Environments

Announcements

Expressions

Types of expressions

An expression describes a computation and evaluates to a value

$18 + 69$

$\frac{6}{23}$

$\sin \pi$

$\log_2 1024$

2^{100}

$\boxed{f(x)}$

$\sqrt{3493161}$

$7 \bmod 2$

$\sum_{i=1}^{100} i$

$\lim_{x \to \infty} \frac{1}{x}$

$\binom{69}{18}$

$|-1869|$

(Demo)

Anatomy of a Call Expression

$$\underbrace{\texttt{add}}_{\textit{Operator}}\ (\ \underbrace{\texttt{2 + 2}}_{\textit{Operand}}\ ,\ \underbrace{\texttt{mul(3, 4)}}_{\textit{Operand}}\ )$$

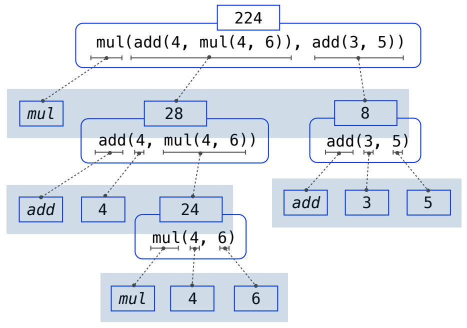
Operators and operands are also expressions

So they evaluate to values

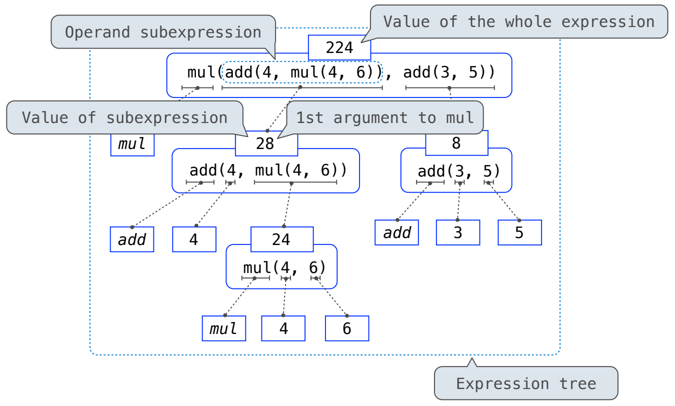
Evaluation procedure for call expressions:

1. Evaluate the operator and then the operand subexpressions
2. Apply the function that is the value of the operator
   to the arguments that are the values of the operands

Evaluating Nested Expressions



Evaluating Nested Expressions



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

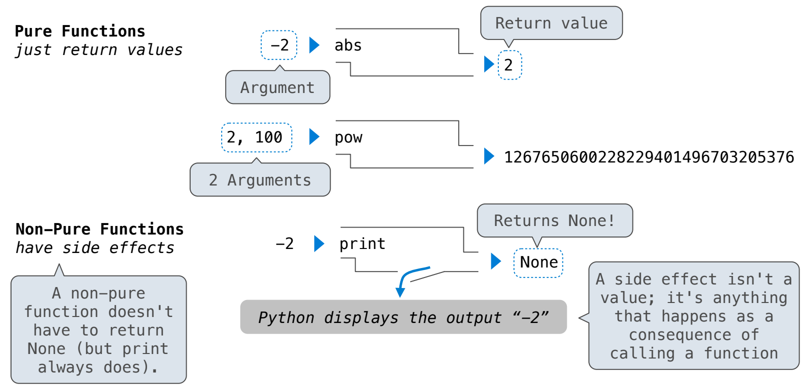
```
>>> def does_not_return_square(x):
...     x * x
...
>>> does_not_return_square(4)
>>> sixteen = does_not_return_square(4)
>>> sixteen + 4
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'NoneType' and 'int'
```

No **return**

None value is not displayed

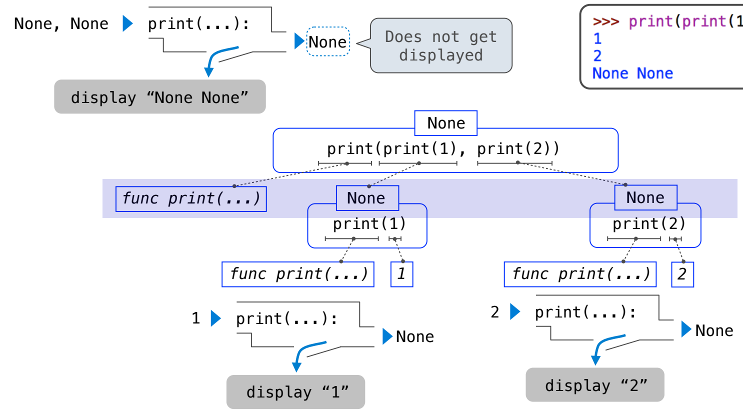
The name **sixteen** is now bound to the value **None**

Pure Functions & Non-Pure Functions


Pure Functions
just return values
-2
abs
Return value
2
Argument
2, 100
pow
1267650600228229401496703205376
2 Arguments
Non-Pure Functions
have side effects
-2
print
Returns None!
None
Python displays the output "-2"
A non-pure function doesn't have to return None (but print always does).
A side effect isn't a value; it's anything that happens as a consequence of calling a function

Nested Expressions with Print

```
>>> print(print(1), print(2))
1
2
None None
```



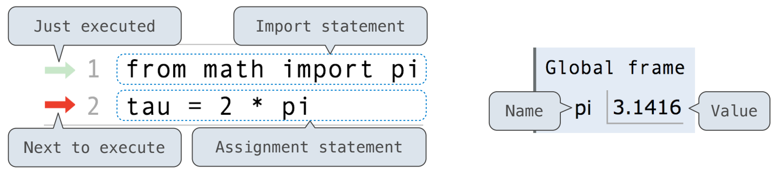
Names, Assignment, and User-Defined Functions

(Demo)

Environment Diagrams

Environment Diagrams

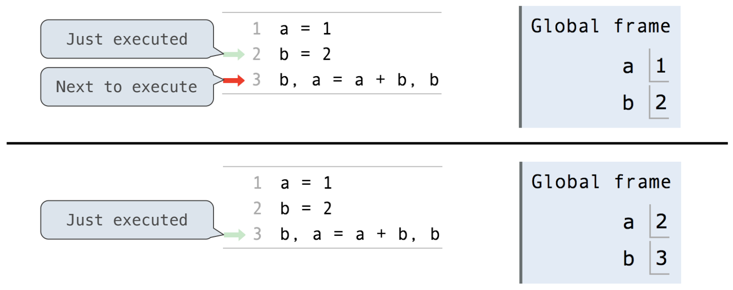
Environment diagrams visualize the interpreter’s process.


```
from math import pi
tau = 2 * pi
```

Code (left):	**Frames (right):**
Statements and expressions	Each name is bound to a value
Arrows indicate evaluation order	Within a frame, a name cannot be repeated

(Demo: tutor.cs61a.org)

Assignment Statements


```
a = 1
b = 2
b, a = a + b, b
```

Execution rule for assignment statements:

1. Evaluate all expressions to the right of = from left to right.

2. Bind all names to the left of = to those resulting values in the current frame.

Calling Functions

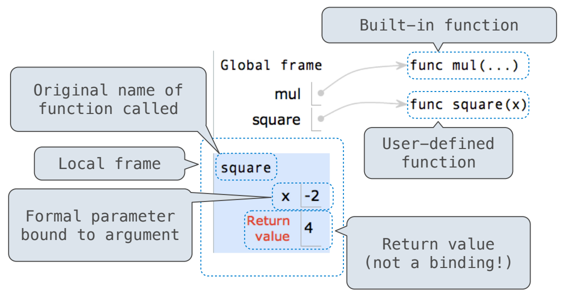
(Demo: tutor.cs61a.org)

Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

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



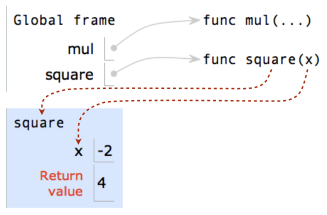
Calling User-Defined Functions

Procedure for calling/applying user-defined functions (version 1):

1. Add a local frame
2. Bind the function's formal parameters to its arguments in that frame
3. Execute the body of the function in that new environment

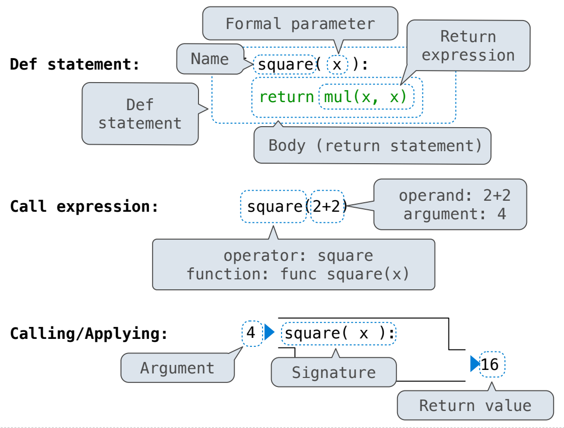
```
from operator import mul
def square(x):
    return mul(x, x)
square(-2)
```

A function’s signature has all the information needed to create a local frame



Life Cycle of a User-Defined Function

What happens?



Def statement:

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:

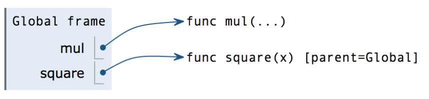
A new frame is created!

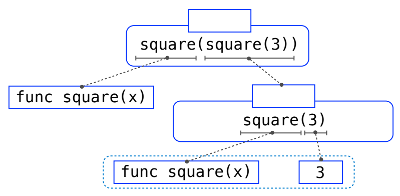
Parameters bound to arguments

Body is executed

Multiple Environments in One Diagram!

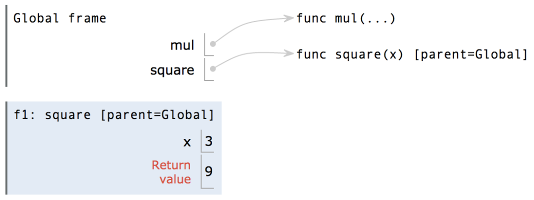
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```

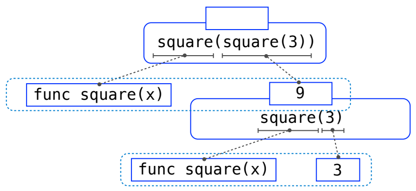




Multiple Environments in One Diagram!

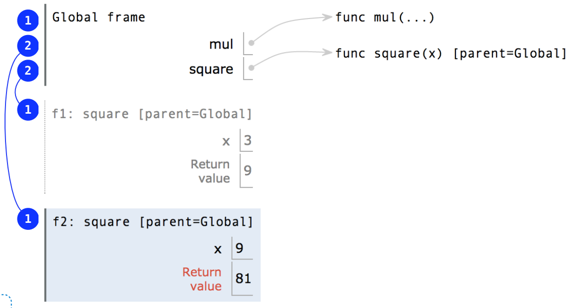
```
from operator import mul
def square(x):
    return mul(x, x)
square(square(3))
```



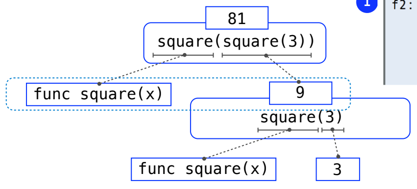


Multiple Environments in One Diagram!

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



1
2
2
1
1
Global frame
mul
square
func mul(...)
func square(x) [parent=Global]
f1: square [parent=Global]
x 3
Return value 9
f2: square [parent=Global]
x 9
Return value 81

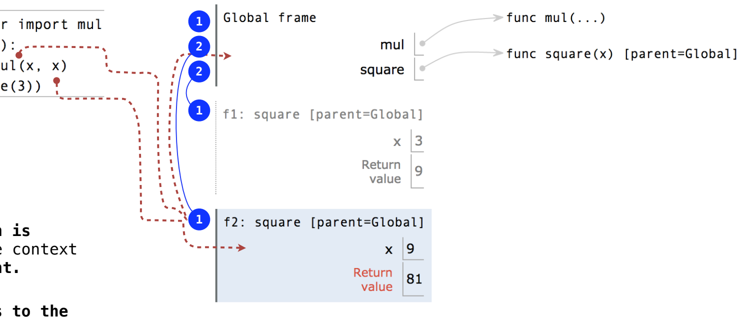


An **environment** is a sequence of frames.

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

Names Have No Meaning Without Environments

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



1
2
2
1
1
Global frame
mul
square
func mul(...)
func square(x) [parent=Global]
f1: square [parent=Global]
x 3
Return value 9
f2: square [parent=Global]
x 9
Return value 81

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 **OR**
- A local frame, then the global frame

(Demo)