# Lecture #13: More Sequences and Strings



## Odds and Ends: Multi-Argument Map

- Python's built-in map function actually applies a function to one *or more* sequences:

```
>>> from operator import *
>>> tuple(map(abs, (-1, 2, -4, 5))
(1, 2, 4, 5)
>>> tuple(map(add, (1, 2, 3, 18), (5, 2, 1)))
(6, 4, 4)
```

- That is, 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:

```
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)
```



## Defining multi-argument map: zip and F(*S)

- Defining map requires
  - The library function zip:

```
>>> tuple(zip((1, 2), (3, 4), (5, 6, 7)))
((1, 3, 5), (2, 4, 6))
```

  - And Python's "apply" and multi-argument syntax:

```
>>> def multi_arg(*args): print(args)
>>> multi_arg()
[]
>>> multi_arg(1)
[1]
>>> multi_arg(3, 4, 5)
[3, 4, 5]
>>> def two_argument_function(x, y): return 2*x + 3*y
>>> two_argument_function(3, 4)
18
>>> two_argument_function( *(3, 4) )
18
```

```
• def map(func, *sequences):
      return (func(*S) for S in zip(*sequences))
```



## Odds and Ends: Membership

- Built-in Python sequences support the membership operation:

```
>>> 5 in (2, 3, 5, 7, 11, 13, 17, 19)
True
>>> 6 not in (2, 3, 5, 7, 11, 13, 17, 19)
True
>>> (3, 2) in ((1, 2), (3, 4), (6, 5), (2, 3))
False
>>>
```



## Representing Multi-Dimensional Structures

- How do we represent a two-dimensional table (like a matrix)?
- Answer: use a sequence of sequences (such as a 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]]
```



## The Game of Life: Another Problem

- J. H. Conway's Game of Life is an example of a *cellular automaton* on an infinite grid of squares.
- Each square may be occupied or unoccupied.
- One genertion of cells is computed from the preceding according to a simple rule:
  - An occupied empty square with 2 or 3 occupied neighbor squares in one generation remains occupied in the next.
  - An empty square with exactly 3 occupied neighbor squares in one generation becomes occupied in the next.
  - All other squares become or remain unoccupied in the next generation.
- One can build arbitrary computations from these simple rules, resulting in remarkable patterns.
- (See `http://www.youtube.com/watch?v=C2vgICfQawE`)

## Counting Neighbors

- Consider the problem of computing the number of occupied neighbors of each cell on a grid.
- We'll use a slight modification: a finite grid that wraps around: the top row is adjacent to the bottom, and the left column adjacent to the right.
- Example (1 indicates occupancy; blank squares are 0):

**Board**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| | | 1 | 1 | 1 | | | |
| | | 1 | 1 | 1 | | | |
| | | 1 | | 1 | | | |
| | | | | | | | |
| | | 1 | | | 1 | 1 | |
| | | | | | 1 | 1 | |
| | | | | | | | |
| | | | | | | | |

**Neighbor Count**

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 0 | 2 | 3 | 5 | 3 | 2 | 0 | 0 |
| 0 | 3 | 4 | 7 | 4 | 3 | 0 | 0 |
| 0 | 2 | 2 | 5 | 2 | 2 | 0 | 0 |
| 0 | 2 | 2 | 3 | 2 | 3 | 2 | 1 |
| 0 | 1 | 0 | 1 | 2 | 3 | 3 | 2 |
| 0 | 1 | 1 | 1 | 2 | 3 | 3 | 2 |
| 0 | 0 | 0 | 0 | 1 | 2 | 2 | 1 |
| 0 | 1 | 2 | 3 | 2 | 1 | 0 | 0 |



## Strategy (I): Map2

- Suppose that we have a function like map that operates on sequeuces of sequeuces.

```
def map2(f, A, B):
    """Given that A and B are 2-dimensional sequences, the result of
    applying f to corresponding elements of A and B(as a tuple of tuples).
    Extra rows or columns in one or the other argument are thrown away.
    >>> map2(add, ((1, 2, 3), (4, 5, 6)), ((7, 8, 9), (10, 11, 12)))
    ((8, 10, 12), (14, 16, 18))
    """
    return tuple(map(lambda ra, rb: tuple(map(f, ra, rb)),
                     A, B))
```

- With this, we can find the number of neighbors of each cell (with a little help).



## Strategy (II): rotate2

- *Rotating* a sequence right by $N$ means moving its last $N$ values to the front, shifting the rest over.
- Rotating left by $N$ moves the first $N$ values to the end.
- We rotate 2D lists in two directions: rotating the rows and the columns:

```
def rotate2(A, dr, dc):
    """Given that A is a 2-dimensional sequence the result of rotating each
    row of A by DC columns and each column by DR rows. That is, a new
    2D tuple, B, in which B[r+dr][c+dc] is A[r][c], wrapping at the ends.
    >>> rotate2( ((1, 2, 3), (4, 5, 6), (7, 8, 9), (10, 11, 12)), (1, -1))
    ((11, 12, 10), (2, 3, 1), (5, 6, 4), (8, 9, 7))"""
    def rotate(R, d):
        # Negative slice indices count from the right.
        if d < 0:
            return R[-len(R)-d: ] + R[0: -d]
        else:
            return R[-d:] + R[0: len(R)-d]
    rows = tuple(map(lambda row: rotate(row, dc), A))
    return rotate(rows, dr)
```



## Strategy (III): Adding Up Neighbors

- Now we can find number of neighbors (with wrap-around) by shifting and adding:

**A =**

| | | | | |
|---|---|---|---|---|
| | 1 | 1 | 1 | |
| | 1 | 1 | 1 | |
| | 1 | | 1 | |
| | | | | |

**neighbor_count(A) =**

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | | |
| 1 | | 1 | | |
| | | | | |
| 1 | 1 | 1 | | |

+

| | | | | |
|---|---|---|---|---|
| | 1 | 1 | 1 | |
| | 1 | | 1 | |
| | | | | |
| | 1 | 1 | 1 | |

+

| | | | | |
|---|---|---|---|---|
| | | 1 | 1 | 1 |
| | | 1 | | 1 |
| | | | | |
| | | 1 | 1 | 1 |

+

| | | | | |
|---|---|---|---|---|
| 1 | 1 | 1 | | |
| 1 | 1 | 1 | | |
| 1 | | 1 | | |
| | | | | |

+

| | | | | |
|---|---|---|---|---|
| | | 1 | 1 | 1 |
| | | 1 | 1 | 1 |
| | | 1 | | 1 |
| | | | | |

+

| | | | | |
|---|---|---|---|---|
| | | | | |
| 1 | 1 | 1 | | |
| 1 | 1 | 1 | | |
| 1 | | 1 | | |

+ . . .



## Finally, neighbor_count

Putting it all together:

```
def neighbor_count(A):
    """Given a life board A, the number of neighbors corresponding to each
    cell as a tuple of tuples, assuming the board wraps around.
    >>> neighbor_count(((0, 0, 0, 0),
    ...                 (0, 1, 0, 0),
    ...                 (0, 1, 1, 0),
    ...                 (0, 0, 0, 0)))
    ((1, 1, 1, 0), (2, 2, 3, 1), (2, 2, 2, 1), (1, 2, 2, 1))
    """
    sum2 = lambda A, B: map2(add, A, B)
    neighbors = ((-1, -1), (-1, 0), (-1, 1),
                 (0, -1),           (0, 1),
                 (1, -1),  (1, 0),  (1, 1))
    return reduce(sum2,
                  map(lambda d: rotate2(A, d[0], d[1]),
                      neighbors))
```



## Strings: A Specialized Type of Sequence

- Strings are sequences of characters, with a good deal of special syntax.
- Rather odd property: the base cases are circular. Characters are themselves strings of length 1!
- The usual operations on tuples apply also to strings:

```
>>> "abcd"[0]
'a'
>>> len("abcd")
4
>>> "abcd"[1:3]
'bc'
>>> "ab" + "cd"
'abcd'
>>> "x" * 5
"xxxxx"
>>> for c in "abcd":
        print(c, end=", ")
a, b, c, d,
```

## Modified Operations

- Membership is not quite the same for strings:

```
>>> 'b' in ('a', 'b', 'c', 'd')     # A sequence, not a string
True
>>> 'bc' in ('a', 'b', 'c', 'd')
False
# But...
>>> 'b' in 'abcd'
True
>>> 'bc' in 'abcd'                   # in Finds substrings
True
```

- The substring is generally more important than the character, in other words.



## Numerous Functions and Methods

- The calls str(x) and x.__str__() convert values of any type into strings that depict them:

```
>>> str(3+7)
'10'              A string, not an int
```

- The methods reflect common manipulations from "real life":

```
>>> "i can't find my shift key".capitalize()
'I can't find my shift key'.capitalize()
>>> "cHaNge".upper() + " CaSe".lower() + " raNDomLY".swapcase()
'CHANGE case RAndOMly'
>>> '1234'.isnumeric() and 'abcd'.isalpha()
True
>>> 'SNAKEeyes'.upper().endswith('YES')
True
>>> '{x} + {y} = {answer}'.format(answer=7, x=3, y=4)
'3 + 4 = 7'
>>> " ".join(map(lambda x: x.capitalize(), "a bunch of words".sp
'A Bunch Of Words'
```



## A Cast of Thousands

- Python3 uses Unicode as its basic character set: an international standard comprising most alphabets (dead and alive).
- Characters have standard numbers (indicating position in the character set) and names. The Python ord and chr convert from character to number and back.
- Getting your computer to actually render them all properly, however, is another matter entirely, which is outside Python.
- The character codes from 0–127 (7-bit codes) are known as ASCII (American Standard Code for Information Interchange). Everything you typically type uses this subset.
- Nice property: 1 byte (8 bits) per character.
- This is lost with Unicode, but since there is an extra bit, we can *encode* larger character codes (UTF-8).



## Denoting Characters and Strings

- You've seen string literals all along. Python has 8 (!) styles. Consider the string

```
\begin{quote}
"I'd rather be in Philadelphia."
\end{quote}
```

which we can write:

```
>>> "\\begin{quote}\n\"I'd rather be in Philadelphia.\"\n\\end{quote}"
>>> '\\begin{quote}\n"I\'d rather be in Philadelphia."\n\\end{quote}'
>>> """\\begin{quote}
...  "I'd rather be in Philadelphia."
...  \\end{quote}"""
>>>  '''\\begin{quote}
...  "I'd rather be in Philadelphia."
...  \\end{quote}"""
>>>  r"""\begin{quote}
...  "I'd rather be in Philadelphia."
...  \end{quote}"""
```



## Escapes

- The \ escape allows us to introduce special, non-graphical characters" newline \n, tab \t
- Or to insert quoting characters.
- Or Unicode characters:

```
"\u006b\u03b1\u03b2\u03b3\u03b6\u05d1\u05d0\u8071\u8072"
"\u263a\u2639"
```

[Try printing this on your home computer].



## Strings as Sequences

- Most string operations are variations on the sequence operations we've seen.
- Example: take a string, break it into lines, indent the lines by $N$ spaces, glue the lines back together, and return the result

```
def indent_lines(s, n):
    """The result of indenting each line in s by n spaces."""
    return "\n".join(map(lambda line: " " * n + line,
                         s.split('\n')))
```

- Use it to indent a file:

```
print(indent_lines(open("afile").read(), 4))
```

- An even more general manipulation: regular expressions:

```
import re
def indent_lines(s, n):
    return re.sub(r'(?m)^', ' ' * n, s)
```

Further exploration left to the reader. E.g., see 13.py

## Observation: Sequences as Conventional Interfaces

- Python 3 defines map, reduce, and filter on sequences just as we did on rlists.
- So to compute the sum of the even Fibonacci numbers among the first 12 numbers of that sequence, we could proceed like this:

```
First 20 integers:
    0   1   2   3   4   5   6   7   8   9   10   11
Map fib:
    0   1   1   2   3   5   8  13  21  34   55   89
Filter to get even numbers:
    0           2           8          34
Reduce to get sum:
    44
```

- ...or:

```
reduce(add, filter(is_even, map(fib, range(12))))
```

- Why is this important? Sequences are amenable to *parallelization*.



## An aside: Streams in Unix

- Many Unix utilities operate on *streams of characters,* which are sequences.
- With the help of pipes, one can do amazing things. One of my favorites:

```
tr -c -s '[:alpha:]' '[\n*]' < FILE | \
sort | \
uniq -c | \
sort -n -r -k 1,1 | \
sed 20q
```

which prints the 20 most frequently occuring words in *FILE*, with their frequencies, most frequent first.