The term sequence refers generally to a data structure consisting of an indexed collection of values. That is, there is a first, second, third value (which CS types call #0, #1, #2, etc.). A sequence may be finite (with a length) or infinite. As an object, it may be mutable (elements can change) or immutable. There are numerous alternative interfaces (i.e., sets of operations) for manipulating it. And, of course, numerous alternative implementations.

Today: immutable, finite sequences, recursively defined. A recursive definition allows us to define structures with recursive thinking. For example, a sequence of sequences is defined recursively as an empty sequence or a first and sequence of the remaining sequences.

A Recursive Definition

A possible definition: a sequence consists of
• an empty sequence, or
• a first element and a sequence consisting of the rest of the sequence or tail.

Example: height of an object
The approach to implementing functions on them
The cases in the recursive definition of list often suggests a recursive algorithm.

Implementation with Pairs

An obvious implementation uses two-element tuples (pairs), such as those defined in lecture 8. The result is a linked list.

Box-and-Pointer Diagrams for Linked Lists

Diagrammatically, one gets structures like this:

Q = make rlist(5, make rlist(3, empty rlist))
L = make rlist(8, make rlist(Q, make rlist(empty rlist, empty rlist)))

or
Q = make rlist(5, make rlist(3))
L = make rlist(8, make rlist(Q, make rlist(empty rlist)))

From Recursive Structure to Recursive Algorithm

The cases in the recursive definition often suggest a recursive approach to implementing functions on them.

Example: length of an rlist:

```python
def len_rlist(s):
    if isempty(s):
        return 0
    else:
        return 1 + len_rlist(rest(s))
```

Q: Why do we know the comment is accurate?
A: Because we assume the comment is accurate!

An example of reasoning by structural induction. . . or recursive thinking about data structures...
Another Example: Selection
• Want to extract item #k from an rlist (number from 0).

Recursively:
```
def getitem_rlist(s, i):
    """Return the element at index 'i' of recursive list 's'.""
    if i == 0:
        return first(s)
    else:
        return getitem_rlist(rest(s), i-1)
```

Iterative Version of getitem_rlist
```
def getitem_rlist(s, i):
    """Return the element at index 'i' of recursive list 's'.""
    while i != 0:
        s, i = rest(s), i-1
    return first(s)
```

On to Higher Orders!

Map implemented
```
def map_rlist(f, s):
    """The rlist of values F(x) for each element x of rlist S (in the same order.)""
    if isempty(s):
        return empty_rlist()
    else:
        return make_rlist(f(first(s)), map_rlist(f, rest(s)))
```

• So map_rlist(lambda x: x**2, L) produces a list of squares.
• [Python 3 produces a different kind of result from its map function; we'll get to it.]
Filtering

Filtering

Filtering (II)

Filtering (III)

Filtering (IV)

Filtering (V)

Filtering

Python's Sequences

Python's Sequences

Python's Sequences

Python's Sequences
Sequence Features

• For now, we emphasize computation by construction rather than modification. The interesting characteristics include:

  – Explicit Construction:
    - \( t = (2, 0, 9, 10, 11) \) # Tuple
    - \( L = [2, 0, 9, 10, 11] \) # List
    - \( R = \text{range}(2, 13) \) # Integers 2-12.
    - \( R_0 = \text{range}(13) \) # Integers 0-12.
    - \( E = \text{range}(2, 13, 2) \) # Even integers 2-12.
    - \( S = \text{"Hello, world!\"} \) # Strings (sequences of characters)

  – Indexing:
    - \( t[-1] == t[len(t)-1] == 11 \)
    - \( S[1] == \text{"e\"} \)

  – Slicing:
    - \( t[1:4] == (t[1], t[2], t[3]) == (0, 9, 10) \)
    - \( t[2:] == t[2:len(t)] == (9, 10, 11) \)
    - \( t[::2] == t[0:len(t):2] == (2, 9, 11) \)
    - \( t[::-1] == (11, 10, 9, 0, 2) \)
    - \( S[0:5] == \text{"Hello\"} \)
    - \( S[0:5:2] == \text{"Hlo\"} \)
    - \( S[4:0:-1] == \text{"olleH\"} \)
    - \( R[2:5] == \text{range}(4, 7) \)
    - \( E[1 : 5] == \text{range}(4, 12, 2) \)

Sequence Combination and Conversion

• Sequence types can be converted into each other where needed:

  - \( \text{list}( (1, 2, 3) ) == [1, 2, 3] \)
  - \( \text{tuple}([1, 2, 3]) == (1, 2, 3) \)
  - \( \text{list}(\text{range}(2, 10, 2)) == [2, 4, 6, 8] \)
  - \( \text{list}(\text{"ABCD\"}) == [\text{\'A\"}, \text{\'B\"}, \text{\'C\"}, \text{\'D\"}] \)

• One can construct certain sequences (tuples, lists, strings) from smaller ones:

  - \( A = [1, 2, 3, 4] \)
  - \( B = [7, 8, 9] \)
  - \( A + B == [1, 2, 3, 4, 7, 8, 9] \)
  - \( A[1:3] + B[1:] == [1, 2, 3, 8, 9] \)
  - \( (1, 2, 3, 4) + (7, 8, 9) == (1, 2, 3, 4, 7, 8, 9) \)
  - \( \text{"Hello,\"} + \text{\" \"} + \text{\"world\"} == \text{\"Hello, world\"} \)
  - \( (1, 2, 3, 4) + 3 \) ERROR (why?)

Sequence Iteration: For Loops

• We can write more compact and clear versions of while loops:

```python
>>> t = (2, 0, 9, 10, 11)
>>> s = 0

>>> for x in t:
    s += x

>>> print(s)
32
```

• Iteration over numbers is really the same, conceptually:

```python
>>> s = 0

>>> for i in range(1, 10):
    s += i

>>> print(s)
45
```

Higher-Order Manipulation of Sequences

• Python 3 defines map (just as on rlists), as well as accumulate (called reduce in the module functools), and filter, 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:

  \( \text{reduce(add, filter(\text{iseven}, \text{map(fib, range(12)}) )}) \)

  # or
  \( \text{sum(filter(\text{iseven}, \text{map(fib, range(12)}) )}) \)

  # Specialized reduction

• Why is this important? Sequences are amenable to parallelization.

List Comprehensions

• In fact, one doesn't often need map and filter because Python has a succinct syntax for expressing their application: the list comprehension.

• Full form:

  \[ <expression> \text{for} <var> \text{in} <sequence expression> \text{if} <boolean expression> \]

• Example: Squares of the prime numbers up to 100.

  \[ [ x*x \text{for} x \text{in} \text{range}(101) \text{if} \text{isprime(x)} ] \]

• A different variety is the generator, which can be useful in reductions:

  \( \text{sum}( ( x*x \text{for} x \text{in} \text{range}(101) \text{if} \text{isprime(x)} ) ) \)

  ... because it does not actually construct the list. More on generators later.

An aside: Sequences in Unix

• Many Unix utilities operate on streams of characters, which are sequences. One of my favorites:

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

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