Peg Solitaire

Depth First Recursive Search for Solutions

Using Python

Greg Smith
greg@scoug.com

Presented to the Programming SIG
of the
Southern California OS/2 Users Group

April 19, 2003

Why Python?

- Open Source
- Interpreted byte codes – VM approach similar to Pascal, Java, Perl, etc.
- Clean, if somewhat unusual syntax
    - No Begin/End for blocks
    - No {} for blocks
    - Consistent use of the semicolon ; (It isn't used)
- Can use an Object Oriented approach – if you want to.
- Because it is there – Another interesting language to learn.

Python Data Types

- Immutable objects (constant) data
    - Numbers
        - Integers – Plain Integers and Long Integers
        - Floating Point
        - Complex Numbers
    - Strings – ASCII and Unicode
    - Tuples
- Other objects (data types)
    - Lists – The items of a list are arbitrary Python objects.
    - Dictionaries – A dictionary represents finite sets of objects indexed by arbitrary values. The arbitrary key must be an immutable type.

Python Gotcha

- Assignment is a *reference* to the object

```
>>> a = [ 1, 3, 5, 7 ]
>>> b = a
>>> print a
[1, 3, 5, 7]
>>> print b
[1, 3, 5, 7]
>>> b[2] = 73
>>> print b
[1, 3, 73, 7]
>>> print a
[1, 3, 73, 7]
>>> a = [ 7, 5, 3, 1 ]
>>> print b
[1, 3, 73, 7]
>>> print a
[7, 5, 3, 1]
```

Python Slice Operator

- Slice operations make copies of the list objects that they reference

```
>>> a = [ "cat", "dog", "canary", "ferret", "hamster" ]
>>> b = a[1:3]
>>> print a
['cat', 'dog', 'canary', 'ferret', 'hamster']
>>> print b
['dog', 'canary']
>>> c = a[1:]
>>> print c
['dog', 'canary', 'ferret', 'hamster']
>>> d = a[:]
>>> print d
['cat', 'dog', 'canary', 'ferret', 'hamster']
>>> d[1] = "bad dog"
>>> print d
['cat', 'bad dog', 'canary', 'ferret', 'hamster']
>>> print a
['cat', 'dog', 'canary', 'ferret', 'hamster']
```

Data Structure for Board Layout

```
>>> board = [ "--xxx--", \
...                    "--xxx--", \
...                    "xxxxxxx", \
...                    "xxx.xxx", \
...                    "xxxxxxx", \
...                    "--xxx--", \
...                    "--xxx--"  ]
>>> print board
['--xxx--', '--xxx--', 'xxxxxxx', 'xxx.xxx', 'xxxxxxx', '--xxx--', '--xxx--']
>>> for line in board:
...         print line
...
--xxx--
--xxx--
xxxxxxx
xxx.xxx
xxxxxxx
--xxx--
--xxx--
```

Data Structure for Board Layout

- However, strings are immutable objects. So we replace each string with a list to give our data structure for the board.

```
board   = [ [ " ", " ", "x", "x", "x", " ", " " ],
            [ " ", " ", "x", "x", "x", " ", " " ],
            [ "x", "x", "x", "x", "x", "x", "x" ],
            [ "x", "x", "x", ".", "x", "x", "x" ],
            [ "x", "x", "x", "x", "x", "x", "x" ],
            [ " ", " ", "x", "x", "x", " ", " " ],
            [ " ", " ", "x", "x", "x", " ", " " ] ]
```

Data Structure for Tracking Moves

- Moves are tracked in a list of board positions that acts as a stack.

```
stack = []
stack.append(firstmove)
```

- The recursive routine takes the stack as its argument. Moves are generated and added to the stack. After processing a move, it is popped off the stack and another is tried.

```
def  traverse ( stack ):
    .......
        #generate move
        stack.append(copy.deepcopy(newboard)
        traverse ( stack )
        stack.pop()
    .......
```