• Home Page •

The Iterator Pattern ¶

A "Behavioral Pattern" from the Gang of Four book

Verdict

> Python supports the Iterator Pattern at the most fundamental level available to a programming language: it's built into Python's syntax.
>
> But to support not only the object-based Iterator Pattern but also its own legacy iteration protocol, Python relegates the actual iteration object protocol to a pair of dunder methods. Instead of calling these directly, programmers are expected to invoke iteration through a pair of builtin functions.

The least expressive computer languages make no attempt to hide the inner workings of their data structures. In those languages, if you want to visit every element in an array, you have to generate the integer indexes yourself.

Code in such languages struggles to stay at the high level of describing programmer intent. Instead, the flow of each thought is interrupted with low-level data structure details. When code can't abstract away the mechanics of iteration, it becomes more difficult to read; it's more liable to errors, because the same iteration details need to be repeated over and over; and changing how a data structure is traversed requires finding and changing every place in the code where that data structure is iterated over.

The remedy is encapsulation. The Iterator Pattern proposes that the details about how a data structure is traversed should be moved into an "iterator" object that, from the outside, simply yields one item after another without exposing the internals of how the data structure is designed.

Iterating with the "for" loop ¶

Python's `for` loop abstracts the Iterator Pattern so thoroughly that most Python programmers are never even aware of the object design pattern that it enacts beneath the surface. The `for` loop performs repeated assignment, running its indented block of code once for each item in the sequence it is iterating over.

```
some_primes = [2, 3, 5]
for prime in some_primes:
    print(prime)
```

```
2
3
5
```

The above loop has performed a series of three assignment statements `prime = 2`, `prime = 3`, and `prime = 5`, running the indented block of code after each assignment. The block can include the C-style loop control statements `break` to leave the loop early and `continue` to skip back to the top of the loop.

Because `for` is a repeated assignment statement, it has the same flexibility as Python’s normal `=` assignment operator: by listing several names separated by commas, you can upgrade from assigning to a single name to unpacking a whole tuple. This lets you skip a separate unpacking step.

```
elements = [('H', 1.008), ('He', 4.003), ('Li', 6.94)]

# You’re not limited to a single name like “tup”...
for tup in elements:
    symbol, weight = tup
    print(symbol, weight)

# ...instead, unpack right inside the "for" statement
for symbol, weight in elements:
    print(symbol, weight)
```

Famously, this can be coupled with the Python dictionary’s `item()` method to easily visit each dictionary key and value without the expense of a key lookup at the top of each loop.

```
d = {'H': 1.008, 'He': 4.003, 'Li': 6.94}

# You don’t need to...
for symbol in d.keys():
    weight = d[symbol]
    print(symbol, weight)

# ...instead, you can simply:
for symbol, weight in d.items():
    print(symbol, weight)
```

The `for` loop combines such admirable concision and expressiveness that Python not only supports it as a stand-alone statement, but has incorporated it into four different expressions — Python’s famous “comprehensions” that build lists, sets, dictionaries, and generators directly from inline loops:

```
[symbol for symbol, weight in d.items() if weight > 5]
{symbol for symbol, weight in d.items() if weight > 5}
{symbol: weight for symbol, weight in d.items() if weight > 5}
list(symbol for symbol, weight in d.items() if weight > 5)
```

Python’s decision to re-use its `for` statement syntax inside of expressions — instead of going to the trouble of inventing a separate special-purpose syntax for inline iteration — makes the language simpler, easier to learn, and easier to remember.

The pattern: the iterable and its iterator ¶

Let's now step behind the `for` loop and learn about the design pattern that powers it. The traditional Iterator Pattern involves three kinds of object.

First, there's a *container* object.

Second, the container's internal logic lets it corral and organize a number of *item* objects.

Finally, there's the key to the pattern: instead of the container inventing its own unique method calls for stepping through its items — which would force the programmer to learn a different approach for every container — it offers sequential access to its items through a generic *iterator* object that implements the exact same interface as the iterator of every other kind of container.

Python provides a pair of builtins that let you step behind the `for` loop and pull the levers of iteration yourself.

- `iter()` takes a container object as its argument and asks it to build and return a new iterator object. If the argument you pass isn't actually a container, a `TypeError` is raised: `object is not iterable`.
- `next()` takes the iterator as its argument and, each time it's called, returns the next item from the container. Once the container has no more objects to return, the exception `StopIteration` is raised.

To manually reenact the previous section's first `for` loop, we make a single call to `iter()` followed by four calls to `next()`:

```
>>> it = iter(some_primes)
>>> it
<list_iterator object at 0x7f072ffdb518>
>>> print(next(it))
2
>>> print(next(it))
3
>>> print(next(it))
5
>>> print(next(it))
Traceback (most recent call last):
  ...
StopIteration
```

These are precisely the actions that were taken by Python's `for` loop. Of course, the real `for` loop doesn't hard-code exactly the right number of `next()` calls like I did here — instead, `for` is implemented as something like the following `while` loop:

```
it = iter(some_primes)
while True:
    try:
        prime = next(it)
    except StopIteration:
            break
    else:
        print(prime)
```

```
2
3
5
```

My first question when I learned the Iterator Pattern was: why does the iterator need to be a separate object from the container? Why can't each container simply include a counter inside that reminds it which object to yield next?

The answer is that a single internal counter would work as long as only one `for` loop at a time were ever iterating over a container. But there are many situations where several `for` loops are working on the same container at once. For example, to generate all combinations of two coin flips, a programmer might write concentric loops:

```
sides = ['heads', 'tails']
for coin1 in sides:
    for coin2 in sides:
        print(coin1, coin2)
```

```
heads heads
heads tails
tails heads
tails tails
```

If the Python list object `sides` had tried to support iteration using only a single internal counter, then the inner `for` loop would have used up all of the items and left the outer `for` loop without any further items to visit. Instead, we are given a separate iterator on each call to `iter()`.

```
>>> it1 = iter(sides)
>>> it2 = iter(sides)
>>> it1
<list_iterator object at 0x7fa8b8b292b0>
>>> it2
<list_iterator object at 0x7fa8b8b29400>
>>> it1 is it2
False
```

Not only concentric loops, but multiple threads of control — whether operating system threads or coroutines — also offer plenty of circumstances under which an object might be operated on by several `for` loops at the same time, so each of those `for` loops will also need their own iterator to avoid throwing each other off track.

A twist: objects which are their own iterator ¶

Each time you pass a normal Python container like a list, set, or dictionary to `iter()`, you receive a new iterator object that starts visiting the container's items over again from the beginning.

Some Python objects, however, exhibit a different behavior.

The Python file object is a good example. It conveniently yields lines of text when you iterate across it with a `for` loop. But, unlike a list or dictionary, it doesn't start over again at the first line when traversed with a new `for` loop. Instead, it remembers its previous place in the file and continues yielding lines from where you previously left off.

The fact that a file picks up where it left off can be used to loop over a file in phases. A simple example of a file format that's composed of different sections is the traditional UNIX mailbox file:

```
From jdoe@machine.example Fri Nov 21 09:55:06 1997
From: John Doe <jdoe@machine.example>
To: Mary Smith <mary@example.net>
Subject: Saying Hello
Date: Fri, 21 Nov 1997 09:55:06 -0600

This is a message just to say hello.
So, "Hello".
```

This file needs to be parsed in three phases because each section is delimited by different rules. The initial "envelope line" that starts with the word `From` is always a single standalone line. The header follows, consisting of a series of colon-separated names and values which is terminated by a single blank line. The body comes last and ends at either the next envelope `From` line or the end of the file. Here's how we might parse the first message from a mailbox file using only `for` loops:

```
def parse_email(f):
    for line in f:
        envelope = line
        break
    headers = {}
    for line in f:
        if line == '\n':
            break
        name, value = line.split(':', 1)
        headers[name.strip()] = value.lstrip().rstrip('\n')
    body = []
    for line in f:
        if line.startswith('From'):
            break
        body.append(line)
    return envelope, headers, body
```

This convenient pattern — in which we tackle each section of the file with its own `for` loop that does a `break` once it's done processing its section — is possible because each time we start up another loop, the file object continues reading from right where we left off.

```
with open('email.txt') as f:
    envelope, headers, body = parse_email(f)

print(headers['To'])
print(len(body), 'lines')
```

```
Mary Smith <mary@example.net>
2 lines
```

How does the file object subvert normal iteration and preserve state between one `for` loop and the next?

We can see the answer by stepping behind the `for` loop, calling `iter()` ourselves, and examining the result:

```
>>> f = open('email.txt')
>>> f
<_io.TextIOWrapper name='email.txt' mode='r' encoding='UTF-8'>
>>> it1 = iter(f)
>>> it1
<_io.TextIOWrapper name='email.txt' mode='r' encoding='UTF-8'>
>>> it2 = iter(f)
>>> it2
<_io.TextIOWrapper name='email.txt' mode='r' encoding='UTF-8'>
>>> it1 is it2 is f
True
```

The file object is taking advantage of a technicality: the rule that `iter()` must return an iterator never says that the iterator can't be the iterable object itself! Instead of creating a separate iterator object to keep up with which line from the file should be returned next, the file itself serves as its own iterator and yields a single continuous series of lines, delivering the next available line to whichever of possibly many consumers is the first to invoke `next()`.

In the case of the file object, the decision to collapse the iterable and iterator together into a single object is driven by the behavior of the underlying operating system. Not every kind of file, after all, supports rewind and fast forward — Unix terminals and pipes don't, for example — so the Python file object simply lets the operating system keep up with the current line and never tries to rewind on its own.

But what if you wanted the ability to iterate in separate phases with other kinds of object, and not just files?

For example, what if you pass a plain list of lines to `parse_email()`? Then the routine breaks — because the second and third `for` loops, instead of continuing from where the previous loop stopped, instead start again at the beginning of the list. Among other problems, this prevents the function from finding the email's body:

```
with open('email.txt') as f:
    lines = list(f)

envelope, headers, body = parse_email(lines)
print(headers['To'])
print(len(body), 'lines')
```

```
Mary Smith <mary@example.net>
```

```
0 lines
```

How can we support gradual iteration over a standard container like a Python list?

The answer is another extension to the traditional Iterator Pattern: in Python, iterators return themselves if passed to `iter()`! This means that iterators themselves can be passed to `for` loops, and lets you write programs that switch back and forth whenever they want between manual single steps with `next()` and automatic iteration with a `for` loop:

```
it = iter(some_primes)
print(next(it))
for prime in it:
    print(prime)
    break
print(next(it))
```

```
2
3
5
```

Because the above code handles only a single iterator that we requested ourselves with `iter()`, both its manual `next()` calls as well as its `for()` loop are all advancing the same iterator along through the 3 items in the underlying list. The `for` loop works because the iterator object `it` returns itself when asked for its iterator:

```
>>> it
<list_iterator object at 0x7f4fdce6c5f8>
>>> iter(it)
<list_iterator object at 0x7f4fdce6c5f8>
>>> iter(it) is it
True
```

So we can successfully use our `parse_email()` routine with a Python list if, instead of passing the underlying list, we instead pass an iterator that we've already constructed. With that change, the routine runs as successfully on a list as it originally did on an open file:

```
with open('email.txt') as f:
    lines = list(f)

it = iter(lines)
envelope, headers, body = parse_email(it)
print(headers['To'])
print(len(body), 'lines')
```

```
Mary Smith <mary@example.net>
2 lines
```

If you ever really do implement a routine like `parse_email()`, it's better not to make your caller remember to pass an iterator. Instead, have them pass the container and call `iter()` yourself.

Implementing an Iterable and Iterator ¶

How can a class implement the Iterator Pattern and plug in to Python's native iteration mechanisms `for`, `iter()`, and `next()`?

- The container must offer an `__iter__()` method that returns an iterator object. Supporting this method makes a container an *iterable*.
- Each iterator must offer a `__next__()` method (in old Python 2 code, the spelling is `next()` without the dunder) that returns the next item from the container each time it is called. It should raise `StopIterator` when there are no further items.
- Remember the special case we learned about in the previous section — that some users pass iterators to a `for` loop instead of passing the underlying container? To cover this case, each iterator is also required to offer an `__iter__()` method that simply returns itself.

We can see all of these requirements working together by implementing our own simple iterator!

Note that there is no requirement that the items yielded by `__next__()` be stored as persistent values inside the container, or even exist until `__next__()` is called. This lets us offer a very simple Iterator Pattern example without even implementing storage in the container:

```
class OddNumbers(object):
    "An iterable object."

    def __init__(self, maximum):
        self.maximum = maximum

    def __iter__(self):
        return OddIterator(self)

class OddIterator(object):
    "An iterator."

    def __init__(self, container):
        self.container = container
        self.n = -1

    def __next__(self):
        self.n += 2
        if self.n > self.container.maximum:
            raise StopIteration
        return self.n

    def __iter__(self):
        return self
```

With these three simple methods — one for the container object, and two for its iterator — an `OddNumbers` container is now eligible for full membership in Python's rich iteration ecosystem. It will work seamlessly with a `for` loop:

```
numbers = OddNumbers(7)

for n in numbers:
    print(n)
```

```
1
3
5
7
```

And it works with the `iter()` and `next()` builtins.

```
it = iter(OddNumbers(5))
print(next(it))
print(next(it))
```

```
1
3
```

And it can even dance with the comprehensions!

```
print(list(numbers))
print(set(n for n in numbers if n > 4))
```

```
[1, 3, 5, 7]
{5, 7}
```

Three simple methods are all that's needed to unlock access to Python's syntax-level support for iteration.

Python's extra level of indirection ¶

But, one question: why does Python even have `iter()` and `next()`?

The standard Iteration Pattern, out in the wider world of programming languages, involves no builtin functions. The pattern instead talks directly about object behavior: a container needs to implement one method, and an iterable needs to implement another. Why didn't Python simply specify the names of the methods themselves, and let users call them directly using the well-understood `obj.method()` notation?

The reason is that Python had a legacy iteration mechanism to support.

Originally, Python's `for` loop only supported containers that were integer-indexed, like the Python list and tuple. Its underlying operation looked something like this:

```
>>> some_primes[0]
2
>>> some_primes[1]
3
>>> some_primes[2]
5
>>> some_primes[3]
Traceback (most recent call last):
  ...
IndexError: list index out of range
```

It was the final `IndexError` that told the original Python `for` loop it was finished. To iterate over a container that wasn't always addressable by sequential integer indices, like a dictionary, you had to first ask for a `list` of keys, values, or items, then iterate over the integer-indexed list object — because integer indexes were all the `for` loop understood.

When the time came for PEP-234 to incorporate the Iterator Pattern into Python 2.2, there was a problem. If the `for` loop was now going to start calling `__iter__()` on the container it was given, what would happen to all of the simpler containers that programmers had built over the years — that accepted integer indices, trusting that this made them iterable?

Happily, all problems in computer science can be solved by another level of indirection![1]

Python decided to interpose a pair of builtins in between users and the unfortunate fact that the language features two iteration patterns — one legacy, and one the object based Iterator Pattern itself. The PEP decreed that:

- The `for` loop would now prefer the Iterator Pattern. If a container offered `__iter__()`, then the `for` loop would use the iterator it returned.
- To support old containers, `for` would fall back to looking for a `__getitem__()` method and, if it existed, passing it the integers 0,1,2,... to receive items until it received an `IndexError`.
- To expose this mechanism to Python programmers without every programmer needing to hand-implement this carefully crafted fallback themselves, `iter()` was introduced. It gives programmers access the underlying operation "create an iterator from an iterable" that would otherwise have only been natively available to the `for` loop itself.
- Finally, `next()` was added in a later version of Python so that both halves of the Iterator Pattern — and not just the container half — would be symmetrically covered by builtins.

While they were at it, they also tossed an obscure convenience into `iter()` that involves passing it two arguments instead of one. Since it's not related to the classic Iterator Pattern itself, I'll recommend you read about it in the Standard Libary documention for `iter()` if you're curious, and then experiment with it for yourself.
