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:

  \[
  \begin{align*}
  -5 & \Rightarrow \text{abs(number)} : 5 \\
  (29, 13) & \Rightarrow \text{add(left, right)} : 42 
  \end{align*}
  \]

- 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:

  \[
  -5 \Rightarrow \text{print('''')} : \text{None}
  \]

  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 \text{add}(2, 3):

  \[
  \begin{array}{c|c}
  \text{add} & \text{Operator} \\
  \hline
  2 & \text{Operand 0} \\
  3 & \text{Operand 1} \\
  \end{array}
  \]

- 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 \text{mul(add(2, \text{mul}(0x4, 0x6)), add(0x3, 0x5))}.

In the following sequence, values are shown in boxes.

1. \text{mul(add(2, \text{mul}(0x4, 0x6)), add(0x3, 0x5))}
2. \text{mul(add(2, 24), add(0x3, 0x5))}
3. \text{mul(26, add(0x3, 0x5))}
4. \text{mul(26, 25)}
5. \text{208}

Example: Print

What about an expression with side effects?
1. \text{print(print(1), print(2))}
2. \text{print(' • • ', (None, print(2)))}
3. \text{print(' • • ', (None, print(2)))}
   and print '1'.
4. \text{print(' • • ', (None, print(2)))}
   and print '1'.
5. \text{print(' • • ', (None, None))}
   and print '2'.
6. \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 \text{add}, \text{mul}, or \text{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
  \[ x = 3 \]
  \[ y = x \times 3 \]
  \[ z = y^x \]
  by treating each assignment (=) as a **definition**.

• Thus, we get
  \[ x = 3 \]
  \[ y = 3 \times 3 \]
  \[ z = y \times y \]

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

Substitution and Functions

• Now consider a simple function definition:
  ```python
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
  ```python
  print((\lambda x, y : (xy)^x)(3, 2))
  ```

• What happens?

Substitution and Formal Parameters

• A function call such as
  ```python
  (\lambda x, y : (xy)^x)(3, 2)
  ```
  from last slide is like a **simultaneous assignment** to or substitution for `x` and `y`.

• So we replace the whole expression with
  ```python
  (3 \times 2)^3
  ```
  and (eventually), just 216.

Getting Fancy

• What about this?
  ```python
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`:
  ```python
def incr(n):
    def f(x):
      return n + x
    return f
print(incr(5)(6))
print((\lambda n : return \lambda x : n + x)(5)(6))
```

• The 5 now gets substituted for `n`:
  ```python
  print((\lambda x : 5 + x)(6))
  ```

• And 6 for `x`:
  ```python
  print(5 + 6)
  ```

• Finally giving
  ```python
  print(11)
  ```

Trouble

• Alas, this relatively simple (if tedious) approach doesn't work.

• Example:
  ```python
  x = 4
  x = 8
  print(x)
  ```

• If we just substitute for the first `x`, first as before:
  ```python
  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 substitute 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:

<table>
<thead>
<tr>
<th>Pre-defined</th>
<th>Imported</th>
<th>Assigned</th>
<th>Assigned by def</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>abs:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pi: 3.1415926</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>radius: 10</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>square:</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

  return mul(x, x)

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 \(x^2 + 2\), ...).
- 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 \(\text{square}(\text{mul}(x, x))\) after executing

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

x = -2
```

- Evaluating the subexpressions \(x\), \(\text{mul}\), and \(\text{square}\) takes values from the expression's 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`.

```
Global
mul:
  x: -2
  square:
    x: 4

square(x)
  return mul(x, x)
```

```
16
mul(4, 4)
```