Lecture #12: Python Sequences: Tuples
Recursive Lists vs. Tuples

- Rlists tend to divide problems into “what to do with the first item” and “what to do with the rest of the list.”
- This reflects the operations on them (first, rest, rlist).
- But accessing items of a Python tuple is uniform (x[i]), and the style of algorithms is correspondingly different.
For loops

- Python `for` loops operate on sequences of various types:

  ```python
  for targets in sequence expression:
    repeated statements
  ```

- First, evaluate `sequence expression` to get a sequence of values.
- Then, for each value, \( V \), in that sequence (left-to-right):
  - Assign \( V \) to `targets`.
  - Execute the `repeated statements`.

- Usually, `targets` is just a simple variable,
- But can be anything that can go to the left of an assign operator.
### Examples

<table>
<thead>
<tr>
<th>Program</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>L = (1, 2, 3)</code></td>
<td>1</td>
</tr>
<tr>
<td><code>for i in L:</code></td>
<td>2</td>
</tr>
<tr>
<td><code>print(i)</code></td>
<td>3</td>
</tr>
<tr>
<td><code>for i in 1, 2, 3:</code></td>
<td></td>
</tr>
<tr>
<td># Same as for i in (1, 2, 3):</td>
<td></td>
</tr>
<tr>
<td><code>print(i)</code></td>
<td></td>
</tr>
<tr>
<td><code>for p in (0, &quot;Fwd&quot;), (2, &quot;Back&quot;), (3, &quot;Turn&quot;):</code></td>
<td>Fwd 0</td>
</tr>
<tr>
<td><code>print(p[1], p[0])</code></td>
<td>Back 2</td>
</tr>
<tr>
<td><code>print(p[1], p[0])</code></td>
<td>Turn 3</td>
</tr>
<tr>
<td><code>for lft, rght in (0, &quot;Fwd&quot;), (2, &quot;Back&quot;), (3, &quot;Turn&quot;):</code></td>
<td>Fwd 0</td>
</tr>
<tr>
<td><code>print(rght, lft)</code></td>
<td>Back 2</td>
</tr>
<tr>
<td><code>print(rght, lft)</code></td>
<td>Turn 3</td>
</tr>
</tbody>
</table>
Ranges

- A range in Python is a kind of sequence, and works that way in a for loop.

```python
>>> range(1, 5) # The integers from 1 up to but not including 5.
range(1, 5)
>>> range(5) # Shorthand for range(0, 5)
range(0, 5)
>>> L = range(1, 5)
>>> tuple(L) # Convert to tuple
(1, 2, 3, 4)
>>> L[2]
3
>>> len(L)
4
>>> tuple(range(0, 10, 2))
(0, 2, 4, 6, 8)
>>> tuple(range(5, 0, -1))
(5, 4, 3, 2, 1)
>>> for i in range(3, 8):
...     print(i, end="; ")
3; 4; 5; 6; 7;
```
Operations on Python Sequences

• **Add up the values in sequence L:**

```python
sum = 0
for p in L:
    sum += p  # Or sum = sum + p
```

• **reduction**: generalization to operations other than addition:

```python
from operator import *

def reduce(f, seq, init):
    """If SEQ is a sequence of length n>=0, returns sn, where s0 = INIT, s1=F(s0, SEQ[0]), s2=F(s1, SEQ[1]), ..."
    result = ___
    for p in seq:
        result = __________
    return result
```

```python
>>> L = (2, 3, 4)
>>> reduce(add, L, 0)
9
>>> reduce(mul, L, 1)
24
>>> reduce(lambda x, y: rlist(y, x), L, empty_rlist)
(4, (3, (2, None)))
"""
```
Operations on Python Sequences

• Add up the values in sequence \( L \):

```python
sum = 0
for p in L:
    sum += p  # Or sum = sum + p
```

• \textit{reduction}: generalization to operations other than addition:

```python
from operator import *
def reduce(f, seq, init):
    """If SEQ is a sequence of length \( n \geq 0 \), returns \( s_n \), where \( s_0 = INIT \), \( s_1 = F(s_0, SEQ[0]) \), \( s_2 = F(s_1, SEQ[1]) \), ...\n    >>> L = (2, 3, 4)
    >>> reduce(add, L, 0)
    9
    >>> reduce(mul, L, 1)
    24
    >>> reduce(lambda x, y: rlist(y, x), L, empty_rlist)
    (4, (3, (2, None)))
    """
    result = init
    for p in seq:
        result = __________
    return result
```
Operations on Python Sequences

- **Add up the values in sequence** \( L \):

  ```python
  sum = 0
  for p in L:
      sum += p  # Or sum = sum + p
  ```

- **reduction**: generalization to operations other than addition:

  ```python
  from operator import *
  def reduce(f, seq, init):
      # (See also functools.reduce)
      """If SEQ is a sequence of length n>=0, returns s_n, where
      s_0 = INIT, s_1=F(s_0, SEQ[0]), s_2=F(s_1, SEQ[1]), ...
      >>> L = (2, 3, 4)
      >>> reduce(add, L, 0)
      9
      >>> reduce(mul, L, 1)
      24
      >>> reduce(lambda x, y: rlist(y, x), L, empty_rlist)
      (4, (3, (2, None)))
      """
      result = init
      for p in seq:
          result = f(result, p)
      return result
  ```
Building Tuples: the Basics

- As for **rlists**, can construct tuples from other tuples:

  ```python
  >>> L = (1, 2)
  >>> L + (3, 4)  # Extend
  (1, 2, 3, 4)
  >>> L + (5,)
  (1, 2, 5)
  >>> R = (3, 4, 5, 6, 7, 8)
  >>> R[0:3] + L + R[4:]  
  ```
Building Tuples: the Basics

• As for rlists, can construct tuples from other tuples:

```python
>>> L = (1, 2)
>>> L + (3, 4)  # Extend
(1, 2, 3, 4)
>>> L + (5,)
(1, 2, 5)
>>> R = (3, 4, 5, 6, 7, 8)
>>> R[0:3] + L + R[4:]
(3, 4, 5, 1, 2, 7, 8)
```
Building Tuples: ‘map’ and ‘filter’

So we could write these (they’re actually builtin functions):

```python
def map(f, seq):
    """Assuming SEQ is the sequence containing s1, s2, ..., sn,
    returns the tuple (F(s1), F(s2), ..., F(sn))."""
    result = ()
    for p in seq:
        result = result + (f(p), )
    return result

def filter(pred, seq):
    """Assuming SEQ is the sequence containing s1, s2, ..., sn,
    returns tuple containing only those si for which PRED(si) is true."""
    result = ()
    for p in seq:
        result = result + (f(p), )
    return result
```
Building Tuples: ‘map’ and ‘filter’

- So we could write these (they’re actually built-in functions):

```python
def map(f, seq):
    """Assuming SEQ is the sequence containing s1, s2, ..., sn,
    returns the tuple (F(s1), F(s2), ..., F(sn)).""
    result = ()
    for p in seq:
        result = result + (f(p), )
    return result

def filter(pred, seq):
    """Assuming SEQ is the sequence containing s1, s2, ..., sn,
    returns tuple containing only those si for which PRED(si) is true.""
    result = ()
    for p in seq:
        result = result + (p,) if pred(p) else result
    return result
```
Basic Comprehensions

- But building large tuples this way becomes horrendously slow, much slower than the analogous operations on rlists [why?].
- We’ll see one way to deal with that problem in the next lecture (mutable sequences), but for now…
- Python has a couple of ways of specifying a list in one expression:

  \[
  ( \text{expression} \text{ for } \text{targets} \text{ in } \text{sequence expression} )
  \]

  creates a sequence that’s kind of like

  \[
  \text{seq} = ()
  \]

  for \text{targets} in \text{sequence expression}:

  \[
  \text{seq} = \text{seq} + (\text{expression}, )
  \]

  …but much faster. (It’s actually a \textit{generator}, not a tuple, but can be converted to one with \texttt{tuple}. More on this later.)

- For example,

  >>> \texttt{tuple((k * k for k in range(5)))}
  
  \[
  (0, 1, 4, 9, 16)
  \]
More Elaborate Comprehensions

• It's possible to use multiple for clauses as well:
  ```python
  >>> tuple( ((i, j) for i in range(2) for j in range(3)) )
  ((0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2))
  >>> tuple( ((i, j) for i in range(3) for j in range(i + 1)) )
  ((0, 0), (1, 0), (1, 1), (2, 0), (2, 1), (2, 2))
  ```

• Finally, if clauses filter a range:
  ```python
  >>> tuple( ( k for k in range(50) if k % 7 == 0 or k % 11 == 0 ) )
  (0, 7, 11, 14, 21, 22, 28, 33, 35, 42, 44, 49)
  # filter is built in
  >>> tuple(filter(lambda x: k % 7 == 0 or k % 11 == 0, range(50)))
  (0, 7, 11, 14, 21, 22, 28, 33, 35, 42, 44, 49)
  # Now you try to get the same result:
  >>> def filter(pred, seq): return __________________________
  ```
More Elaborate Comprehensions

• It's possible to use multiple for clauses as well:

```python
>>> tuple( ((i, j) for i in range(2) for j in range(3)) )
((0, 0), (0, 1), (0, 2), (1, 0), (1, 1), (1, 2))
>>> tuple( ((i, j) for i in range(3) for j in range(i + 1)) )
((0, 0), (1, 0), (1, 1), (2, 0), (2, 1), (2, 2))
```

• Finally, if clauses filter a range:

```python
>>> tuple( ( k for k in range(50) if k % 7 == 0 or k % 11 == 0 ) )
(0, 7, 11, 14, 21, 22, 28, 33, 35, 42, 44, 49)
# filter is built in
>>> tuple(filter(lambda x: k % 7 == 0 or k % 11 == 0, range(50)))
(0, 7, 11, 14, 21, 22, 28, 33, 35, 42, 44, 49)
# Now you try to get the same result:
>>> def filter(pred, seq): return (x for x in seq if pred(x))
```
def partition(L, x):
    """Returns result of rearranging the elements of L so that all items < X appear before all items >= X, and all are otherwise in their original order.
    >>> L = (0, 9, 6, 2, 5, 11, 1)
    >>> partition(L, 5)
    (0, 2, 1, 9, 6, 5, 11)
    """
    return
A Sequence Problem

def partition(L, x):
    """Returns result of rearranging the elements of L so that all items < X appear before all items >= X, and all are otherwise in their original order.
>>> L = (0, 9, 6, 2, 5, 11, 1)
>>> partition(L, 5)
(0, 2, 1, 9, 6, 5, 11)
"""
    return tuple((y in L if y < x)) + tuple((y in L if y >= x))
def collapse_runs(L):
    """Return result of removing the second and subsequent consecutive duplicates of values in L,
    >>> x = (1, 2, 2, 1, 1, 1, 2, 0, 0)
    >>> collapse_runs(x)
    (1, 2, 1, 2, 0)
    """
    return

def collapse_runs(L):
    """Return result of removing the second and subsequent consecutive duplicates of values in L,
    >>> x = (1, 2, 2, 1, 1, 1, 2, 0, 0)
    >>> collapse_runs(x)
    (1, 2, 1, 2, 0)
    """
    return tuple((L[k] for k in range(len(L)) if k==0 or L[k-1]!=L[k]))