Lecture #2: Functions, Expressions, Environments

- From last lecture: *Values* are data we want to manipulate and in particular,
- *Functions* are values that perform computations on values.
- *Expressions* denote computations that produce values.
- Today, we'll look at them in some detail at how functions operate on data values and how expressions denote these operations.
- As usual, although our concrete examples all involve Python, the actual concepts apply almost universally to programming languages.

Functions

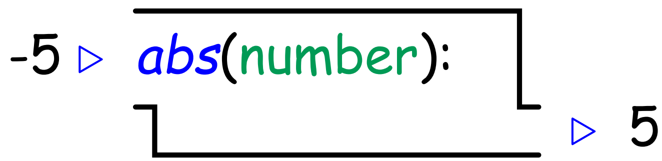
- We're going to use this notation to denote functions:

abs(number):   *add*(left, right)

- The green parenthesized lists indicate the number of *parameter values* or *inputs* the functions operate on (this information is also known as a function's *signature*).
- For our purposes, the blue name is simply a helpful comment to suggest what the function does, and the specific (green) parameter names are likewise just helpful hints.
- (Python actually maintains this *intrinsic name* and the parameter names internally, but this is not a universal feature of programming languages).

Pure Functions

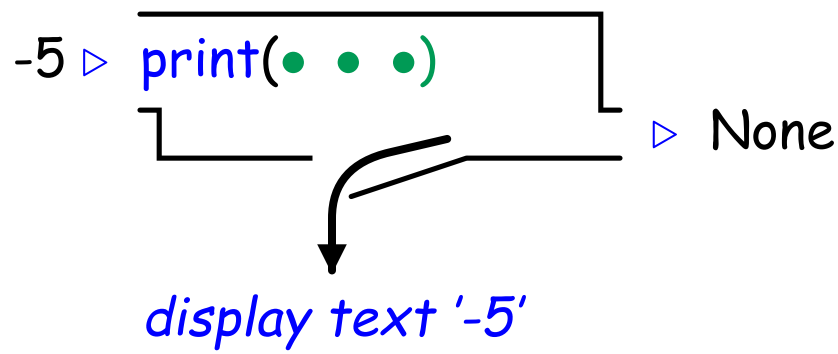
- The fundamental operation on function values is to *call* or *invoke* them, which means giving them one value for each formal parameter and having them produce the result of their computation on these values:



- These two functions are *pure:* their output depends only on their input parameters' values, and they do nothing in response to a call but compute a value.

Impure Functions

- Functions may do additional things when called besides returning a value.
- We call such things *side effects.*
- Example: the built-in print function:



- Displaying text is print's side effect. It's value, in fact, is generally useless (always the null value).

Call Expressions

- A call expression denotes the operation of calling a function.
- Consider add(2, 3):

$$\underbrace{\text{add}}_{\textit{Operator}}(\underbrace{2}_{\textit{Operand 0}},\underbrace{3}_{\textit{Operand 1}})$$

- The operator and the operands are all themselves expressions (recursion again).
- To evaluate this call expression:
  - Evaluate the operator (let's call the value C);
  - Evaluate the operands in the order they appear (let's call the values P_0 and P_1)
  - Call C (which must be a function) with parameters P_0 and P_1.
- Together with the definitions for base cases (mostly literal expressions and symbolic names), this describes how to evaluate any call.

Example: From Expression to Value

Let's evaluate the expression mul(add(2, mul(0x4, 0x6)), add(0x3, 005)).
In the following sequence, values are shown in $\boxed{\text{boxes}}$.
Everything outside a box is an expression.

- $\underbrace{\text{mul}}(\underbrace{\underbrace{\text{add}}(\underbrace{2}, \underbrace{\text{mul}(\underbrace{0x4}, \underbrace{0x6})})}, \underbrace{\text{add}(\underbrace{0x3}, \underbrace{005})})$
- $\boxed{\text{mul(left, right)}}$ (add(2, mul(0x4, 0x6)), add(0x3, 005))
- $\boxed{\text{mul(left, right)}}$ ($\boxed{\text{add(left, right)}}$ ($\boxed{2}$, $\boxed{\text{mul(left, right)}}$ ($\boxed{4}$, $\boxed{6}$)), add(0x3, 005))
- $\boxed{\text{mul(left, right)}}$ ($\boxed{\text{add(left, right)}}$ ($\boxed{2}$, $\boxed{24}$), add(0x3, 005))
- $\boxed{\text{mul(left, right)}}$ ($\boxed{26}$, add(0x3, 005))
- $\boxed{\text{mul(left, right)}}$ ($\boxed{26}$, $\boxed{\text{add(left, right)}}$ ($\boxed{3}$, $\boxed{5}$))
- $\boxed{\text{mul(left, right)}}$ ($\boxed{26}$, $\boxed{8}$)
- $\boxed{208}$.

Example: Print

What about an expression with side effects?

1. print(print(1), print(2))

2. $\boxed{\text{print}(\bullet\ \bullet\ \bullet)}$ ($\boxed{\text{print}(\bullet\ \bullet\ \bullet)}$ ($\boxed{1}$), print(2))

3. $\boxed{\text{print}(\bullet\ \bullet\ \bullet)}$ ($\boxed{\text{None}}$, print(2))

   and print '1'.

4. $\boxed{\text{print}(\bullet\ \bullet\ \bullet)}$ ($\boxed{\text{None}}$, $\boxed{\text{print}(\bullet\ \bullet\ \bullet)}$ ($\boxed{2}$))

5. $\boxed{\text{print}(\bullet\ \bullet\ \bullet)}$ ($\boxed{\text{None}}$, $\boxed{\text{None}}$))

   and print '2'.

6. $\boxed{\text{None}}$

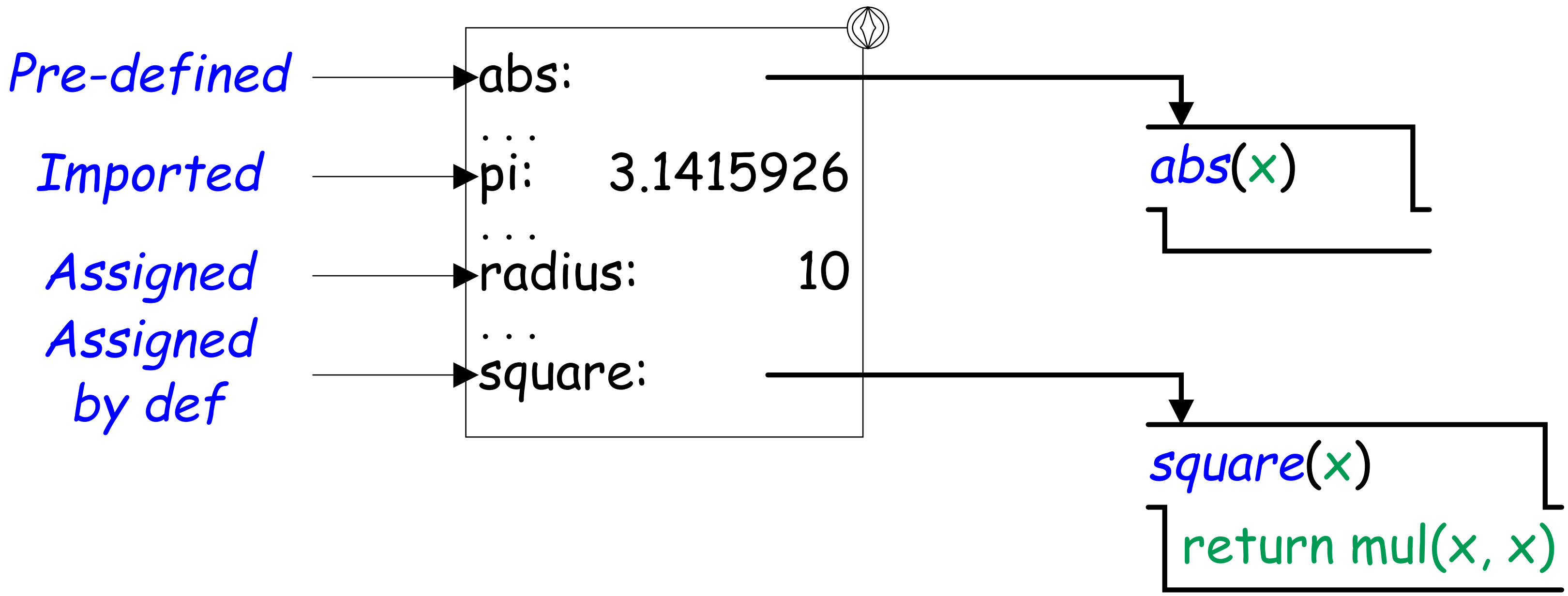
   and print 'None None'.

Names

- Evaluating expressions that are literals is easy: the literal's text gives all the information needed.
- But how did I evaluate names like add, mul, or print?
- Deduction: there must be another source of information.
- We'll use the concept of an *environment* to explain this.

Environments

- An environment is a mapping from names to values.
- We say that a name is *bound to* a value in this environment.
- In its simplest form, it consists of a single *global environment frame*:



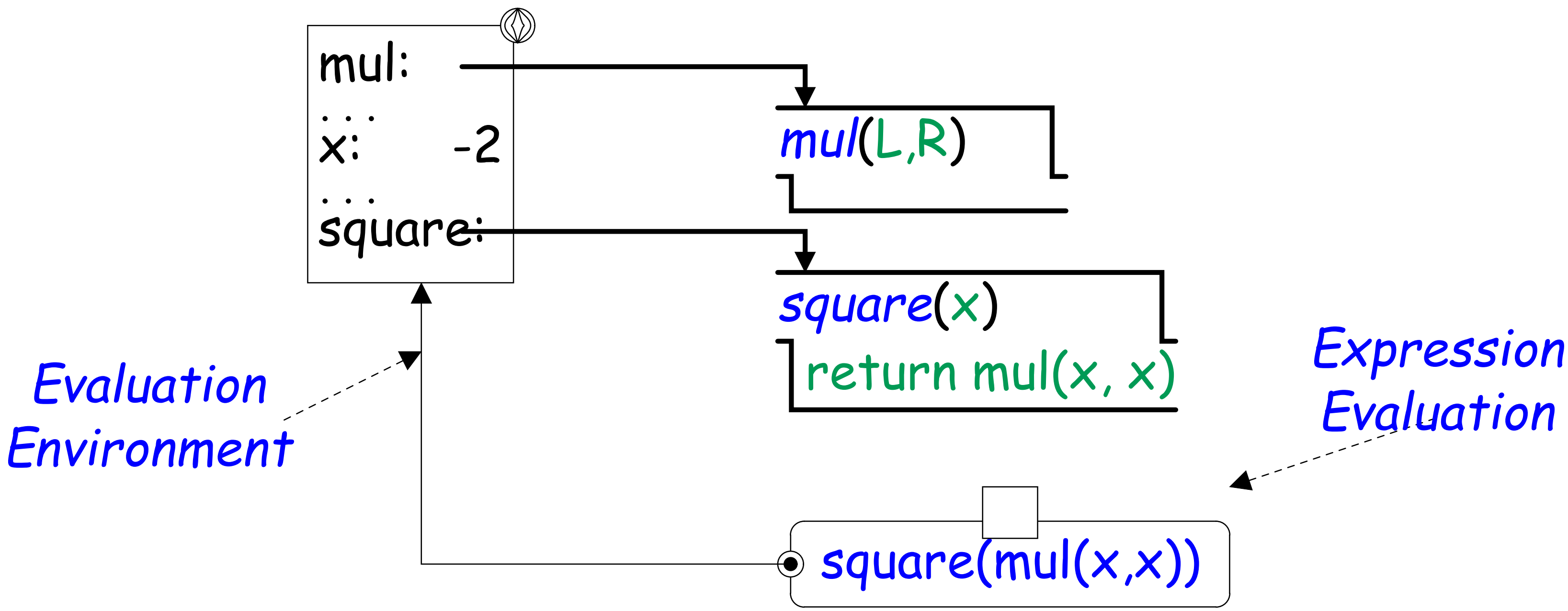
Environments and Evaluation

- Every expression is evaluated in an environment, which supplies the meanings of any names in it.
- Evaluating an expression typically involves first evaluating its subexpressions (the operators and operands of calls, the operands of conventional expressions such as x*(y+z), ...).
- These subexpressions are evaluated in the same environment as the expression that contains them.
- Once their subexpressions (operator + operands) are evaluated, calls to user-defined functions must evaluate the expressions and statements from the definition of those functions.

Evaluating User-Defined Function Calls

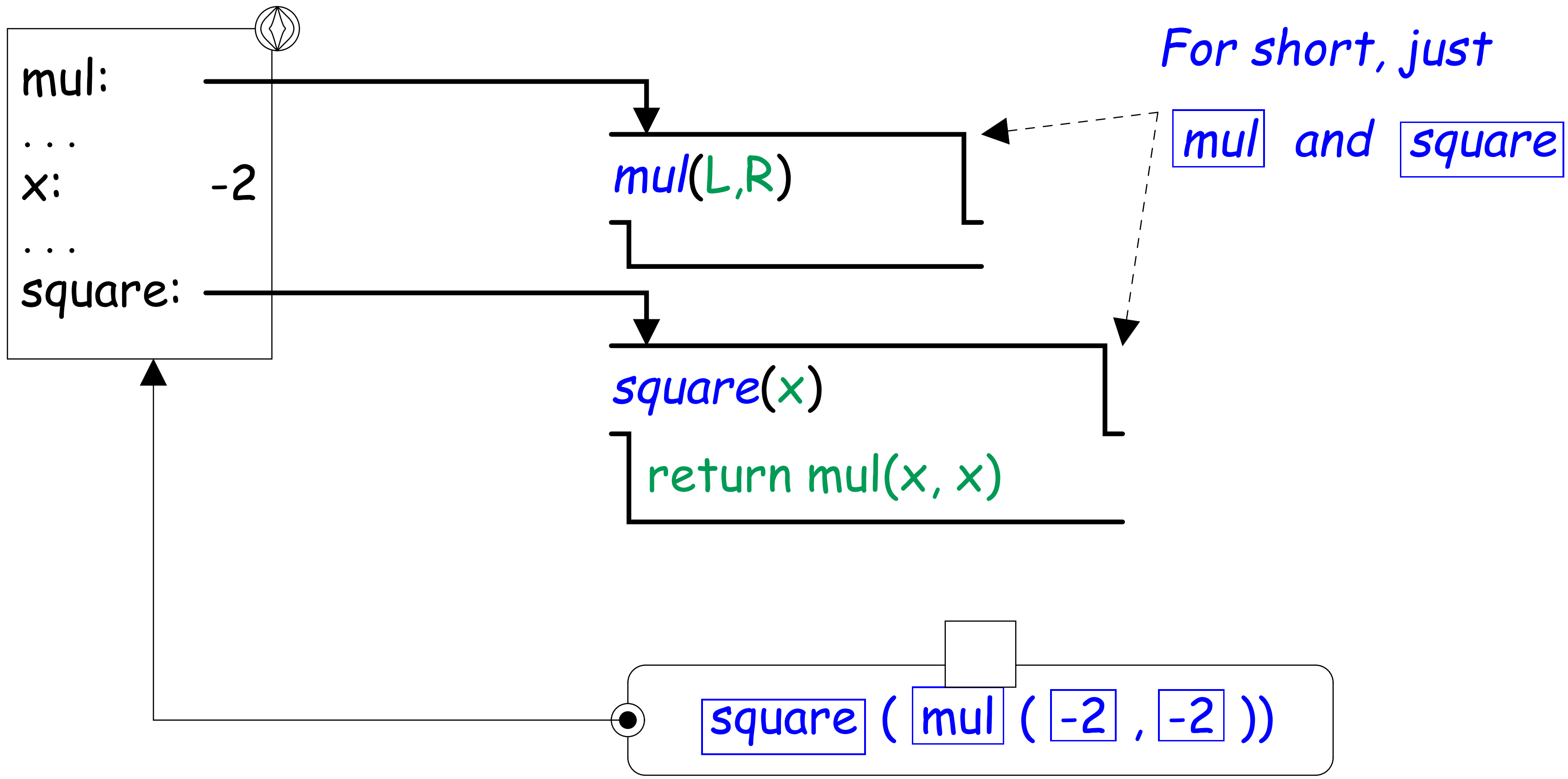
- Consider the expression square(mul(x, x)) after executing

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



Evaluating User-Defined Function Calls (II)

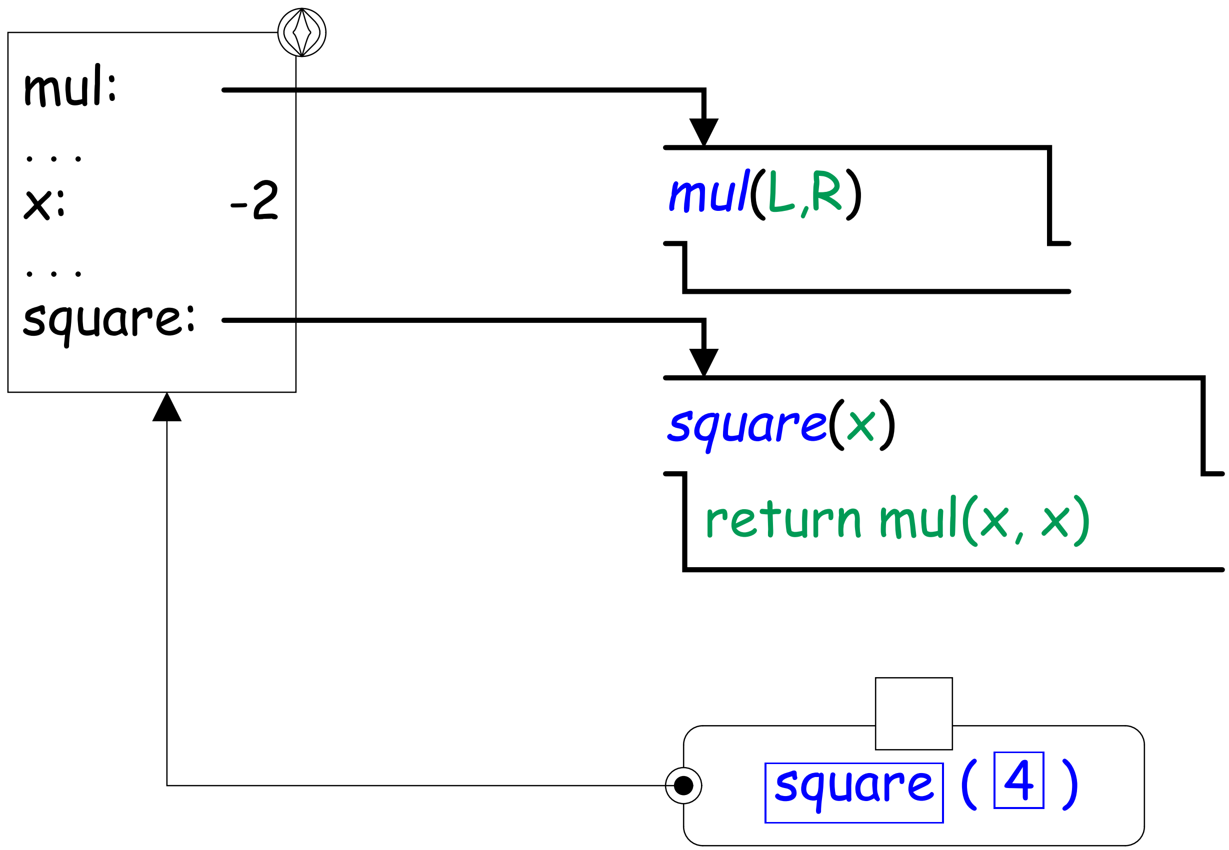
- First evaluate the subexpressions of square(mul(x, x)) in the global environment:



- Evaluating subexpressions x, mul, and square takes values from the expression's environment.

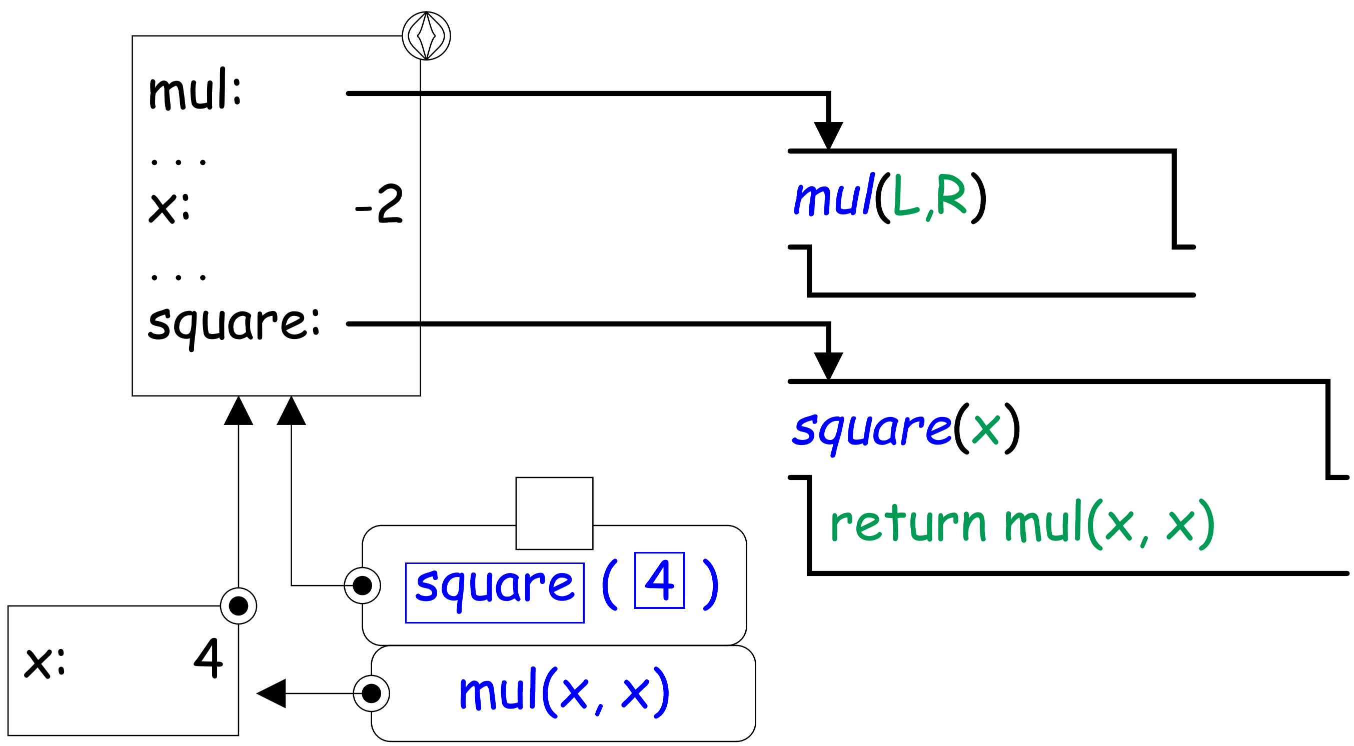
Evaluating User-Defined Functions Calls (III)

- Then perform the primitive multiply function:



Evaluating User-Defined Functions Calls (IV)

- To explain parameter to user-defined square function, extend environment with a *local environment frame*, attached to the frame in which square was defined (the global one in this case), and giving x the operand value.
- Now replace original call with evaluating body of square in the new local environment.



Evaluating User-Defined Functions Calls (V)

- When we evaluate mul(x, x) in this new environment, we get the same value as before for mul, but the local value for x.