Think Python 2e, Chapter 10 Notes

Lists

October 4, 2022

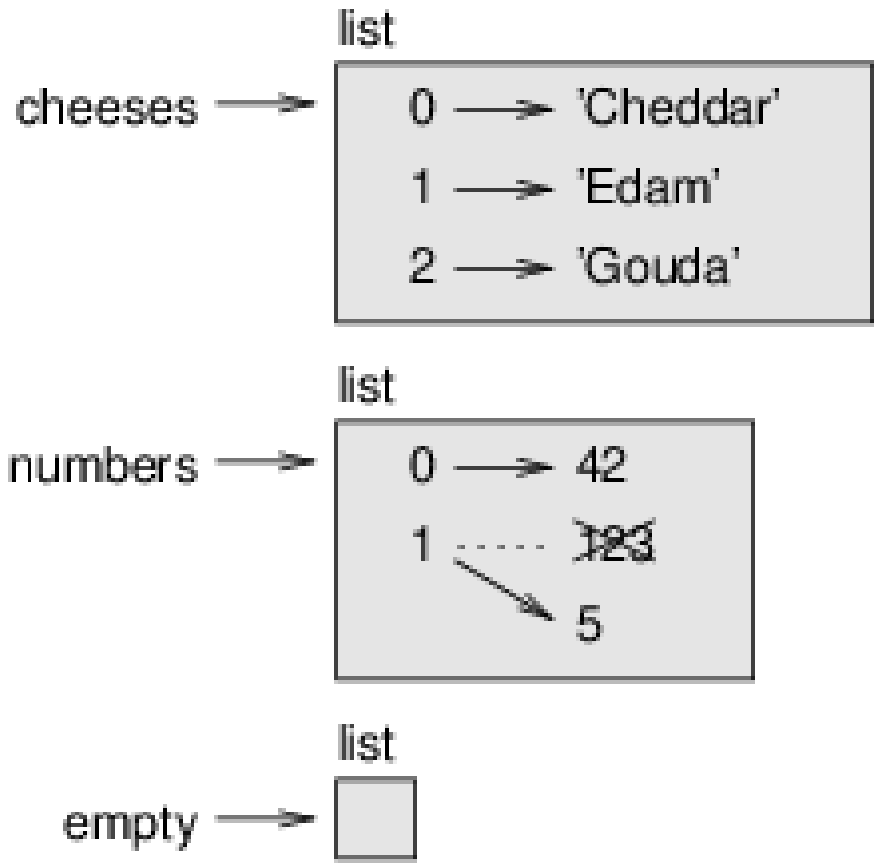
Lists

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> numbers = [42, 123]
>>> empty = []
>>> print(cheeses, numbers, empty)
['Cheddar', 'Edam', 'Gouda'] [42, 123] []
>>> mixed_list = ['spam', 2.0, 5, [10, 20]]
```

- A sequence of values, called **elements** or **items**
- Values of elements in a list can be anything.
- A list in a list is called **nested**
- The empty list is just `[]`

Lists are mutable

```
1 >>> cheeses = ['Cheddar', '
      Edam', 'Gouda']
2 >>> numbers = [42, 123]
3 >>> empty = []
4 >>> numbers[1] = 5
```



Indexing the same in lists and strings

- Any integer expression can be used as an index.
- If you try to read or write an element that does not exist, you get an IndexError.
- If an index has a negative value, it counts backward from the end of the list.
- Slicing works the same.

```
>>> foo = ['hello', 'there', 'how', 'are', 'you?']
>>> foo[2:4]
['how', 'are']
```

The in operator also works on lists.

```
>>> cheeses = ['Cheddar', 'Edam', 'Gouda']
>>> 'Edam' in cheeses
True
>>> 'Brie' in cheeses
False
```

The math operators also work on lists.

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
>>> [1, 2, 3] * 3
[1, 2, 3, 1, 2, 3, 1, 2, 3]
```

List traversal

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

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

Traversal only happens at top level

```
>>> x = [1,2,3]
>>> y = ['hello','bye']
>>> a = [99, x, y, 100]
>>> a
[99, [1, 2, 3], ['hello', 'bye'], 100]
>>> for item in a:
                print(item)
99
[1, 2, 3]
['hello', 'bye']
100
```

List methods

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

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

t2 is unmodified

List methods

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

List methods

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

NO! $\Rightarrow$ `t = t.sort()` $\Leftarrow$ BAD!

Reduce

Converting a list of numbers to a single number.

```
1 def add_all(t):
2     total = 0
3     for x in t:
4         total += x
5     return total
```

The augmented assignment statement:

```
1 total += x
```

equivalent to

```
1 total = total + x
```

Reduce

Converting a list of numbers to a single number.

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

The augmented assignment statement:

```
total += x
```

equivalent to

```
total = total + x
```

This is so common python provides a builtin function:

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

Reduce

Converting a list of strings to a single string.

```
1 def cat_all(t):
2     total = ''
3     for x in t:
4         total += x
5     return total
```

```
1 >>> cat_all(['hello','there','how','are','you'])
2 'hellotherehowareyou'
```

Reduce

Converting a list of strings to a single string.

```
def cat_all(t):
    total = ''
    for x in t:
        total += x
    return total
```

```
>>> cat_all(['hello','there','how','are','you'])
'hellotherehowareyou'
```

This is so common python provides a builtin:

```
>>> x = ['hello', 'there', 'how', 'are', 'you']
>>> ''.join(x)\
'hello/there/how/are/you'
>>> '   '.join(x)
'hello   there   how   are   you'\
```

Map

Traverse one list while building another.

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

```
>>> capitalize_all(['hello','there'])
['Hello', 'There']
```

Filter

Select only some elements of a list.

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

```
>>> only_upper(['hello', 'BOB', 'hello', 'HELLO']
['BOB', 'HELLO']
```

Map, Filter, Reduce

- Paradigms of functional programming
- Structure complex programs
- Used in large-scale parallel programming

Deleting elements: pop

Deletes and returns one element.

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

Without optional parameter, deletes the last one

```
>>> t = [1,2,3]
>>> t.pop()
3
>>> t
[1, 2]
```

Deleting elements: `del`

If you don't need the removed value, use `del`

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

Can delete slices

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

Deleting elements: `remove`

If you know the element but not the index, use `remove`

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

The return value is `None`

Lists and Strings

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

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

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

Lists and Strings

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

Objects and values

```
1 >>> a = 'banana'
2 >>> b = 'banana'
3 >>> a == b
4 True
```

Which is the case?

Objects and values

```
>>> a = 'banana'
>>> b = 'banana'
>>> a == b
True
```

Which is the case?

Check with `is` operator:

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

Python only creates one string.

Lists are different

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

- They are **equivalent** but not **identical**.
- We say that an **object** has a **value**.

Aliasing

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

- Association of a variable with an object is called a **reference**
- If there is more than one reference, an object is **aliased**.
- Changes to one change all aliases:

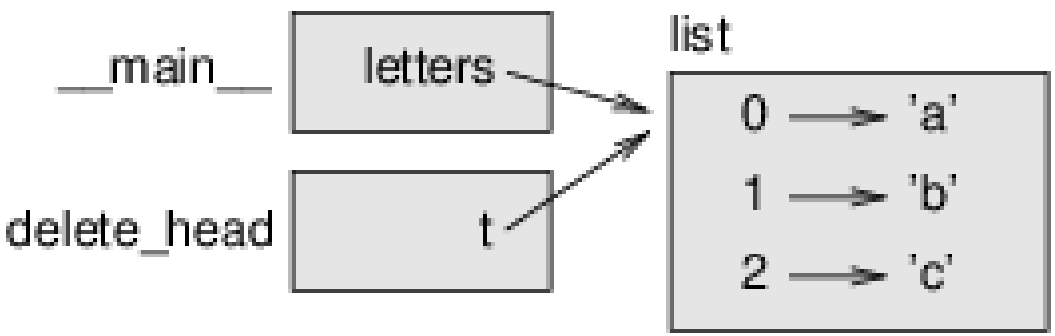
```
>>> b[0] = 42
>>> a
[42, 2, 3]
```

- Aliasing is dangerous

List arguments are aliased

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

```
1 >>> letters = ['a', 'b', 'c']
2 >>> delete_head(letters)
3 >>> letters
4 ['b', 'c']
```



Destructive operations

```
>>> t1 = [1, 2]
>>> t2 = t1.append(3)
>>> t1
[1, 2, 3]
>>> t2
None
>>> t3 = t1 + [4]
>>> t1
[1, 2, 3]
>>> t3
[1, 2, 3, 4]
```

- The append operation destroys the original list `t1`
- The + operation does not

Functions that (don't) modify lists

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

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

Usually better to create new lists

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

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

Most list methods return None

```
1 word = word.strip()          # RIGHT!
2 t = t.sort()                 # WRONG!
```

Most list methods return None

```
word = word.strip()          # RIGHT!
t = t.sort()                 # WRONG!
```

Sometimes desctructive and nondestrucitve versions exist

```
>>> t = [3,5,2,1,4]
>>> s = sorted(t)
>>> t
 [3,5,2,1,4]
>>> s
[1,2,3,4,5]
```

Pick an idiom and stick with it

- There are too many confusing ways to do things sometimes.
- Removing elements from lists:
  - `pop`
  - `remove`
  - `del`
  - slice assignment

Pick an idiom and stick with it

- There are too many confusing ways to do things sometimes.
- Removing elements from lists:
  - `pop`
  - `remove`
  - `del`
  - slice assignment
- **Pick one!**

Make copies to avoid aliasing

```
>>> t = [3, 1, 2]
>>> t2 = t[:]
>>> t2.sort()
>>> t
[3, 1, 2]
>>> t2
[1, 2, 3]
```

Vocabulary

list: A sequence of values.

element: One of the values in a list (or other sequence), also called items.

nested list: A list that is an element of another list.

accumulator: A variable used in a loop to add up or accumulate a result.

augmented assignment: A statement that updates the value of a variable using an operator like $+=$.

Vocabulary

reduce: A processing pattern that traverses a sequence and accumulates the elements into a single result.

map: A processing pattern that traverses a sequence and performs an operation on each element.

filter: A processing pattern that traverses a list and selects the elements that satisfy some criterion.

Vocabulary

object: Something a variable can refer to. An object has a type and a value.

equivalent: Having the same value.

identical: Being the same object (which implies equivalence).

reference: The association between a variable and its value.

aliasing: A circumstance where two or more variables refer to the same object.

delimiter: A character or string used to indicate where a string should be split.