61A Lecture 2

Monday, August 27, 2012
Lightning Review: Types of Expressions

Primitive expressions:  
- Number: 2
- Name: `add`
- String: `'hello'`

Call expressions:

```
max(min(pow(3, 5), -4), min(1, -2))
```

One big nested call expression

```
max(2, 3)
```
Lightning Review: Expression Trees

\[
\text{max}(\text{min}(\text{pow}(3, 5), -4), \text{min}(1, -2))
\]

Operand 0 “subexpression”

Leaves are primitive expressions

Expression tree
The Print Function

(Demo)
Pure Functions & Non-Pure Functions

Pure Functions

**Argument**
-2

**abs(number):**

**Return value**
2

2, 100

**pow(x, y):**

1267650600228229401496703205376

Non-Pure Functions

**Side effect**
-2

**print(...):**

None

Python displays the output “-2”

The Interactive interpreter displays all return values except None.
Nested Expressions with Print

None, None ➔ print(...):

None

display “None None”

None

print(print(1), print(2))

None

>>> print(print(1), print(2))
1
2
None None

1 ➔ print(...):

None

display “1”

2 ➔ print(...):

None

display “2”
The Elements of Programming

- Primitive Expressions and Statements
  - The simplest building blocks of a language

- Means of Combination
  - Compound elements are built from simpler ones

- Means of Abstraction
  - Compound elements can be named and manipulated as units
Names, Assignment, and User-Defined Functions

(Demo)
Environment Diagrams

Environment diagrams visualize the interpreter’s process.

**Code (left):**
Statements and expressions
Next line is highlighted

**Frames (right):**
A name is bound to a value
In a frame, there is at most one binding per name

(Demo)

Example: [http://goo.gl/sQabt](http://goo.gl/sQabt)
User-Defined Functions

Named values are a simple means of abstraction.

Named *expressions* are a more powerful means of abstraction.

Function “signature” indicates how many parameters:

```python
>>> def <name>(<formal parameters>):
  return <return expression>
```

Function “body” defines a computational process.

Execution procedure for def statements:

1. Create a function value with signature `<name>(<formal parameters>)`.
2. Bind `<name>` to that value in the current frame.
Calling User-Defined Functions

Procedure for applying user-defined functions (version 1):

1. Add a local frame
2. Bind formal parameters to arguments in that frame
3. Execute the body of the function in the new environment

Example:
```python
from operator import mul

def square(x):
    return mul(x, x)

square(-2)
```

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)
Calling User-Defined Functions

Procedure for applying user-defined functions (version 1):

1. Add a local frame
2. Bind formal parameters to arguments in that frame
3. Execute the body of the function in the new environment

Example:

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

A function’s signature has all the information to create a local frame

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)
Looking Up Names In Environments

Every expression is evaluated in the context of an environment.

So far, the current environment is either:

- The global frame alone, or
- A local frame, followed by the global frame.

**Most important two things I’ll say all day:**

An environment is a sequence of frames.

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

E.g., to look up some name in the body of the square function:

- Look up the name in the local frame.
- If not found, look it up in the global frame.

(Built-in names like “print” are in the global frame too, but we don’t draw them in environment diagrams.)

(Demo)
**Formal Parameters**

```python
def square(x):
    return mul(x, x)

def square(y):
    return mul(y, y)
```

**Formal parameters have local scope**

Example: [http://goo.gl/OapJa](http://goo.gl/OapJa)