The Usefulness of Python: Containers

Yung-Yu Chen

yyc@solvcon.net

Python Hsinchu User Group

June 3, 2013

What's a Container

- OK, here's some C++ code for an `int` list:

```
1  #include <iostream>
2  #include <list>
3  void print(std::list<int> a) {
4      for(std::list<int>::iterator it=a.begin();
5              it!=a.end(); it++)
6          std::cout << ' ' << *it;
7      std::cout << '\n';
8  };
9  int main() {
10     std::list<int> a;
11     // put 9,8,7,6,5,4,3,2,1 onto the list.
12     for(int i=0; i<9;++i)
13         a.push_back(9-i);
14     print(a); // here the list contains (9,8,7,6,5,4,3,2,1).
15     a.sort(); // in the <list> library!
16     print(a); // here the list contains (1,2,3,4,5,6,7,8,9).
17 };
```

Code is taken from `http://www.csci.csusb.edu/dick/samples/stl.html#Lists`.

What Does It Take for a List?

- Many many things.
- A very simple (and primitive) implementation takes 157 lines of C code.
- The first 10 lines:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdbool.h>
4
5  struct test_struct {
6      int val;
7      struct test_struct *next;
8  };
9
10 struct test_struct *head = NULL;
```

- And it's also a list for `int` only!

Code is taken from `http://www.thegeekstuff.com/2012/08/c-linked-list-example/`.

Let's See a Python List

- 2 (or 1?) lines of code:

```
1  a = [9, 8, 7, 6, 5, 4, 3, 2, 1]
2  print(type(a), a, sorted(a), list(reversed(sorted(a))))
```

  - This slide becomes a bit too blank.
- It's so good to have Python.

What Are Python Containers?

- In fact there is no such a type called "container" in Python.
  - It's just a common name to refer objects that can hold other objects.
  - Like `list`, `stack`, `vector`, `map`, etc., in C++.
  - The dynamicity makes containers in Python more powerful than in C++.
- Three categories of Python containers:
  - Sequence types (`list`, `tuple`, etc.)
  - Set types (`set` and `fronzeset`).
  - Mapping types (`dict`).
- They fit your brains.

list: A General Sequence Type

- str might be the most used sequence type, but it is for character strings only.
  - list can hold anything.
- [] or list() constructs a list for you:

```
1  >>> la = []
2  >>> lb = list()
3  >>> print(la, lb)
4  ([], [])
```

- Some built-ins return a list:

```
1  >>> a = range(10)
2  >>> print(type(a), a)
3  (<type 'list'>, [0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Sort a list

- Simple sort:

```
1  >>> a = [87, 82, 38, 56, 84]
2  >>> b = sorted(a) # b is a new list.
3  >>> print(b)
4  [38, 56, 82, 84, 87]
5  >>> a.sort() # this method does in-place sort.
6  >>> print(a)
7  [38, 56, 82, 84, 87]
```

- Not-so-simple sort:

```
1  >>> a = [('a', 0), ('b', 2), ('c', 1)]
2  >>> print(sorted(a)) # sorted with the first value.
3  [('a', 0), ('b', 2), ('c', 1)]
4  >>> print(sorted(a, key=lambda k: k[1])) # use the second.
5  [('a', 0), ('c', 1), ('b', 2)]
```

List Comprehension

- List comprehension is a very useful technique to construct a list from another iterable:

```
1  >>> values = [10.0, 20.0, 30.0, 15.0]
2  >>> print([it/10 for it in values])
3  [1.0, 2.0, 3.0, 1.5]
```

- List comprehension can even be nested:

```
1  >>> values = [[10.0, 1.0], [20.0, 2.0], [30.0, 3.0], [15.0,
       1.5]]
2  >>> print([jt for it in values for jt in it])
3  [10.0, 1.0, 20.0, 2.0, 30.0, 3.0, 15.0, 1.5]
```

But usually nested list comprehension is not easy to understand, and not a good idea.

Recipe: Calculation

- Built-in calculation functions for iterables:

```
1  >>> values = [10.0, 20.0, 30.0, 15.0]
2  >>> min(values), max(it for it in values)
3  (10.0, 30.0)
4  >>> sum(values)
5  75.0
6  >>> sum(values)/len(values)
7  18.75
```

tuple: Immutable Sequence

- A tuple can also hold anything, but cannot be changed once constructed.
- It can be created with () or tuple():

```
>>> ta = (1)
>>> print(type(ta), ta)
(<type 'int'>, 1)
>>> ta = (1,)
>>> print(type(ta), ta)
(<type 'tuple'>, (1,))
>>> ta[0] = 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

set: Distinct Elements

- A set holds any hashable element, and its elements are distinct:

```
>>> sa = {1, 2, 3}
>>> print(type(sa), sa)
(<type 'set'>, set([1, 2, 3]))
>>> print({1, 2, 2, 3}) # no duplication is possible.
set([1, 2, 3])
>>> len({1, 2, 2, 3})
3
```

- It's unordered:

```
>>> [it for it in {3, 2, 1}]
[1, 2, 3]
>>> [it for it in {3, 'q', 1}]
['q', 1, 3]
>>> 'q' < 1
False
```

Add/Remove Elements

- Add elements after construction of the set:

```
>>> sa = {1, 2, 3}
>>> sa.add(1)
>>> sa
set([1, 2, 3])
>>> sa.add(10)
>>> sa
set([1, 2, 3, 10])
```

- Remove elements:

```
>>> sa = {1, 2, 3, 10}
>>> sa.remove(5) # err with non-existing element
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 5
>>> sa.discard(2) # really discard an element
>>> sa
set([1, 3, 10])
```

Set Operations

- Subset or superset:

```
1  >>> {1, 2, 3} < {2, 3, 4, 5} # subset
2  False
3  >>> {2, 3} < {2, 3, 4, 5}
4  True
5  >>> {2, 3, 4, 5} > {2, 3} # superset
6  True
```

- Union and intersection:

```
1  >>> {1, 2, 3} | {2, 3, 4, 5} # union
2  set([1, 2, 3, 4, 5])
3  >>> {1, 2, 3} & {2, 3, 4, 5} # intersection
4  set([2, 3])
5  >>> {1, 2, 3} - {2, 3, 4, 5} # difference
6  set([1])
```

Recipe: Count Unique Elements

- A set can be used with a sequence to quickly calculate unique elements:

```
>>> data = [1, 2.0, 0, 'b', 1, 2.0, 3.2]
>>> sorted(set(data))
[0, 1, 2.0, 3.2, 'b']
```

- But there's a problem: It doesn't support unhashable objects:

```
>>> data = [dict(a=200), 1, 2.0, 0, 'b', 1, 2.0, 3.2]
>>> set(data)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'dict'
```

Read the Python Cookbook for a solution :-)

dict: The Mapping Type

- A dict stores any number of key-value pairs. It is the most used Python container since it's everywhere for Python namespace.

```
>>> {'a': 10, 'b': 20} == dict(a=10, b=20)
True
>>> da = {1: 10, 2: 20} # any hashable can be a key
>>> da[1] + da[2]
30
>>> class SomeClass(object):
...     pass
...
>>> print(type(SomeClass().__dict__))
<type 'dict'>
```

Access Contents

- To test whether something is in a dictionary or not:

```
>>> da = {1: 10, 2: 20}
>>> 3 in da
False
```

- Access a key-value pair:

```
>>> da[3] # it fails for 3 is not in the dictionary
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 3
>>> print(da[3] if 3 in da else 30) # works but wordy
30
>>> da.get(3, 30) # it's the way to go
30
>>> da # indeed we don't have 3 as a key
{1: 10, 2: 20}
>>> da.setdefault(3, 30) # how about this?
30
>>> da # we added 3 into the dictionary!
{1: 10, 2: 20, 3: 30}
```

Iterating a dict

- Iterating a dict automatically gives you its keys:

```
1  >>> da = {1: 10, 2: 20}
2  >>> ','.join('%s'%key for key in da)
3  '1,2'
4  >>> ','.join('%d'%da[key] for key in da)
5  '10,20'
```

- items() and iteritems() give you both key and value at once:

```
1  >>> da.items() # returns a list
2  [(1, 10), (2, 20)]
3  >>> da.iteritems() # returns an iterator
4  <dictionary-itemiterator object at 0xa35050>
5  >>> ','.join('%s:%s'%(key, value) for key, value in da.
       iteritems())
6  '1:10,2:20'
```

Dictionary Views

- A dictionary view changes with the dictionary:

```
>>> da = {1: 10, 2: 20}
>>> daiit = da.iteritems() # an iterator
>>> daiit
<dictionary-itemiterator object at 0xa350a8>
>>> davit = da.viewitems() # a view object
>>> davit
dict_items([(1, 10), (2, 20)])
>>> da[3] = 30 # change the dictionary
>>> ','.join('%s:%s'%(key, value) for key, value in daiit)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 1, in <genexpr>
RuntimeError: dictionary changed size during iteration
>>> ','.join('%s:%s'%(key, value) for key, value in davit)
'1:10,2:20,3:30'
```

Collections Abstract Base Classes

- OK, what I really wanted to talked about in this talk was the collections ABCs.
  - But it seemed impossible to prepare a talk about them in 3 hours.
  - Sorry.
  - I stop here.
- When you want to make your own containers (data structures), they are extremely handy.
  - For example, an OrderedSet that uses `collections.MutableSet`.
- Maybe next time.

Thanks!