Announcements

- HW5 due on Wednesday

- Trends project out
  - Partners are required; find one in lab or on Piazza
  - Will not work in IDLE
  - New bug submission policy; see Piazza
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
- \(\text{first}(s)\) returns \(f\), and
- \(\text{rest}(s)\) 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: [http://goo.gl/fVhbF](http://goo.gl/fVhbF)
Implementing the Sequence Abstraction
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.
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))
```

**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.
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.
Python Sequence Abstraction
Built-in sequence types provide the following behavior.
### Python Sequence Abstraction

Built-in sequence types provide the following behavior

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

<table>
<thead>
<tr>
<th>Length</th>
<th>&gt;&gt;&gt; len(a), len(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(3, 4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element selection</th>
<th>&gt;&gt;&gt; a[1], b[-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2, 7)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slicing</th>
<th>&gt;&gt;&gt; a[1:3], b[1:1], a[:2], b[1:]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((2, 3), (), (1, 2), (5, 6, 7))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Membership</th>
<th>&gt;&gt;&gt; 2 in a, 4 in a, 4 not in b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(True, False, False)</td>
</tr>
</tbody>
</table>
# Python Sequence Abstraction

Built-in sequence types provide the following behavior:

| Type-specific constructor | >>> \(a = (1, 2, 3)\)  
<table>
<thead>
<tr>
<th></th>
<th>&gt;&gt;&gt; (b = \text{tuple}([4, 5, 6, 7]))</th>
</tr>
</thead>
<tbody>
<tr>
<td>Length</td>
<td>&gt;&gt;&gt; (\text{len}(a), \text{len}(b))</td>
</tr>
<tr>
<td></td>
<td>(3, 4)</td>
</tr>
<tr>
<td>Element selection</td>
<td>&gt;&gt;&gt; (a[1], b[-1])</td>
</tr>
<tr>
<td></td>
<td>(2, 7)</td>
</tr>
<tr>
<td>Slicing</td>
<td>&gt;&gt;&gt; (a[1:3], b[1:1], a[:2], b[1:])</td>
</tr>
<tr>
<td></td>
<td>((2, 3), (), (1, 2), (5, 6, 7))</td>
</tr>
<tr>
<td>Membership</td>
<td>&gt;&gt;&gt; (2 \text{ in } a, 4 \text{ in } a, 4 \text{ not in } b)</td>
</tr>
<tr>
<td></td>
<td>(True, False, False)</td>
</tr>
</tbody>
</table>
## Python Sequence Abstraction

Built-in sequence types provide the following behavior

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

<table>
<thead>
<tr>
<th>Length</th>
<th>&gt;&gt;&gt; len(a), len(b)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(3, 4)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Element selection</th>
<th>&gt;&gt;&gt; a[1], b[-1]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(2, 7)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Slicing</th>
<th>&gt;&gt;&gt; a[1:3], b[1:1], a[:2], b[1:]</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>((2, 3), (), (1, 2), (5, 6, 7))</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Membership</th>
<th>&gt;&gt;&gt; 2 in a, 4 in a, 4 not in b</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(True, False, False)</td>
</tr>
</tbody>
</table>
Sequence Iteration
Python has a special statement for iterating over the elements in a sequence
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
```
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
```

Name bound in the first frame of the current environment
For Statement Execution Procedure
for <name> in <expression>:  
<suite>
for <name> in <expression>:
    <suite>

1. Evaluate the header <expression>, which must yield an iterable value.
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
Sequence Unpacking in For Statements

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

>>> same_count = 0
```
Sequence Unpacking in For Statements

A sequence of fixed-length sequences

```python
>>> pairs = [(1, 2), (2, 2), (2, 3), (4, 4)]
>>> same_count = 0
```
Sequence Unpacking in For Statements

A sequence of fixed-length sequences

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

>>> same_count = 0

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

>>> same_count
2
```
A sequence of fixed-length sequences

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

>>> same_count = 0

A name for each element in a fixed-length sequence

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

>>> same_count
2
A sequence of fixed-length sequences

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

>>> same_count = 0

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

>>> same_count
2
```
The Range Type
The Range Type

A range is a sequence of consecutive integers.*
The Range Type

A range is a sequence of consecutive integers.*

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

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

* Ranges can actually represent more general integer sequences.
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)

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

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

\[\text{range}(-2, 3)\]

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

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

\texttt{range(-2, 3)}

* Ranges can actually represent more general integer sequences.
The Range Type

A range is a sequence of consecutive integers.*

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

\[ \text{range}(-2, 3) \]

* Ranges can actually represent more general integer sequences.
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

* Ranges can actually represent more general integer sequences.
A range is a sequence of consecutive integers.*

\[ \ldots, -5, -4, -3, -2, -1, 0, 1, 2, 3, 4, 5, \ldots \]

Length: ending value - starting value

Element selection: starting value + index

* Ranges can actually represent more general integer sequences.
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

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

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

* Ranges can actually represent more general integer sequences.
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)
```

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

* Ranges can actually represent more general integer sequences.
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

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

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

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

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

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

>>> '您好'
'您好'
```
String Literals

```python
>>> 'I am string!'
'I am string!'
>>> "I've got an apostrophe"
"I've got an apostrophe"
>>> '您好'
'您好'
```

Single- and double-quoted strings are equivalent
String Literals

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

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

>>> '您好'
'您好'

>>> """The Zen of Python claims, Readability counts. Read more: import this.""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.'
String Literals

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

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

>>> '您好'
'您好'

>>> """The Zen of Python
claims, Readability counts.
Read more: import this."""
'The Zen of Python\\nclaims, Readability counts.\\nRead more: import this.'
```

A backslash "escapes" the following character

Single- and double-quoted strings are equivalent
String Literals

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

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

>>> '您好'
'您好'

>>> """The Zen of Python claims, Readability counts. Read more: import this."""
'The Zen of Python\nclaims, Readability counts.\nRead more: import this.

A backslash "escapes" the following character

"Line feed" character represents a new line

Single- and double-quoted strings are equivalent
Strings Are Sequences
Strings Are Sequences

```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```
Strings Are Sequences

```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!
Strings Are Sequences

```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

The `in` and `not in` operators match substrings
Strings Are Sequences

```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

The `in` and `not in` operators match substrings

```python
>>> 'here' in "Where's Waldo?"
True
```
Strings Are Sequences

```python
>>> city = 'Berkeley'
>>> len(city)
8
>>> city[3]
'k'
```

An element of a string is itself a string!

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
Sequence Arithmetic

Some Python sequences support arithmetic operations
Sequence Arithmetic

Some Python sequences support arithmetic operations

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

Some Python sequences support arithmetic operations

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

Some Python sequences support arithmetic operations

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

>>> "Don't repeat yourself! " * 2
"Don't repeat yourself! Don’t repeat yourself! "
```
Sequence Arithmetic

Some Python sequences support arithmetic operations

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

>>> "Don't repeat yourself! " * 2
"Don't repeat yourself! Don’t repeat yourself! "
```
Sequence Arithmetic

Some Python sequences support arithmetic operations

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

>>> "Don't repeat yourself! " * 2
"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)
```
Sequences as Conventional Interfaces
Sequences as Conventional Interfaces

We can apply a function to every element in a sequence
Sequences as Conventional Interfaces

We can apply a function to every element in a sequence
This is called *mapping* the function over the sequence
Sequences as Conventional Interfaces

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

```python
>>> fibs = tuple(map(fib, range(8)))
>>> fibs
(0, 1, 1, 2, 3, 5, 8, 13)
```
Sequences as Conventional Interfaces

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

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

We can extract elements that satisfy a given condition
Sequences as Conventional Interfaces

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

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

We can extract elements that satisfy a given condition

```python
>>> even_fibs = tuple(filter(is_even, fibs))
>>> even_fibs
(0, 2, 8)
```
Sequences as Conventional Interfaces

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

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

We can extract elements that satisfy a given condition:

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

We can compute the sum of all elements.
We can apply a function to every element in a sequence. This is called *mapping* the function over the sequence.

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

We can extract elements that satisfy a given condition.

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

We can compute the sum of all elements.

```python
>>> sum(even_fibs)
10
```
Sequences as Conventional Interfaces

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

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

We can extract elements that satisfy a given condition:

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

We can compute the sum of all elements:

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

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

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

Python-specific construct; more general than 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.
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
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
Generator Expressions

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

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

\[(\text{<map exp>} \text{ for } \text{<name>} \text{ in } \text{<iter exp>} \text{ if } \text{<filter exp>})\]
Generator Expressions

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

\[(<\text{map \ exp}> \ \text{for} \ <\text{name}> \ \text{in} \ <\text{iter \ exp}> \ \text{if} \ <\text{filter \ exp}>)\]

• Evaluates to an iterable.
Generator Expressions

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

\[(<\text{map exp}> \text{ for } <\text{name}> \text{ in } <\text{iter exp}> \text{ if } <\text{filter exp}>)\]

• Evaluates to an iterable.
• \(<\text{iter exp}>\) is evaluated when the generator expression is evaluated.
Generator Expressions

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

```
(<map exp> for <name> in <iter exp> if <filter exp>)
```

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

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

`(map exp for name in iter exp if filter exp)`

• 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)`
Generator Expressions

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

\((\text{map exp} \text{ for } \<name> \text{ in } \text{iter exp} \text{ if } \text{filter exp})\)

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

No-filter version: \((\text{map exp} \text{ for } \<name> \text{ in } \text{iter exp})\)

Precise evaluation rule introduced in Chapter 4.
Reducing a Sequence
Reducing a Sequence

Reduce is a higher-order generalization of max, min, and sum.
Reducing a Sequence

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

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

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

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

First argument:
A two-argument function
Reducing a Sequence

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

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

First argument: A two-argument function

Second argument: an iterable object
Reducing a Sequence

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

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

First argument: A two-argument function

Second argument: an iterable object

Optional initial value as third argument
Reducing a Sequence

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

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

Like accumulate from Homework 2, but with iterables

- First argument: A two-argument function
- Second argument: an iterable object
- Optional initial value as third argument
Reducing a Sequence

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

```python
>>> from operator import mul
>>> from functools import reduce
>>> 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)
```
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))
>>> tuple(combinations(a, 2))
((1, 2), (1, 3), (2, 3))
```

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