Lecture #10: Sequences and Comprehensions

Announcements:

- HW4, Project #2 available.
- All people needing test accommodations should email me this week.
- Needed: student with undergrad physics course who can type equations in Latex or Microsoft equation writer to help finish an answer book for a new introductory physics text.
- **CSUA Hackathon**: Code any 18 hour project of your choice!
  
  When: 1800 Friday 2/17 to 1200 Saturday 2/18.
  Location: Wozniak Lounge + Overflow rooms
  Teams of 4! Registration is day-of.
  Private github repo provided!
Extension of Map

• Homework #4 uses a version of \texttt{map} that takes multiple arguments:

\begin{verbatim}
>>> from operator import *
>>> tuple(map(add, (1, 2, 3, 18), (5, 2, 1)))
(6, 4, 4)
\end{verbatim}

• That is, \texttt{map} takes a function of $N$ arguments plus $N$ sequences and applies the function to the corresponding items of the sequences (throws away extras, like 18).

• So, how do we do this:

\begin{verbatim}
def deltas(L):
    """Given that \texttt{L} is a sequence of \texttt{N} items, return the (\texttt{N-1})-item sequence \texttt{(L[1]-L[0], L[2]-L[1],...)}."""
    return ________________________________
\end{verbatim}
def deltas(L):
    """Given that L is a sequence of N items, return
    the (N-1)-item sequence (L[1]-L[0], L[2]-L[1],...)."""

    return map(sub, tuple(L)[1:], L)

>>> deltas((1, 2, 4, 3, 9))
<map object at 0x82b9ccc>
>>> tuple(deltas((1, 2, 4, 3, 9)))
(1, 2, -1, 6)
“Map Objects”??

- We say that `map` and `filter` operate on and return *sequences*.
- In fact, as these lectures have said, there are many forms of sequences, with different interfaces (i.e., different possible operations).
- `map` and `filter` return objects that look a bit like rlists, with a first item and subsequent items.
- `except` that you only get one bite at the first item.
- We’ll get into why and how later.
- For now, we can convert these objects into tuples (with `tuple`) or lists (with `list`) when we need to print them, subscript them, or slice them.
- `map`, `filter`, and `reduce`, meanwhile, can handle any kind of sequence as input.
Representing Multi-Dimensional Structures

• How do we represent a two-dimensional table (like a matrix)?
• Answer: use a sequence of sequences (typically a list of lists or tuple of tuples).
• The same approach is used in C, C++, and Java.
• Example:

\[
\begin{bmatrix}
  1 & 2 & 0 & 4 \\
  0 & 1 & 3 & -1 \\
  0 & 0 & 1 & 8 \\
\end{bmatrix}
\]

becomes

\[
( (1, 2, 0, 4), (0, 1, 3, -1), (0, 0, 1, 8) )
\]

# or

\[
[[1, 2, 0, 4], [0, 1, 3, -1], [0, 0, 1, 8]]
\]

# or (for old Fortran hands):

\[
[[1, 0, 0], [2, 1, 0], [0, 3, 1], [4, -1, 8]]
\]
Life: Another Problem

- One step in J.H. Conway's game of Life is to count the number of occupied neighbors (0-8) of a given cell on a two-dimensional square grid. The rules then state which cell occupants die and which unoccupied cells give birth based on this count.

- Example:

<table>
<thead>
<tr>
<th>Board</th>
<th>NeighborCount</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>0 1 2 3 2 1 0 0</td>
</tr>
<tr>
<td>*</td>
<td>0 2 3 5 3 2 0 0</td>
</tr>
<tr>
<td>*</td>
<td>0 3 4 7 4 3 0 0</td>
</tr>
<tr>
<td>*</td>
<td>0 2 2 5 2 2 0 0</td>
</tr>
<tr>
<td>*</td>
<td>0 2 2 3 2 3 2 1</td>
</tr>
<tr>
<td>*</td>
<td>0 1 0 1 2 3 3 2</td>
</tr>
<tr>
<td>*</td>
<td>0 1 1 1 2 3 3 2</td>
</tr>
<tr>
<td>*</td>
<td>0 0 0 0 1 2 2 1</td>
</tr>
</tbody>
</table>
Computing the Count

• Suppose that a board is a list of lists containing 1 (for ‘*’) and 0 for blank:

\[
\begin{array}{rrrrrrrr}
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\end{array}
\]

• Now we want:

```python
def neighbors(board):
    """A list of list of integers, NC, such that NC[i][j]
    is the number of occupied neighbor cells of board[i][j]."""
    return ________________________________
```
def add3(x, y, z): return x + y + z
def with_border(B):
    """Life board B with a layer of 0’s around the edges."""
    m, n = len(board), len(board[0])
    return [ [0] * (n+2) ] \
            + list(map(lambda row: [0] + row + [0], B)) \
            + [ [0] * (n+2) ]

def neighbors(board):
    """A list of list of integers, NC, such that NC[i][j]
    is the number of occupied neighbor cells of board[i][j]."""
    board = with_border(board)
    return ______________________________
Comprehensions

• Another way to create sequences is to specify them with a description of the elements.

• We already do that with list and tuple displays:

  [1, 2, 3, 4, 5, 6, 8]
  (9, 16, 25, 36, 49, 64, 81)
  [1, 2, 3, 2, 4, 6, 3, 6, 9]

• But we can also use *comprehensions*: formulas that generate the elements:

  [x for x in range(1, 9) ]
  tuple( (x**2 for x in range(3, 10)) )
  [x * y for x in range(1,4) for y in range(1, 4)]
Another Approach to Neighbors

def neighbors(board):
    """A list of list of integers, NC, such that NC[i][j]
    is the number of occupied neighbor cells of board[i][j]."""

    m = len(board)
    n = len(board[0])
    B = with_border(board)
    return [[B[i-1][j-1]+B[i-1][j]+B[i-1][j+1]
             +B[i][j-1]+B[i][j+1]
             +B[i+1][j-1]+B[i+1][j]+B[i+1][j+1]
             for j in range(1, n+1)]
            for i in range(1, m+1)]