Lecture #3: Recap of Function Evaluation; Control

**Announcement:** Triangle Fraternity for Engineers, Architects, and Scientists Spring Rush 2012 This Week! Visit caltriangle.org for information.
Summary: Environments

• **Environments** map names to values.

• They consist of chains of **environment frames**.

• An environment is either a *global frame* or a first (local) frame chained to a *parent environment* (which is itself either a global frame or . . . ).

• We say that a name is **bound to** a value in a frame.

• The **value (or meaning) of a name** in an environment is the value it is bound to in the first frame, if there is one, . . .

• . . . or if not, the meaning of the name in the parent environment
A Sample Environment Chain

<table>
<thead>
<tr>
<th>In</th>
<th>Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>x</td>
</tr>
<tr>
<td>Environ 1</td>
<td>2</td>
</tr>
<tr>
<td>Environ 2</td>
<td>3</td>
</tr>
</tbody>
</table>

Last modified: Fri Mar 2 00:45:29 2012
Environments: Binding and Evaluation

• Every expression and statement is evaluated (executed) in an environment, which determines the meaning of its names.

• Subexpressions (pieces) of an expression are evaluated in the same environment as the expression.

• Assigning to a variable binds a value to it in (for now) the first frame of the environment in which the assignment is executed.

• Def statements bind a name to a function value in the first frame of the environment in which the def statement is executed.

• Calling a user-defined function creates a new local environment and binds the operand values in the call to the parameter names in that environment.
Example: Evaluation of a Call: sum_square(3,4)

```plaintext
square:
... mul, abs...
... sum_square:

sum_square(x, y)
return square(x) + square(y)

square(x)
return x * x

square(3)
x*x
9

square(4)
x*x
16

sum_square(3, 4)
25

x: 3
y: 4

A
B
```

Last modified: Fri Mar 2 00:45:29 2012
What’s Left?

• So far, all our environments have had at most two frames.

• We’ll see how longer chains of frames come about in upcoming lectures, ...

• But the machinery is now all present to handle them.

• Looking ahead, there are still two constructs—global and nonlocal—that will require additions.

• But we could build anything with what we already have.
What Does This Do?

def id(x):
    return x

print(id(id)(id(13)))
def id(x):
    return x

print(id(id)(id(13)))

• We’ll denote the user-defined function value created by `def id():...` by the shorthand `id`.

• Evaluation proceeds like this:

\[
\begin{align*}
\text{id(id)(id(13))} & \Rightarrow \text{id(id(id(id(id(id(id(13)))))))} \\
& \Rightarrow \text{id(id(13))} \\
& \Rightarrow \text{id(13)} \\
& \Rightarrow 13
\end{align*}
\]

(because `id` returns its argument).

• *Important:* There is nothing new on this slide! Everything follows from what you’ve seen so far.
Control

- The expressions we've seen evaluate all of their operands in the order written.

- While there are very clever ways to do everything with just this [challenge!], it's generally clearer to introduce constructs that control the order in which their components execute.

- A control expression evaluates some or all of its operands in an order depending on the kind of expression, and typically on the values of those operands.

- A statement is a construct that produces no value, but is used solely for its side effects.

- A control statement is a statement that, like a control expression, evaluates some or all of its operands, etc.

- We typically speak of statements being executed rather than evaluated, but the two concepts are essentially the same, apart from the question of a value.
Conditional Expressions (I)

- The most common kind of control is *conditional evaluation (execution)*.

- In Python, to evaluate

  \[ \text{TruePart} \; \text{if} \; \text{Condition} \; \text{else} \; \text{FalsePart} \]

  - First evaluate *Condition*.
  - If the result is a "true value," evaluate *TruePart*; its value is then the value of the whole expression.
  - Otherwise, evaluate *FalsePart*; its value is then the value of the whole expression.

- Example: If \( x \) is 2:

  \[ \frac{1}{x} \text{if} \; x \neq 0 \text{else} \; 1 \]

  \[ \Rightarrow \frac{1}{x} \text{if} \; \text{True} \text{else} \; 1 \]

  \[ \Rightarrow \frac{1}{2} \]

  \[ \Rightarrow 0.5 \]

  If \( x \) is 0:

  \[ \frac{1}{x} \text{if} \; x \neq 0 \text{else} \; 1 \]

  \[ \Rightarrow \frac{1}{x} \text{if} \; \text{False} \text{else} \; 1 \]

  \[ \Rightarrow 1 \]

  \[ \Rightarrow 1 \]

Last modified: Fri Mar 2 00:45:29 2012
“True Values”

• Conditions in conditional constructs can have any value, not just True or False.

• For convenience, Python treats a number of values as indicating “false”:
  - False
  - None
  - 0
  - Empty strings, sets, lists, tuples, and dictionaries.

• All else is a “true value” by default.

• So, for example: 13 if 0 else 5 and 13 if [] else 5 both evaluate to 5.
Conditional Expressions (II)

• To evaluate

  *Left* and *Right*

  - Evaluate *Left*.
  - If it is a false value, that becomes the value of the whole expression.
  - Otherwise the value of the expression is that of *Right*.

• This is an example of something called “short-circuit evaluation.”

• For example,

  \[
  5 \text{ and } "Hello" \implies "Hello". \\
  [] \text{ and } 1 / 0 \implies [].
  \]
Conditional Expressions (III)

• To evaluate

   \textit{Left} or \textit{Right}

   - Evaluate \textit{Left}.
   - If it is a true value, that becomes the value of the whole expression.
   - Otherwise the value of the expression is that of \textit{Right}.

• Another example of “\textit{short-circuit evaluation}.”

• For example,

\[
\begin{align*}
5 \text{ or } "Hello" & \implies 5, \\
[] \text{ or } "Hello" & \implies "Hello", \\
[] \text{ or } 1 / 0 & \implies ?.
\end{align*}
\]
Conditional Statement

- Finally, this all comes in statement form:

  ```python
  if Condition1:
      Statements1
      ...
  elif Condition2:
      Statements2
      ...
  ...
  else:
      Statementsn
      ...
  ```

- Execute (only) `Statements1` if `Condition1` evaluates to a true value.
- Otherwise execute `Statements2` if `Condition2` evaluates to a true value (optional part).
- ...
- Otherwise execute `Statementsn` (optional part).
def signum(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -1

# Alternative Definition
def signum(x):
    return 1 if x > 0 else 0 if x == 0 else -1
**Indefinite Repetition**

- With conditionals and function calls, we can conduct computations of any length.

- For example, to sum the squares of all numbers from 1 to $N$ (a parameter):

  ```python
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    if N < 1:
        return 0
    else:
        return N**2 + sum_squares(N - 1)
  ```

- This will repeatedly call `sum_squares` with decreasing values (down to 1), adding in squares:

  ```
  sum_squares(3) => 3**2 + sum_squares(2)
  => 3**2 + 2**2 + sum_squares(1)
  => 3**2 + 2**2 + 1**2 + sum_squares(0)
  => 3**2 + 2**2 + 1**2 + 0 => 14
  ```
Explicit Repetition

- But in the Python, C, Java, and Fortran communities, it is more usual to be explicit about the repetition.

- The simplest form is **while**

  ```python
  while Condition:
      Statements
  ```

  means “If condition evaluates to a true value, execute statements and repeat the entire process. Otherwise, do nothing.”

- So our sum-of-squares becomes:

```python
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    while N >= 1:
        result += N**2  # Or result = result + N**2
        N -= 1  # Or N = N-1
    return result
```
Did You Notice The Difference?

- OK: I cheated in the interests of brevity. In the recursive version, you actually add up the squares starting from the small end.

- So to be true to the original, I would write:

```python
def sum_squares(N):
    """The sum of K**2 for K from 1 to N (inclusive)."""
    result = 0
    k = 1
    while k <= N:
        result += k**2
        k += 1
    return result
```