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:

```
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

Program	Output

`L = (1, 2, 3)`
`for i in L:`
`  print(i)`
	1
	2
	3

`for i in 1, 2, 3:`
`# Same as for i in (1, 2, 3):`
`  print(i)`
	1
	2
	3

`for p in (0, "Fwd"), (2, "Back"), (3, "Turn"):` `  print(p[1], p[0])`	Fwd 0 Back 1 Turn 3
`for lft, rght in (0, "Fwd"), (2, "Back"), (3, "Turn"):` `  print(rght, lft)`	Fwd 0 Back 1 Turn 3

Ranges

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

```
>>> 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:

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

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

```
from operator import *
def reduce(f, seq, init):  # (See also functools.reduce)
    """If SEQ is a sequence of length n>=0, returns sn, where
    s0 = INIT, s1=F(s0, SEQ[0]), s2=F(s1, 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 = ____
    for p in seq:
        result = ____________
```

Operations on Python Sequences

- Add up the values in sequence L:

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

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

```
from operator import *
def reduce(f, seq, init):  # (See also functools.reduce)
    """If SEQ is a sequence of length n>=0, returns sn, where
    s0 = INIT, s1=F(s0, SEQ[0]), s2=F(s1, 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 = _____________
```

Operations on Python Sequences

- Add up the values in sequence L:

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

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

```
from operator import *
def reduce(f, seq, init):  # (See also functools.reduce)
    """If SEQ is a sequence of length n>=0, returns sn, where
    s0 = INIT, s1=F(s0, SEQ[0]), s2=F(s1, 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)
```

Building Tuples: the Basics

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

```
>>> 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:

```
>>> 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, 6, 7, 8)
```

Building Tuples: map and filter

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

```
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(f, pred):
    """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 = _____________________________________
    return result
```

Building Tuples: map and filter

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

```
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(f, pred):
    """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:

  ```
  ( expression for targets in sequence expression }
  ```

  creates a sequence that's kind of like

  ```
  seq = ()
  for targets in sequence expression:
      seq = seq + (expression, )
  ```

  ...but much faster. (It's actually a *generator*, not a tuple, but can be converted to one with tuple. More on this later.)

- For example,

  ```
  >>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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:

```
>>> 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))
```

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 _____
```

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))
```

Another Sequence Problem

- For this one, you'll probably want to use the operator **in** on sequences (and other things):

```
>>> 3 in (1, 3, 8, 7)
True
>>> 2 not in (1, 3, 8, 7)
True
>>> 2 in (1, 3, 8, 7)
False
>>> 2 in (x in range(10) if x % 3 == 0)
False
```

```
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 _____
```

Another Sequence Problem

- For this one, you'll probably want to use the operator **in** on sequences (and other things):

```
>>> 3 in (1, 3, 8, 7)
True
>>> 2 not in (1, 3, 8, 7)
True
>>> 2 in (1, 3, 8, 7)
False
>>> 2 in (x in range(10) if x % 3 == 0)
False
```

```
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]) )
```