Iterate, Iterate, Iterate

Moshe Zadka – https://cobordism.com

Agenda

- What
- How
- How not
- Why

Iteration in Python is subtle. First, let's talk about what iteration is: what are the moving parts that make it work. Then, cover how to use it – using language features, built-in constructs, standard library modules, an third party modules. Iteration also has some so-called "foot-guns": way to do things that cause problems. Finally, the "why" part: why should you use it.

0.1 Iteration protocol

0.1.1 Iterable vs. Iterator

```
from typing import TypeVar, Protocol, Any, Generic
from dataclasses import dataclass

T = TypeVar("T")
```

Iterable or Iterator

```
class Iterable(Protocol[T]):
    def __iter__(self) -> Iterator[T]:
        ...

class Iterator(Iterable[T]):
    def __next__(self) -> T:
        """Raises StopIteration"""
```

The `__iter__` method returns `self`

When it comes to iterators, there are two related "protocols" that are important to understand. They refer to each other, so they have to be explained as a pair.

The first is the *iterable* protocol. This protocol has one method: `__iter__`. This method is supposed to return an iterator. The *iterator* protocol specializes the iterable protocol. It has another method, `__next__`, which returns the next item or raises StopIteration. On an iterator, the `__iter__` method returns `self`.

0.1.2 `iter`

Functions

```
def iter(thing):
    return thing.__iter__()

def next(thing):
    return thing.__next__()

del iter
del next
```

The iter function is the right way to get at an object's `__iter__`. The real function is slightly more complicated, since it can call native objects' iteration slot.

0.1.3 `__iter__`

Making it iterable

```
@dataclass(frozen=True)
class TwoInts:
    thing_1: int
    thing_2: int

    def __iter__(self) -> Iterator[T]:
        return iter([self.thing_1, self.thing_2])
```

One way to implement the `__iter__` method is to return an iterator of a different object which gets the same things. This is sometimes the correct thing to do, although not always.

As a teaching example, this works well!

Making it iterable

```
dec = TwoInts(1, 10)
iter_dec = iter(dec)
print("thing_1", next(iter_dec))
print("thing_2", next(iter_dec))
```

```
thing_1 1
thing_2 10
```

This is one way to use an iterable object. Call iter(), and then call next() repeatedly on the result.

0.1.4 `__next__`

Moving Forward

```
class TwoIntIter:
    def __init__(self) -> None:
        self.things = [1, 10]

    def __next__(self) -> int:
        try:
            return self.things.pop(0)
        except IndexError:
            raise StopIteration() from None
```

When writing a fresh iterator, it has to be *stateful.* Each next() call, in general, returns a different item. In this example, the state is the list itself.

Moving Forward

```
iter_dec = TwoIntIter()
print(next(iter_dec))
print(next(iter_dec))
```

```
1
10
```

In this case, there is no need (although it is still allowed) to call iter. The rest of the code is the same as the last example.

0.1.5 `next`

Moving Forward: Usage

```
iter_dec = iter([1, 10])
print(next(iter_dec))
print(next(iter_dec))
try:
    next(iter_dec)
except Exception as exc:
    print("Stopped", repr(exc))
```

```
1
10
Stopped StopIteration()
```

In more realistic examples of iteration, we do not know in advance the number of items. The StopIteration exception has to be caught.

0.1.6 `for`

for: Using iterators

```
class TwoIntIter:
    def __init__(self) -> None:
        self.things = [1, 10]

    def __iter__(self) -> Iterator[int]:
        return self

    def __next__(self) -> int:
        try:
            ret_value = self.things.pop(0)
        except IndexError:
            print("__next__: End of iteration")
            raise StopIteration() from None
        print("__next__: Returning", ret_value)
        return ret_value
```

In order to show how for uses iterator, this iterator example focuses on a lot of side effects – printing. This is generally not a good idea. In this specific case, it helps illustrate when the function gets called.

for: Using iterators

```
iter_dec = TwoIntIter()
print("for: Before")
for val in iter_dec:
    print("for: Got", val)
print("for: After")
```

```
for: Before
__next__: Returning 1
for: Got 1
__next__: Returning 10
for: Got 10
__next__: End of iteration
for: After
```

Note thatin this example, the iterator is called before each iteration. When the exception is raised, the for loop ends.

0.1.7 Sequence constructors

Sequence constructors: Using iterators

```
res = list(TwoIntIter())
print("Result is", res)
```

```
__next__: Returning 1
__next__: Returning 10
__next__: End of iteration
Result is [1, 10]
```

Another common way to consume operators is via the constructor for sequences. These constructors will consume the items in this iterator and construct a sequence from them.

This is a useful way to "buck-stop" the iterator. After constructing the sequence, the iterator logic is no longer involved.

0.2 Generators

0.2.1 Generator vs. Iterator

Generator vs. Iterator

Iterator: Protocol

Annoying to implement (state)

Generator: Mechanism to build iterators

The iterator protocol is useful. It is also non-trivial to implement well. Because it has to keep track of state, it can be easy to get messy with it.

One way to avoid this need for manually managing the state of the iteration is to use generators. In generators, program state is the iterator state.

0.2.2 `yield`

Generators and yield

```
def two_int_iter():
    print("Begin")
    yield 1
    print("Middle")
    yield 10
    print("End")
```

This generators, except for the print statements, is similar to the last one. Note that in this case, there is no explicit storage of a 2-element list.

Generators and yield

```
iter_dec = two_int_iter()
print("for: Before")
for val in iter_dec:
    print("for: Got", val)
print("for: After")
```

```
for: Before
Begin
for: Got 1
```

```
Middle
for: Got 10
End
for: After
```

When consuming it with for, it is possible to see exactly when the generator resumes execution.

0.2.3 `yield from`

yield from

```
def three_int_iter():
    yield from two_int_iter()
    yield 100

list(three_int_iter())
```

```
Begin
Middle
End
```

```
[1, 10, 100]
```

In a generator, yield from is a way to reuse any iterator. The iterator being reused can be another generator or not.

0.2.4 `deque(maxlen=0)`

Mindless consumption

```
from collections import deque

deque(two_int_iter(), maxlen=0)
```

```
Begin
Middle
End
```

```
deque([], maxlen=0)
```

Because generators have side-effects, sometimes we need only the side effects. A sequence constructor for doing this is deque(maxlen=0). It consumes a generator without storing any elements.

0.3 Combining iterators

Iterable to Iterator

Input: any iterable

Output: on-demand iterator

One of the most useful thing about iterators is the concept of “iterator algebra”. There are many functions that take in an iterable, or several iterables, and return iterators. Since iterators are also iterables, the result of one function can be fed to others.

Much like in “numeric algebra”, where a few operations that work on elements and return elements of the same type, this idea allows breaking up code into multiple, more easily digestible, parts. Moreover, again like in “numeric algebra”, Python comes with useful operations out of the box.

0.3.1 `enumerate`

enumerate

```
list(enumerate(iter([1, 10, 100])))
```

```
[(0, 1), (1, 10), (2, 100)]
```

The function enumerate takes an iterable and returns an iterator that yields a count as well as the original value. This is often useful for indexing or other purposes.

0.3.2 `map`

map

```
from operator import add
from functools import partial

list(map(
    partial(add, 1),
    iter([10, 20]),
))
```

```
[11, 21]
```

The map function takes some iterables and applies a function to the items one-by-one. This is useful to post-process results from iterators.

0.3.3 `filter`

filter

```
from operator import ge
from functools import partial

list(filter(
    partial(ge, 5),
    iter([1, 10, 3]),
))
```

```
[1, 3]
```

The filter function takes an iterable and returns an iterator with only the elements that satisfy a condition. It does *not* modify the elements. It is sometimes used in combination with map.

0.3.4 `zip`

zip

```
list(zip(
    iter([10, 100]),
    iter([2, 4]),
))
```

```
[(10, 2), (100, 4)]
```

The zip function takes iterables and returns an iterator which yields tuples – one from each iterable.

0.3.5 Comprehensions

zip

```
double_plus_one = (
    2 * a + 1
    for a in
    iter([1, 10])
)

list(double_plus_one)
```

```
[3, 21]
```

For artbitrary code, generator comprehensions can be used. They allow filtering (with if), looping over multiple things, and more. They are often useful together with zip.

0.4 `itertools`

```
import itertools
```

itertools

standard

Advanced iterator algebra

Beyond the built-in functions and syntax, the iteratools module provides some more useful iteration algebra primitives. There are more than can be easily covered, but here is a small taste.

0.4.1 `chain`

chain

```
list(
    itertools.chain(
        iter([1, 2, 3]),
        iter([4, 5]),
    )
)
```

```
[1, 2, 3, 4, 5]
```

The itertools .chain function is similar to sequence addition. It returns an iterator which yields elements from the first iterable, and then the second. It can take any number of iterables.

0.4.2 `islice`

islice

```
list(itertools.islice(two_int_iter(), 0, 1))
```

```
Begin
```

```
[1]
```

The islice function is similar to sequence slicing. It supports first element, last element, and step. Note that because it uses iterators, it will *not* consume all elements. This makes it particularly useful to cap the length of long (or infinite) iterators.

0.4.3 `count`

count

```
list(itertools.islice(itertools.count(), 0, 5))
```

```
[0, 1, 2, 3, 4]
```

The count function returns a count. This is an infinite count, but can be cut down with islice.

0.5 `more_itertools`

```
import more_itertools
```

more-itertools

When itertools is not enough

There are also useful third-party libraries with even more useful iterator algebra primitives. One of them has the unassuming name more_itertools. It contains quite a few useful functions. Once again, this is a small taste.

0.5.1 `chunked`

chunked

```
list(more_itertools.chunked(iter([0, 1, 2, 3]), 2))
```

```
[[0, 1], [2, 3]]
```

The chuncked function is useful for batching elements from an iterator. For example, this is useful for a remote API that takes several items in a batch. Without much code modification, you can tune the size of the batch.

0.5.2 `distribute`

distribute

```
evens, odds = more_itertools.distribute(
    2,
    iter([0, 1, 2, 3]),
)
print(list(evens))
print(list(odds))
```

```
[0, 2]
[1, 3]
```

The distribute function distributes elements between several iterators. This can be used for parallel processing, for example.

0.5.3 `peekable`

peekable

```
some_nums = more_itertools.peekable(
    iter([0, 1, 2, 3, 4])
)

for x in some_nums:
    if some_nums.peek() == 3:
        break
    print(x)
```

```
0
1
```

Sometimes it's useful to be able to know "what comes next". This can be useful when processing a stream, to only process as much as needed, and leave the stream in a good state for the next consumer.

0.5.4 `windowed`

windowed

```
samples = more_itertools.windowed(
    iter([0, 3, 7, 10, 12]),
    2
)

for start, end in samples:
    print(end - start)
```

```
3
4
3
2
```

The windowed function gives a moving window over the iterator. As in the example shown, it can be used to get the difference between successive samples.

0.6 Avoid Pitfalls

0.6.1 Iterators are (weirdly) mutable

Iterators are (weirdly) mutable

```
things = iter([1, 2, 3])
print("first time", " ".join(map(str, things)))
print("second time", " ".join(map(str, things)))
```

```
first time 1 2 3
second time
```

Iterators are often used in "functional style" programming. This is good, and functional style programming often works well. However, note that iterators themselves are mutable. A common pitfall is to try to iterate more than once.

Iterator mutation: solution one

```
things = [1, 2, 3]
print("first time", " ".join(map(str, iter(things))))
print("second time", " ".join(map(str, iter(things))))
```

```
first time 1 2 3
second time 1 2 3
```

The best solution sometimes requires refactoring. Pass in the container, not the iterator. This is efficient but can be awkward, depending on the context.

Iterator mutation: solution two

```
things = iter([1, 2, 3])
saved_things = list(things)
print("first time", " ".join(map(str, saved_things)))
print("second time", " ".join(map(str, saved_things)))
```

```
first time 1 2 3
second time 1 2 3
```

The easy, but expensive, solution is to "freeze" the iterator into a list internally. This can require non-trivial amounts of memory.

0.6.2 Iterators and mutation

Dictionary, iteration, and mutation

```
things = iter(stuff := dict(a=1, b=2))
stuff.pop("a")
try:
    list(things)
except Exception as exc:
    print(repr(exc))
```

```
RuntimeError('dictionary changed size during iteration')
```

If a dictionary is modified *after* the iterator is constructed, but before the iterator is consumed, this iterator is a footgun.this iterator is a footgun.

0.6.3 The last yield matters

The last yield matters

```
def gen():
    yield "hello"
    print("going")
    yield "goodbye"
print(type(gen()))
```

```
<class 'generator'>
```

The last yield matters

```
def gen():
    # yield "hello"
    print("going")
    yield "goodbye"
print(type(gen()))
```

```
<class 'generator'>
```

The last yield matters

```
def gen():
    # yield "hello"
    print("going")
    # yield "goodbye"
print(type(gen()))
```

```
going
<class 'NoneType'>
```

When debugging, sometimes you need to comment out code. Every yield that you comment out makes the generator generate *less* but does not change what it is. The *last yield* to be commented out does.

The last yield matters

```
def gen():
    if False:
        yield # Fake
    # yield "hello"
    print("going")
    # yield "goodbye"
print(type(gen()))
```

```
<class 'generator'>
```

If you are doing it, put a yield under "if False".

0.7 Summary

0.7.1 Iterators and Generators

Iterators and Generators

Iterators: Useful interface

Generators: Useful way to implement interface

Iterators provide a useful common interface for just-in-time execution. Generators are a useful way to write iterators.

0.7.2 Algebra for Just-in-time

Algebra of Just-in-time

Functions that accept and return iterators

Composable just-in-time tools

Part of the usefulness of the iterator concept is that it lends itself to an algebra. This allows expressing abstract computation that needs to happen step by step while still supporting composability.

0.7.3 Ecosystem

Ecosystem

Research

Consume

Contribute

This useful abstraction, together with Python's popularity, means there is a big ecosystem. Research to learn what things you can get "for free". Consume them as you would any other PyPI dependency. If you do find a useful abstraction that's missing, you can explore contributing it to an existing PyPI package or upload your own.