Lists

Chapter 10

A List is a Sequence

- Sequence of values of any type (**elements** or **items**)

```
[10, 20, 30, 40]
['SDU', 'KBTU', 'NU']

['university', 1996, 4.0, [20, 25]]
```

- **Nested** list

```
[]
```

- **Empty** list

A List is a Sequence (cont.)

```
>>> cheeses = [' Cheddar' , ' Edam' , ' Gouda' ]
>>> numbers = [42, 123]
>>> empty = []

>>> print(cheeses, numbers, empty)
[' Cheddar' , ' Edam' , ' Gouda' ] [42, 123] []
```

Lists are Mutable

```
>>> cheeses[0]
' Cheddar'

>>> numbers = [42, 123]
>>> numbers[1] = 5
>>> numbers
[42, 5]
```

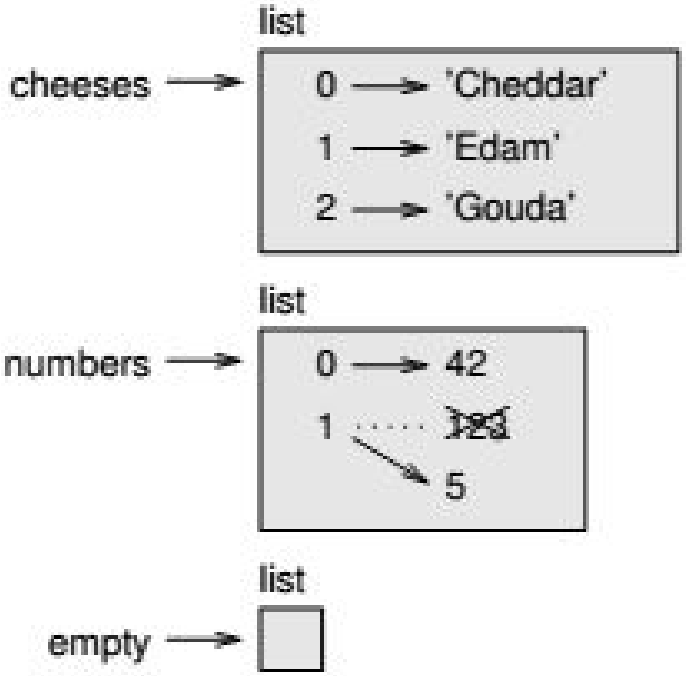

Figure 10.1: State diagram.

`in` Operator

```
>>> cheeses = [' Cheddar' , ' Edam' , ' Gouda' ]

>>> ' Edam' in cheeses
True

>>> ' Brie' in cheeses
False
```

Traversing a List

- To read:

```
for cheese in cheeses:
    print(cheese)
```

- To modify:

```
for i in range(len(numbers)):
    numbers[i] = numbers[i] * 2
```

Traversing a List (cont.)

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


t = [True, False, [‘x’,’y’], [20, 25, 30]]
>>> len(t)
```

List Operations

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

>>> [0] * 4
[0, 0, 0, 0]
>>> [1, 2, 3] * 3          # repetition
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List Slices

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

- Make a copy before modifying

List Slices (cont.)

```
>>> t = [' a' , ' b' , ' c' , ' d' , ' e' , ' f' ]

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

>>> t[1:3] = [1,2,3]
>>> t
???
```

List Methods

```
>>> t1 = [' a' , ' b' , ' c' ]
>>> t1.append(' d' )
>>> t1
[' a' , ' b' , ' c' , ' d' ]

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

>>> t2
```

List Methods (cont.)

```
>>> t = [' d' , ' c' , ' e' , ' b' , ' a' ]


>>> t.sort()
>>> t
[' a' , ' b' , ' c' , ' d' , ' e' ]

>>> t = t.sort()
>>> t
```

- Most list methods are void; they modify the list and return `None`.

Reduce, Map, and Filter

- Reduce

```
def add_all(t) :
    total = 0          # accumulator
    for x in t:
        total += x     # total = total + x
    return total


>>> t = [1, 2, 3]
>>> sum(t)
6
```

Reduce, Map, and Filter (cont.)

- Map

```
def capitalize_all(t):
    res = []          # a kind of accumulator
    for s in t:
        res.append(s.capitalize())
    return res
```

Reduce, Map, and Filter (cont.2)

- Filter

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

- Most common list operations can be expressed as a combination of map, filter and reduce.

Deleting Elements

- If you know the index:

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

>>> x = t.pop()
```

Deleting Elements (cont.)

- If you know the element to remove (but not the index):

```
>>> t = [' a' , ' b' , ' c' ]
>>> t.remove(' b' )          # returns None
>>> t
[' a' , ' c' ]
```

Deleting Elements (cont.2)

- If you don't need the removed value:

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


>>> del t[1:4]
>>> t
[' a' , ' f' ]
```

Lists and Strings

- To convert from a string to a list:

```
>>> s = ' spam'
>>> t = list(s)
>>> t
[' s' , ' p' , ' a' , ' m' ]
```

- Avoid using `list` or `l` as a list variable name

Lists and Strings (cont.)

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

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

Lists and Strings (cont.2)

```
>>> s = ' spam-spam-spam'
>>> delimiter = '-'          # a character used for splitting
>>> t = s.split(delimiter) # optional argument
>>> t
[' spam' , ' spam' , ' spam' ]
```

Lists and Strings (cont.3)

```
>>> t = [' pining' , ' for' , ' the' , ' fjords' ]
>>> delimiter = ' '
>>> s = delimiter.join(t)  # inverse of split method
>>> s
' pining for the fjords'
```

Objects and Values

```
a = ' banana'
b = ' banana'
```

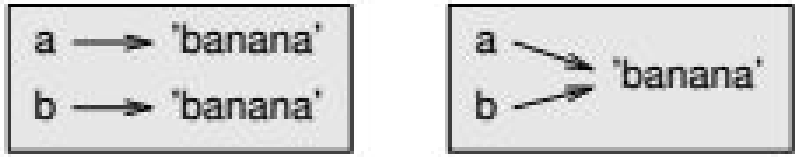

Figure 10.2: State diagram.

```
>>> a is b
```

Objects and Values (cont.)

```
>>> a = [1, 2, 3]
>>> b = [1, 2, 3]

>>> a is b
False
```

Figure 10.3: State diagram.

- The two lists are **equivalent**, but not **identical**

Aliasing

```
>>> a = [1, 2, 3]
>>> b = a

>>> b is a
True
```

Figure 10.4: State diagram.

- Two **references** to the same object.
- An object with more than one references is called to be **aliased** (has several names)

Aliasing (cont.)

```
>>> a = [1, 2, 3]
>>> b = a

>>> b[0] = 42
>>> a
```

- It is safer to avoid aliasing.

```
a = ' banana'
b = ' banana'
```

List Arguments

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

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

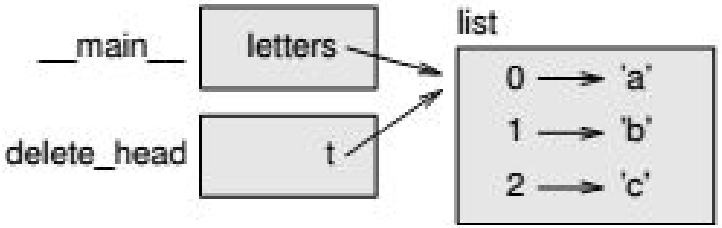

Figure 10.5: Stack diagram.

List Arguments (cont.)

```
def bad_delete_head(t):
    t = t[1:]          # WRONG!
```

```
>>> t4 = [1, 2, 3]
>>> bad_delete_head(t4)
>>> t4
[1, 2, 3]
```

List Arguments (cont.2)

```
def tail(t):
    return t[1:]

>>> letters = [' a' , ' b' , ' c' ]
>>> rest = tail(letters)
>>> rest
[' b' , ' c' ]
```