The Sequence Abstraction

- red, orange, yellow, green, blue, indigo, violet.
- 0, 1, 2, 3, 4, 5, 6.

There isn’t just one sequence type (in Python or in general).
This abstraction is a collection of behaviors:

**Length.** A sequence has a finite length.

**Element selection.** A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

The sequence abstraction is shared among several types, including tuples.

Recursive Lists

**Constructor:**
```python
def rlist(first, rest):
    """Return a recursive list from its first element and the rest."""
```

**Selectors:**
```python
def first(s):
    """Return the first element of recursive list s."""

def rest(s):
    """Return the remaining elements of recursive list s."""
```

**Behavior condition(s):**
If a recursive list s is constructed from a first element f and a recursive list r, then
- first(s) returns f, and
- rest(s) returns r, which is a recursive list.

Implementing Recursive Lists Using Pairs

A recursive list is a pair

- The first element of the pair is the first element of the list
- The second element of the pair is the rest of the list

None represents the empty list

Example: [1, 2, 3, 4]

Implementing the Sequence Abstraction

**Length.** A sequence has a finite length.

**Element selection.** A sequence has an element corresponding to any non-negative integer index less than its length, starting at 0 for the first element.

```python
def len_rlist(s):
    """Return the length of recursive list s.""
    if s == empty_rlist:
        return 0
    return 1 + len_rlist(rest(s))

def getitem_rlist(s, i):
    """Return the element at index i of recursive list s.""
    if i == 0:
        return first(s)
    return getitem_rlist(rest(s), i - 1)
```
**Python Sequence Abstraction**

Built-in sequence types provide the following behavior:

<table>
<thead>
<tr>
<th>Type-specific constructor</th>
<th>Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>a = (1, 2, 3)</code></td>
<td>A list: more on this later</td>
</tr>
<tr>
<td><code>b = tuple([4, 5, 6, 7])</code></td>
<td></td>
</tr>
</tbody>
</table>

- **Length**
  - `len(a), len(b)`
  - `(3, 4)`

- **Element selection**
  - `a[1], b[-1]`
  - `(2, 7)`

- **Slicing**
  - `a[1:3], b[1:1], a[:2], b[1:]`
  - `((2, 3), (), (1, 2), (5, 6, 7))`

- **Membership**
  - `2 in a, 4 in a, 4 not in b`
  - `(True, False, False)`

**Sequence Iteration**

Python has a special statement for iterating over the elements in a sequence:

```python
def count(s, value):
    total = 0
    for elem in s:
        if elem == value:
            total += 1
    return total
```

**For Statement Execution Procedure**

```python
for <name> in <expression>:
    <suite>
```

1. Evaluate the header `<expression>`, which must yield an iterable value.
2. For each element in that sequence, in order:
   A. Bind `<name>` to that element in the first frame of the current environment.
   B. Execute the `<suite>`.

**Sequence Unpacking in For Statements**

```python
for x, y in pairs:
    if x == y:
        same_count += 1
```

- A sequence of fixed-length sequences
- A name for each element in a fixed-length sequence
- Each name is bound to a value, as in multiple assignment

**The Range Type**

A range is a sequence of consecutive integers.

..., -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, ...

```
range(-2, 3)
```

Length: ending value - starting value
Element selection: starting value + index

```python
>>> tuple(range(-2, 3))
(-2, -1, 0, 1, 2)
```

With a 0 starting value

```python
>>> tuple(range(4))
(0, 1, 2, 3)
```

*Ranges can actually represent more general integer sequences.*

**String Literals**

```python
>>> 'I am string!'
'I am string!'
```

Single- and double-quoted strings are equivalent

```python
>>> "I've got an apostrophe" < "I've got an apostrophe"
'您好'
```

"The Zen of Python claims, Readability counts. Read more: import this." ***

A backslash "escapes" the following character

"Line feed" character represents a new line
Strings Are Sequences

```python
>>> city = 'Berkeley'
>>> len(city)
8
```  
The `in` and `not in` operators match substrings

```python
>>> 'here' in 'Where's Waldo?'
True
```

Why? Working with strings, we care about words, not characters

Sequences as Conventional Interfaces

```python
>>> fibs = tuple(map(fib, range(8)))
```  
We can extract elements that satisfy a given condition

```python
>>> even_fibs = tuple(filter(is_even, fibs))
```  
We can compute the sum of all elements

```python
>>> sum(even_fibs)
10
```

Both `map` and `filter` produce an iterable, not a sequence

Generator Expressions

```python
One large expression that combines mapping and filtering to produce an iterable

```  
- Evaluates to an iterable.
- `iter exp` is evaluated when the generator expression is evaluated.
- Remaining expressions are evaluated when elements are accessed.

No-filter version: `(map exp for <name> in <iter exp>)`

Precise evaluation rule introduced in Chapter 4.

Sequence Arithmetic

```python
Some Python sequences support arithmetic operations

```  
```python
>>> city = 'Berkeley'
>>> city + ', CA'
'Berkeley, CA'
```

```python
>>> "Don't repeat yourself! " * 2
"Don't repeat yourself! Don't repeat yourself!"
```

```python
>>> (1, 2, 3) * 3
(1, 2, 3, 1, 2, 3, 1, 2, 3)
```

```python
>>> (1, 2, 3) + (4, 5, 6, 7)
(1, 2, 3, 4, 5, 6, 7)
```

Iterables

```python
Iterables provide access to some elements in order but do not provide length or element selection
```

Many built-in functions take iterables as argument

```python
tuple  Construct a tuple containing the elements
map    Construct a map that results from applying the given function to each element
filter Construct a filter with elements that satisfy the given condition
sum    Return the sum of the elements
min    Return the minimum of the elements
max    Return the maximum of the elements
```

For statements also operate on iterable values.

Reducing a Sequence

```python
Reduce is a higher-order generalization of max, min, and sum.

```  
```python
>>> from operator import mul
>>> from functools import reduce
```  
```python
>>> reduce(mul, (1, 2, 3, 4, 5), 1)
120
```

Like `accumulate` from Homework 2, but with iterables

```python
def accumulate(combiner, start, n, term):
    return reduce(combiner, map(term, range(1, n + 1)), start)
```
Create an iterable of fixed-length sequences
>>> a, b = (1, 2, 3), (4, 5, 6, 7)
>>> for x, y in zip(a, b):
...     print(x + y)
...     5
...     7
...     9

The `itertools` module contains many useful functions for working with iterables
>>> from itertools import product, combinations
>>> tuple(product(a, b[:2]))
((1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5))
>>> tuple(combinations(a, 2))
((1, 2), (1, 3), (2, 3))

Produces tuples with one element from each argument, up to length of smallest argument