8.14 实现自定义容器¶

问题¶

你想实现一个自定义的类来模拟内置的容器类功能，比如列表和字典。但是你不确定到底要实现哪些方法。

解决方案¶

`collections` 定义了很多抽象基类，当你想自定义容器类的时候它们会非常有用。 比如你想让你的类支持迭代，那就让你的类继承 `collections.Iterable` 即可：

```
import collections
class A(collections.Iterable):
    pass
```

不过你需要实现 `collections.Iterable` 所有的抽象方法，否则会报错:

```
>>> a = A()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class A with abstract methods __iter__
>>>
```

你只要实现 `__iter__()` 方法就不会报错了(参考4.2和4.7小节)。

你可以先试着去实例化一个对象，在错误提示中可以找到需要实现哪些方法：

```
>>> import collections
>>> collections.Sequence()
Traceback (most recent call last):
    File "<stdin>", line 1, in <module>
TypeError: Can't instantiate abstract class Sequence with abstract methods \
__getitem__, __len__
>>>
```

下面是一个简单的示例，继承自上面Sequence抽象类，并且实现元素按照顺序存储：

```
class SortedItems(collections.Sequence):
    def __init__(self, initial=None):
        self._items = sorted(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        return self._items[index]

    def __len__(self):
        return len(self._items)

    # Method for adding an item in the right location
    def add(self, item):
        bisect.insort(self._items, item)


items = SortedItems([5, 1, 3])
print(list(items))
print(items[0], items[-1])
items.add(2)
print(list(items))
```

可以看到，SortedItems跟普通的序列没什么两样，支持所有常用操作，包括索引、迭代、包含判断，甚至是切片操作。

这里面使用到了 `bisect` 模块，它是一个在排序列表中插入元素的高效方式。可以保证元素插入后还保持顺序。

讨论¶

使用 `collections` 中的抽象基类可以确保你自定义的容器实现了所有必要的方法。并且还能简化类型检查。 你的自定义容器会满足大部分类型检查需要，如下所示：

```
>>> items = SortedItems()
>>> import collections
>>> isinstance(items, collections.Iterable)
True
>>> isinstance(items, collections.Sequence)
True
>>> isinstance(items, collections.Container)
True
>>> isinstance(items, collections.Sized)
True
>>> isinstance(items, collections.Mapping)
False
>>>
```

`collections` 中很多抽象类会为一些常见容器操作提供默认的实现， 这样一来你只需要实现那些你最感兴趣的方法即可。假设你的类继承自 `collections.MutableSequence` ，如下：

```
class Items(collections.MutableSequence):
    def __init__(self, initial=None):
        self._items = list(initial) if initial is not None else []

    # Required sequence methods
    def __getitem__(self, index):
        print('Getting:', index)
        return self._items[index]

    def __setitem__(self, index, value):
        print('Setting:', index, value)
        self._items[index] = value

    def __delitem__(self, index):
        print('Deleting:', index)
        del self._items[index]

    def insert(self, index, value):
        print('Inserting:', index, value)
        self._items.insert(index, value)

    def __len__(self):
        print('Len')
        return len(self._items)
```

如果你创建 `Items` 的实例，你会发现它支持几乎所有的核心列表方法(如append()、remove()、count()等)。 下面是使用演示：

```
>>> a = Items([1, 2, 3])
>>> len(a)
Len
3
>>> a.append(4)
Len
Inserting: 3 4
>>> a.append(2)
Len
Inserting: 4 2
>>> a.count(2)
Getting: 0
Getting: 1
Getting: 2
Getting: 3
Getting: 4
Getting: 5
2
```

```
>>> a.remove(3)
Getting: 0
Getting: 1
Getting: 2
Deleting: 2
>>>
```

本小节只是对Python抽象类功能的抛砖引玉。`numbers` 模块提供了一个类似的跟整数类型相关的抽象类型集合。 可以参考8.12小节来构造更多自定义抽象基类。