Lecture #2: Functions, Expressions, Environments

Public-Service Announcement I

"Net Impact Berkeley (NIB) is a not-for-profit, student-run consulting group on the UC Berkeley campus, part of the global Net Impact community of over 600,000 professional leaders. Every semester, NIB teams conduct semester-long consulting projects for a wide range of clents, including for-profit, non-profit, and social enterprises.

Come out to our Info Sessions and Case Workshop on Weds, 1/27 (10 Evans) and Thurs, 1/28 (145 Dwinelle).

Applications are due by FRIDAY, 1/29 at 12PM (Noon) to nib.berkeley.edu. If you have any questions feel free to email our VP of Finance and Operations, Zamzama Azizi, at zamzamaazizi@berkeley.edu."

Public-Service Announcement II

"Berkeley Consulting is a student-run consulting group on campus. We are a group of 30 students that complete 4 projects a semester for Fortune 500 firms, startups, and nonprofit organizations. We solve problems for and provide solutions to companies from all industries like Google, Dropbox and Khan Academy. We are currently recruiting and would love to have you join us! We are looking for students from all majors who are driven, critical thinkers, team players, and able to think outside the box. If you are interested in joining up please visit `bc.berkeley.edu` for more information. Also make sure to come to one of our info sessions on January 26th and 28th to learn more and attend our case workshop on January 29th to prepare for the interview process. We hope to see you at one of our events next week!"

Public-Service Announcement III

"CALPIRG has been here at Berkeley since 1976 working to protect the environment, promote democracy and make college more affordable.

This spring our top priority is to save the statewide ban on plastic bags. Before any plastic bag bans were passed here in CA 19 billion plastic bags washed into the Pacific Ocean each year. To date 138 cities and counties have ban plastic bags. But now out of state plastic companies are working to overturn plastic bag bans and have spent $3 million to date.

Unfortunately, it's working. This past summer Huntington Beach repealed their plastic bag ban.

This is going to be a big fight and we need all the help we can get. Apply to be a CALPIRG Intern today at `http://calpirgstudents.org/page/ca/campus-internships`.

You can also intern on our other campaigns to stop the overuse of antibiotics on factory farms, save the bees, get Berkeley to go solar, and alleviate hunger and poverty."

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 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

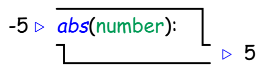
- For this lecture, we're going to use this notation to show function *values* (which are created by evaluating function *definitions*):

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

  (We'll simplify this in a bit to make it easier to write.)
- 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

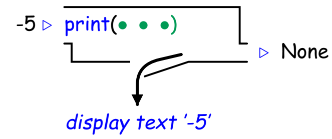
- The fundamental operation on function values is to *call* or *invoke* them, which means giving them one value for each formal parameter and having them produce the result of their computation on these values:



- 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. Its 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.

- mul(add(2, mul(0x4, 0x6)), add(0x3, 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. $\boxed{\text{print(• • •)}}$ ($\boxed{\text{print(• • •)}}$ ($\boxed{1}$), print(2))
3. $\boxed{\text{print(• • •)}}$ ($\boxed{\text{None}}$, print(2))
   and print '1'.
4. $\boxed{\text{print(• • •)}}$ ($\boxed{\text{None}}$, $\boxed{\text{print(• • •)}}$ ($\boxed{2}$))
5. $\boxed{\text{print(• • •)}}$ ($\boxed{\text{None}}$, $\boxed{\text{None}}$))
   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 first try a simple approach: *substitution* of values for names.
- This won't cover all the cases, however, and so we'll introduce the concept of an *environment*.

Substitution

- Let's try to explain the effect of

```
x = 3
y = x * 3
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).*

Substitution and Functions

- Now consider a simple function definition:

```
def compute(x, y):
    return (x * y) ** x
print(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 $(xy)^x$."
- Here, I'll use a common notation for that (due to Church):

$\lambda\, x, y : (xy)^x$.

- So after substitution for compute, we have

print((λ $x, y : (xy)^z$) (3, 2))

- Now what?

Substitution and Formal Parameters

- A function call such as

(λ $x, y : (xy)^z$) (3, 2)

from last slide is like a *simultaneous assignment* to or substitution for x and y.

- So we replace the whole expression with

$(3 \cdot 2)^3$

and (eventually), just 216.

Getting Fancy

- What about this?

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

print(incr(5)(6))
```

- The incr function returns a function. The argument to **print** then calls this function on 6.
- What happens?

Answer

- First, deal with incr:

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

print(incr(5)(6))        print((λ n: return λ x:  n + x)(5)(6))
```

- The 5 now gets substituted for n:

```
print((λ x:  5 + x)(6)
```

- And 6 for x:

```
print(5 + 6)
```

- Finally giving

```
print(11)
```


Trouble

- Alas, this relatively simple (if tedious) approach doesn't work.
- Example:

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

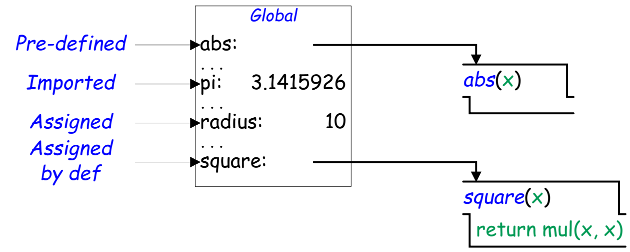
- If we just substitute for the first x, first as before:

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

- ...we get a wrong result (4 instead of 8).
- After one substitution, x isn't around any more to susbstitute for.
- We need something stronger.

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*:



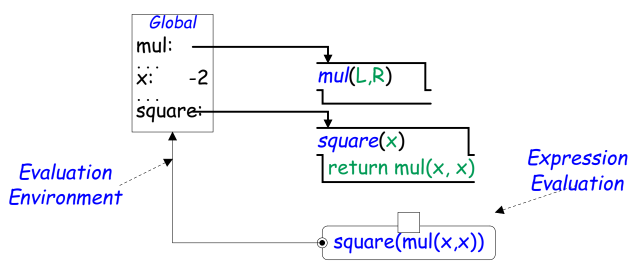
Environments and Evaluation

- 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

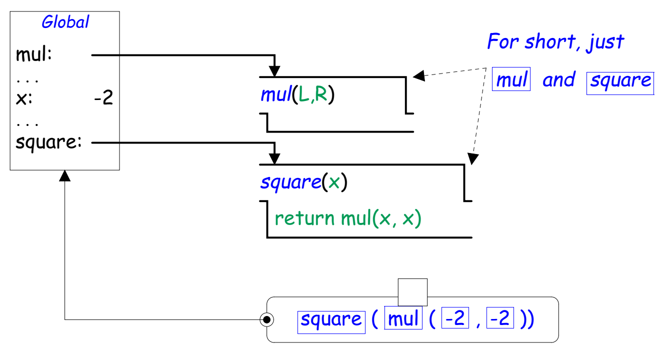
- Consider the expression square(mul(x, x)) after executing

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



Evaluating User-Defined Function Calls (II)

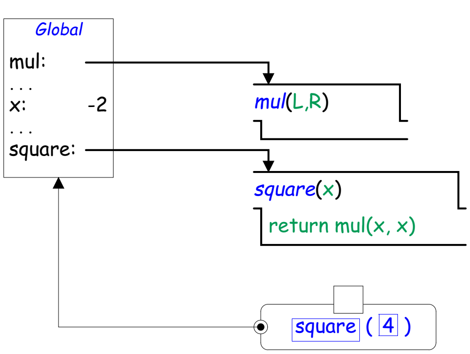
- First evaluate the subexpressions of square(mul(x, x)) in the global environment:



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

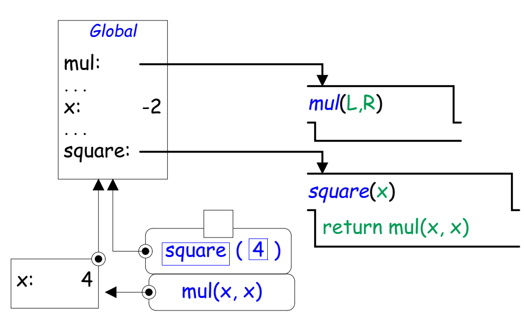
Evaluating User-Defined Functions Calls (III)

- 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.