Chapter_10_Lists

March 6, 2024

A for loop over an empty list never runs the body:

[1]:
```
for x in []:
    print('This never happens.')
```

The + operator concatenates lists:

[2]:
```
a = [1, 2, 3]
>>> b = [4, 5, 6]
>>> c = a + b
c
```

[2]:
```
[1, 2, 3, 4, 5, 6]
```

The * operator repeats a list a given number of times:

[3]:
```
[0] * 4
```

[3]:
```
[0, 0, 0, 0]
```

[4]:
```
[1, 2, 3] * 3
```

[4]:
```
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

[5]:
```
>>> t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3]
['b', 'c']
>>> t[:4]
['a', 'b', 'c', 'd']
>>> t[3:]
['d', 'e', 'f']
```

[5]:
```
['d', 'e', 'f']
```

If you omit the first index, the slice starts at the beginning. If you omit the second, the slice goes to the end. So if you omit both, the slice is a copy of the whole list.

[6]:
```
t[:]
```

[6]:
```
['a', 'b', 'c', 'd', 'e', 'f']
```

[7]:
```
 t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> t[1:3] = ['x', 'y']
>>> t
```

[7]:
```
['a', 'x', 'y', 'd', 'e', 'f']
```

Python provides methods that operate on lists. For example, append adds a new element to the end of a list:

[8]:
```
t = ['a', 'b', 'c']
>>> t.append('d')
>>> t
```

[8]:
```
['a', 'b', 'c', 'd']
```

extend takes a list as an argument and appends all of the elements:

[9]:
```
 t1 = ['a', 'b', 'c']
>>> t2 = ['d', 'e']
>>> t1.extend(t2)
>>> t1
```

[9]:
```
['a', 'b', 'c', 'd', 'e']
```

This example leaves t2 unmodified.

sort arranges the elements of the list from low to high:

[10]:
```
t = ['d', 'c', 'e', 'b', 'a']
>>> t.sort()
>>> t
```

[10]:
```
['a', 'b', 'c', 'd', 'e']
```

Most list methods are void; they modify the list and return None. If you accidentally write t = t.sort(), you will be disappointed with the result.

[11]:
```
t = t.sort()
```

[12]:
```
type(t)
```

[12]:
```
NoneType
```

0.1 Map, filter and reduce

To add up all the numbers in a list, you can use a loop like this:

[13]:
```
def add_all(t):
    total = 0
    for x in t:
        total += x # augmented assignment statement
```

```
    return total
```

Adding up the elements of a list is such a common operation that Python provides it as a built-in function, sum:

[14]:
```
t = [1, 2, 3]
>>> sum(t)
```

[14]:
```
6
```

An operation like this that combines a sequence of elements into a single value is sometimes called reduce.

Sometimes you want to traverse one list while building another. For example, the following function takes a list of strings and returns a new list that contains capitalized strings:

[15]:
```
def capitalize_all(t):
    res = []
    for s in t:
        res.append(s.capitalize())
    return res
```

res is initialized with an empty list; each time through the loop, we append the next element. So res is another kind of accumulator.

An operation like capitalize_all is sometimes called a map because it "maps" a function (in this case the method capitalize) onto each of the elements in a sequence.

Another common operation is to select some of the elements from a list and return a sublist. For example, the following function takes a list of strings and returns a list that contains only the uppercase strings:

[16]:
```
def only_upper(t):
    res = []
    for s in t:
        if s.isupper():
            res.append(s)
    return res
```

An operation like only_upper is called a filter because it selects some of the elements and filters out the others

0.2 Deleting elements

There are several ways to delete elements from a list. If you know the index of the element you want, you can use pop:

[17]:
```
t = ['a', 'b', 'c']
x = t.pop(1)
t
```

[17]:
```
['a', 'c']
```

[18]:
```
x
```

[18]:
```
'b'
```

pop modifies the list and returns the element that was removed. If you don't provide an index, it deletes and returns the last element.

If you don't need the removed value, you can use the del operator:

If you know the element you want to remove (but not the index), you can use remove:

[19]:
```
t = ['a', 'b', 'c']
>>> t.remove('b')
>>> t
```

[19]:
```
['a', 'c']
```

The return value from remove is None.

To remove more than one element, you can use del with a slice index:

[20]:
```
t = ['a', 'b', 'c', 'd', 'e', 'f']
>>> del t[1:5]
>>> t
```

[20]:
```
['a', 'f']
```

The list function breaks a string into individual letters. If you want to break a string into words, you can use the split method:

[21]:
```
s = 'pining for the fjords'
>>> t = s.split()
>>> t
```

[21]:
```
['pining', 'for', 'the', 'fjords']
```

An optional argument called a delimiter specifies which characters to use as word boundaries. The following example uses a hyphen as a delimiter:

[22]:
```
 s = 'spam-spam-spam'
>>> delimiter = '-'
>>> t = s.split(delimiter)
>>> t
```

[22]:
```
['spam', 'spam', 'spam']
```

join is the inverse of split. It takes a list of strings and concatenates the elements. join is a string method, so you have to invoke it on the delimiter and pass the list as a parameter:

[23]:
```
t = ['pining', 'for', 'the', 'fjords']
>>> delimiter = ' '
>>> s = delimiter.join(t)
>>> s
```

[23]:
```
'pining for the fjords'
```

[24]:
```
>>> a = 'banana'
>>> b = 'banana'
>>> a is b
```

[24]:
```
True
```

In this example, Python only created one string object, and both a and b refer to it. But when you create two lists, you get two objects:

[25]:
```
a = [1, 2, 3]
>>> b = [1, 2, 3]
>>> a is b
False
```

[25]:
```
False
```

In this case we would say that the two lists are equivalent, because they have the same elements, but not identical, because they are not the same object. If two objects are identical, they are also equivalent, but if they are equivalent, they are not necessarily identical.

[26]:
```
 a = [1, 2, 3]
>>> b = a
>>> b is a
```

[26]:
```
True
```

The association of a variable with an object is called a reference. In this example, there are two references to the same object.

An object with more than one reference has more than one name, so we say that the object is aliased. If the aliased object is mutable, changes made with one alias affect the other:

When you pass a list to a function, the function gets a reference to the list. If the function modifies the list, the caller sees the change. For example, delete_head removes the first element from a list:

[27]:
```
def delete_head(t):
    del t[0]
```

[28]:
```
letters = ['a', 'b', 'c']
>>> delete_head(letters)
>>> letters
```

[28]:
```
['b', 'c']
```

It is important to distinguish between operations that modify lists and operations that create new lists. For example, the append method modifies a list, but the + operator creates a new list.

[29]:
```
t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
```

[29]:
```
[1, 2, 3]
```

[30]:
```
print(t2)
```

```
None
```

The return value from append is None.

[31]:
```
t3 = t1 + [4]
>>> t1
```

[31]:
```
[1, 2, 3]
```

[32]:
```
t3
```

[32]:
```
[1, 2, 3, 4]
```

The result of the operator is a new list, and the original list is unchanged. This difference is important when you write functions that are supposed to modify lists

[ ]:
```

```