What is an abstract data type (ADT)AÇÃO?
• We need to guarantee that constructor and selector functions together specify the right behavior.
• Behavior condition: If we construct rational number \( x \) from numerator \( n \) and denominator \( d \), then \( \text{numerator}(x)/\text{denom}(x) \) must equal \( n/d \).
• An abstract data type is some collection of selectors and constructors, together with some behavior condition(s).
• If behavior conditions are met, the representation is valid.

You can recognize data types by behavior, not by bits.

Tuple-based pair implementation

```python
def pair(x, y):
    """Return a tuple-based pair."""
    return (x, y)

def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p[i]
```

Functional pair implementation

```python
def pair(x, y):
    """Return a functional pair."""
    def dispatch(m):
        if m == 0:
            return x
        elif m == 1:
            return y
    return dispatch

def getitem_pair(p, i):
    """Return the element at index i of pair p."""
    return p[i]
```

The pair ADT
To implement our rational number abstract data type, we used a two-element tuple (also known as a pair).

What is a pair?
Constructors, selectors, and behavior conditions:

- If a pair \( p \) was constructed from elements \( x \) and \( y \), then
  - \*getitem_pair(p, 0) returns \( x \), and
  - \*getitem_pair(p, 1) returns \( y \).

Together, selectors are the inverse of the constructor
Generally true of container types. Not true for rational numbers because of GCD.

Announcements
• Homework 4 due tonight
• Homework 5 is out, due Friday
• Midterm is Thursday, 7pm
• Thanks for coming to the potluck!
Using a pair

```python
>>> p = pair(1, 2)
>>> item_pair(p, 0)
1
>>> item_pair(p, 1)
2
```

If a pair `p` was constructed from elements `x` and `y`, then
- `item_pair(p, 0)` returns `x`, and
- `item_pair(p, 1)` returns `y`.

This pair representation is valid!

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.

Tuples in environment diagrams

Tuples introduce new memory locations outside of a frame

We use `box-and-pointer` notation to represent a tuple
- Tuple itself represented by a set of boxes that hold values
- Tuple value represented by a pointer to that set of boxes

---

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

```python
A recursive list is a pair

1, 2, 3, 4
```

```
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
```

---

Implementing the Sequence Abstraction

```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)
```

**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.
Break!

• We’re transitioning from concepts to Python vocabulary

## Python sequence abstraction

Built-in sequence types provide the following behavior

<table>
<thead>
<tr>
<th>Type-specific constructor</th>
<th><code>a = (1, 2, 3)</code></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>b = tuple([4, 5, 6, 7])</code></td>
<td></td>
</tr>
</tbody>
</table>

### Length

| `len(a)` | `(3,)` |
| `len(b)` | `(4,)` |

### Element selection

<table>
<thead>
<tr>
<th><code>a[1]</code>, <code>b[-1]</code></th>
<th><code>(2, 7)</code></th>
</tr>
</thead>
</table>

### Slicing

<table>
<thead>
<tr>
<th><code>a[1:3]</code>, <code>b[1:1]</code>, <code>a[:2]</code>, <code>b[1:]</code></th>
<th><code>((2, 3), (), (1, 2), (5, 6, 7))</code></th>
</tr>
</thead>
</table>

### Membership

<table>
<thead>
<tr>
<th><code>2 in a</code>, <code>4 in a</code>, <code>4 not in b</code></th>
<th><code>(True, False, False)</code></th>
</tr>
</thead>
</table>

### 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**

```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>`.

   Demo: [http://goo.gl/cWX38](http://goo.gl/cWX38)

### Sequence unpacking in for statements

```python
>>> pairs = [(1, 2), (2, 3), (4, 4)]
```

A name for each element in a fixed-length sequence

```python
>>> same_count = 0
```

A sequence of fixed-length sequences

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

Each name is bound to a value, as in multiple assignment

```python
>>> same_count
2
```

### The range type

A range is a sequence of consecutive integers:

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

Length: ending value - starting value

Element selection: starting value + index

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

Tuple constructor

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

With a 0 starting value

*Ranges can actually represent more general integer sequences.*
String literals

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

>>> "I've got an apostrophe"
"I've got an apostrophe"
```

A backslash "escapes" the following character

"Line feed" character represents a new line

Strings are sequences

```
>>> city = 'Berkeley'
>>> len(city)
8

>>> city[3] < An element of a string is itself a string!
'k'
```

The in and not in operators match substrings

```
>>> 'here' in 'where's Waldo?'
True
```

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

Sequence arithmetic

Some Python sequences support arithmetic operations

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

>>> "Don't repeat yourself! " * 2  Repeat twice
"Don't repeat yourself! Don't repeat yourself!"

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

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

Concatenate

Repeat twice

Sequences as conventional interfaces

We can apply a function to every element in a sequence
This is called mapping the function over the sequence

```
>>> fibs = tuple(map(fib, range(8)))
>>> fibs
(0, 1, 1, 2, 3, 5, 8, 13)
```

We can extract elements that satisfy a given condition

```
>>> even_fibs = tuple(filter(is_even, fibs))
>>> even_fibs
(0, 2, 8)
```

We can compute the sum of all elements

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

Both map and filter produce an iterable, not a sequence

Iterables

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

Python-specific construct; more general than a sequence

Many built-in functions take iterables as argument

- 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.

Generator expressions

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

```
(\langle map exp \rangle for \langle name \rangle in \langle iter exp \rangle if \langle filter exp \rangle)
```

• Evaluates to an iterable.
• \langle iter exp \rangle is evaluated when the generator expression is evaluated.
• Remaining expressions are evaluated when elements are accessed.

No-filter version: (\langle map exp \rangle for \langle name \rangle in \langle iter exp \rangle)

Precise evaluation rule introduced in Chapter 4.
More Functions on Iterables (Bonus)

Create an iterable of fixed-length sequences

```python
>>> 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

```python
>>> from itertools import product, combinations

>>> tuple(product(a, b[:2]))
((1, 4), (1, 5), (2, 4), (2, 5), (3, 4), (3, 5))
```

```python
>>> tuple(combinations(a, 2))
((1, 2), (1, 3), (2, 3))
```