The Iterator Design Pattern

A fundamental pattern that is critical for allowing easy traversal of classes that contain collections of data

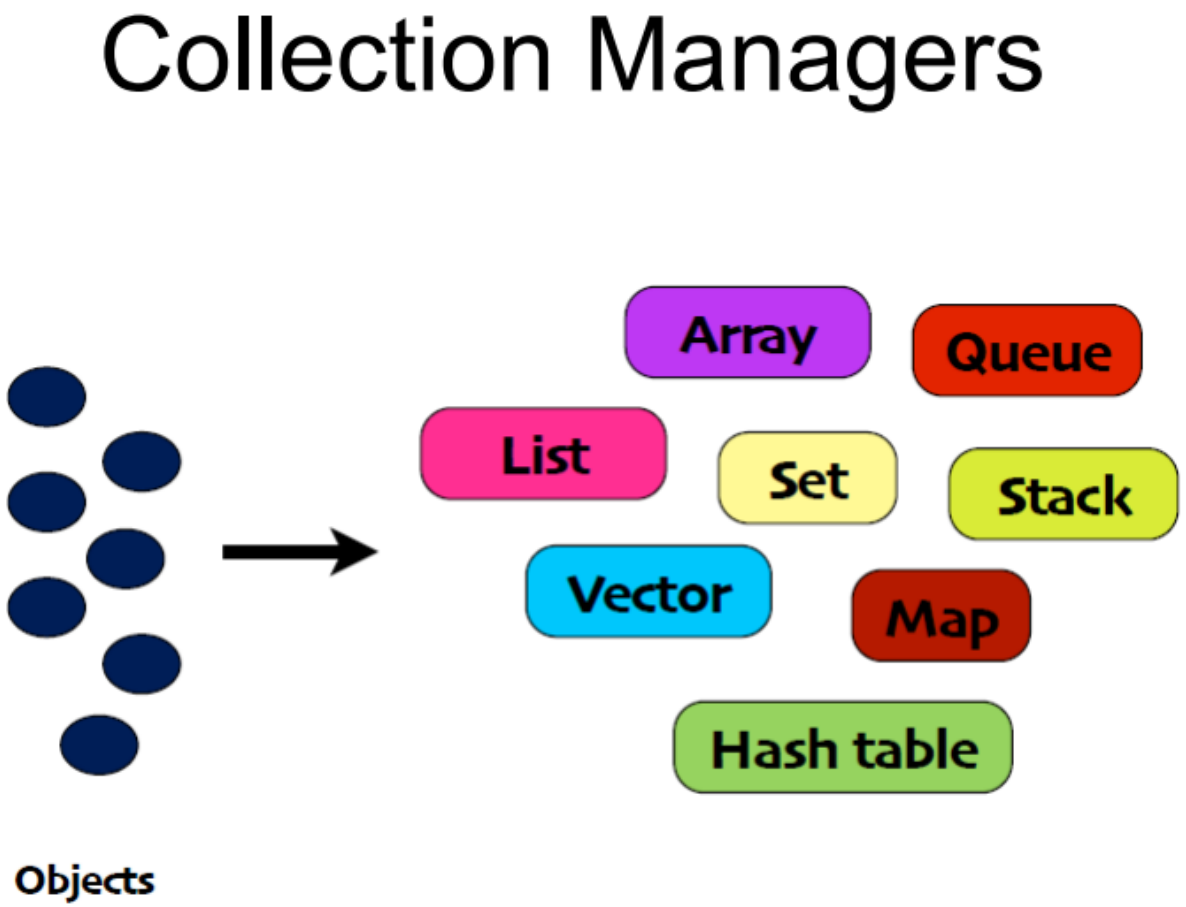
Iterator Definition

- Provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation
- Used to traverse a container and access the container’s elements
- Decouples algorithms from containers
- Allows you to access the elements of a collection object in ‘sequential’ manner without any need to know its underlying representation

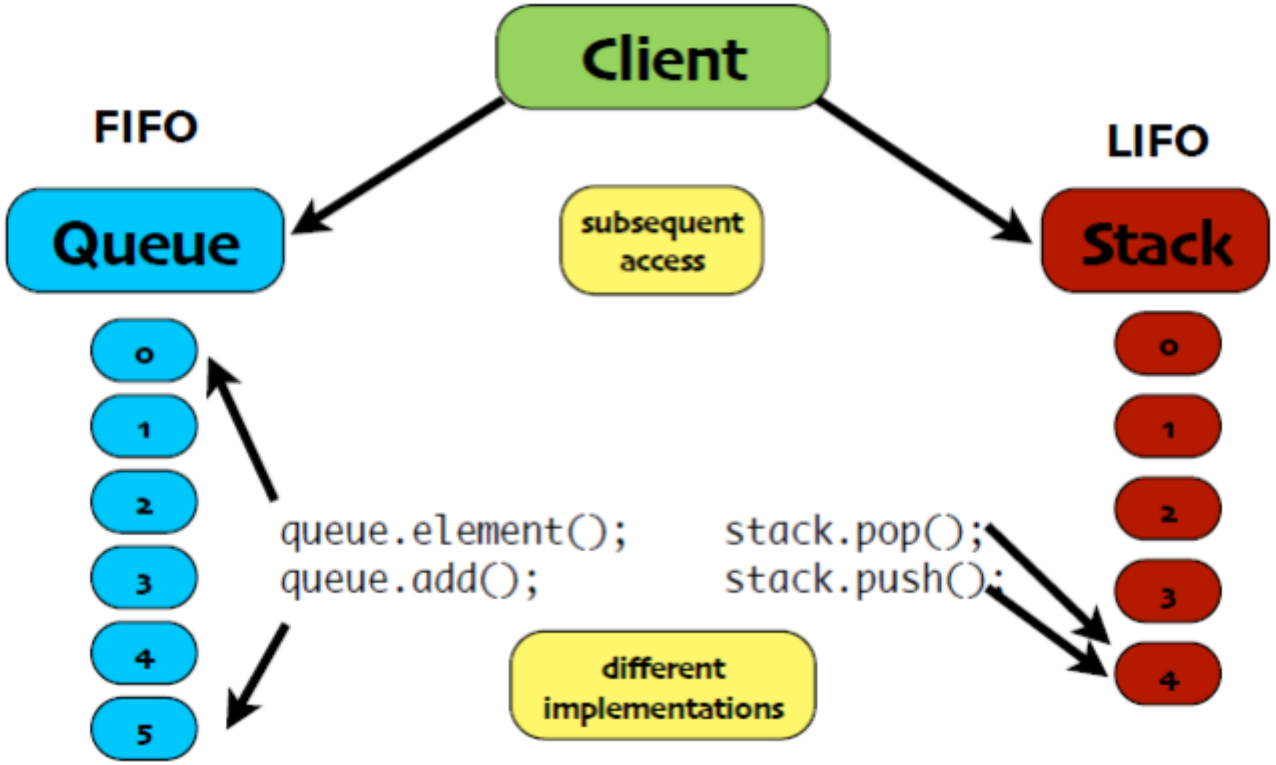
Problem

- Aggregate objects such as something that contains a list internally (or just a list itself!) should allow a way to traverse its elements without exposing its internal structure (encapsulation!)
- This kind of thing is supported in Python and Java (and C#) – the for loop (foreach in C#) takes advantage of this behavior on objects designed with this in mind

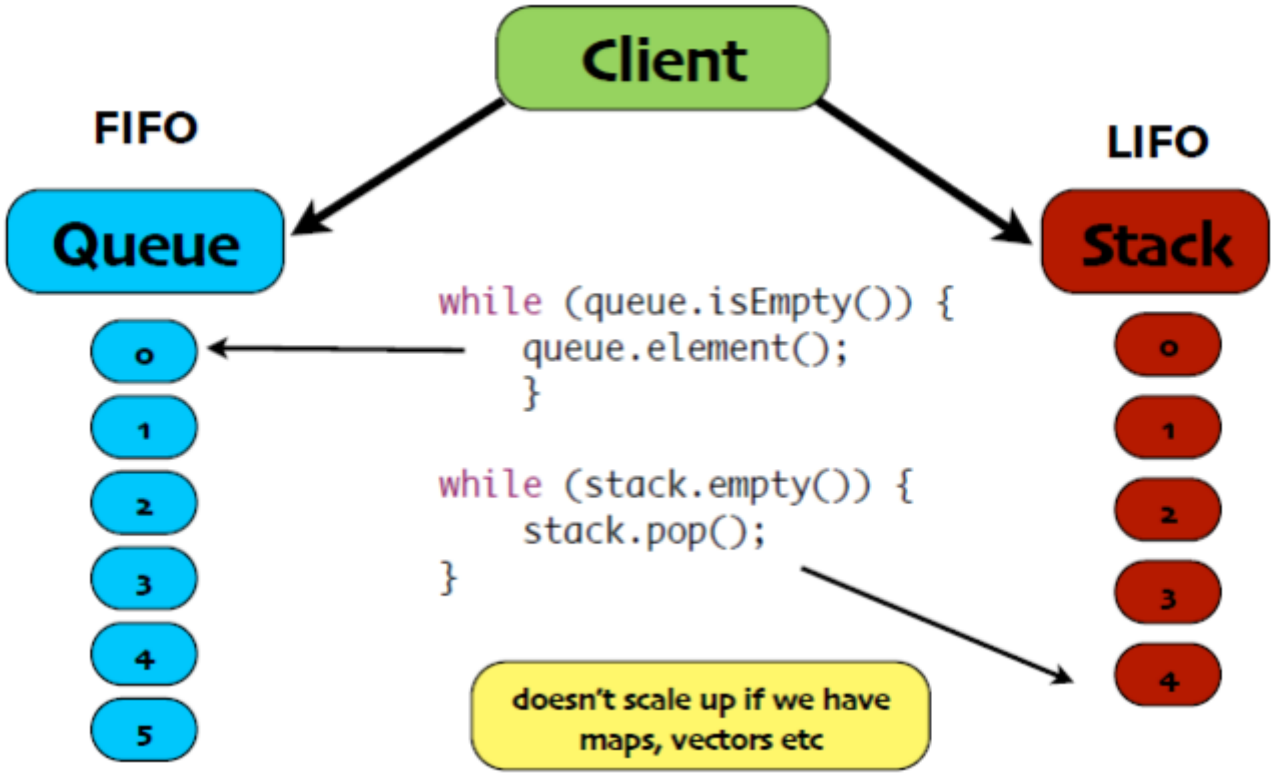
Collections and Managers



Sample Problem

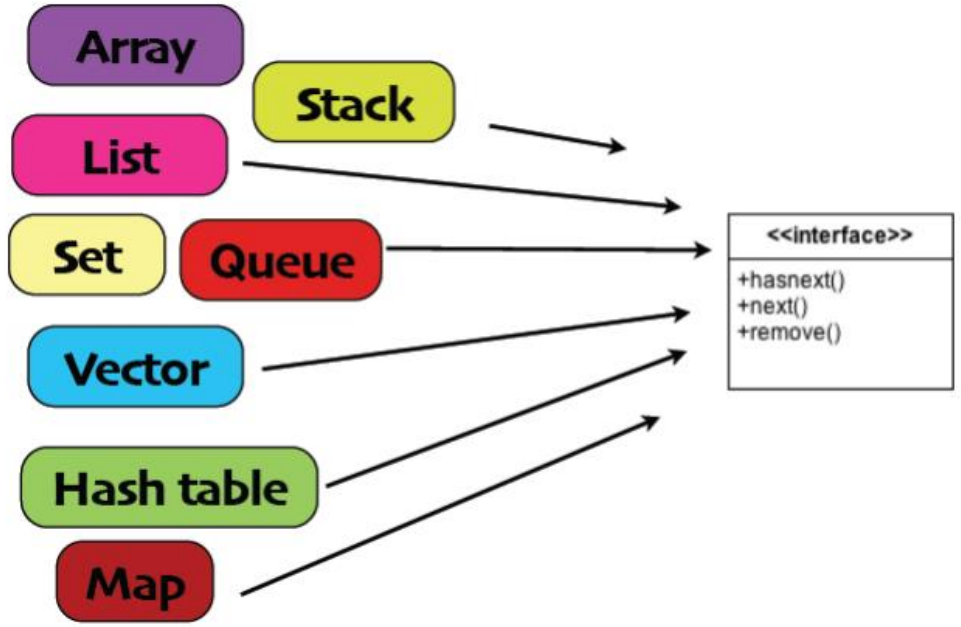


Cumbersome – z z z z z z z z z z

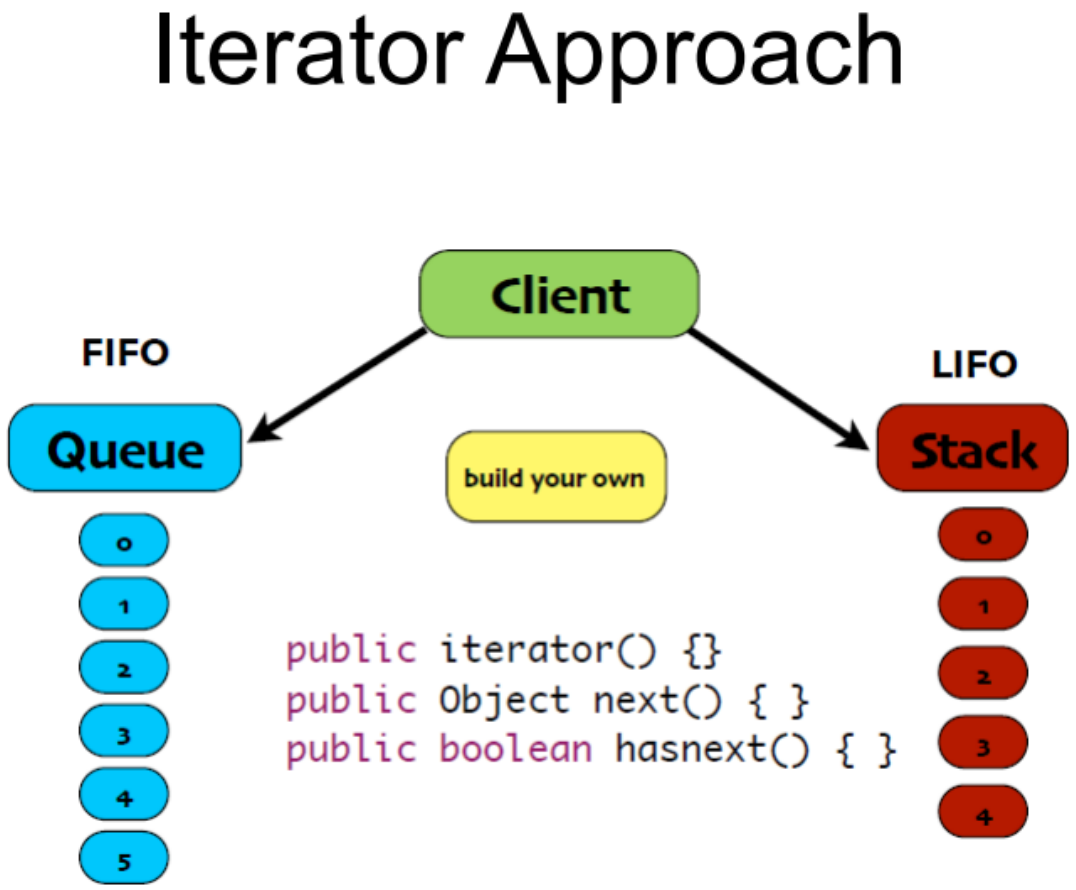


Let’s build an interface to represent basic behavior!

One interface to rule them all, one interface to find them, one interface to bring them all and in the API bind them!!!



Iterator Approach



Iterator Implementation

- Some class implements Iterator interface (next, hasNext, remove)
- Aggregate class provides a method that generates an iterator object (typically createIterator or iterator) from the above class
- By placing iteration information in its own class this allows the aggregate to ‘go about its business’ without having to provide other classes with info on how to traverse its underlying collection
- Iterator helps enforce the Single Responsibility Principle (a class should only have one reason to change – a class should do what that class is about and nothing else)

Single Responsibility Principle

Iterators in Python

- Python has built in support for Iterator
- The class needs to implement the following methods
  - _ _ iter _ _ (used to initialize the iterator)
  - _ _ next _ _ (used to return current item, advance ‘count’ in preparation for next item, and report when the collection has been traversed)

Example 1

```
class MyNumbers:
  def __iter__(self):
    self.a = 1
    return self

  def __next__(self):
    if self.a <= 20:
      x = self.a
      self.a += 1
      return x
    else:
      raise StopIteration
```

Example 1 continued

```
myclass = MyNumbers()
myiter = iter(myclass)

for x in myiter:
  print(x)
```

Example 2

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max=0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self
```

Example 2

```
def __next__(self):
    if self.n <= self.max:
        result = 2 ** self.n
        self.n += 1
        return result
    else:
        raise StopIteration
```

Example 2

```
# create an object
numbers = PowTwo(3)

# create an iterable from the object
i = iter(numbers)

# Using next to get to the next iterator element
print(next(i))
print(next(i))
print(next(i))
print(next(i))
print(next(i))
```

Example 2

```
for i in PowTwo(5):
    print(i)
```

```
1
2
4
8
16
32
```

Iterator Closing Thoughts

- It allows other classes to work with an object of your collection class and traverse its items without having to know how those items are represented underneath
- Creates decoupled designs
- Follows encapsulation principles
- Simplifies your code base

UML Class Diagram