The Iterator Pattern

Intermediate Application Development

Otago Polytechnic
Dunedin, New Zealand
Kaiako: Tom Clark

Introduction

We do this all the time:

```
for thing in aggregate:
    do_stuff_with(thing)
```

1. What kind of objects can do this?
2. How does this work, anyway?

Iterables

Objects that can be used in `for` loops like this are called *Interables*. This means that they are capable of supplying *Iterators*. In other words, they implement the Iterator Pattern.

“Provide a way to access the elements of an aggregate object sequentially without exposing its underlying implementation.” (*GoF*)

This pattern is so fundamental that it is supported in the core Python language. Many common objects implement Iterator, and it is easy to make your own objects that support it too.

Examples

- Lists
- Dictionaries
- Sets
- Files
- Database query results

Iterators in Action

```
ls = [1, 2, 3]  # ls is an Iterable
itr = iter(ls)  # itr is a list Iterator
next(itr) # returns 1
next(itr) # returns 2
next(itr) # returns 3
next(itr) # raises a StopIteration exception
```

Iterators in Action

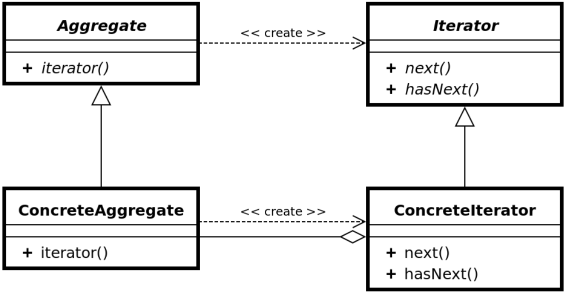
These loops are equivalent.

```
ls = [1, 2, 3]

for i in ls:
    print(i)

itr = iter(ls)
while True:
    try:
        print(next(itr))
    except StopIteration:
        break
```

Structural Diagram



Why a Seperate Iterator?

There is an important reason why an Iterable like a list produces a seperate Iterator. We may want to obtain two Iterators from the same list. We expect that each iterator produces the list values independently.

A File object is a notable exception to this. It is its own Iterator. This is because we expect to be able to read from an open File up to a point and then later resume reading from where we left off.

List vs File

```
ls = [1, 2, 3]

# these both print the same thing
for i in ls:
    print(i)

for i in ls:
    print(i)

with open(’filename’) as f: # f is an iterator
    for line in f:
        print(line)
        if line == ’\n’:
            break

    # this picks up reading where the loop above stopped
    for line in f:
        print(line)
```

Programming Activity

1. Pull the course materials repo.
2. Create a new branch, `12-practical` in your practicals repo.
3. Add a subdirectory, `12-practical` and copy `11-practical.ipynb` from the class materials into it.
4. Open a shell, cd to this directory, and run `jupyter notebook` to open the notebook. Complete the first two questions.
5. We will discuss results in 20ish minutes.

Make a Class Iterable

We make a class iterable by implementing the `__iter__` method. It must return an Iterator.

```
class Cattery:

  def __init__(self, cats):
      self._cats = set(cats)

  def __iter__(self):
      return iter(self._cats)
```

Make an Iterator

Since an Iterable needs to provide an Iterator, sometimes we make our own Iterator class. To make a class an Iterator, implement a `__next__` method.

```
class SkipperIterator:

 def __init__(self, lst):
     self._lst = lst
     self._next_index = 0

 def __next__(self):
     try:
         result = self._lst[self._next_index]
         self._next_index += 2
         return result
     except IndexError:
         raise StopIteration
```

Using the Iterator

Now we just need an Iterable that uses our Iterator

```
class Skipper:

  def __init__(self, lst):
      self._lst = list(lst)

  def __iter__(self):
      return SkipperIterator(self._lst)
```

Notice how the Iterable and the Iterator are tightly coupled. That’s typical of this pattern.