# Chapter 09 - The Iterator Pattern

April 4, 2021

## 0.1 Design Patterns in Brief

- design patterns are an attempt to bring this same formal definition for correctly designed structure to software problems
- there are common problems faced by developers, the desing principle then is a suggestion as to the idea solution for that problem, in terms of OOP design

## 0.2 Iterators

- an iterator is an object with a `next()` method and a `done()` method
- `done()` returns True if there there are no items left in the sequence

- in pyhton, iteration is a special feature, so the method gets a special name `__next__`
- this method can be accessed using the `next(iterator)` built-in
- rather than a `done` method, python iterator protocol raises `StopIteration` to notify the loop that it has complete
- we also have the `foriteminiterator` syntax to actually access items in an iterator instead of messing around with a while loop

## 0.3 The Iterator Protocol

- the `iterator` abstract base class, in the `collections.abc` module
- any class that provides an `__iter__` method is iterable

[2]:
```
class CapitalIterable:
    def __init__(self, string):
        self.string = string

    def __iter__(self):
        return CapitalIterator(self.string)

class CapitalIterator:
    def __init__(self, string):
        self.words = [w.capitalize() for w in string.split()]
        self.index = 0

    def __next__(self):
        if self.index == len(self.words):
            raise StopIteration()
```

```
        word = self.words[self.index]
        self.index += 1
        return word

    def __iter__(self):
        return self

iterable = CapitalIterable('the quck brown fox jumps over the lazy dog')
iterator = iter(iterable)

while True:
    try:
        print(next(iterator))
    except StopIteration:
        break
```

```
The
Quck
Brown
Fox
Jumps
Over
The
Lazy
Dog
```

- the `iterable` is an object with elements that can be looped over
- the `iterator` represents a specific location in that iterable
- some items have been constructed and some have not
- the iterables might be the same but the `iterators` might be at different locations
- each time `next()` is called on the iterator, it returns another token from the iterable, in order

## 0.4 Comprehensions

- powerful syntaxes that allow us to transform or filter an iterable object in as little as one line of code
- the resultant object can be a perfectly normal list, set or dictionary or it can be a generator expression that can be effciently consumed while keeping just one element in memory at a time

[6]:
```
# non - comprehensive method
input_strings = ["1", "5", "28", "131", "3"]
output_integers = []
for num in input_strings:
    output_integers.append(int(num))
```

[7]:
```
# using comprehension
input_strings = ["1", "5", "28", "131", "3"]
```

```
output_integers = [int(num) for num in input_strings]
```

- list comprehensions are faster than a `for` loop when looping over a large number of itmes

[9]:
```
output_integers = [int(num) for num in input_strings if len(num) < 3]
```

## 0.5 Set and Dictionary COmprehensions

- one way to create a `set` is to wrap a list comprehension inside of `set()` constructor

## 0.6 Generator Expressions

- sometimes we want to process a new sequence without pulling a new list, set or dictionary into system memory
- when processing one item at a time, we only need the current object available in memory at any one moment
- `generator expression` use the same syntax as comprehensions but they dont create a final container object
- to create a generator expression, wrap the comprehension in `()` instead of `[]` or `{}`

[13]:
```
import sys

def log_parse():
    inname = sys.argv[1]
    outname = sys.argv[2]

    with open(inname) as infile:
        with open(outname, "w") as outfile:

            warnings = (l for l in infile if 'WARNING' in l)
            for l in warnings:
                outputfile.write(l)
```

- generator xpressions are most useful inside function calls
- we can call `sum`, `min` or `max` on a generator expression instead of a list, since these functions process one object at a time
- try to use generator expressions as much as possible

## 0.7 Generators

- generator comprehensions are a sort of comprehensions too
- they compress the more advanced generator syntax into one line

[ ]:
```
import sys

def warnings_filter(insequence):
    for l in insequence:
        if "WARNING" in l:
            yield l.replace("\tWARNING", "")
```

```
inname = sys.argv[0]
outname = sys.argv[1]
with open(inname) as infile:
    with open(outname, "w") as outfile:
        filter = WarningFilter(infile)
        for l in filter:
            outfile.write(l)
```

- `yelid` is key to generators
- when we see a yeild in a function, it takes that function and wraps it up in an object
- think of a yeild statement simmilar to the return statement
- it exits the function and returns a line
- unlike `return`, when the function is called again via `next()` it will start where it left off -on the line after the yeild statement

the `yeild` is basically creating an object - the power of `yeild` comes from when you have to make multiple calls to `yeild` in a single function, on each loop, the generator will simply pick up the most recent `yeild` and continue to the next one

### 0.8 Yeild items from another Iterable

- when ever possible we should operate on iterators as input, the same way function could be regardless of whether the log line came from a file, memory or the web

- when the code encounters a directory, it recursively asks `walk()` to generate a list of all the files subordinate to each of its children and then yeilds all the data plus its own filename
- in the simple case that it has encountered a normal file, it just yields that name

[ ]:
```
class File:
    def __init__(self, name):
    self.name = name

class Folder(File):
    def __init__(self, name):
    super().__init__(namme)
    self.children = []

def walk(file):
    if isinstance(file, Folder):
        yield file.name + "/"
        for f in file.children:
            yield from walk(f)
        else:
            yield file.name
```

### 0.9 Coroutines

- outside the `asyncio` module, they are not used all that often
- used in asynchronous programming or concurent programming
- the simple ofbject could be used by a scoring application for a baseball team
- seprate tallies could be kept for each team and their score could be incremented by the number of runs accumulated at the end of every half-innings

- we first construct two `tally` objects one for each team
- they look like functions but as with generator objects, the fact that there is a yeild statement inside the function tells python to put a great deal of effort into turning the simple function into an object

[23]:
```python
def tally():
    score = 0
    while True:
        increment = yield score
        score += increment

white_sox = tally()
blue_jays = tally()

print("next(white_sox)")
print(next(white_sox))
print("")
print("next(blue_jays)")
print(next(blue_jays))
print("")
print("white_sox.send(3)")
print(white_sox.send(3))
print("")
print("blue_jays.send(2)")
print(blue_jays.send(2))
print("")
print("white_sox.send(2)")
print(white_sox.send(2))
print("")
print("blue_jays.send(4)")
print(blue_jays.send(4))
```

```
next(white_sox)
0

next(blue_jays)
0

white_sox.send(3)
3
```

```
blue_jays.send(2)
2

white_sox.send(2)
5

blue_jays.send(4)
6
```

- when we call the `next()` on each of the coroutne objects, this does the same things as calling next on any generator, which is to say, it executes each line of code until it encounters a yeild statement
- then it returns the value at that point, and then pauses untill the next time `next()` is called

- this `yeild` function looks like its supposed to return a vlaue and assign it to a variable
- that is exactly whats happening
- the coroutine is still paused at the `yeild` statement and waiting to be activated again by another call to `next()`
- we dont call `next()` we call `send()`
- except we dont call `next()`
- we call the method `send()`
- the method send() does exactly the same thing as next() except that in addition to advancing the generator, the value is what gets assigned to the left side of the `yeild` statement

**High Overview Of Steps** - `yeild` occurs and the generator pauses - `send()` occurs from the outside the function and the generator wakes up - the value sent in is assigned to the left side of the `yeild` statement - the generator continues processing untill it encounters another `yeild` statement

- the `yeild` becomes the most recent value of our `send`

  A generator only produces values, while a coroutine can also consume them

### 0.10 Closing Coroutines and throwing exceptions

- coroutines don't normally follow the iteration mechanism
- instead of pulling data through one until an exception is encountered, data is usally pushed into it using a `send`
- the entity doing the pushing i s normally the one in charge of telling the corouutine when its finished

- it is done using the `close()` method on the coroutine in question
- when `close()` is called, it will raise a `GeneratorExit` exception at the point the coroutine was waiting for a value to be sent in
- normally you want to wrap your `yeild` statements for `coroutines` in a try/finally block
- if we ant to raise an error inside the couroutine, we can use the `throw()` method

### 0.11 The relationship between coroutines, generators and fucntions

- `coroutines` are considered more general and `generators` are considered more specialized
- a `coroutine` is a routine that can have data passed in at one or more points and get it out at one or more points

- a `function` is the simplest type of coroutine; while function can have multiple returns, only one is executed
- a `generator` is a type of coroutine that can have data passed in one point, but can pass data out at multiple points
- note with a generator, you cannot pass data out, but with a coroutine, you can use `send` to pass data in
- functions are callable and return data, with generators, you have to explicity use next and with coroutines you have to use send