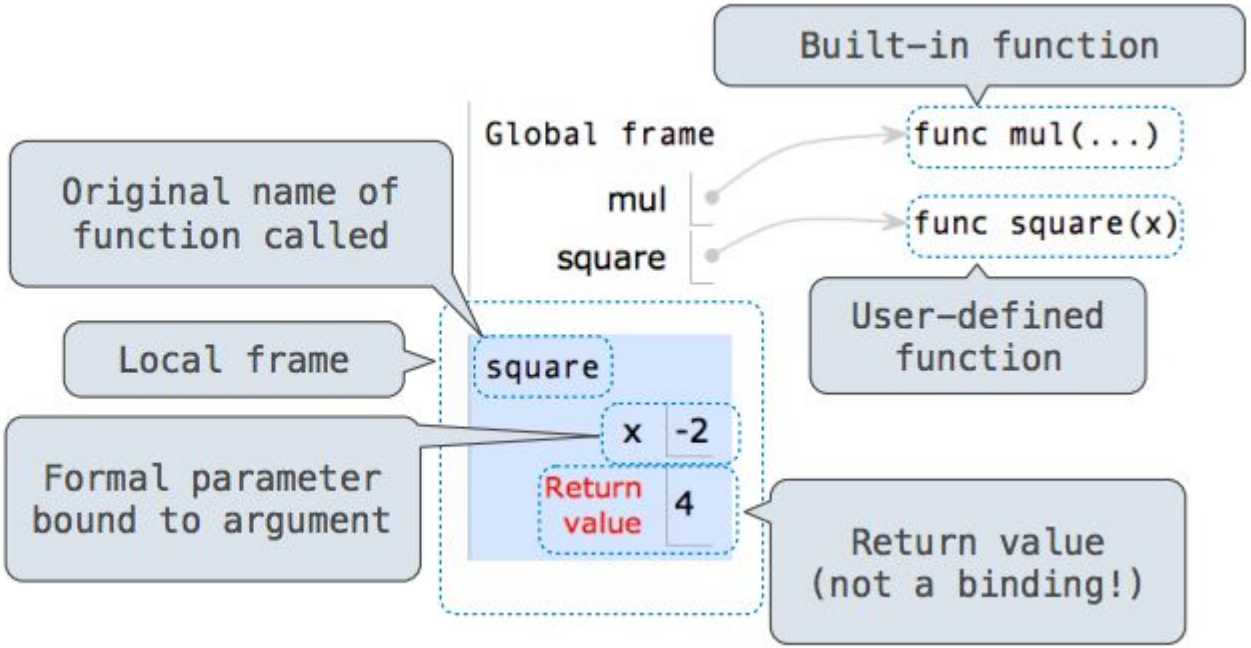
Lecture 3

Environments

Many slides were borrowed from John DeNero

Last time: calling user - defined functions

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



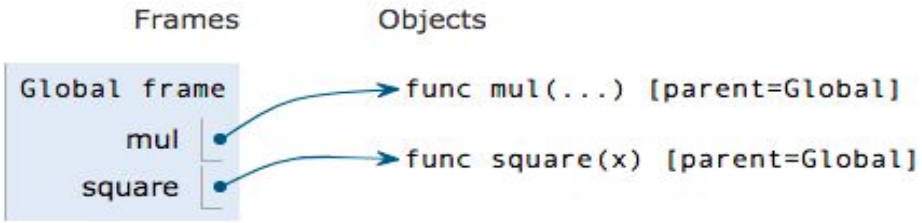
Multiple Environments

Multiple Environments in one diagram (John's)

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

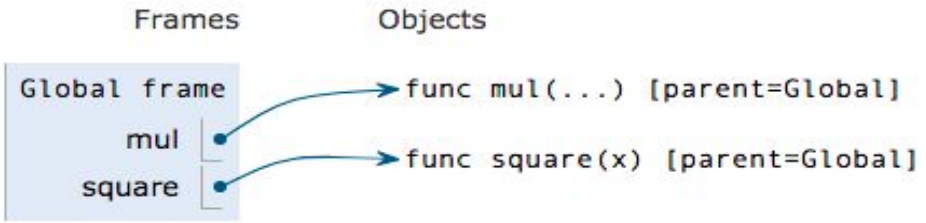
Multiple Environments in one diagram

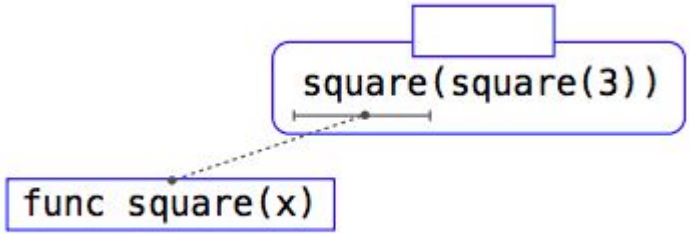
```
1 from operator import mul
2 def square (x):
3     return mul(x,x)
4 square(square(3))
```



Multiple Environments in one diagram

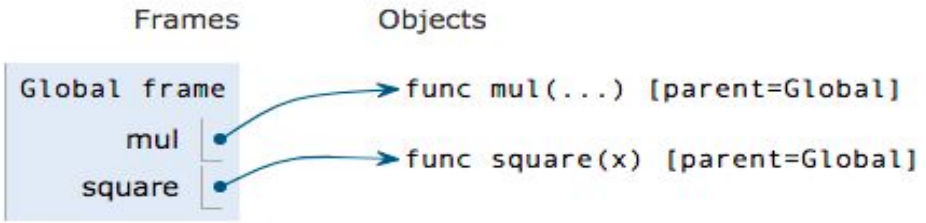
```
1 from operator import mul
2 def square (x):
3     return mul(x,x)
4 square(square(3))
```

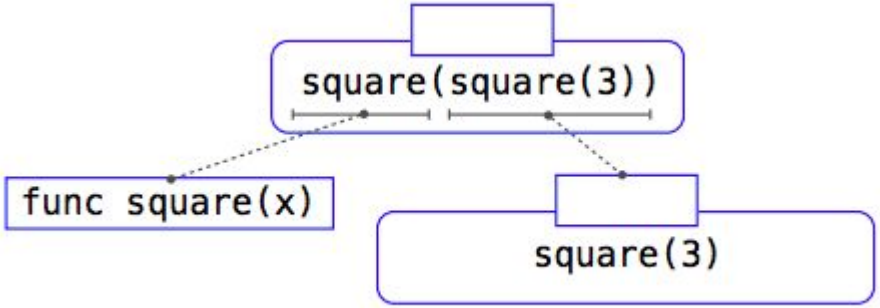




Multiple Environments in one diagram

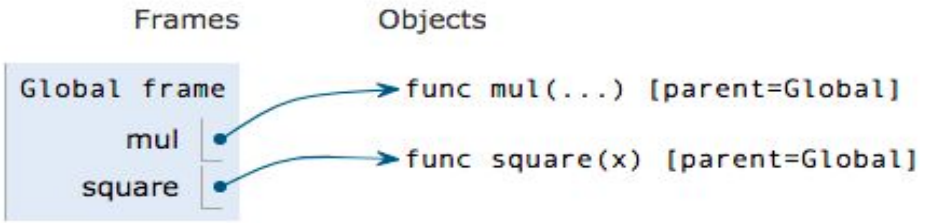
```
1 from operator import mul
2 def square (x):
3     return mul(x,x)
4 square(square(3))
```

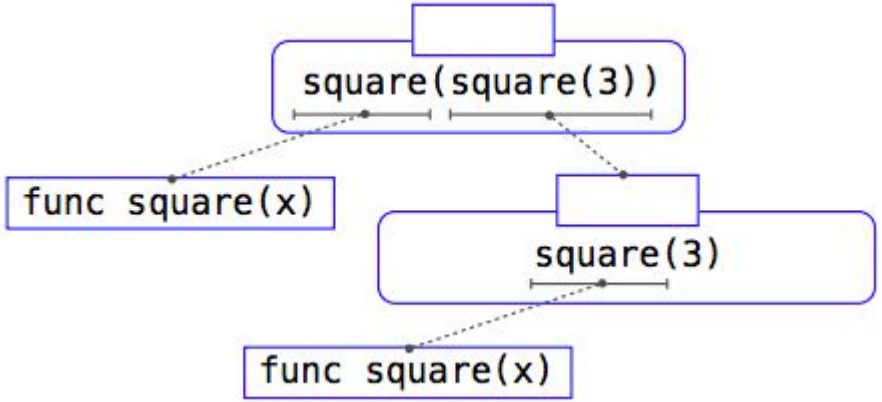




Multiple Environments in one diagram

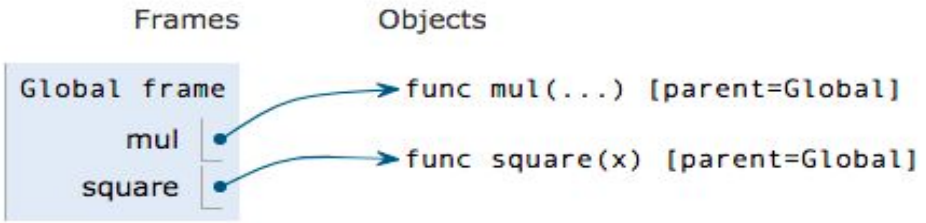
```
1 from operator import mul
2 def square (x):
3     return mul(x,x)
4 square(square(3))
```

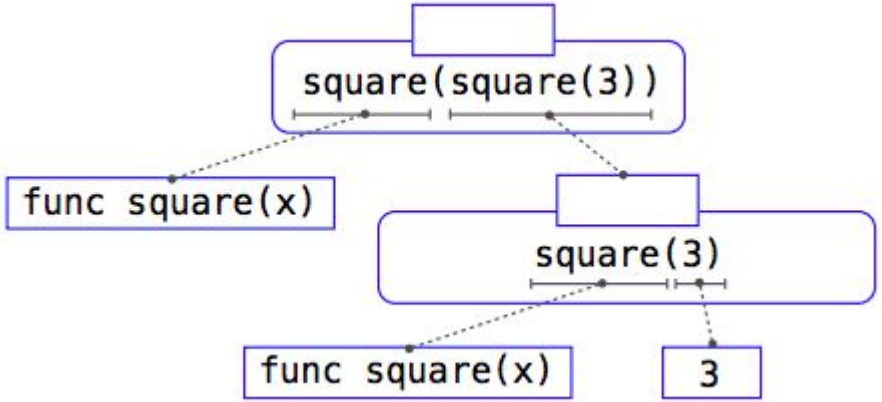




Multiple Environments in one diagram

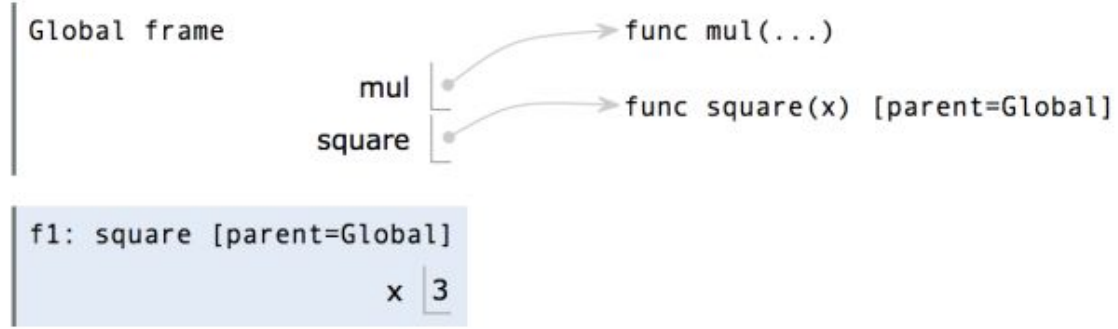
```
1 from operator import mul
2 def square (x):
3     return mul(x,x)
4 square(square(3))
```

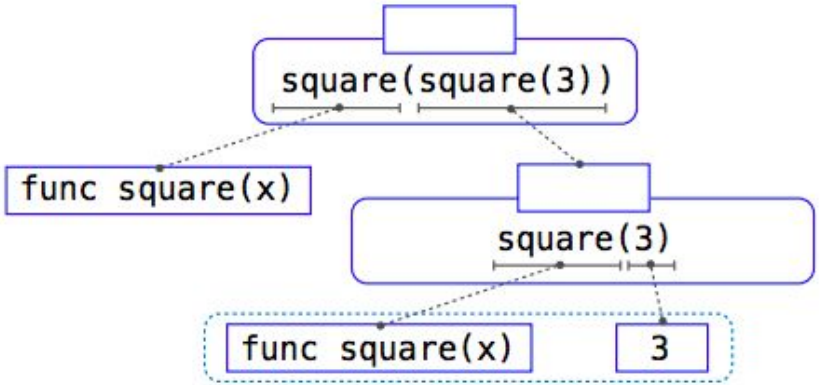




Multiple Environments in one diagram

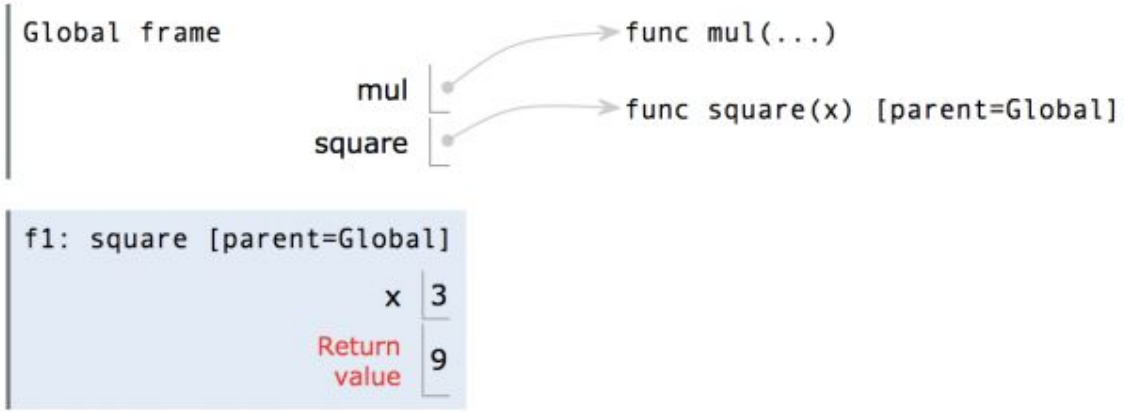
```
1 from operator import mul
2 def square(x):
3     return mul(x, x)
4 square(square(3))
```

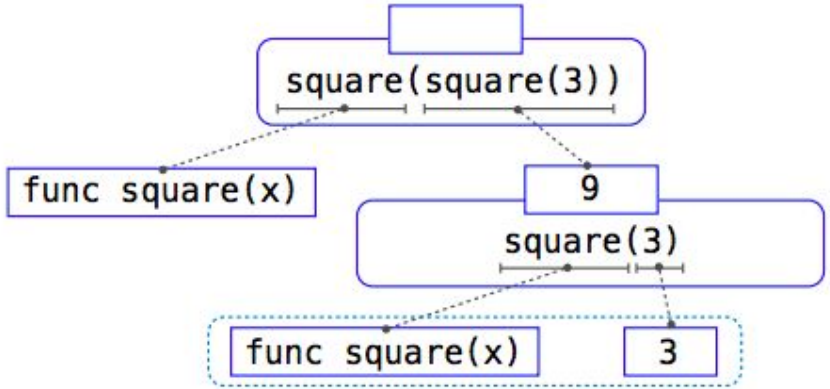




Multiple Environments in one diagram

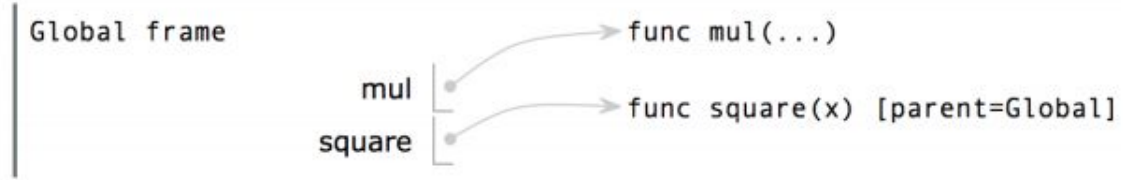
```
1  from operator import mul
2  def square(x):
3      return mul(x, x)
4  square(square(3))
```

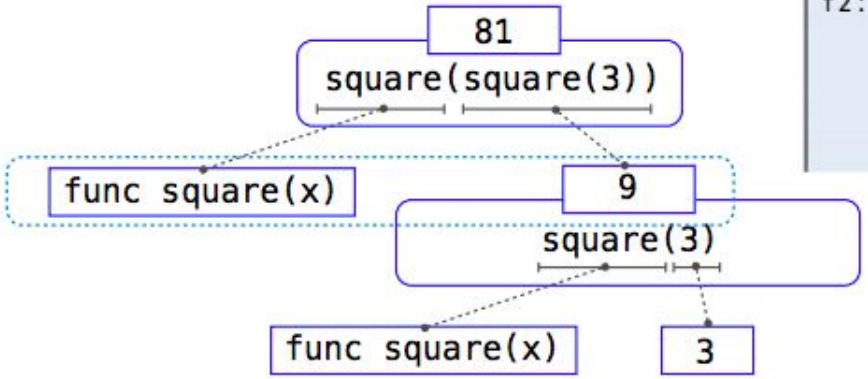




Multiple Environments in one diagram

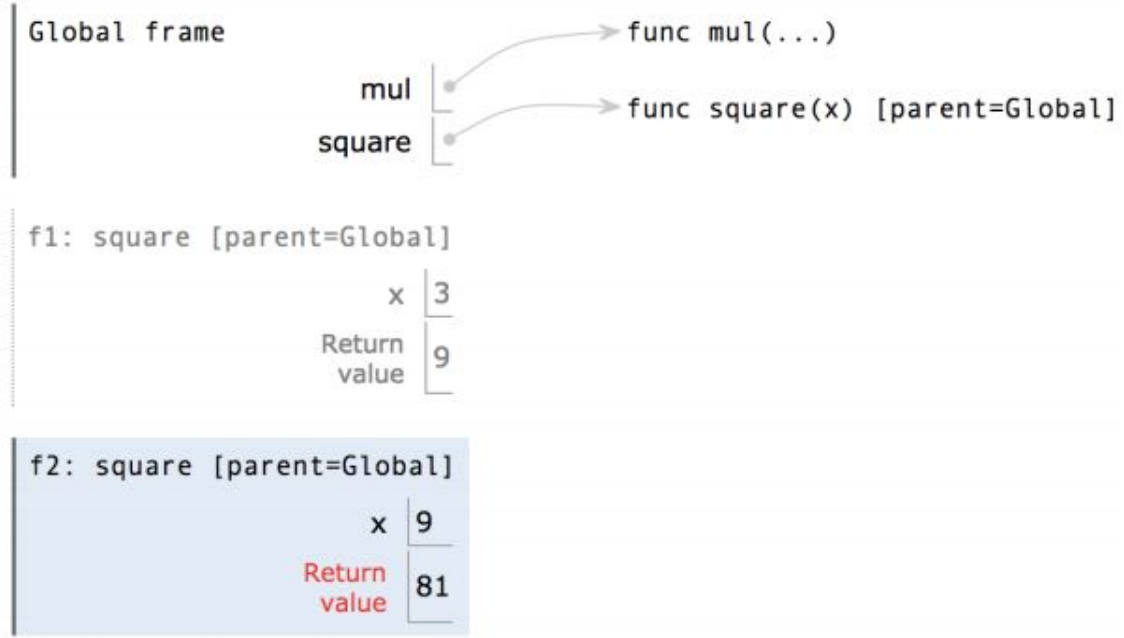
```
1  from operator import mul
2  def square(x):
3      return mul(x, x)
4  square(square(3))
```

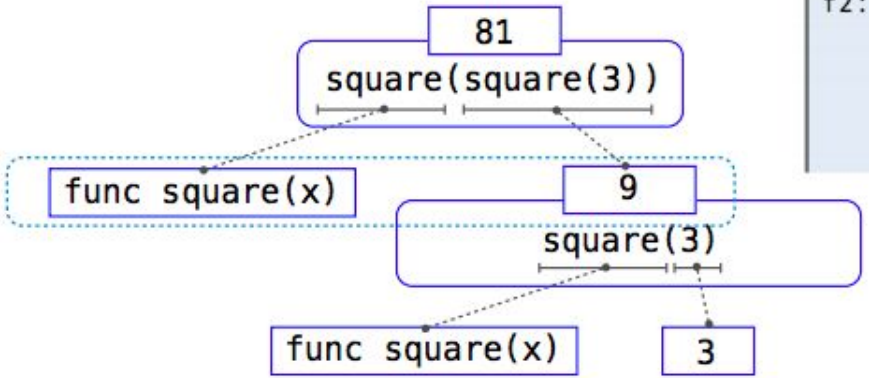




Multiple Environments in one diagram

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



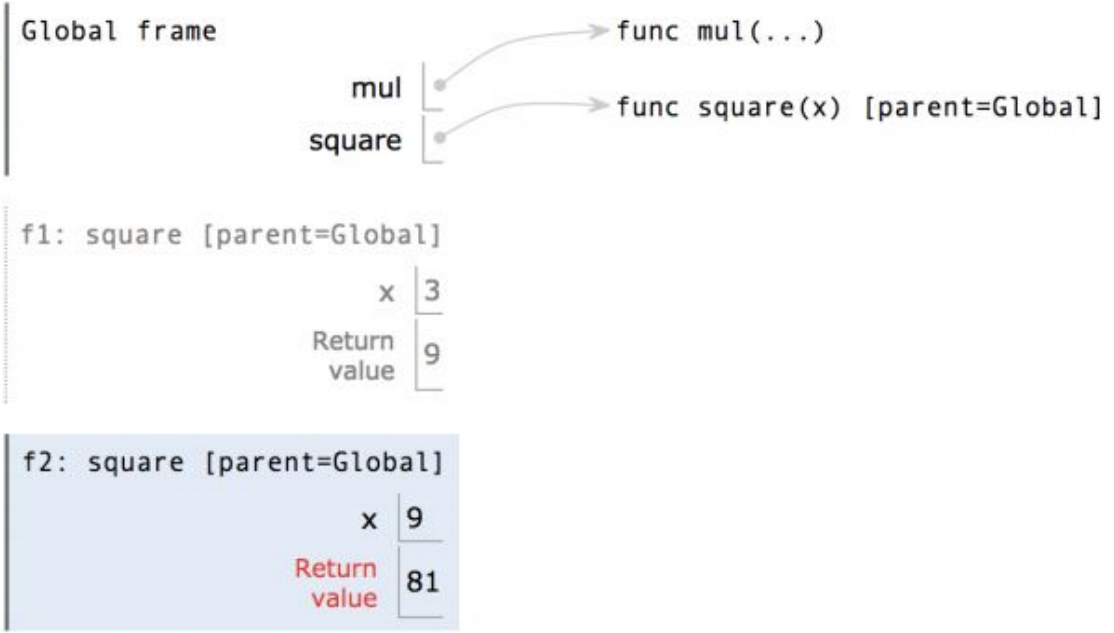


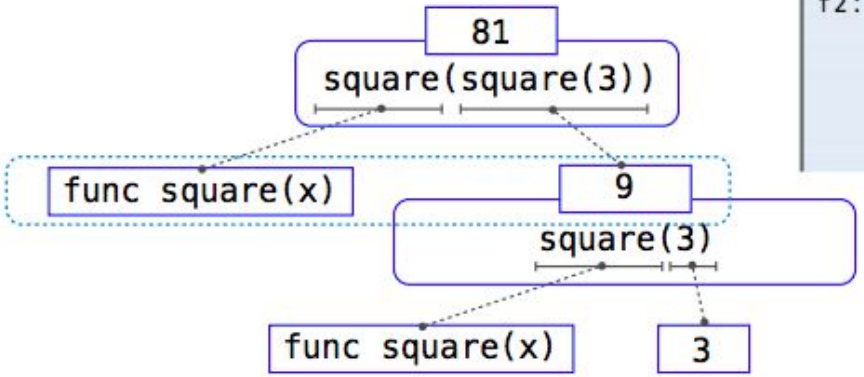
An environment is a sequence of frames.

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

Multiple Environments in one diagram

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



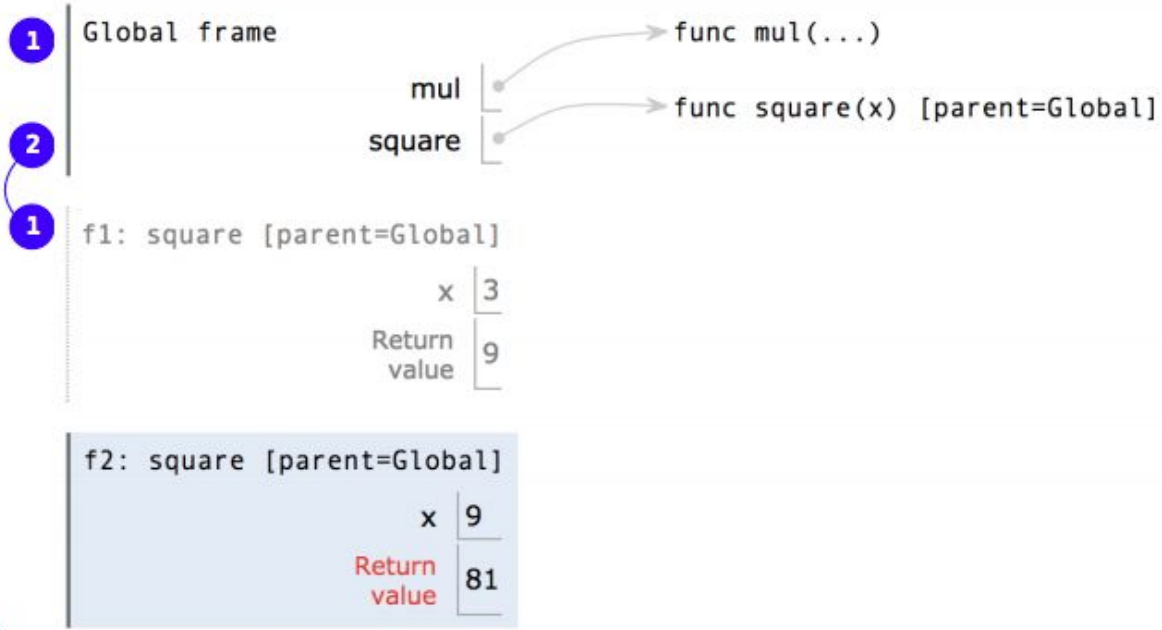


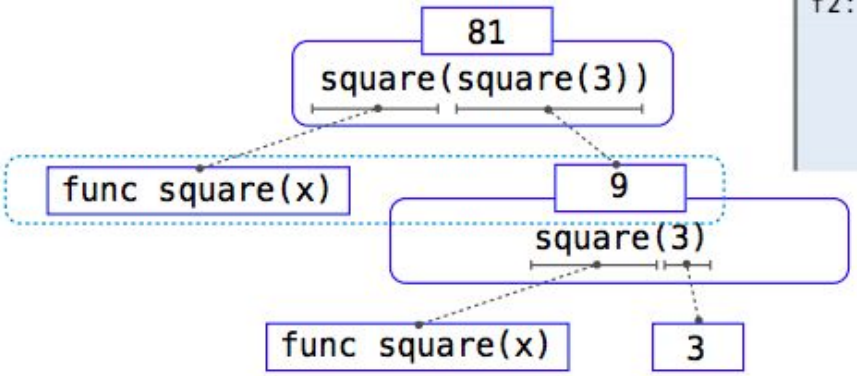
An environment is a sequence of frames.

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

Multiple Environments in one diagram

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



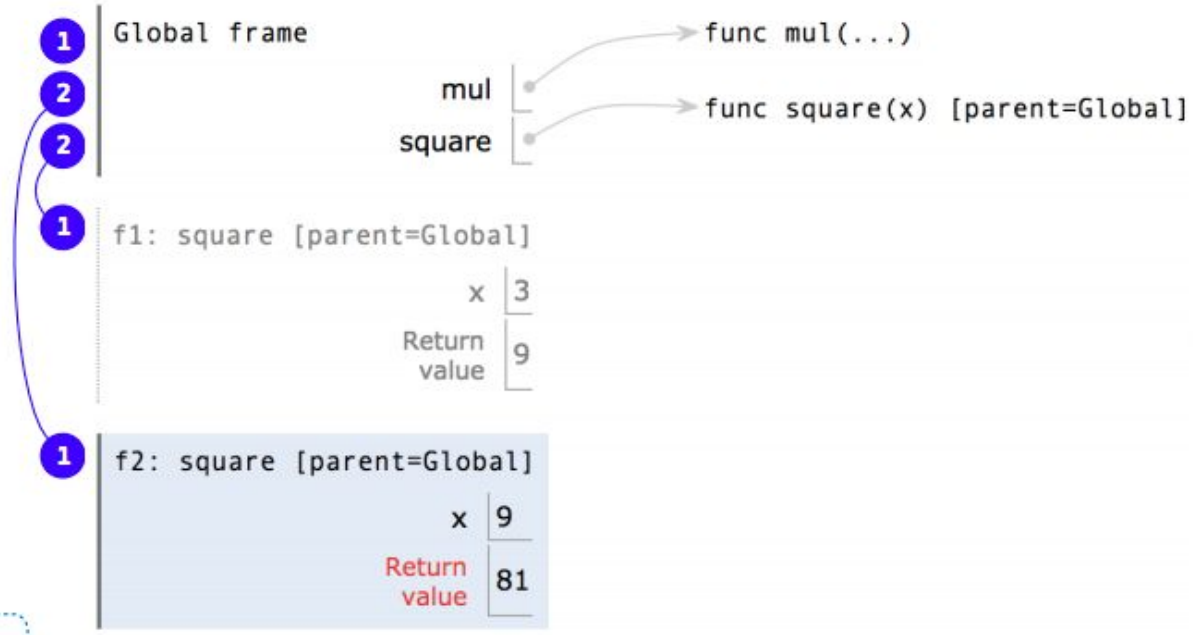


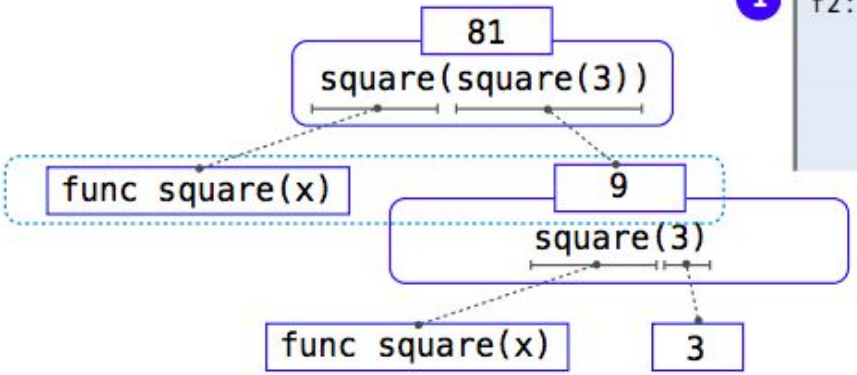
An environment is a sequence of frames.

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

Multiple Environments in one diagram

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



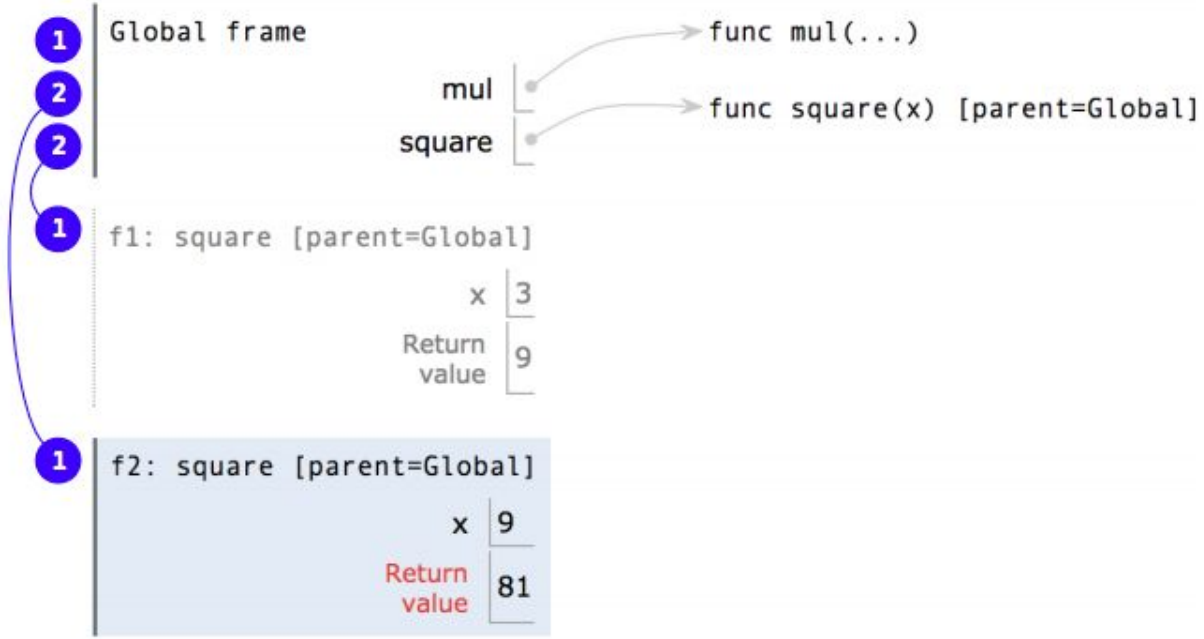


An environment is a sequence of frames.

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

Names Have No Meaning Without Environments

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



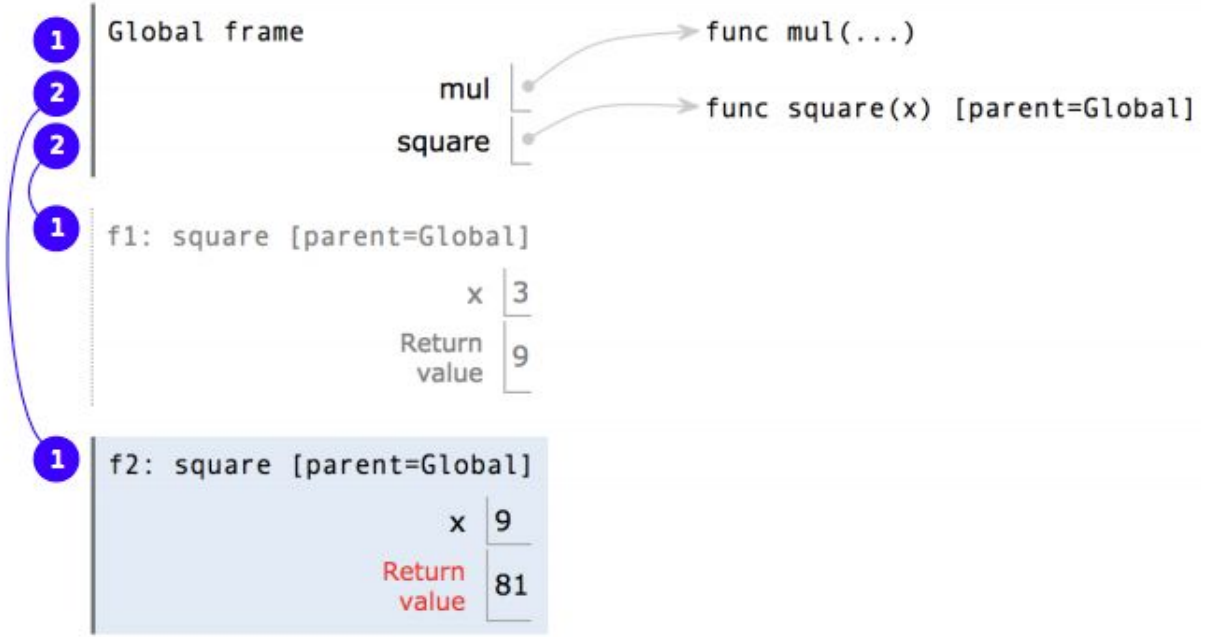
Every expression is evaluated in the context of an environment.

An environment is a sequence of frames.

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

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.

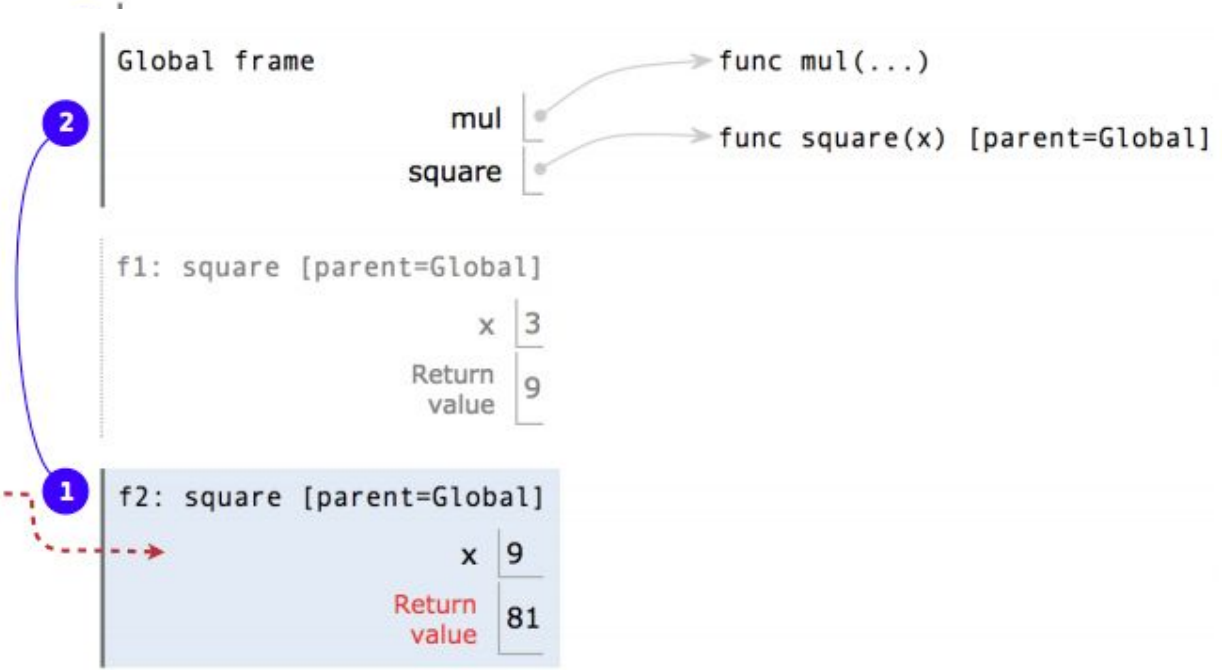
A name evaluates to the value bound to that name in the earliest frame of the current environment in which that name is found.

An environment is a sequence of frames.

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

Names Have No Meaning Without Environments

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



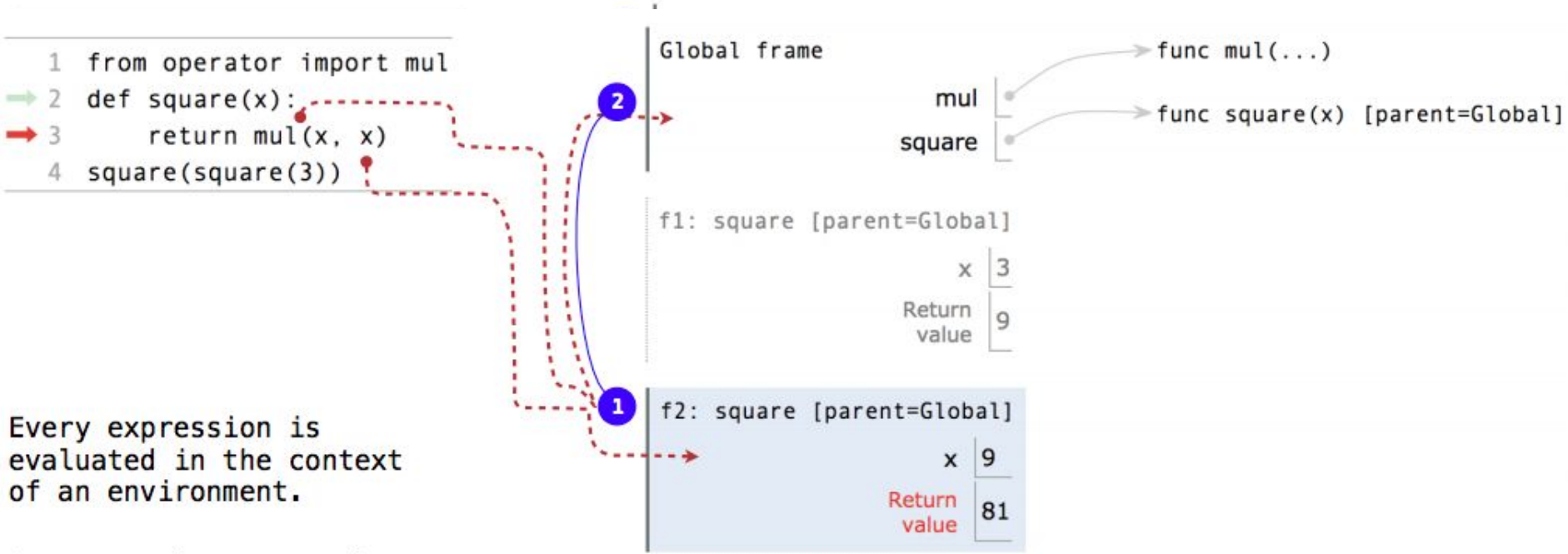
Every expression is evaluated in the context of an environment.

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

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

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.

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)
```

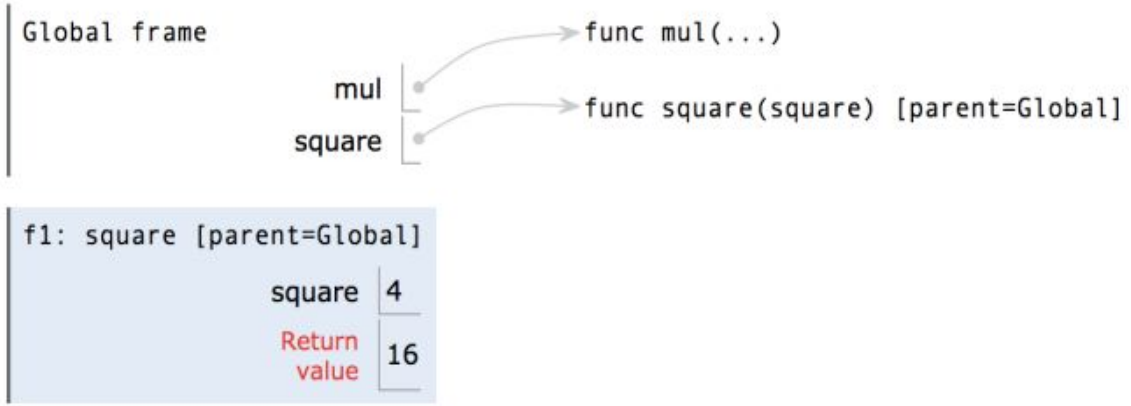
Every expression is evaluated in the context of an environment.

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

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)
```



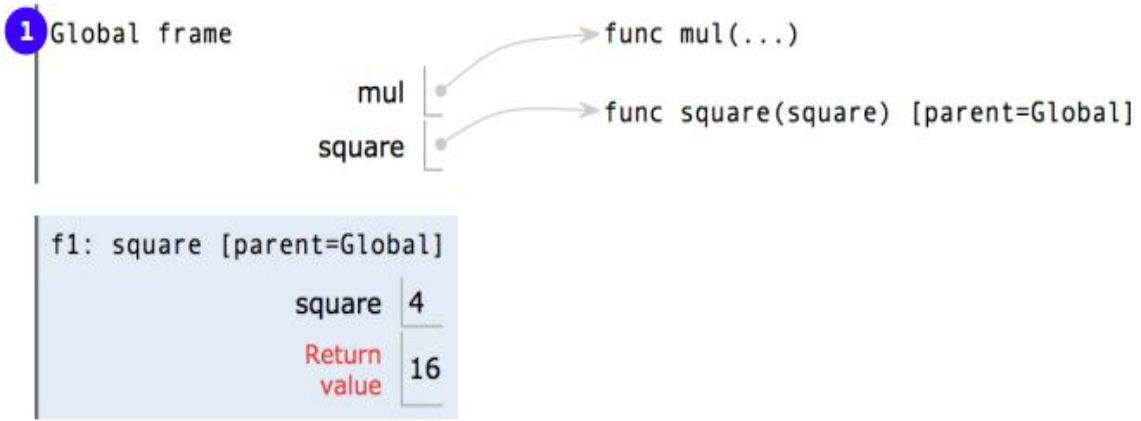
Every expression is evaluated in the context of an environment.

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

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)
```



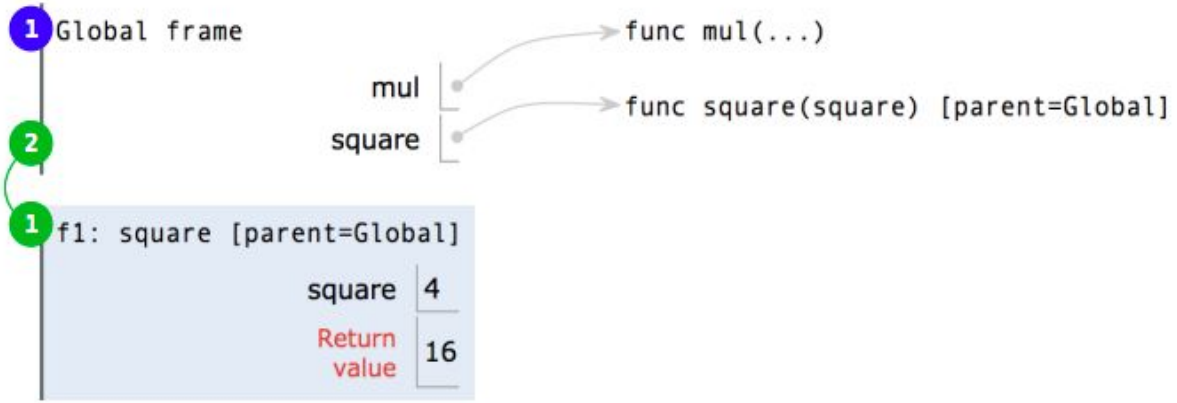
Every expression is evaluated in the context of an environment.

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

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)
```



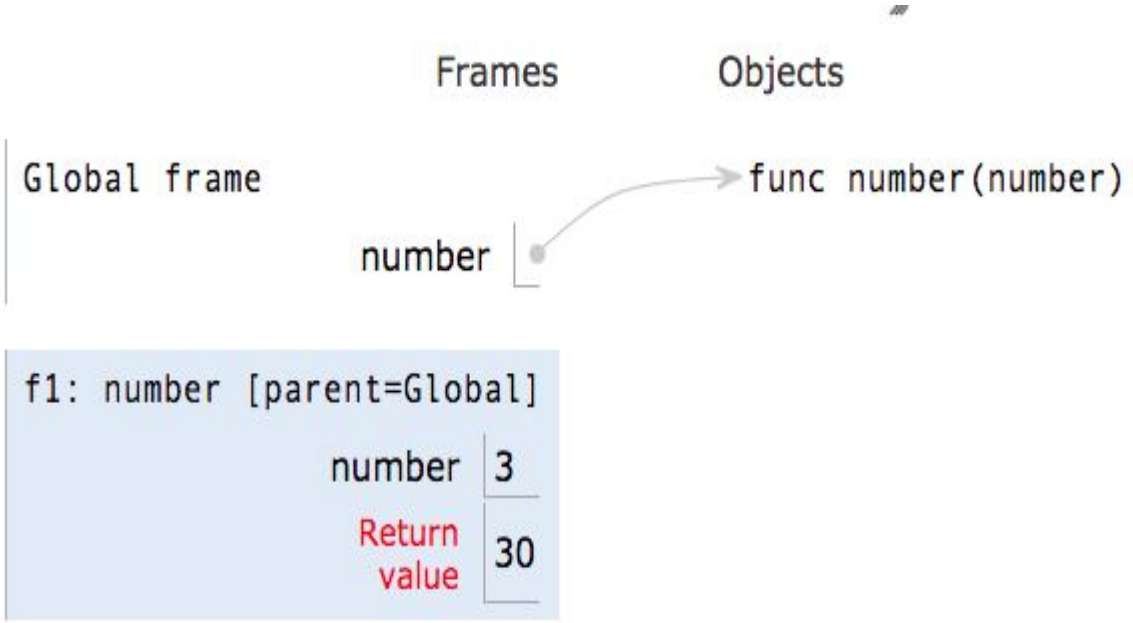
Every expression is evaluated in the context of an environment.

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

Last time: Question

```
def number (number):
    return number ** number + number
```

```
result = number(3)
print(result)
```



Question

```
def test (test):
    return test * test + test


test = 3
test(test)
```

What will be a result of the function call?

A: 12

B: 30

C: Unpredicted value

D: Error

E: Nothing will be printed

(Demo)

Quick check

Did everything make sense?

A: Yes, I got it
B: More or less, need to review it
C: Totally lost me
D: Did not pay attention.
E: Other

Programming so far

Expectation

What you want the program to do

Reality

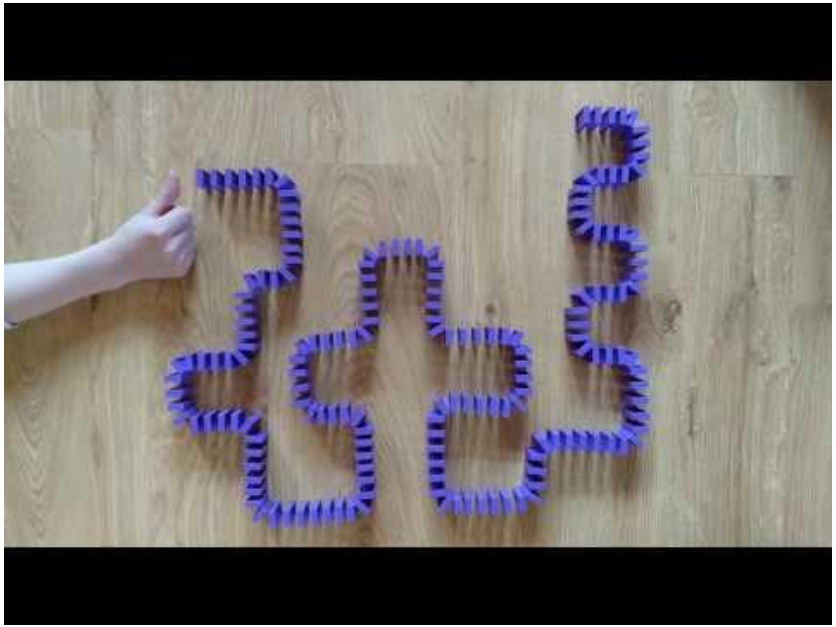
What the program actually does

Programming so far

Expectation

What you want the program to do

Reality



What the program actually does


TEST
&
CODE

Test-Driven Development

- Write the test of a function before you write the function.
  - A test will clarify the arguments, return and behavior of a function.
  - Tests can help identify tricky edge cases.

- Develop incrementally and test each piece before moving on.
  - You can't depend upon code that hasn't been tested.
  - Run your old tests again after you make new changes.

Comments and Docstrings

- **Comments**: #
- **Docstring**: “ ” ” … “ ” ”
  - occurs as the first statement in a module, function, class, or method definition.
- **Difference**:
  - These descriptions are what is returned by Python when you type `help(object)`
  - Allows you to write doctests

Doctests

(demo)

Various

Division

True Division

```
>>> 300/10
30.0
>>> 450/5
90.0
>>> 234/12
19.5
>>>
```

Integer Division

```
>>> 300//10
30
>>> 450//5
90
>>> 234//12
19
>>>
```

Question

```
>>> def multiple_return (a,b):
...     return a+b, a-b
...
>>> sum, diff = multiple_return(10, 5)
>>> print (sum, diff)
```

What will be printed?

A: 5, 15
B: 15, 5
C: Something else
D: Error

For loop. Outputs?

```
list = [0, 4, 0, 6, 5]

for i in range(len(list)):

     print(i)
```

```
list = [0, 4, 0, 6, 5]

for i in list:

     print(i)
```

```
list = [0, 4, 0, 6, 5]

for i in range(len(list)):

     print(list[i])
```

```
list = [0, 4, 0, 6, 5]

for i in list:

     print(list[i])
```

Break statement

- It terminates the current loop and resumes execution at the next statement

- The most common use for **`break`** is when some external condition is triggered requiring a hasty exit from a loop.

- The break statement can be used in both ***while*** and ***for*** loops.

What does this code do?

```
text = input("User, type a word: ")

for letter in text:

   if letter == 'h':

      break

   print ('Current Letter :', letter)

print ('the End')
```

Print: new line vs space

```
>>> for i in range(4):
...     print(i)
...
0
1
2
3
```

```
>>> for i in range(4):
...     print(i, end=" ")
...
0 1 2 3 >>>
```

Designing functions

A Guide to Designing Function

- Give each function exactly one job, but make it apply to many related situations

VS

A Guide to Designing Function

- Don’t repeat yourself (DRY). Implement a process just once, but execute it many times.

A Guide to Designing Function

- Define functions generally

```
>>> round(1.23)
1
```

```
>>> round(1.23, 1)
1.2
```

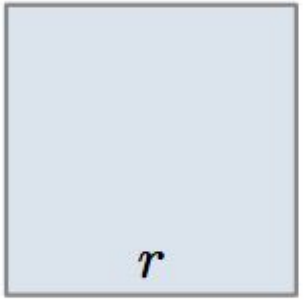
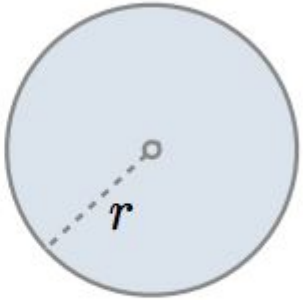
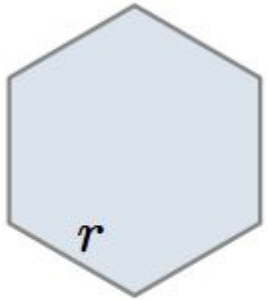
```
>>> round(1.23, 0)
1
```

```
>>> round(1.23, 5)
1.23
```

Higher - Order functions

Motivation

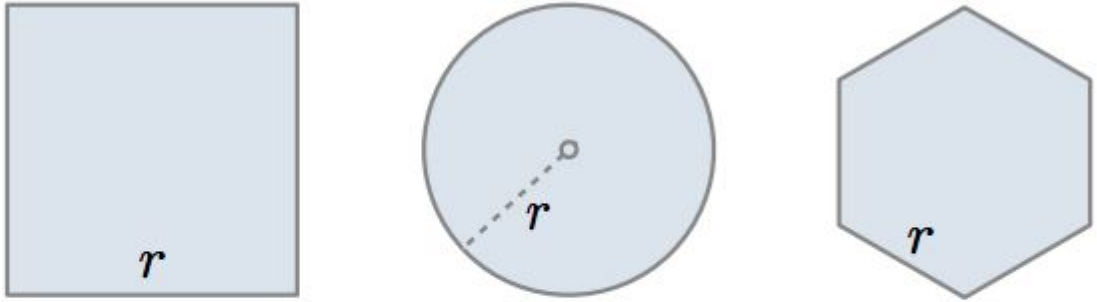
Regular geometric shapes relate length and area.

Shape:			
Area:	$1 \cdot r^2$	$\pi \cdot r^2$	$\frac{3\sqrt{3}}{2} \cdot r^2$

Motivation

Regular geometric shapes relate length and area.

Shape:



Area: $1 \cdot r^2$ $\pi \cdot r^2$ $\frac{3\sqrt{3}}{2} \cdot r^2$

Finding common structure allows for shared implementation

```
from math import pi, sqrt

def area_square(r):
    """Return the area of a square with side length R."""
    return r * r
```

```
def area_circle(r):
    """Return the area of a circle with radius R."""
    return r * r * pi
```

```
def area_hexagon(r):
    """Return the area of a regular hexagon with side length R."""
    return r * r * 3 * sqrt(3) / 2
```

Does it work properly for **all** r?

A: Yes
B: No

```
from math import pi, sqrt

def area_square(r):
    """Return the area of a square with side length R."""
    assert r > 0, "r must be positive"
    return r * r
```

```
def area_circle(r):
    """Return the area of a circle with radius R."""
    return r * r * pi
```

```
def area_hexagon(r):
    """Return the area of a regular hexagon with side length R."""
    return r * r * 3 * sqrt(3) / 2
```

Better way: Asserts

Again, and again

Repeating….:(

Generalize!

```
from math import pi, sqrt

def area_square(r):
    """Return the area of a square .."""
    return r * r

def area_circle(r):
    """Return the area of a circle..."""
    return r * r * pi

def area_hexagon(r):
    """Return the area of a hexagon
..."""
    return r * r * 3 * sqrt(3) / 2
```

```
def area(r, shape_constant):
     """Return area of a shape from length R."""
    assert r > 0, 'A length must be positive'
    return r * r * shape_constant

def area_square(r):
    return area(r, 1)

def area_circle(r):
    return area(r, pi)

def area_hexagon(r):
    return area(r, 3 * sqrt(3) / 2)
```

Textbook example

The common structure among functions may be a computational process, rather than a number.

$$\sum_{k=1}^{5} k = 1 + 2 + 3 + 4 + 5 = 15$$

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 = 225$$

$$\sum_{k=1}^{5} k^5 = 1^5 + 2^5 + 3^5 + 4^5 + 5^5 = 4425$$

Write a function to compute the first expression

$$\sum_{k=1}^{5} k = 1+2+3+4+5 \qquad\qquad = 15$$

```
def sum_naturals(n):     # use a while loop
```

Write a function to compute the second expression

$$\sum_{k=1}^{5} k^3 = 1^3 + 2^3 + 3^3 + 4^3 + 5^3 \qquad = 225$$

```
def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k*k*k, k + 1
        return total
```

Write a function to compute the second expression

$$\sum_{k=1}^{5} k^5 = 1^5 + 2^5 + 3^5 + 4^5 + 5^5 = 4425$$

```
def sum_fifths(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k**5, k + 1
        return total
```

All together

```
def sum_naturals(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k + 1
        return total
```

```
def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k**3, k + 1
        return total
```

```
def sum_fifths(n):
        total, k = 0, 1
        while k <= n:
            total,k = total + k**5, k + 1
        return total
```

All together

```
def sum_naturals(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k, k +
1
        return total
```

```
def sum_fifths(n):
        total, k = 0, 1
        while k <= n:
            total,k = total + k**5, k +
1
        return total
```

```
def sum_cubes(n):
        total, k = 0, 1
        while k <= n:
            total, k = total + k**3, k + 1
        return total
```

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total,k=total + term(k),k + 1
    return total
```

How to call it

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

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total,k = total + term(k), k + 1
    return total
```

```
result = summation (5, cube)
```

All together, again

```
def sum_naturals(n):
    return summation (n, identity)
```

```
def sum_cubes(n):
    return ________________________
```

```
def sum_fifths(n):
    return summation (n, fifth)
```

```
def cube(x):
    return x**3
def identity(x):
    return x
def fifths(x):
    return x**5
```

```
def summation(n, term):
    total, k = 0, 1
    while k <= n:
        total,k=total+term(k),k + 1
    return total
```

```
def cube(k):
    return pow(k, 3)

def summation(n, term):
    """Sum the first n terms of a sequence.

    >>> summation(5, cube)
    225
    """
    total, k = 0, 1
    while k <= n:
        total, k = total + term(k), k + 1
    return total
```

Function of a single argument (*not called "term"*)

A formal parameter that will be bound to a function

The cube function is passed as an argument value

0 + 1 + 8 + 27 + 64 + 125

The function bound to term gets called here

Check Point

```
def calc(n):
    return 2 * n - 1

def question(n, func):
    return n + func(n)



result = question(2, calc)
print(result)
```

What will be printed?

A: 3

B: 5

C: 7

D: Syntax Error

E: Runtime Error

Overall idea

- **Want**: is the ability to build **abstractions** by assigning names to common patterns and then to work in terms of the names directly.

- **Need**: to construct functions that can *accept* other functions as arguments or *return* functions as values.

- Functions that manipulate functions are called *higher-order* functions.