Python
Lists

Stéphane Vialette

LIGM, Université Paris-Est Marne-la-Vallée

September 25, 2012

Outline

Outline

Lists

Description

- Python knows a number of compound data types, used to group together other values.
- The most versatile is the list, which can be written as a list of comma-separated values (items) between square brackets.
- List items need not all have the same type.

Example

```
>>> l = [1, 2, "ab", "cd"]
>>> l
[1, 2, 'ab', 'cd']
>>> type(l)
<type 'list'>
```

Empty list and length

Example

```
>>> # empty list
... []
[]
>>> type([])
<type 'list'>
>>>
>>> # The built-in function len() return the length
... len([])
0
>>> len([1, 2, 3, 4])
4
>>> len([[]])
1
>>> len([[[[[[[]]]]]]])
1
>>>
```

First and last elements

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> # First element is at position 0
... l[0]
'one'
>>> # Last element is at poisition len() - 1
... l[len(l) - 1]
'five'
>>> # Out of range
... l[len(l)]
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IndexError: list index out of range
>>> # Oups ... more on this latter
... l[-1]
'five'
>>>
```

Slice

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> # From the first element to the last one
... l[0:len(l)]
['one', 'two', 'three', 'four', 'five']
>>> # Only shorter
... l[:]
['one', 'two', 'three', 'four', 'five']
>>>
```

Slice

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>>
>>> l[0:]
['one', 'two', 'three', 'four', 'five']
>>> l[1:]
['two', 'three', 'four', 'five']
>>> l[2:]
['three', 'four', 'five']
>>> l[3:]
['four', 'five']
>>> l[4:]
['five']
>>> l[5:]
[]
>>> l[6:]
[]
>>>
```

Slice

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> l[:0]
[]
>>> l[:1]
['one']
>>> l[:2]
['one', 'two']
>>> l[:3]
['one', 'two', 'three']
>>> l[:4]
['one', 'two', 'three', 'four']
>>> l[:5]
['one', 'two', 'three', 'four', 'five']
>>> l[:6]
['one', 'two', 'three', 'four', 'five']
>>>
```

Slice

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> # From the second element to the fourth
... l[1:4]
['two', 'three', 'four']
>>> # The third element, i.e., l[2]
... l[2:3]
['three']
>>> # From the second element to the last one
... l[1:]
['two', 'three', 'four', 'five']
>>> # Idem
... l[1:len(l)]
['two', 'three', 'four', 'five']
>>> # From the first element to the third one
... l[:3] # or l[0:3]
['one', 'two', 'three']
>>>
```

Assignment

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> l[0] = 'zero'
>>> l
['zero', 'two', 'three', 'four', 'five']
>>> l[1] = l[0]
>>> l
['zero', 'zero', 'three', 'four', 'five']
>>> l[2] = ['six', 'seven']
>>> l
['zero', 'zero', ['six', 'seven'], 'four', 'five']
>>> l[3] = l
>>> l
['zero', 'zero', ['six', 'seven'], [...], 'five']
>>> l[3]
['zero', 'zero', ['six', 'seven'], [...], 'five']
>>> l[3][3][3][3][3][3][3][3][3][3][3][3][3][3][3][3][3][3][3][3]
['zero', 'zero', ['six', 'seven'], [...], 'five']
```

Assignment and slide

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> l[2:4] = ['six', 'seven']
>>> l
['one', 'two', 'six', 'seven', 'five']
>>> l[2:4] = ['height']
>>> l
['one', 'two', 'height', 'five']
>>> l[1:3] = []
>>> l
['one', 'five']
>>> l[2:] = l
>>> l
['one', 'five', 'one', 'five']
>>> l[1:3] = l[:3]
>>> l
['one', 'one', 'five', 'one', 'five']
>>> l[:] = [] # l is now the empty list []
```

Negative index

Example

```
>>> l = ['one', 'two', 'three', 'four', 'five']
>>> l[-1], l[-2], l[-3], l[-4], l[-5]
('five', 'four', 'three', 'two', 'one')
>>> l[-6]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
IndexError: list index out of range
>>> l[1:-1]
['two', 'three', 'four']
>>> l[1:-1] = l[2:-2]
>>> l
['one', 'three', 'five']
>>> l[-0]
'one'
>>> l[-0] == l[0], l[+0] == l[0], l[-0] == l[+0]
(True, True, True)
>>>
```

Concatening lists

Example

```
>>> l1 = ['one', 'two']
>>> l2 = ['three', 'four']
>>> l1, l2
(['one', 'two'], ['three', 'four'])
>>> l1 + l2
['one', 'two', 'three', 'four']
>>> l1 + l2 + ['five', 'six', 'seven']
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
>>> l3 = l1 + l2 + ['five', 'six', 'seven']
>>> l1, l2
(['one', 'two'], ['three', 'four'])
>>> l3
['one', 'two', 'three', 'four', 'five', 'six', 'seven']
>>> l1 + [], [] + l1
(['one', 'two'], ['one', 'two'])
>>> [] + [] + []
[]
```

Outline

The list data type has some more methods

Methods

- `list.append(x)`
  Add an item to the end of the list
  Equivalent to `l[len(l):] = [x]`.
- `list.extend(ll)`
  Extend the list by appending all the items in the given list.
  Equivalent to `l[len(l):] = ll`.
- `list.insert(i, x)`
  Insert an item at a given position. The first argument is the index of the element before which to insert, so `l.insert(0, x)` inserts at the front of the list, and `l.insert(len(l), x)` is equivalent to `l.append(x)`.

Methods

Example

```
>>> l = []
>>> l.append(1)
>>> l
[1]
>>> l.append(2)
>>> l.extend([3, 4])
>>> l
[1, 2, 3, 4]
>>> l.extend(l)
>>> l
[1, 2, 3, 4, 1, 2, 3, 4]
>>> l.insert(0, 0)
>>> l
[0, 1, 2, 3, 4, 1, 2, 3, 4]
>>> l.insert(5, 5)
>>> l
[0, 1, 2, 3, 4, 5, 1, 2, 3, 4]
```

The list data type has some more methods

Methods

- `list.remove(x)`
  Remove the first item from the list whose value is `x`. It is an error if there is no such item.
- `list.pop([i])`
  Remove the item at the given position in the list, and return it. If no index is specified, `l.pop()` removes and returns the last item in the list. (The square brackets around the `i` in the method signature denote that the parameter is optional, not that you should type square brackets at that position.
- `list.index(x)`
  Return the index in the list of the first item whose value is `x`. It is an error if there is no such item.

Methods

Example

```
>>> l = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> l.remove(2)
>>> l
[1, 3, 1, 2, 3, 1, 2, 3]
>>> l.remove(2)
>>> l
[1, 3, 1, 3, 1, 2, 3]
>>> l.remove(2)
>>> l
[1, 3, 1, 3, 1, 3]
>>> l.remove(2)
ValueError: list.remove(x): x not in list
>>> l.pop()
3
```

Methods

Example

```
>>> l
[1, 3, 1, 3, 1]
>>> l.pop(1)
3
>>> l
[1, 1, 3, 1]
>>> l.index(1)
0
>>> l.index(3)
2
>>> l[2:].index(1)
1
>>> l.index(2)
ValueError: list.index(x): x not in list
>>> l.index(l.pop())
0
```

The list data type has some more methods

Methods

- `list.count(x)`
  Return the number of times `x` appears in the list.
- `list.sort()`
  Sort the items of the list, in place.
- `list.reverse()`
  Reverse the elements of the list, in place.
- `list.remove(x)`
  Remove the first item from the list whose value is `x`. It is an error if there is no such item.

Methods

Example

```
>>> l = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> l.count(1), l.count(2), l.count(3)
(3, 3, 3)
>>> l.count(4)
0
>>> l.sort()
>>> l
[1, 1, 1, 2, 2, 2, 3, 3, 3]
>>> l.reverse()
>>> l
[3, 3, 3, 2, 2, 2, 1, 1, 1]
>>> l = [1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> l.remove(2)
>>> l
[1, 3, 1, 2, 3, 1, 2, 3]
>>> l.remove(4)
ValueError: list.remove(x): x not in list
```

Outline

Lists as stacks

Example

```
>>> l = []
>>> l.append(1), l.append(2)
>>> l
[1, 2]
>>> l.pop()
2
>>> l
[1]
>>> l.append(3), l.append(4)
>>> l.pop()
4
>>> l
[1, 3]
>>> l.pop(), l.pop()
(3, 1)
>>> l.pop()
IndexError: pop from empty list
```

Lists as queues

Example

```
>>> l = []
>>> l.append(1), l.append(2)
>>> l
[1, 2]
>>> l.pop(0)
1
>>> l
[2]
>>> l.append(3), l.append(4)
>>> l.pop(0)
2
>>> l
[3, 4]
>>> l.pop(0), l.pop(0)
(3, 4)
>>> l.pop(0)
IndexError: pop from empty list
```

Outline

Functional programming

Functions

There are three built-in functions that are very useful when used with lists:

- `filter(function, sequence)`
  returns a sequence consisting of those items from the sequence for which `function(item)` is true.
- `map(function, sequence)`
  calls `function(item)` for each of the sequence's items and returns a list of the return values.
- `reduce(function, sequence)`
  returns a single value constructed by calling the binary function `function` on the first two items of the sequence, then on the result and the next item, and so on.

Functional programming

Example

```
>>> def is_even(x):
...     return x % 2 == 0
...
>>> l = range(12)
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> filter(is_even, l)
[0, 2, 4, 6, 8, 10]
>>> l
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> def f(x):
...     return x % 2 != 0 and x % 3 != 0
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
>>> filter(lambda _: True, range(12))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Functional programming

Example

```
>>> def succ(x):
...     return x+1
>>> map(succ, range(10))
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> map(succ, map(succ, range(10)))
[2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
>>> def square(x):
...     return x*x
>>> map(square, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> map(square, map(succ, range(10)))
[1, 4, 9, 16, 25, 36, 49, 64, 81, 100]
>>> def cube(x):
...     return square(x)*x
>>> map(cube, range(10))
[0, 1, 8, 27, 64, 125, 216, 343, 512, 729]
```

Functional programming

Example

```
>>> def add(x, y):
...     return x+y
>>> reduce(add, range(4))
6
>>> reduce(lambda x, y:x+y, range(4))
6
>>> # A third argument can be passed to indicate the starting value
>>> def my_sum(l):
...     return reduce(lambda x, y: x+y, l, 0)
>>> def my_product(l):
...     return reduce(lambda x, y: x*y, l, 1)
>>> my_sum(range(6))
15
>>> my_product(range(1,6))
120
```

Functional programming

Example

```
>>> def generic(l, f, e):
...     return reduce(f, l, e)
>>> generic(range(1,6), lambda x, y: x+y, 0)
15
>>> generic(range(1,6), lambda x, y: x*y, 1)
120
>>>
>>> def my_sum(l):
...     return generic(l, lambda x, y: x+y, 0)
>>> def my_product(l):
...     return generic(l, lambda x, y: x*y, 1)
>>> my_sum(range(1,6))
15
>>> my_product(range(1,6))
120
```