**5.4 Range and enumerate**

We have seen that the **range** function generates a sequence of integers and that when called as r**ange(n)**, it generates the sequence of integers from 0 through n-1. The type returned from **range** is **range**, although it can be passed to the **list** function if a list consisting of the individual items is desired.

*>>> mr = range(6)*

*>>> print(mr, type(mr))*

range(0, 6) <class 'range'>

*>>> ml = list(mr)*

*>>> print(ml, type(ml))*

[0, 1, 2, 3, 4, 5] <class 'list'>

So, again, **range** does not create a list; it creates an iterator. An interesting aspect of iterators is that the values are created one at a time; the entire iterator is not created first.

The built-in function **iter** will determine whether an argument can be iterated through or not, and if so what type of iterator the argument is. For example, the range variable *mr* is a range iterator, a list [3, 66] is a list iterator, and an integer is not an iterator.

*>>> iter(mr)*

<range\_iterator at 0x7f888bb480f0>

*>>> iter([3, 66])*

<list\_iterator at 0x7f888bb48070>

*>>> iter(33)*

TypeError: 'int' object is not iterable

By storing the iterator object in a variable, the **next** function returns the next value in the iterator, e.g.

*>>> imr = iter(mr)*

*>>> next(imr)*

0

*>>> next(imr)*

1

*>>> next(imr)*

2

When there are no more values, an error message is thrown if **next** is called again.

If two arguments are passed to the **range** function, they are the first value (instead of the default of 0) and the last (minus 1). For example, range(3,7) generates the sequence of integers 3, 4, 5, 6.

*>>> mr = range(3,7)*

*>>> ml = list(mr)*

*>>> print(ml)*

[3, 4, 5, 6]

An integer “step” value can also be specified as a third argument (instead of the default of 1).

*>>> mr = range(3,10,2)*

*>>> ml = list(mr)*

*>>> print(ml)*

[3, 5, 7, 9]

Step values can also be negative.

*>>> ml = list(range(10,2,-2))*

*>>> print(ml)*

[10, 8, 6, 4]

The **range** function is frequently used with **for** loops to specify how many times to execute the action of a loop. More generally, the form of a **for** loop is

for itervar in iterator:

action

The **enumerate** function can be used to return both an index and an item from a sequence such as a list.

*numlist = [4, 52, 33, 11, -3]*

*for i, item in enumerate(numlist):*

*print('Item', i, 'is', item)*

Item 0 is 4

Item 1 is 52

Item 2 is 33

Item 3 is 11

Item 4 is -3

Notice that this gives us two iterator variables: *i*, which is the index, and then *item*, which is the value of that item in the list.

**5.5 Tuples**

Tuples are similar to lists, but are immutable. Tuples are generally created by putting values in parentheses, separated by commas.

*>>> mytup = (2, 11, 33)*

Functions such as **len** and indexing/slicing work the same on tuples as on lists.

*>>> len(mytup)*

3

*>>> mytup[1]*

11

Parentheses are not always necessary:

*>>> newtuple = 5, 19*

*>>> newtuple*

(5, 19)

The concatenation operator can be used to join two tuples together.

*>>> mytup + newtuple*

(2, 11, 33, 5, 19)

An ***empty tuple*** is created using parentheses with nothing inside, e.g.:

*>>> emptup = ()*

*>>> len(emptup)*

0

To create a tuple with one entry, the value must be followed by a comma, and both must be in parentheses.

*>>> onetup = (7,)*

*>>> onetup*

(7,)

The parentheses and comma here are necessary. If the comma is omitted, the value is just an integer:

*>>> print(type((7,)), type((7)))*

<class 'tuple'> <class 'int'>

The type of a tuple is **tuple**, as shown.

Because of this, indexing into a tuple to get one value is not quite the same as slicing to get one value. A slice returns another tuple, even if it only has a length of 1:

*>>> mytup = (2, 11, 33)*

*>>> mytup[0:1]*

(2,)

Indexing using 0 returns just the integer 2:

*>>> mytup[0]*

2

The **list** function can be used to create a list from a tuple:

*>>> tl = list(mytup)*

*>>> tl*

[2, 11, 33]

The **tuple** function can be used to create a tuple from another sequence type such as a string or a list:

*>>> wordlist = ['howdy', 'hi']*

*>>> wordtuple = tuple(wordlist)*

*>>> chartuple = tuple(wordlist[0])*

*>>> print(wordlist, wordtuple, chartuple)*

['howdy', 'hi'] ('howdy', 'hi') ('h', 'o', 'w', 'd', 'y')

Putting values into a tuple is called ***packing***, as in:

*>>> mytup = 2, 11, 33*

or

*>>> mytup = (2, 11, 33)*

The reverse is called ***unpacking***. In order to unpack a tuple, it is necessary to know how many values are in the tuple and to have that many variables on the left-hand side of the assignment operator.

*>>> a, b, c = mytup*

*>>> print(mytup, a, b, c)*

(2, 11, 33) 2 11 33

Unpacking is also possible for other sequence types such as lists and strings.