Scope & Evaluation

June Andrews

May 26, 2010

1. Book?
2. Labs are up and running, see Gautam's email on configuring the pydev plugin for eclipse
3. (M, Tu, W) homework due Thurs 9:30am
4. Memorial Day next monday (Staff Holiday), so no class

- What does this return?

```
In [1]: a_list = ['first', 'second', 'last']
In [2]: a_list[-2] -> ?
In [3]: a_list.append(['new'])
In [4]: a_list.extend(['new'])
In [5]: a_list -> ?
```

- What does this function do?

```
def Mystery(a_list):
    return (not(a_list==[]) or ()) and
            ( (a_list[0]==[] and Mystery(a_list[1:])) or
              (a_list[0], Mystery(a_list[1:])) )
```

- What does this function do?

```
def Mystery(a_list):
    return (not(a_list==[]) or ()) and
           ( (a_list[0]==[] and Mystery(a_list[1:])) or
             (a_list[0], Mystery(a_list[1:])) )
```

- ```
  In [137]: E.Mystery([5, [], 6, 7, [], 9])
  Out[137]: (5, (6, (7, (9, ()))))
  ```
  ```

Proper terminology for yesterday's example.

- Python evaluates terms(can be expressions) in a function to produce values. It does so left to right.
- Better put by the Spring course:

> *Running a [Python functional] program is just evaluating a term. What happens when we evaluate a term? In an imperative (non-functional) language like Java, we sometimes imagine that there is an idea of a "current statement" that is executing. This isn't a very good model for [Python]; it is better to think of [Python] programs as being evaluated in the same way that you would evaluate a mathematical expression.*

Types of errors:

- *Synatx errors:* 'I'm here' (Only type of error Python catches prior to running.)
- *Type errors:* 'a'/4
- *Semantic errors:* 1/0 (Technically, ok, just don't make sense.)
- *Logic errors:* Program design flaws.

- *Scope*:

  the part of the program where the variable stands for the value it is bound to.[Sp10]

  1. *Lexical Scope / Static*: can be determined by reading the code. (Python)
  2. *Dynamical Scope*: develops as code is run. Is determined by the execution stack. (some Perl)

- *Environment*: the set of all variables available in that block of code

```
dog = 1
cat = -1
def OuterFun(a_list):
  dog = 2

  def InnerFun(a_list):
    cat = -3
    a_list[0] = -9

    print 'Inner cat:', cat
    print 'Inner dog:', dog

  InnerFun(a_list)
  print 'Outer dog:', dog
  print 'Outer cat:', cat
```

- Board work of drawing out boxed enivronments for OuterFun.

- Board work of drawing out boxed enivronments for OuterFun.
- ans:

```
Inner cat: -3
Inner dog: 2
Outer dog: 2
Outer cat: -1
```

- Board work of drawing out boxed enivronments for OuterFun.
- ans:

```
Inner cat: -3
Inner dog: 2
Outer dog: 2
Outer cat: -1
```

- And for a non-empty list passed into OuterFun? (Reminder of pass by value and pass by reference.)(Did I need to pass a_list to InnerFun?)

```
dog = 1
cat = -1
def OuterFun(a_list):
  dog = 2

  InnerFun(a_list)
  print 'Outer dog:', dog
  print 'Outer cat:', cat

def InnerFun(a_list):
  cat = -3
  a_list[0] = -9

  print 'Inner cat:', cat
  print 'Inner dog:', dog
```

For classes, variables are only accessible in the class code block, not within methods or consequentially generator functions. They are accessed with *self*.*variable*.

- Works:

```
class A:
  a = 42
  c = a+8
```

- Fails:

```
class A:
  a = 42
  c = [a*i for i in range(10)]
```

- *Unbound or Free Variables*: variables, given all the code, whos value may change. Alternatively, variables whos names act as place holders. ie, input variables
- *Bound Variables*: variables, whos value can be evaluated.

Python rules

- Python works by *reducing* terms to values(non-reducible), in a left to right manner.
- In some cases(such as infinite lists and compound boolean expressions) Python is a 'lazy' evaluator and does not evaluate anything until needed. In other cases as in (x = 1+5) Python is an 'eager' evaluator and stores x as 6, even before x is needed.
- For the most part Python is 'eager.'
- The process of calculating values is referred to as 'reducing' the terms.

```
def PigLatin(sentence):
  spoken = ''
  for  word in sentence.split(' '):
    spoken += Slang(word) + ' '
  return spoken[:-1]

def Slang(word):
  return word[1:]+word[0]+'ay'
```

Board work for substitution example PigLatin('Hey there')

That was simple. A little too simple?
Yes, interpreters are much more complicated. But when code has no side effects(all variables are the same after the computation as before) substitution can be a handy way to think of how your code evaluates.
Don't use the number of substitutions to calculate runtime.

Recursive Function calls

Can unroll.

Better way next time.

Homework if not up by my office hours, then due Friday.