Lecture #2: Functions, Expressions, Environments

Announcements

- HW 1 available. Due next Thursday.
- Piazza post @30 contains an index of useful Piazza threads and Zoom links (such as orientation links).
- Come to the weekly exam prep sections, Fridays 3:10-4:30 (right after Friday lecture). This first week: how exams are structured, studying advice, and our tips for succeeding in the course. Next week we will dive into solving exam level problems.
- DSP students should submit their accommodation letters ASAP.

From Last Time

- 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 in some detail at how functions operate on data values and how expressions denote these operations.
- Why start with functions? Ultimately, function calls do all the work of computation in Python, together with assignments.
- Many constructs in Python that one may also think of as basic are actually "*syntactic sugar*" for function calls. Example:

  ```
  (3 + 7 + 10) * (1000 - 8) / 992 - 17
  ```

  is equivalent (ignoring some necessary imports) to

  ```
  sub(truediv(mul(add(add(3, 7), 10), sub(1000, 8)), 992), 17)
  ```

- As usual, although our concrete examples all involve Python, the actual concepts apply almost universally to programming languages.

Functions as Values

- In grade school, we tend to think of mathematics as having to do with numbers—values are numeric.
- But in mathematics, there are many other kinds of values and standard operations on them.
- For example, the derivative ($\frac{\mathrm{d}}{\mathrm{dx}}$) and integral ($\int$) operators operate on functions to produce new ones.
- And there are sets of functions and sequences of functions as well.
- Originally, functions were not typically treated as full-fledged (or "first class") values in programming languages, but this has changed over the years.
- Python's functions *are* first-class values, and may be assigned to variables, passed to and returned from functions, and stored in data structures.

将函数当作其他数据类型(例如整数,字符串)一样的一等公民.

①函数可以赋值给变量

②函数可以作为参数传递给其他函数

③函数可以从其他函数中返回

④函数可以存储在数据结构中.

Functions Values

- For this lecture, we're going to use the Python tutor's notation for function values:

  func abs(number),   func add(left, right)

- The above are *primitive* (or *native*) function values, provided by Python.
- New functions result from evaluating *function definitions* such as

```
def saxb(a, x, b):    # Header: Name and formal parameters
    return a * x + b  # Body: Computation performed by function
```

  which creates a function value we'll write in this lecture as

  func saxb(a, x, b): return a * x + b

- 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*).

Functions (II)

- For intrinsic functions and those created by `def`, Python actually maintains an *intrinsic name* (such as the saxb in "func saxb(a, x, b).")
- But often in mathematics, functions are *anonymous,* as in "the function that takes three real number, a, x, and b and yields $ax + b$."
- A traditional mathematical notation for such anonymous functions looks like this:

$$\lambda\, a, x, b.\, ax + b$$

  (that's the Greek letter lambda).
- We'll write these values as

  func λ(a, x, b)

  as the Python tutor does, or, when emphasizing the body, as

  func λ(a, x, b): a * x + b
- These anonymous functions are created by Python *lambda expressions* such as:

```
lambda a, x, b: a * x + b
```

Functions as abstractions

sum_squares is defined in terms of the function square, but only relies on the relationship that square defines between its input arguments and output values

The detail of how the square is computed can be suppressed, as far as sum_squares is concerned, square isn't a particular function body, but rather an abstraction of a function ⇒ functional abstraction.

At this level of abstraction, any function that computes the square is equally good.

Aspects of a functional abstraction

To master the use of functional abstraction, it is useful to consider three core attributes

1. domain: the set of arguments it can take
2. range: the set of values it can return
3. intent: the relationship it computes between input and output

Pure Functions

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

-5 → func *abs*(number) → 5

28 ↘
func *add*(left, right) → 42
14 ↗

- 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.

properties of pure function:
① the function returns values are identical for identical arguments
② the function has no side effects

Impure Functions

- Functions may do additional things when called besides returning a value.
- We call such things *side effects.* make some changes to the state of the interpreter or computer
- Example: the built-in print function:

-5 ⟶ func print(*vals) ⟶ None

Side Effect: display '-5'

- Displaying text is print's side effect. Its value, in fact, is generally useless (it always returns the special value None).
- (The notation *vals is Python's way of designating an arbitrary number (0 or more) of parameters.)

Other Kinds of Impurity

- Most side-effects involve changing the value of some variable.
- Example: the function random.randint:

```
>>> from random import randint
>>> randint(0, 100)    # Random number in 0--100.
13
>>> randint(0, 100)    # Different result: Something must have changed!
55
```

- You can deduce from this behavior that there must be some variable hidden away somewhere that changes every time `randint` is called.
- In fact, you can think of printing from the last slide as a change to another invisible variable (containing the "stuff that's been printed").
- We use the hidden-variable idea to describe side effects when doing formal mathematical descriptions of programming languages.

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 resulting value C);
  - Evaluate the operands in the order they appear (let's call the resulting 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 (I)

Let's evaluate the following expression (`0x` and `0o` are Python's way of saying "base 16" and "base 8." I've used them to contrast expressions and values.):

mul(add(2, mul(0x10, 0o10)), add(0x3, 5))

In the following sequence, values (as opposed to expressions) are shown in $\boxed{\text{boxes}}$ to indicate that they need no further evaluation.

mul(add(2, mul(0x10, 0o10)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$(add(2, mul(0x10, 0o10)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$($\boxed{\text{func add(x, y)}}$(2, mul(0x10, 0o10)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$($\boxed{\text{func add(x, y)}}$($\boxed{2}$, mul(0x10, 0o10)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$($\boxed{\text{func add(x, y)}}$($\boxed{2}$, $\boxed{\text{func mul(x, y)}}$(0x10, 0o10)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$($\boxed{\text{func add(x, y)}}$($\boxed{2}$, $\boxed{\text{func mul(x, y)}}$($\boxed{16}$, 0o10)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$($\boxed{\text{func add(x, y)}}$($\boxed{2}$, $\boxed{\text{func mul(x, y)}}$($\boxed{16}$, $\boxed{8}$)), add(0x3, 5))

$\boxed{\text{func mul(x, y)}}$($\boxed{\text{func add(x, y)}}$($\boxed{2}$, $\boxed{128}$), add(0x3, 5))

Example: From Expression to Value (II)

```
[func mul(x, y)]([func add(x, y)]([2], [128]), add(0x3, 5))
[func mul(x, y)]([130], add(0x3, 5))
[func mul(x, y)]([130], [func add(x, y)](0x3, 5))
[func mul(x, y)]([130], [func add(x, y)]([3], 5))
[func mul(x, y)]([130], [func add(x, y)]([3], [5]))
[func mul(x, y)]([130], [8])
[1040]
```

Example: Print

What about an expression with side effects? (Skipping some steps for brevity)

1. print(print(1), print(2))
2. $\boxed{\text{func print(*vals)}}$ ($\boxed{\text{func print(*vals)}}$ ($\boxed{1}$), print(2))
3. $\boxed{\text{func print(*vals)}}$ ($\boxed{\text{None}}$, print(2))
   and print '1'.
4. $\boxed{\text{func print(*vals)}}$ ($\boxed{\text{None}}$, $\boxed{\text{func print(*vals)}}$ ($\boxed{2}$))
5. $\boxed{\text{func print(*vals)}}$ ($\boxed{\text{None}}$, $\boxed{\text{None}}$))
   and print '2'.
6. $\boxed{\text{None}}$
   and print 'None None'.

```
>>> print(print(1), print(2))
1
2
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 first try a simple approach to describing how to handle the evaluation of names: *substitution* of values for names.
- This won't cover all the cases, however, and so we'll introduce the concept of an *environment*.

Substitution

- Python provides several ways to define names: *assignments* of values to names, *function definitions*, and *parameter passing* to functions.
- Let's try to explain the effect of

```
x = 3
y = x * 2
z = y ** x
```

  by treating each assignment (=) as a *definition*.

- Thus, we get

```
x = 3          x = 3          x = 3          x = 3
y = x * 2      y = 3 * 2      y = 6          y = 6
z = y ** x     z = y ** 3     z = 6 ** 3     z = 216
```

- That is, we *replace names by their definitions (values).*

Parameter Substitution and Functions

- Now consider a simple function definition:

```
def compute(x, y):
    return (x * y) ** x
compute(3, 2)
```

- A **def** statement is sort of like an assignment, but specialized to functional values.
- The **def** statement above defines compute to be "the function of x and y that returns $(xy)^x$," or as we have written it before:

  $\boxed{\text{func compute(x, y): return (x * y) ** x}}$

- So evaluation of `compute(3, 2)`, as described previously, eventually gives us

  $\boxed{\text{func compute(x, y): return (x * y) ** x}}(\boxed{3}, \boxed{2})$

- Now what? How do these calls on user-defined functions work once the operands are evaluated?

Substitution and Formal Parameters

- Evaluating a function call such as

  $\boxed{\text{func compute(x, y): return (x * y) ** x}}(\boxed{3}, \boxed{2})$

  from the last slide is like a *simultaneous assignment* to or substitution for x and y—the *formal parameters* of `compute`.

- That is, we replace the whole expression with

  return ($\boxed{3}$ * $\boxed{2}$) ** $\boxed{3}$

  and (eventually), just $\boxed{216}$.

- (Let's just ignore the pesky **return**: it's basically Python syntax to indicate which value the function produces.)

Getting Fancy

- What about this?

```
def incr(n):
    def f(x):
        return n + x
    return f

incr(5)(6)
```

- The incr function returns a function (f), not a number. We then call this function on 6.
- What happens?

Answer (Part I)

- First, deal with incr:

```
def incr(n):
    def f(x):
        return n + x
    return f
incr(5)(6)
```

which first defines `incr` to be the value:

$\boxed{\text{func incr(n): def f...return f}}$

- So when we evaluate `incr(5)`, we end up calling

$\boxed{\text{func incr(n): def f...return f}}(\boxed{5})$

and substitution gives

```
def f(x):
    return 5 + x
return f
```

- That gives us a value for `incr(5)` of

$\boxed{\text{func f(x): return } \boxed{5} + x}$

Answer (Part II)

- So now (after evaluating the argument 6), we evaluate

  $\boxed{\text{func } f(x)\text{: return } \boxed{5} + x}(\boxed{6})$

- resulting in the evaluation of

  $\boxed{5} + \boxed{6}$

- Finally giving $\boxed{11}$

Complication

- What do we get out of

```
def hmmmm(x):
    def f(x):
        return x
    return f
hmmmm(5)(6)
```

- Does this give us 5 or 6?
- This boils down to whether `hmmmm(5)` gives

  func f(x): def f(x): return 5

  or

  func f(x): def f(x): return x

- We opt for the second one.

Free vs. Bound, Hiding

- Again, given

```
def hmmmm(x):
    def f(x):
        return x
    return f
hmmmm(5)(6)
```

Local names: one detail of a function's implementation that shouldn't affect the function's behavior is the implementer's choice of names for the function's formal parameters.

- The inner definition (def f) "protects" the `x` in **return** `x` from substitution when calling `hmmmm`.
- Formally, we say that the substitution caused by `hmmmm(5)` replaces *free occurrences* of `x` in the body of `hmmmm` only.
- A free occurrence of a name in a statement or expression is one that is not defined (*bound*) within that statement.
- Since the `x` in the **return** statement is defined to be the formal parameter of `f`, it is bound, and therefore not replaced.
- Another way to say it is that the inner definition of `x` (in `f`) *hides* the outer one (in `hmmmm`) within the body of `f`.

Trouble

- Alas, even besides these complications, the substitution approach doesn't entirely work.
- Example:

```
x = 4
x = 8
print(x)
```

- If we just substitute for the first x as before:

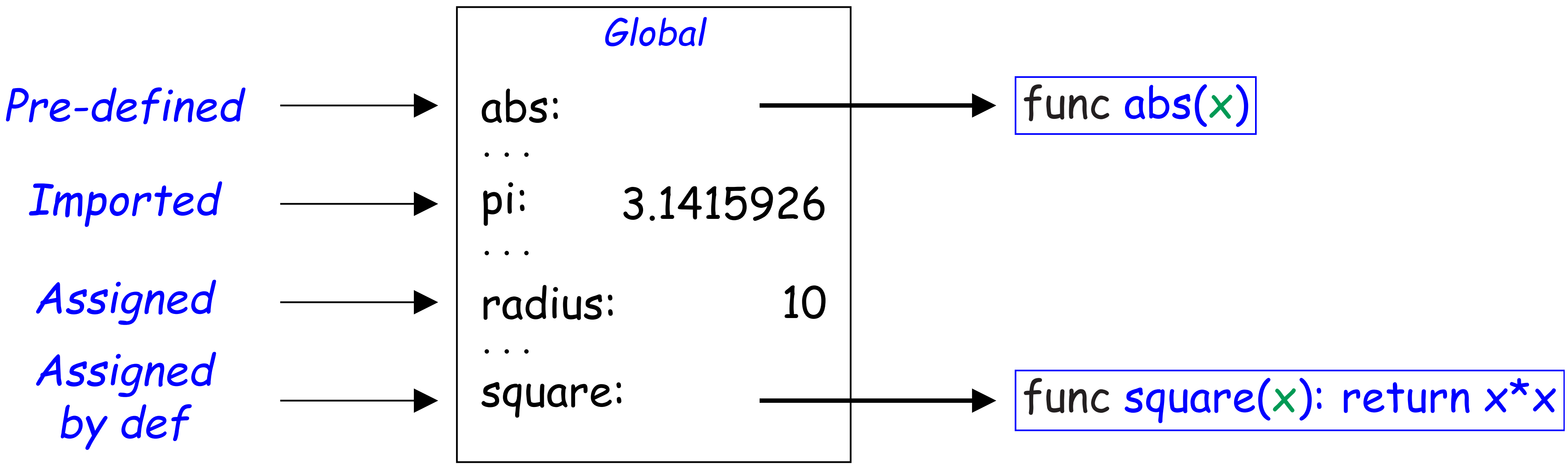
```
x = 4
x = 8    # or even worse: 4 = 8
print(4)
```

- ...we get a wrong result (4 instead of 8).
- After one substitution, x isn't around any more to substitute for.
- We need a more comprehensive answer to the question of how function calls work, one that takes multiple assignments to variables into account.

New Explanation: Environments

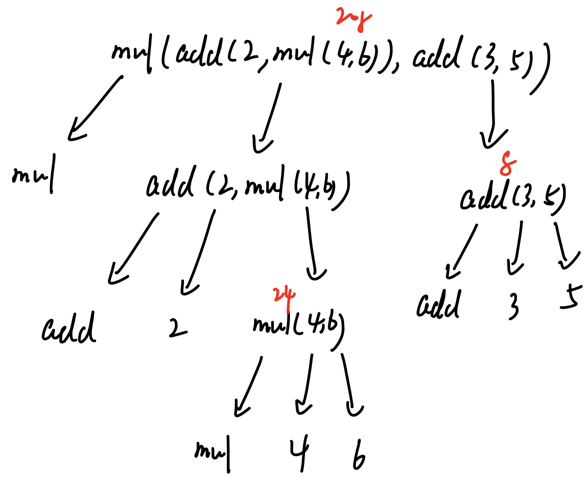
The possibility of binding names to values and later retrieving those values by names means that the interpreter must maintain some sort of memory that keeps track of the names, values and bindings ⇒ environment.

- 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*:




```
from math import pi
radius = 10
def square(x): return x * x
```

To evaluate a call expression, Python will do the following:
① evaluate the operator and operand subexpressions
② apply the function that is the value of the operator subexpression to the arguments that are the values of the operand subexpressions



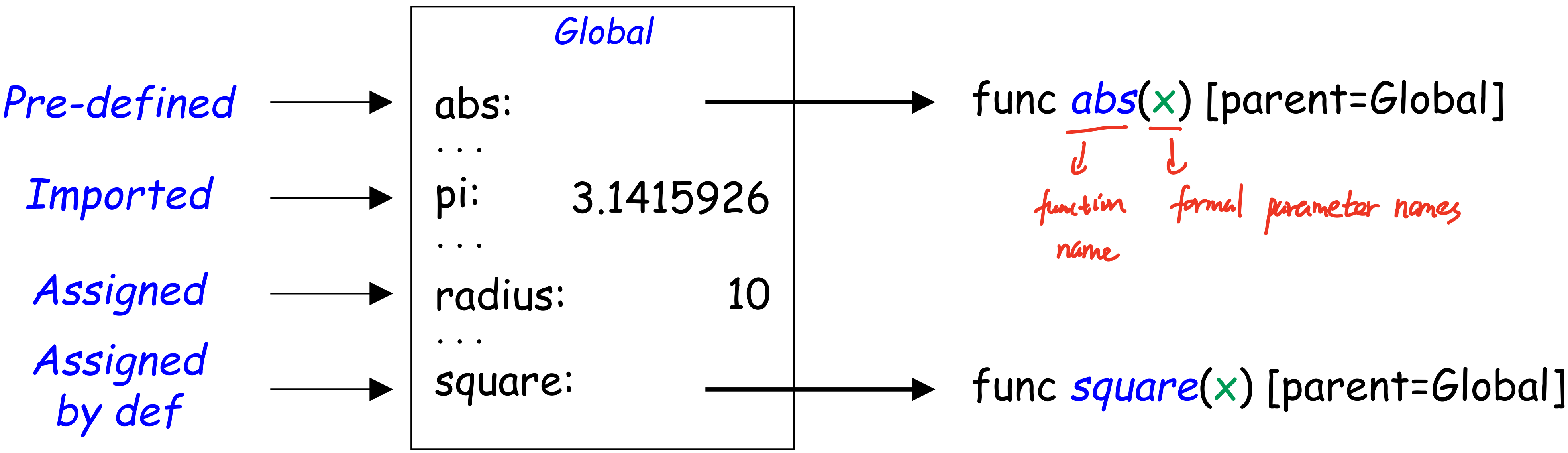
Evaluation procedure is recursive in nature

Expression tree

Environments: an environment in which an expression is evaluated, consists of a sequence of frames, depicted as boxes. Each frame contains bindings, each of which associates a name with its corresponding value. There is a single global frame. Assignment and import statements add entries to the first frame of the current environment.

Diagrams in Python Tutor

- You'll be using the Python Tutor from time to time, which uses a somewhat different notation for function values. Might as well get used to it (we'll explain the "parent=" stuff in a later lecture):




```python
from math import pi
radius = 10
def square(x): return x**2
```

Built-in functions such as mul don't have formal parameter names, so ... is used instead.

The name of the function is repeated twice, one in the frame and again as part of the function itself. 一次在环境帧中 (绑定名称 bound name)
另一次作为函数本身的一部分 (内在名称 intrinsic name)

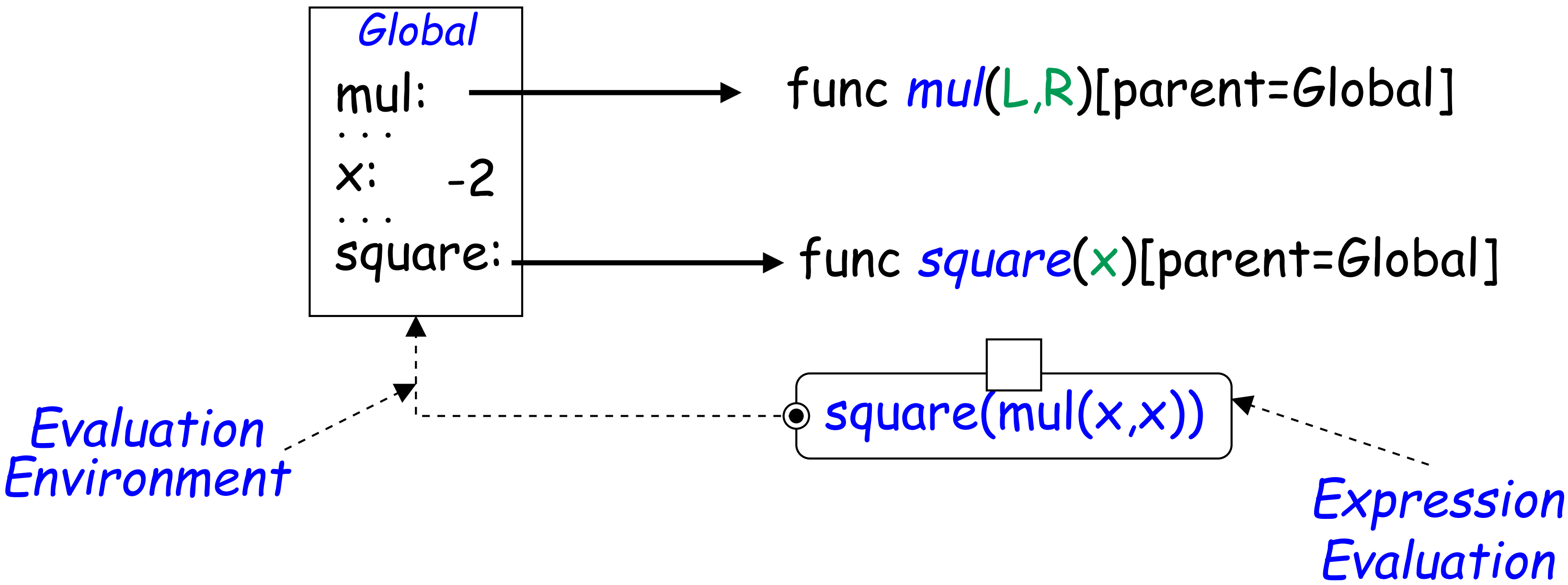
Environments and Evaluation

- 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 definitions of those functions.

Evaluating User-Defined Function Calls

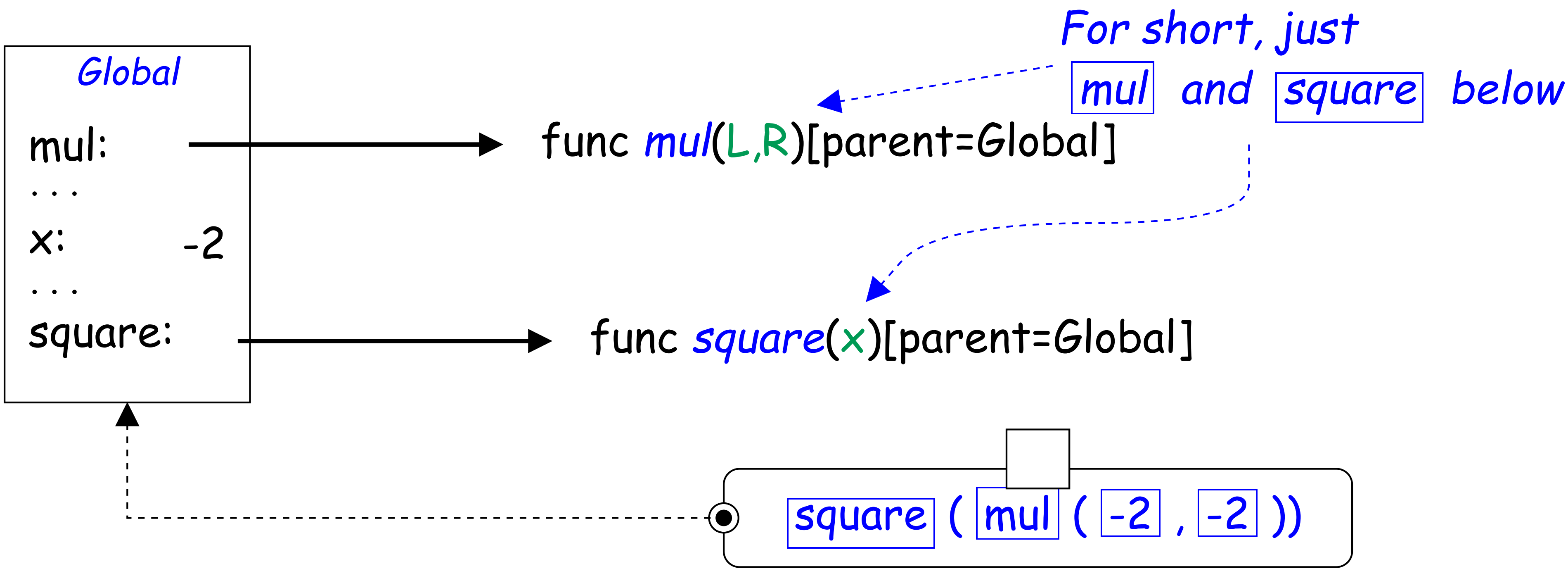
- Consider the expression square(mul(x, x)) after executing

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



Evaluating User-Defined Function Calls (II)

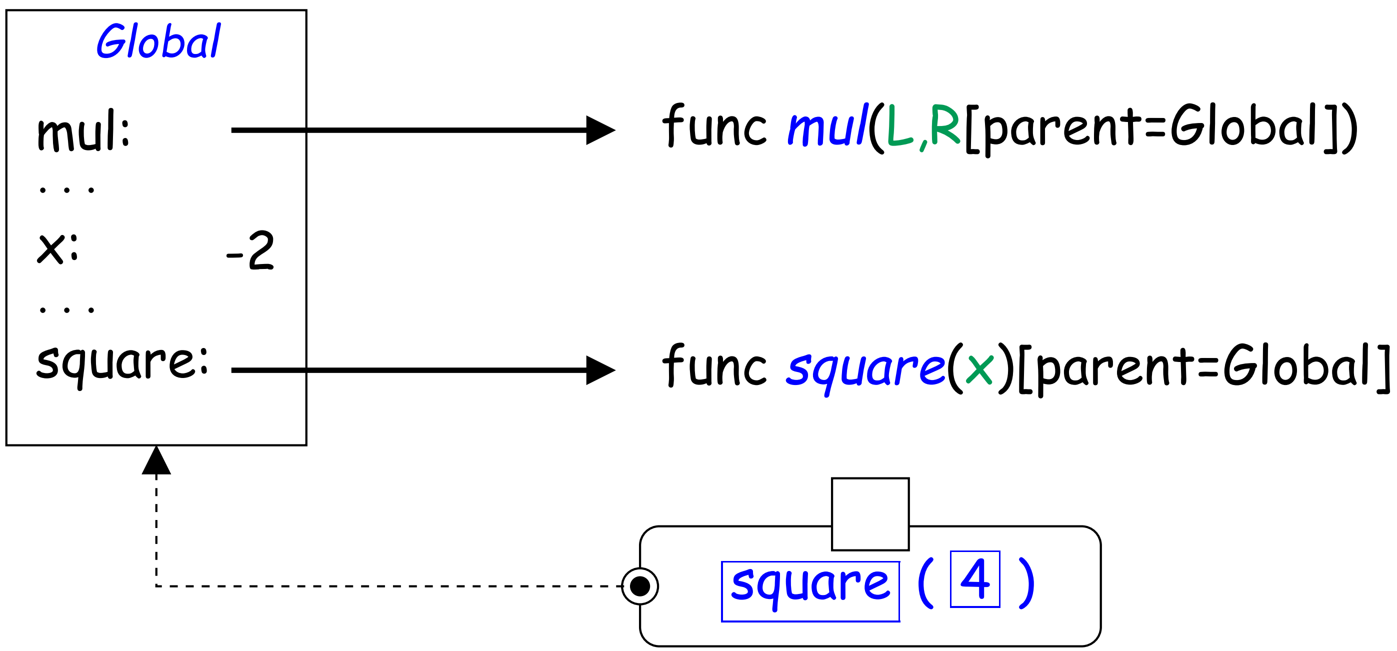
- First evaluate the subexpressions of square(mul(x, x)) in the global environment:



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

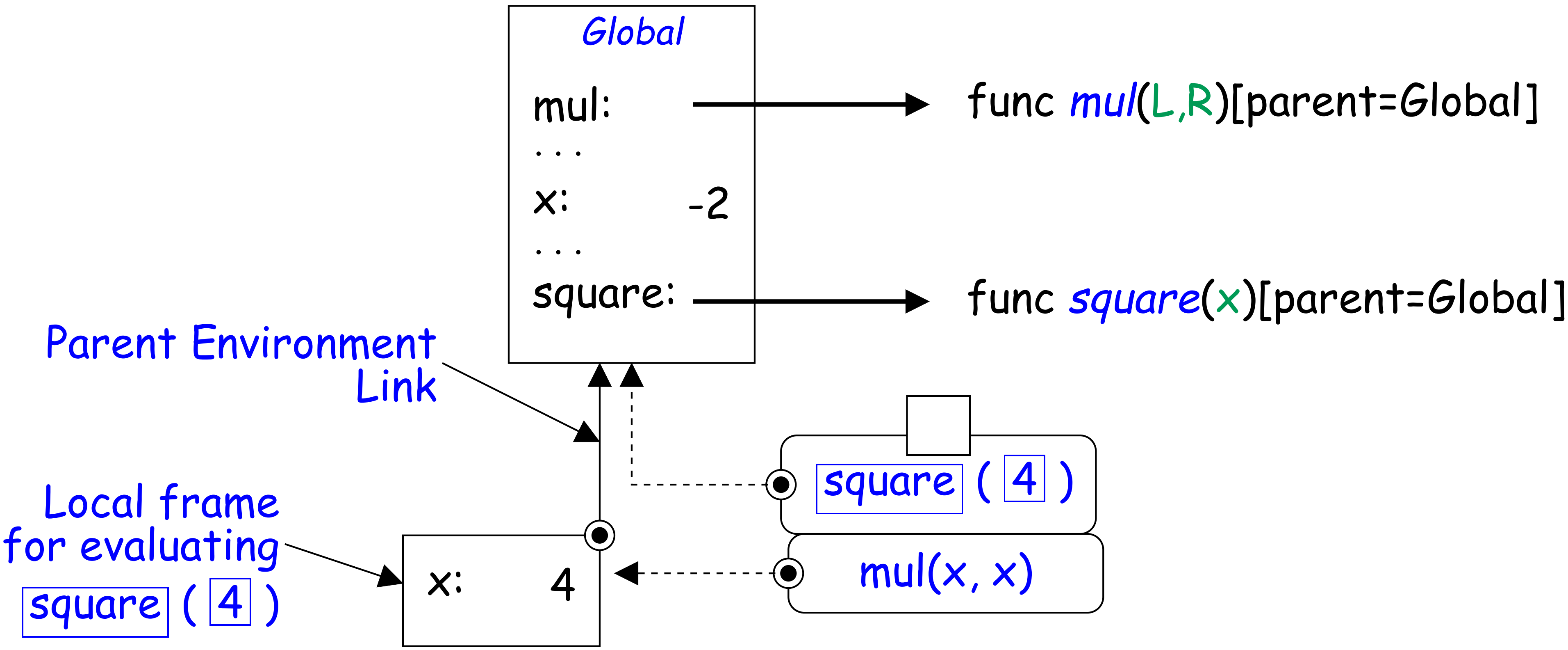
Evaluating User-Defined Functions Calls (III)

- Then perform the primitive multiply function:



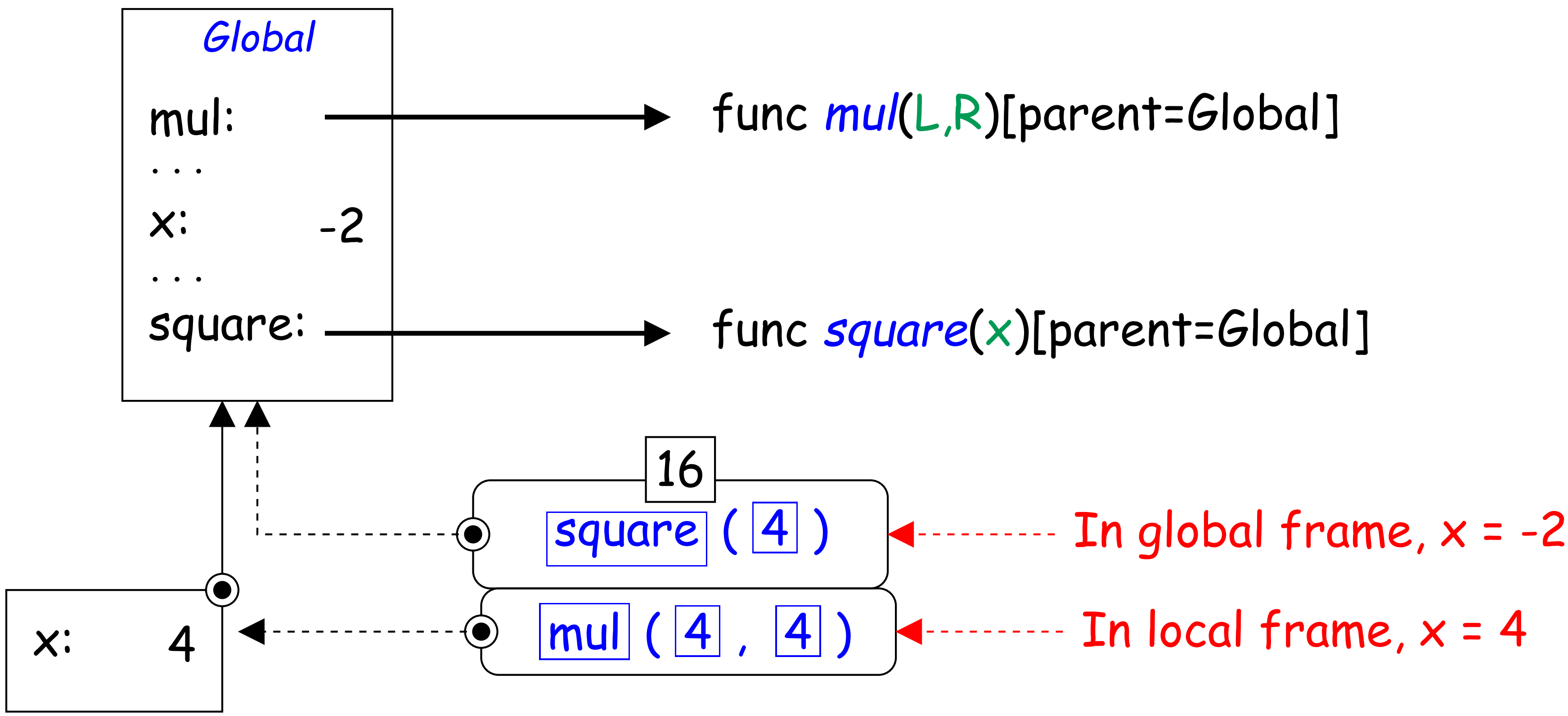
Evaluating User-Defined Functions Calls (IV)

- To explain the parameters to the user-defined square function, extend the 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 the 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.
- When evaluating an identifier in a chain of environments, follow the parent environment links to the first frame containing its definition.



Apply user-defined functions introduces a second local frame, only accessible to that function.

To apply a user-defined functions to some arguments:

① bind the arguments to the name of the functions' formal parameters in a new local frame.

② execute the body of the function in the environment that starts with this frame.

The environment in which the body is evaluated consists of two frames: ① the local frame that contains formal parameter bindings

② the global frame that contains everything else.

Each instance of a function application has its own independent local frame.

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

Global
mul → func mul(...)

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

Global
mul → func mul(...)
square → func square(x)

the entire def statement is processed in a single step. The body of a function isn't executed until the function is called (not when it is defined)

③ the square() function is called with the argument -2, a new frame is created with the formal parameter x bound to the value -2.

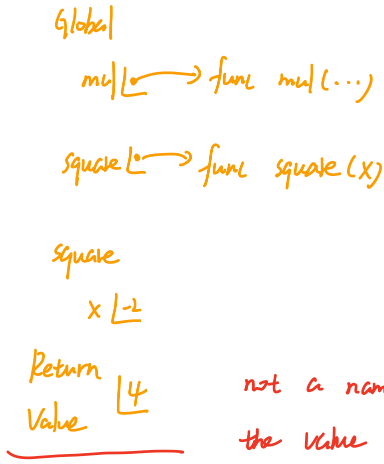
```
  from operator import mul
  def square(x):
      return mul(x, x)
→ square(-2)
```

Global
mul → func mul(...)
square → func square(x)

square
x | -2

③ name x is looked up in the current environment, x evaluates to -2, so the square function returns 4.

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



Both x and mul are bound in this environment, but in different frames

not a name binding, it indicates that the value returned by the function call that creates a frame

① top-level expression square(-2) is evaluated in the global environment

② the return expression mul(x, x) is evaluated in the environment created by calling square

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

So How Does This Help?

- The original problem that led to this whole environment diagram thing was how to deal with:

```
x = 4
x = 8
print(x)
```

- Now it's easy. Each time we assign to *x*, we create a new binding for it in the current evaluation frame (replacing the old one, if any).
- We get the new (last assigned) value when we look up *x* in the modified environment.
- The complication of hiding is also easy to explain: to find the meaning of a name, we follow a chain of environment frames and stop at the first one with the desired definition.

What makes a good function? → reinforce the idea that functions are abstractions

① Each function should have exactly one job, be identifiable with a short name and characterizable in a single line of text.

② Don't repeat yourself is a central tenet of software engineering.

Multiple fragments of codes shouldn't describe redundant logic

③ functions should be defined generally.