Programming technologies

Iterators and Generators

Iterators and Iterables

Iterators and iterables are fundamental components of Python programming.

Python iterable and iterator are different. The main difference between them is, iterable in Python cannot save the state of the iteration, whereas in iterators the state of the current iteration gets saved.

Iterators power and control the iteration process, while iterables typically hold data that you want to iterate over one value at a time.

Every iterator is also an iterable, but not every iterable is an iterator in Python.

Iterators and Iterables

Feature	Iterators	Iterables
Can be used in `for` loops directly	✅	✅
Can be iterated over many times	❌	✅
Support the `iter()` function	✅	✅
Support the `next()` function	✅	❌
Keep information about the state of iteration	✅	❌
Optimize memory use	✅	❌

Iterators

Iterators were added to Python 2.2 through PEP 234. In Python, an iterator is an object that allows you to iterate over collections of data, such as **lists**, **tuples**, **dictionaries**, and **sets**.

Python iterators implement the **iterator design pattern**, which allows you to traverse a container and access its elements. The iterator pattern decouples the iteration algorithms from container data structures.

Iterators take responsibility for two main actions:

- Returning the data from a stream or container one item at a time
- Keeping track of the current and visited items

In summary, an iterator will **yield** each item or value from a collection or a stream of data while doing all the internal bookkeeping required to maintain the state of the iteration process.

Examples

```
a = [1, 2, 3]           # iterable object - have in description __iter__() method
iter_a = iter(a)        # create iterator based on iterable object
print(next(iter_a))     # get 1-st item of iterator
print(next(iter_a))     # get 2-nd item of iterator
print(next(iter_a))     # get 3-rd item of iterator
print(next(iter_a))     # raise StopIteration
#-----------------------------------
a = [1, 2, 3]
for num in a:
    print(num)
#-----------------------------------
a = [1, 2, 3]
iter_a = iter(a)
for num in iter_a:
    print(num)
```

The most generic use case of a Python iterator is to allow iteration over a stream of data or a container data structure. Python uses iterators under the hood to support every operation that requires iteration, including for loops, comprehensions, iterable unpacking, and more.

Examples

```
my_set = {'a', 'b', 'c'}
iter_my_set = iter(my_set)
for item in iter_my_set:
    print(item)
print(next(iter_my_set))

#-----------------------------------------------------------------------------------------------

my_set = {'a', 'b', 'c'}
iter_my_set = iter(my_set)
print(next(iter_my_set))
for item in iter_my_set:
    print(item)
```

Iterator protocol

Python iterators must implement a well-established internal structure known as the **iterator protocol**. A Python object is considered an iterator when it implements two special methods. These two methods make Python iterators work.

Method	Description
.__iter__()	Called to initialize the iterator. It must return an iterator object.
.__next__()	Called to iterate over the iterator. It must return the next value in the data stream. Most of the time, the body of this method looks like: "return self"

Iterators raise StopIteration when all items are already iterated.

Types of Iterators

Using the two methods that make up the iterator protocol in your classes, you can write at least three different types of custom iterators. You can have iterators that:

1. Take a stream of data and yield data items as they appear in the original data
2. Take a data stream, transform each item, and yield transformed items
3. Take no input data, generating new data as a result of some computation to finally yield the generated items

Yielding the Original Data

```
class SequenceIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            item = self._sequence[self._index]
            self._index += 1
            return item
        else:
            raise StopIteration
```

```
for item in SequenceIterator([1, 2, 3, 4]):
    print(item)
#----------------------------------------
sequence = SequenceIterator([1, 2, 3, 4])

# Get an iterator over the data
iterator = sequence.__iter__()
while True:
    try:
        # Retrieve the next item
        item = iterator.__next__()
    except StopIteration:
        break
    else:
        # The loop's code block goes here...
        print(item)
```

Transforming the Input Data

```
class SquareIterator:
    def __init__(self, sequence):
        self._sequence = sequence
        self._index = 0

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < len(self._sequence):
            square = self._sequence[self._index] ** 2
            self._index += 1
            return square
        else:
            raise StopIteration
```

```
for square in SquareIterator([1, 2, 3, 4, 5]):
    print(square)
```

Generating New Data

```
class Squares:
    """Yield n squared numbers from start"""
    def __init__(self, start, n):
        self.i = start
        self.n = n

    def __iter__(self):
        return self

    def __next__(self):
        while self.n > 0:
            self.n -= 1
            i = self.i
            self.i += 1
            return i ** 2
        raise StopIteration
```

```
squares_iter = Squares(1, 6)
for sq in squares_iter:
    print(sq)
```

Generating New Data

```
class FibonacciIterator:
    def __init__(self, stop=10):
        self._stop = stop
        self._index = 0
        self._current = 0
        self._next = 1

    def __iter__(self):
        return self

    def __next__(self):
        if self._index < self._stop:
            self._index += 1
            fib_number = self._current
            self._current, self._next = (
                self._next,
                self._current + self._next,
            )
            return fib_number
        else:
            raise StopIteration
```

```
for fib_number in FibonacciIterator():
    print(fib_number)
```

Potentially Infinite Iterators

```
class FibonacciInfIterator:
    def __init__(self):
        self._index = 0
        self._current = 0
        self._next = 1

    def __iter__(self):
        return self

    def __next__(self):
        self._index += 1
        self._current, self._next = (self._next, self._current + self._next)
        return self._current
```

```
#DO NOT DO THIS IN THE CASE OF AN INFINITE ITERATOR
for fib_number in FibonacciInfIterator():
    print(fib_number)

#USE ONLY next() METHOD
fib_iter = FibonacciInfIterator()
print(next(fib_iter))
print(next(fib_iter))
print(next(fib_iter))
```

Inheriting from collections.abc.Iterator

The **collections.abc** module includes an **abstract base class** (**ABC**) called **Iterator**. You can use this ABC to create your custom iterators quickly.

If you inherit from Iterator, then you don't have to write an **.__iter__()** method because the superclass already provides one with the standard implementation. However, you do have to write your own **.__next__()** method because the parent class doesn't provide a working implementation.

```
from collections.abc import Iterator

class Squares(Iterator):
    """Yield n squared numbers from start"""
    def __init__(self, start, n):
        self.i = start
        self.n = n

    def __next__(self):
        while self.n > 0:
            self.n -= 1
            i = self.i
            self.i += 1
            return i ** 2
        raise StopIteration
```

Cases when the source changes during iteration

```
#example 1
lst = [1, 2, 3]              # iterable object
lst_iter = iter(lst)         # iterator
print(next(lst_iter))        # print 1-st value from
iterator
lst.pop()                    # change the source of iterator
print(lst)                   # print current state of list
print(next(lst_iter))        # print 2-d value from iterator
print(next(lst_iter))        # raise StopIteration
```

```
# example 2, immutable
my_str = "abcd"              # iterable immutable object
my_str_iter = iter(my_str)   # iterator
print(next(my_str_iter))     # print 1-st value from iterator
my_str = "1234"              # change the source of iterator
print(my_str)                # print current state of string
print(next(my_str_iter))     # print 2-d value from iterator
print(next(my_str_iter))     # print 3-d value from iterator
print(next(my_str_iter))     # print 4-th value from iterator
print(next(my_str_iter))     # raise StopIteration
```

Cases when the source changes during iteration

```
# example 3
my_dict = {1: 'a', 2: 'b', 3: 'c'}  # iterable object
dict_iter = iter(my_dict)           # iterator
print(next(dict_iter))              # print 1-st value from iterator
del my_dict[2]                      # change the source of iterator
del my_dict[3]                      # change the source of iterator
print(my_dict)                      # print current state of dictionary
print(next(dict_iter))              # RuntimeError: dictionary changed size during
iteration
```

```
# example 4
my_dict = {1: 'a', 2: 'b', 3: 'c'}  # iterable object
dict_iter = iter(my_dict)           # iterator
print(next(dict_iter))              # print 1-st value from iterator
my_dict['new1'] = 99                # change the source of iterator
my_dict['new2'] = 999               # change the source of iterator
print(my_dict)                      # print current state of dictionary
print(next(dict_iter))              # RuntimeError: dictionary changed size during
iteration
```

Generators

Generator functions are special types of functions that allow you to create iterators using a functional style. Unlike regular functions, which typically compute a value and **return** it to the caller, generator functions return a generator iterator that **yields** a stream of data one value at a time.

In Python, you'll commonly use the term generators to collectively refer to two separate concepts: the **generator function** and the **generator iterator**:

- The generator function is the function that you define using the yield statement.
- The generator iterator is what this function returns.

A generator function returns an iterator that supports the iterator protocol out of the box. So, **generators are also iterators**.

Example

```
def squares(i, n):
    while n > 0:
            n -= 1
            yield i ** 2
            i += 1
```

```
sq_generator_next = squares(1, 6)
print(next(sq_generator_next))
print(next(sq_generator_next))
print(next(sq_generator_next))
print(next(sq_generator_next))
print(next(sq_generator_next))
print(next(sq_generator_next))
print(next(sq_generator_next))    # raise StopIteration
```

Using Generator Expressions to Create Iterators

These are particular types of expressions that return generator iterators. The syntax of a generator expression is almost the same as that of a list comprehension. You only need to turn the square brackets ([]) into parentheses

```
lst_comp = [item for item in range(10)]          # List comprehension

generator_comp = (item for item in range(10))    # Generator expression

#-------------------------------------------------------------------------

generator_expression = (item for item in [1, 2, 3, 4])
for item in generator_expression:
    print(item)
```

Types of Generators

Like class-based iterators, generators allow you to:

1. Yield the input data as is
2. Transform the input and yield a stream of transformed data
3. Generate a new stream of data out of a known computation

```
# example 1
def sequence_generator(sequence):
    for item in sequence:
        yield item

# example 2
def square_generator(sequence):
    for item in sequence:
        yield item**2
```

```
# example 3
def fibonacci_generator(stop=10):
    current_fib, next_fib = 0, 1
    for _ in range(0, stop):
        fib_number = current_fib
        current_fib, next_fib = (
            next_fib, current_fib + next_fib
        )
        yield fib_number
```

Useful links

https://realpython.com/python-iterators-iterables/

https://realpython.com/introduction-to-python-generators/

https://www.w3schools.com/python/python_iterators.asp

https://www.geeksforgeeks.org/iterators-in-python/

https://www.geeksforgeeks.org/python-difference-iterable-iterator/

https://wiki.python.org/moin/Iterator