Lecture #3: Recap of Function Evaluation; Control

Announcements:

• Office Hours are available. See the website for more details.

• Please make sure you have registered your Unix class account (cs61a-...). Also, make sure you have a class account, whether or not you use it.

• Work on the homework (~250 have submitted so far)

• Test #1 scheduled for the 6-8PM, Thursday, 2/18

• Do not publicly post code on Piazza. We’re working on a better way to submit your bugs to us.
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

Environ. 2

Global

Environ. 1

Global

Environ. 1's first frame

Environ. 1's parent

Environ. 2's parent

Environ. 2's first frame

Value of

<table>
<thead>
<tr>
<th>In</th>
<th>Value of</th>
</tr>
</thead>
<tbody>
<tr>
<td>Global</td>
<td>x  1</td>
</tr>
<tr>
<td></td>
<td>y  12</td>
</tr>
<tr>
<td>Environ 1.</td>
<td>x  2</td>
</tr>
<tr>
<td></td>
<td>y  12</td>
</tr>
<tr>
<td>Environ 2.</td>
<td>x  3</td>
</tr>
<tr>
<td></td>
<td>y  12</td>
</tr>
</tbody>
</table>
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)`

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

square(x)
return x * x

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

sum_square(3, 4)

25
square(x)+square(y)

9
square(3)
x*x

16
square(4)
x*x

A  x: 3
B  x: 4

A  y: 4

A

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 (And Why)?

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

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

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

• Evaluation proceeds like this:
  \[
  \text{id(id)(id(13))}
  \Rightarrow \text{id(id(id))(id(id(13)))}
  \Rightarrow \text{id(13)}
  \text{(because id returns its argument)}.
  \Rightarrow 13
  \text{(again because id returns its argument)}.

• \textbf{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

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

  - 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:

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

  \[
  \frac{1}{x} \text{ if } 2 \neq 0 \text{ else } 1
  \]

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

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

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

  \[
  \Rightarrow 0.5
  \]

If \( x \) is 0:

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

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

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

\[
\Rightarrow 1
\]

\[
\Rightarrow 1
\]
• 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
  
  \textit{Left} and \textit{Right}
  
  - Evaluate \textit{Left}.
  - If it is a false value, that becomes the value of the whole expression.
  - Otherwise the value of the expression is that of \textit{Right}.

• This is an example of something called “\textit{short-circuit evaluation}.”

• For example,

  \[ 5 \text{ and } "Hello" \Rightarrow "Hello". \]
  \[ [] \text{ and } 1 / 0 \Rightarrow []. \]
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,
  \[
  5 \text{ or } "Hello" \quad \Rightarrow \quad 5 .
  \]
  \[
  [] \text{ or } "Hello" \quad \Rightarrow \quad "Hello" .
  \]
  \[
  [] \text{ or } 1 / 0 \quad \Rightarrow \quad ? .
  \]
Conditional Statement

• Finally, this all comes in statement form:

  if $\text{Condition1}$:
    $\text{Statements1}$
    ...
  
  elif $\text{Condition2}$:
    $\text{Statements2}$
    ...
  
  ...
  
  else:
    $\text{Statementsn}$
    ...

• Execute (only) $\text{Statements1}$ if $\text{Condition1}$ evaluates to a true value.

• Otherwise execute $\text{Statements2}$ if $\text{Condition2}$ evaluates to a true value (optional part).

• ...

• Otherwise execute $\text{Statementsn}$ (optional part).
Example

# Alternative Definition

def signum(x):
    if x > 0:
        return 1
    elif x == 0:
        return 0
    else:
        return -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: Execute here

```python
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.”

• The effect is (nearly) identical to

```python
def loop():
    if Condition:
        Statements
    loop()

loop()  # Start things off
```

• ...except that (for most Python implementations) the latter eventually runs out of memory; and we’ll have to do something about assignments to variables (more on that later).
Sum_squares Iteratively?

- Our original **sum_squares** was

  ```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)
  ```

- How do we do the same thing with a **while** loop?

  ```python
  def sum_squares(N):
      """The sum of K**2 for K from 1 to N (inclusive).""
  ```
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
Another Way

• Alternatively, I can make this a little shorter by adding the other way:

```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
```

Execute here