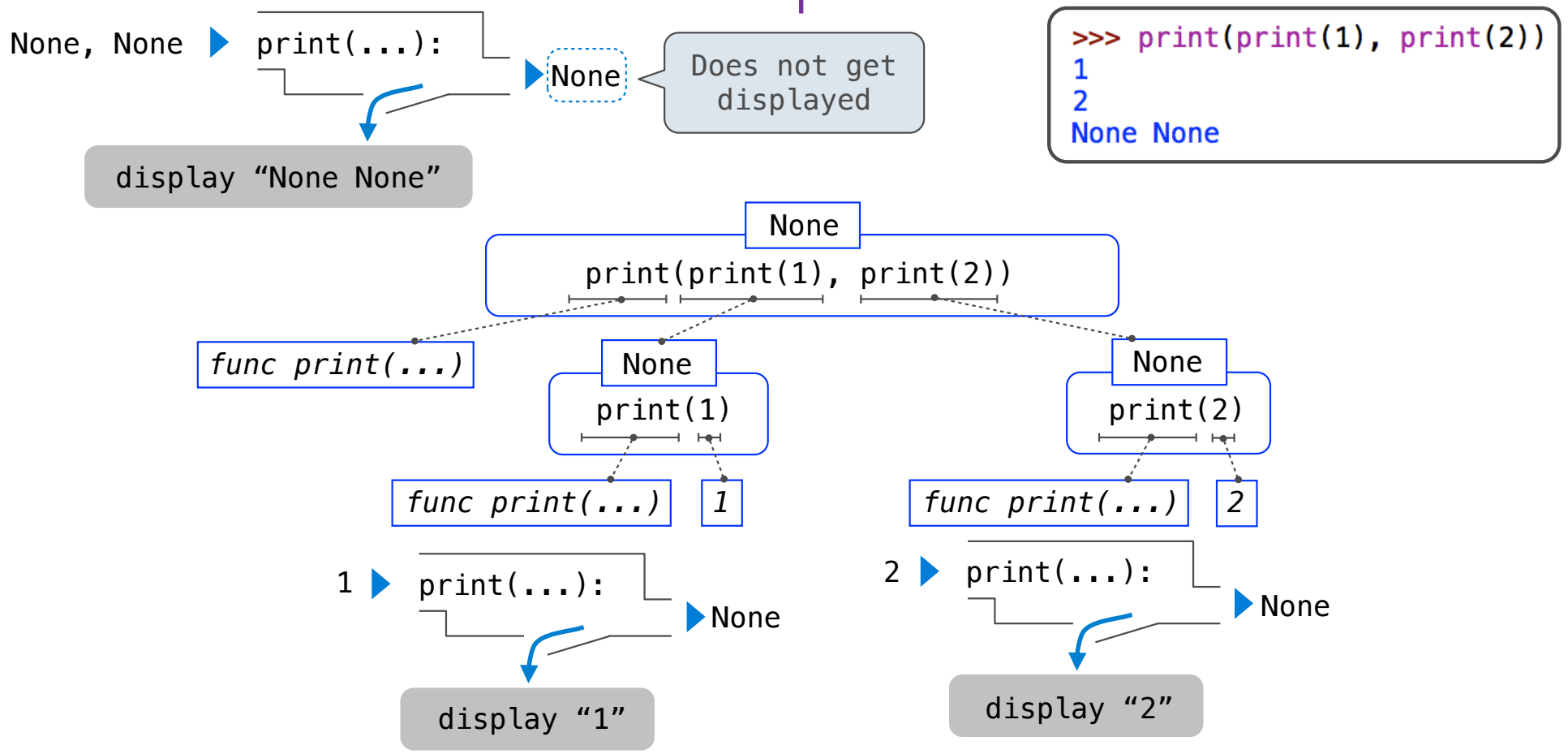
Non-Pure Functions
have side effects



(Demo)

Careful: None is *not displayed* by the interpreter as the value of an expression

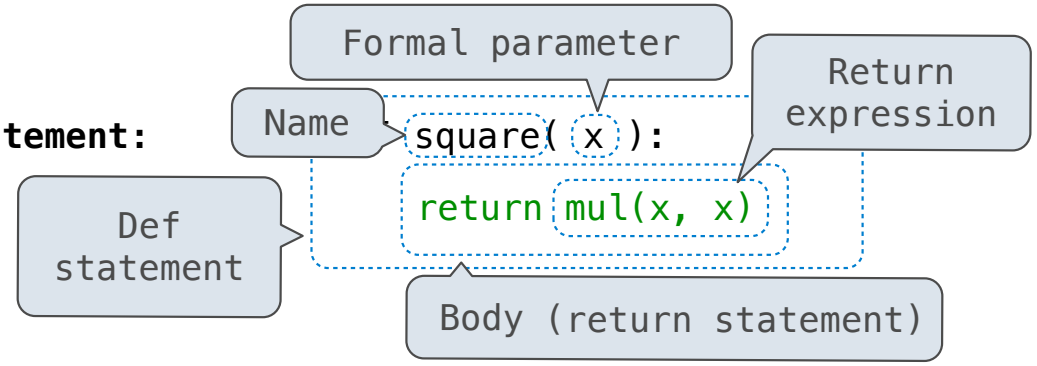
Nested Expressions with Print



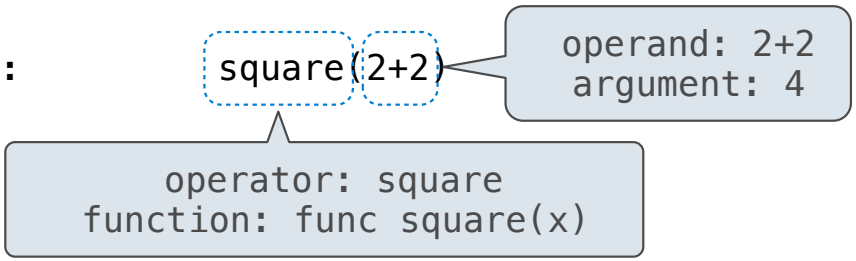
Multiple Environments

Life Cycle of a User-Defined Function

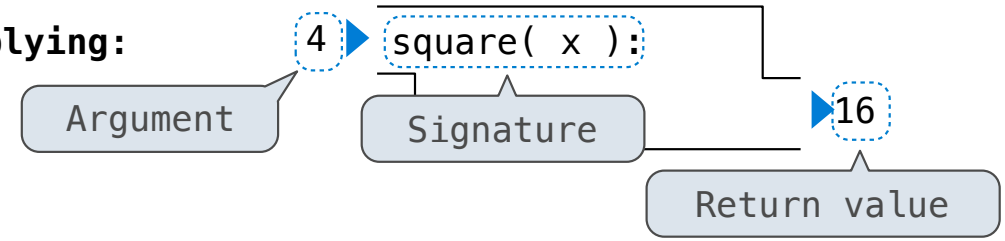
Def statement:



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)

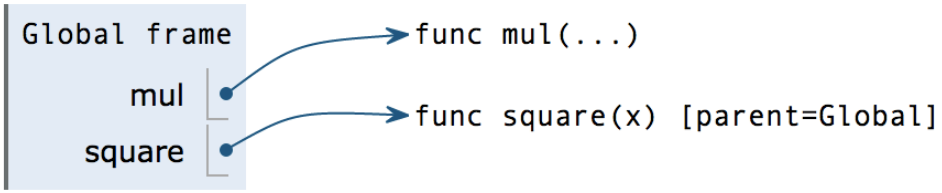
A new frame is created!

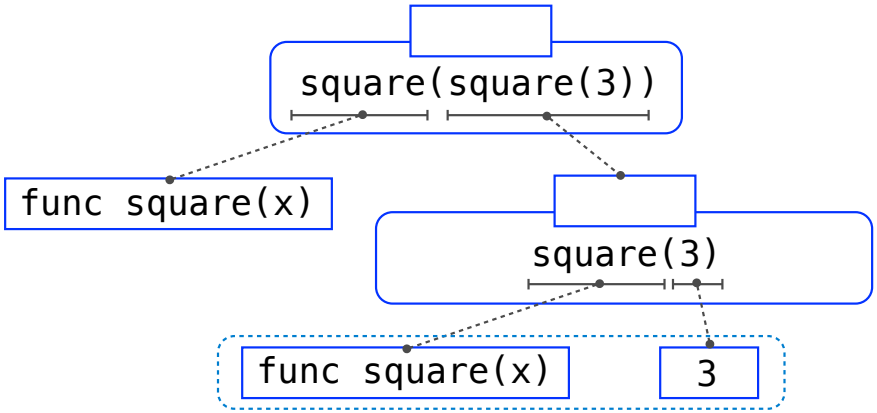
Parameters bound to arguments

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))
```

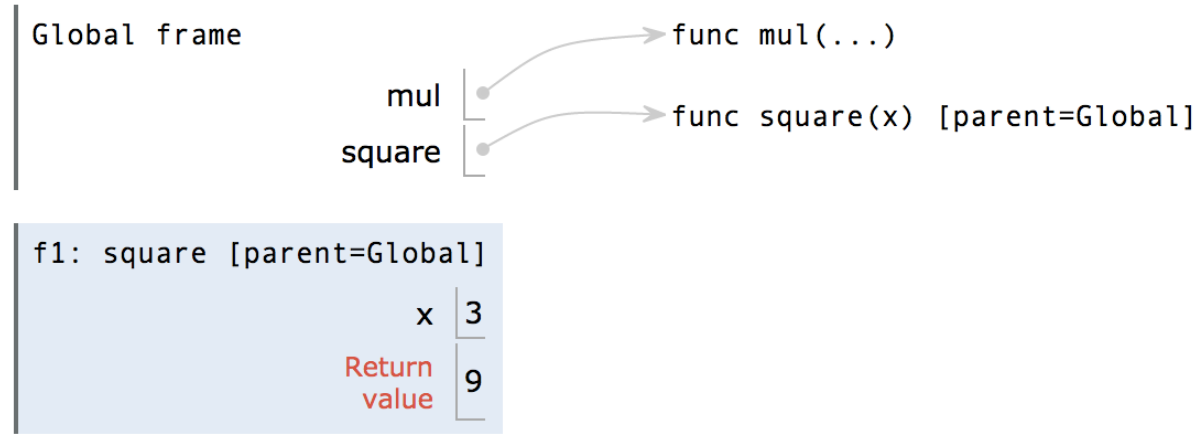


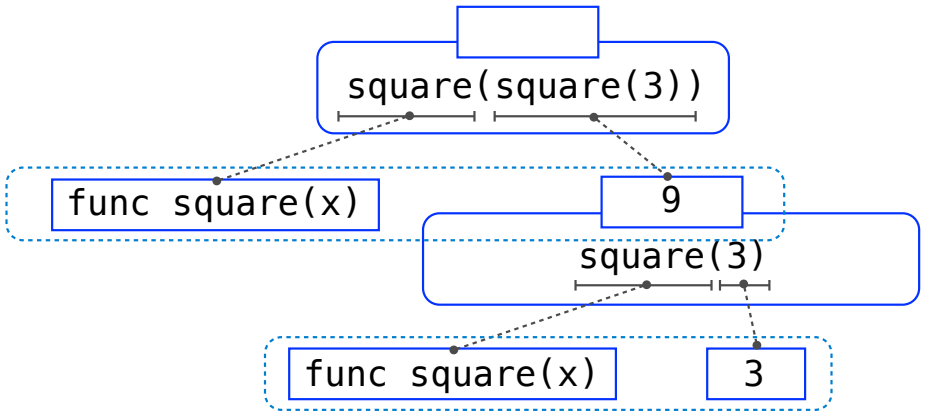


Interactive Diagram

Multiple Environments in One Diagram!

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

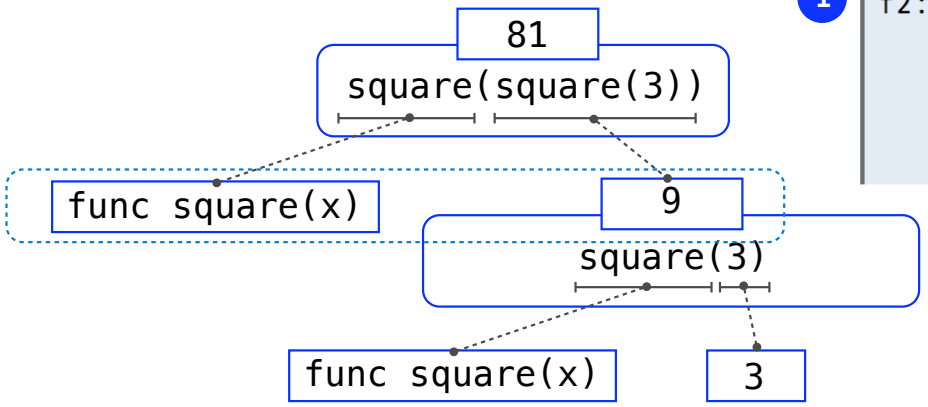


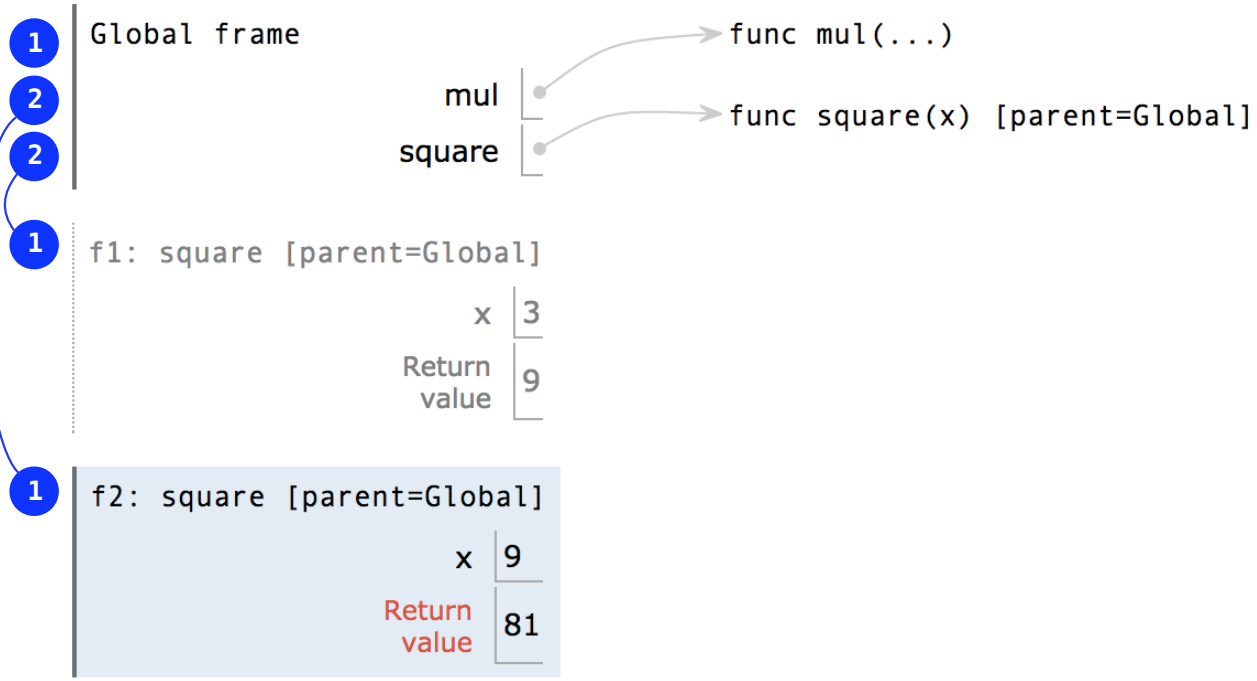


Interactive Diagram

Multiple Environments in One Diagram!

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





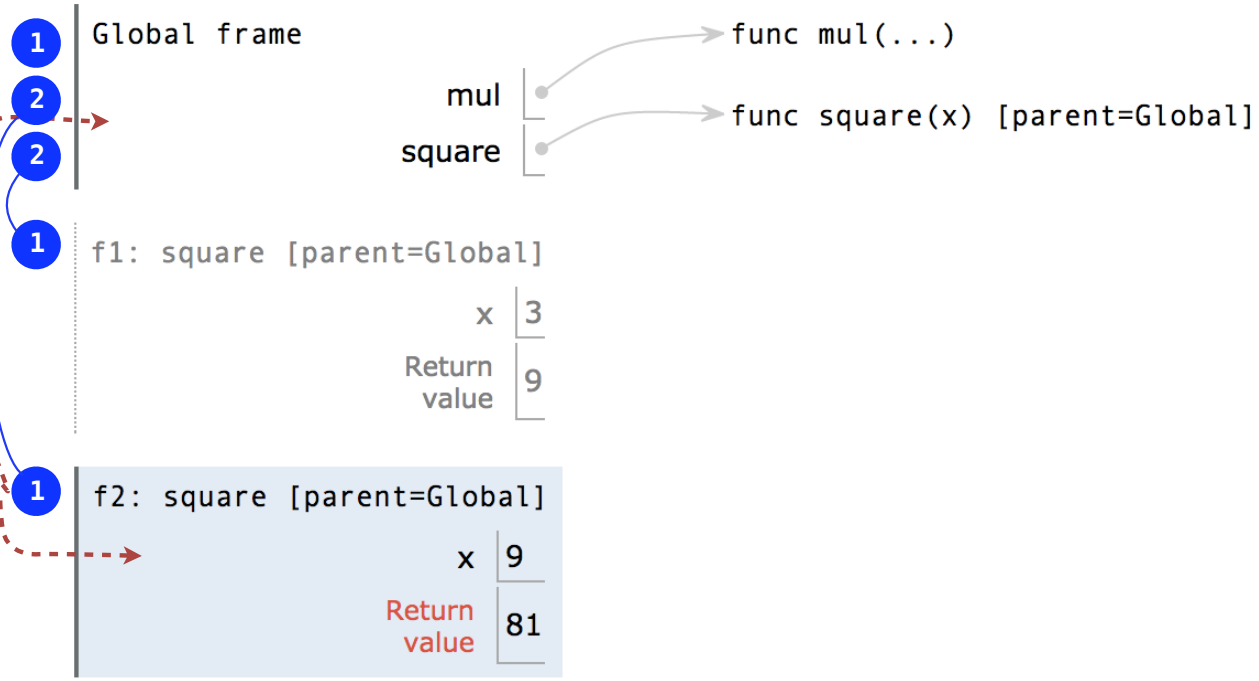
An environment is a sequence of frames.

- 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.

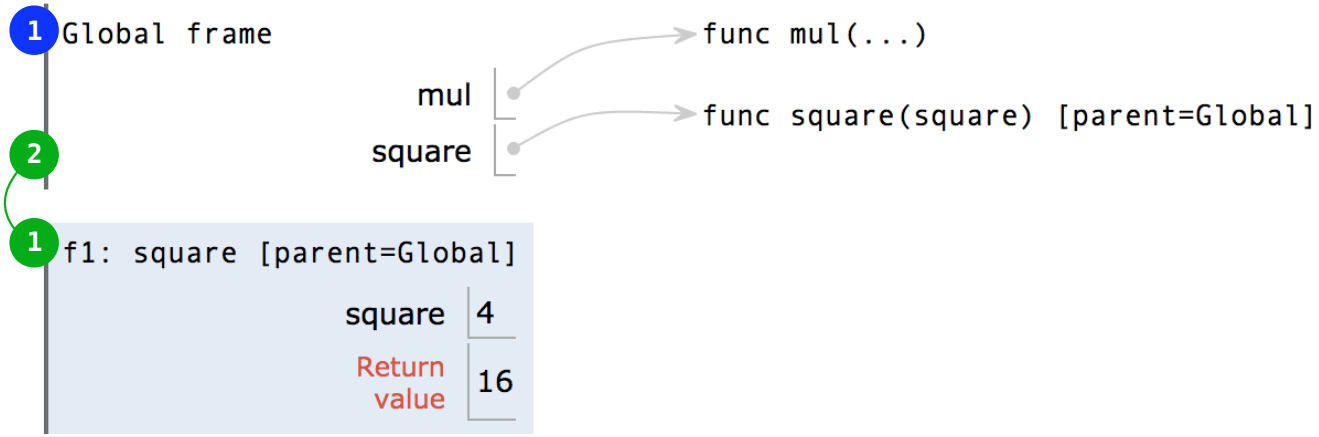
- The global frame alone
- A local, then the global frame

Interactive Diagram

Names Have 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

Division

Multiple Return Values

Source Files

Doctests

Default Arguments

(Demo)

Division

```
>>> 2013 / 10
201.3
>>> 2013 // 10
201
>>> from operator import truediv, floordiv
>>> truediv(2013, 10)
201.3
>>> floordiv(2013, 10)
201
>>> 2013 % 10
3
>>> from operator import mod
>>> mod(2013, 10)
3
```

Multiple Return Values

```
>>> def divide_exact(n, d):
...     return n // d, n % d
...
>>> quotient, remainder = divide_exact(2013, 10)
>>> quotient
201
>>> remainder
3
```

Source Files

```
~/lec$ python3 -i ex.py
>>> q
201
>>> r
3
```

```
1 """Our first Python source file."""
2
3 from operator import floordiv, mod
4
5 def divide_exact(n, d):
6     return floordiv(n, d), mod(n, d)
7
8 q, r = divide_exact(2013, 10)
```

Source Files and Doctests

This is a documentation, including
Doc string: first line below the def line
Doctest

```
bash
~/lec$ python3 -m doctest ex.py
~/lec$ python3 -m doctest -v ex.py
Trying:
    q, r = divide_exact(2013, 10)
Expecting nothing
ok
Trying:
    q
Expecting:
    201
ok
Trying:
    r
Expecting:
    3
ok
1 items had no tests:
    ex
1 items passed all tests:
   3 tests in ex.divide_exact
3 tests in 2 items.
3 passed and 0 failed.
Test passed.
~/lec$
```

Test passed

```
1  """Our first Python source file."""
2
3  from operator import floordiv, mod
4
5  def divide_exact(n, d):
6      """Return the quotient and remainder of dividing N by D.
7
8      >>> q, r = divide_exact(2013, 10)
9      >>> q
10     201
11     >>> r
12     3
13     """
14     return floordiv(n, d), mod(n, d)
15
```

all cap letter refer to
formal parameters

```
**********************************************************
**********
File "./ex.py", line 11, in ex.divide_exact
Failed example:
    r
Expected:
    2
Got:
    3
1 items had no tests:
    ex
**********************************************************
**********
1 items had failures:
   1 of   3 in ex.divide_exact
3 tests in 2 items.
2 passed and 1 failed.
***Test Failed*** 1 failures.
```

error in here ther

Test failed

Default Arguments

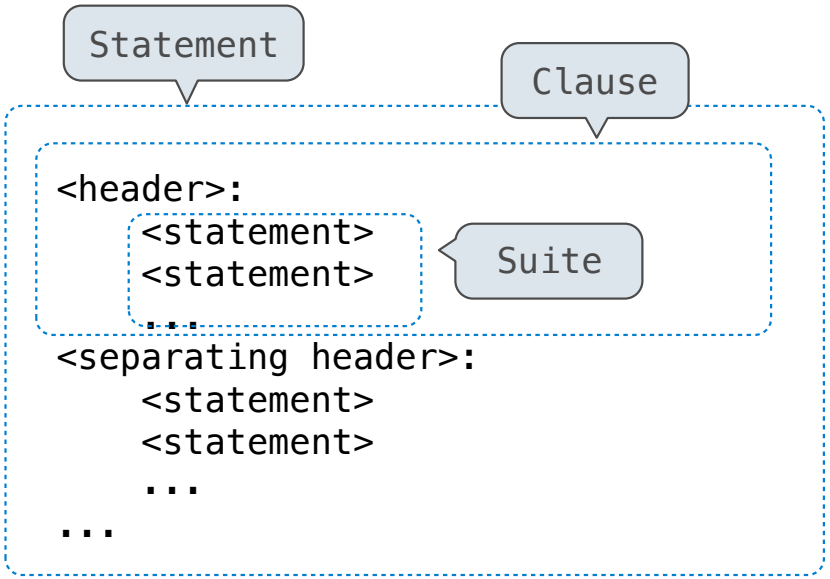
```
"""Our first Python source file."""

from operator import floordiv, mod

def divide_exact(n, d=10):
    """Return the quotient and remainder of dividing N by D.

    >>> q, r = divide_exact(2013, 10)
    >>> q
    201
    >>> r
    3
    """
    return floordiv(n, d), mod(n, d)

```

Conditional Statements

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>
    ...
...
```

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)

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:

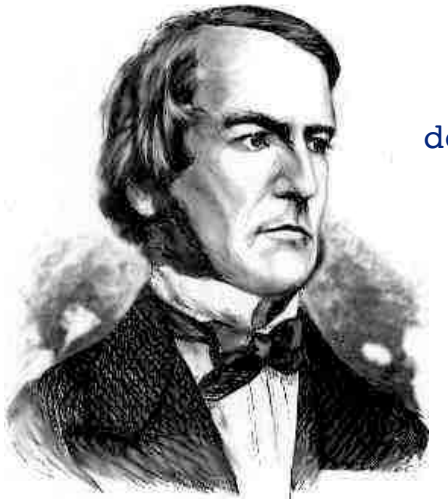
1. Always starts with "if" clause.
2. Zero or more "elif" clauses.
3. Zero or one "else" clause,
   always at the end.

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
```

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

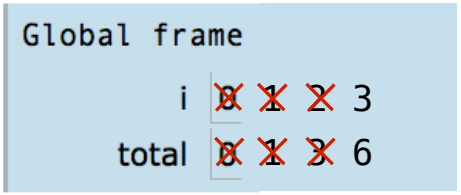
Iteration

While Statements

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.