Python *iterators*
generators
context managers

🐍

Overview

- *Iterators, iterator protocol and itertools*
- *Generators*
- *Context managers*

Iterators

In object-oriented programming,

the ***iterator pattern*** is a ***design pattern***

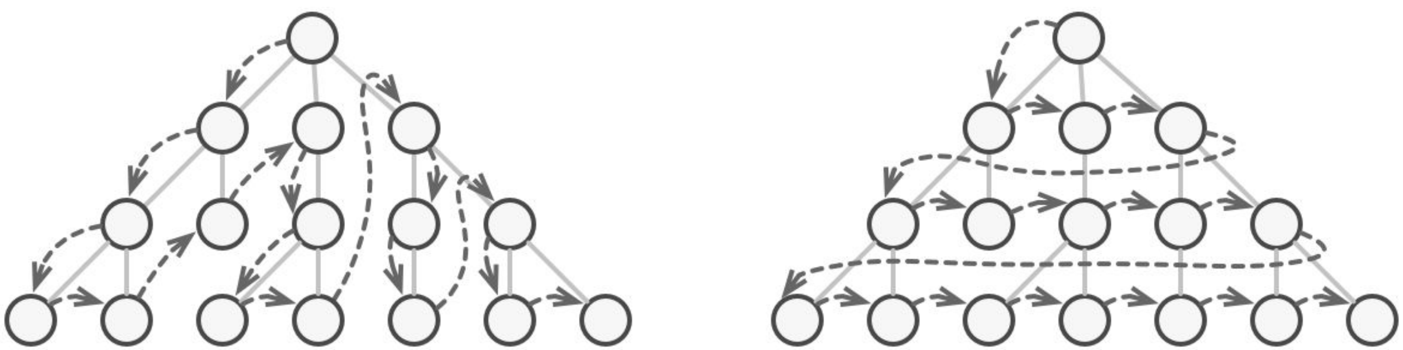
in which an iterator is used to traverse a container

and access the container's elements

Iterators

Collections are one of the most used data types in programming

Nonetheless, a collection is just a container for a group of objects



The same collection can be traversed in several different ways.

Iterators

```
>>> collection = [1, 2, 3, 4, 5]

>>> for x in collection:
...     print(x)
```

Iterators

```
>>> iterator = collection.__iter__()

>>> while True:
...     try:
...         x = iterator.__next__()
...     except StopIteration:
...         break
...
...     print(x)
```

Iterators

Here we should support standard iterator protocol:

- We should be able to get *next* element from iterator
- We should understand if iterator *has next* element

Iterator protocol

container.**__iter__**()

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types.

Iterator protocol

container.**__iter__**()

Return an iterator object. The object is required to support the iterator protocol described below. If a container supports different types of iteration, additional methods can be provided to specifically request iterators for those iteration types.

Iterator protocol

The iterator objects themselves are required to support the following two methods:

iterator.**__iter__**()

Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.

iterator.**__next__**()

Return the next item from the container. If there are no further items, raise the StopIteration exception.

Iterator protocol

Python defines several iterator objects to support iteration over general and specific sequence types, dictionaries, and other more specialized forms

Iterator

```
>>> class custom_range:
...     def __init__(self, n):
...         self.n = n
...         self.i = 0
...
        def __iter__(self):
...         return self

...     def __next__(self):
...         self.i += 1
...         if self.i > self.n:
...             raise StopIteration
...         return self.i
```

Iterator

```
>>> class custom_range:
    …

>>> for i in custom_range(10):
...     print(i)
1
2
3
4
5
6
7
8
9
10
```

Iterator

```
>>> class NextExample:
...     def __init__(self, step):
...         self.step = step
...         self.i = 0
...
...     def __next__(self):
...         self.i += self.step
...         if self.i > 10:
...             raise StopIteration
...         return self.i

>>> nt = NextExample(2)

>>> next(nt)
2
>>> next(nt)
4
...
```

Iterator

```
>>> class NextExample:
...     def __init__(self, step):
...         self.step = step
...         self.i = 0
...
...     def __next__(self):
...         self.i += self.step
...         if self.i > 10:
...             raise StopIteration
...         return self.i

>>> next(nt)
10

>>> next(nt)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "<stdin>", line 9, in __next__
StopIteration
```

Iterator

```
>>> class NextExample:
...     def __init__(self, step):
...         self.step = step
...         self.i = 0
...
...     def __next__(self):
...         self.i += self.step
...         if self.i > 10:
...             raise StopIteration
...         return self.i

>>> next(nt)
10

>>> next(nt, None)
None
```

Iterator

```
>>> lst = []

>>> iter(lst).__class__
<class 'list_iterator'>
```

Iterator

```
>>> class custom_range_iterator:
...     def __init__(self, n):
...         self.n = n
...         self.i = 0
...
...     def __iter__(self):
...         return self
...
...     def __next__(self):
...         self.i += 1
...         if self.i > self.n:
...             raise StopIteration
...         return self.i

>>> class custom_range:
...     def __init__(self, n):
...         self.n = n
...
...     def __iter__(self):
...         return custom_range_iterator(self.n)
```

Separate iterator and iterable

```
>>> rng = custom_range(10)

>>> sum(rng)
55
>>> sum(rng)
55


>>> it = custom_range_iterator(10)

>>> sum(it)
55
>>> sum(it)
0
```

Separate iterator and iterable

```
>>> it = custom_range_iterator(10)

>>> sum(it)
55

Iterator is exhausted here
>>> sum(it)
0
```

Iterator

```
>>> from collections.abc import Iterator

>>> issubclass(custom_range_iterator, Iterator)
True

>>> issubclass(custom_range, Iterator)
False

>>> issubclass(list, Iterator)
False

>>> issubclass(dict, Iterator)
False
```

Separate iterator and iterable

```
>>> class Tree:
...     def __init__(self, ..., depth_first=True):
...         self.depth_first = depth_first
...         ...

...     def __iter__(self):
...         if self.depth_first:
...             return TreeDepthFirstIterator(...)
...         return TreeIterator(...)
```

Iterator *in*

```
>>> dir(custom_range)
['__class__', '__delattr__', '__dict__', '__dir__', '__doc__',
'__eq__', '__format__', '__ge__', '__getattribute__', '__gt__',
'__hash__', '__init__', '__init_subclass__', '__iter__',
'__le__', '__lt__', '__module__', '__ne__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__', '__weakref__'] #
no __contains__
```

```
>>> 3 in custom_range(10) # O(n)
True
```

```
>>> 3 in custom_range(2) # O(n)
False
```

Example of builtin iterators

```
>>> en = enumerate([1, 2, 3, 4])

>>> l = [i for i in en]

>>> l
[(0, 1), (1, 2), (2, 3), (3, 4)]


>>> l = [i for i in en]
>>> l
[]
```

Iterable without *__iter__*

```
>>> class ItemIterable:
...     def __init__(self, lst):
...         self.lst = lst
...
...     def __getitem__(self, k):
...         return self.lst[k]

>>> for item in item_iterable:
...     print(item)
1
2
3
4
```

Iterable without *__iter__*

```
>>> dir(item_iterable)
['__class__', '__delattr__', '__dict__', '__dir__',
'__doc__', '__eq__', '__format__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__init__', '__init_subclass__', '__le__', '__lt__',
'__module__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',
'__str__', '__subclasshook__', '__weakref__', 'lst']
```

Iterable without *__iter__*

```
>>> class ItemIterable:
...     def __init__(self, lst):
...         self.lst = lst
...
...     def __getitem__(self, k):
...         return self.lst[k]
...
...     def __iter__(self):
...         return iter(reversed(self.lst))

>>> item_iterable = ItemIterable([1, 2, 3, 4])

>>> for item in item_iterable:
...     print(item)
4
3
2
1
```

Summary

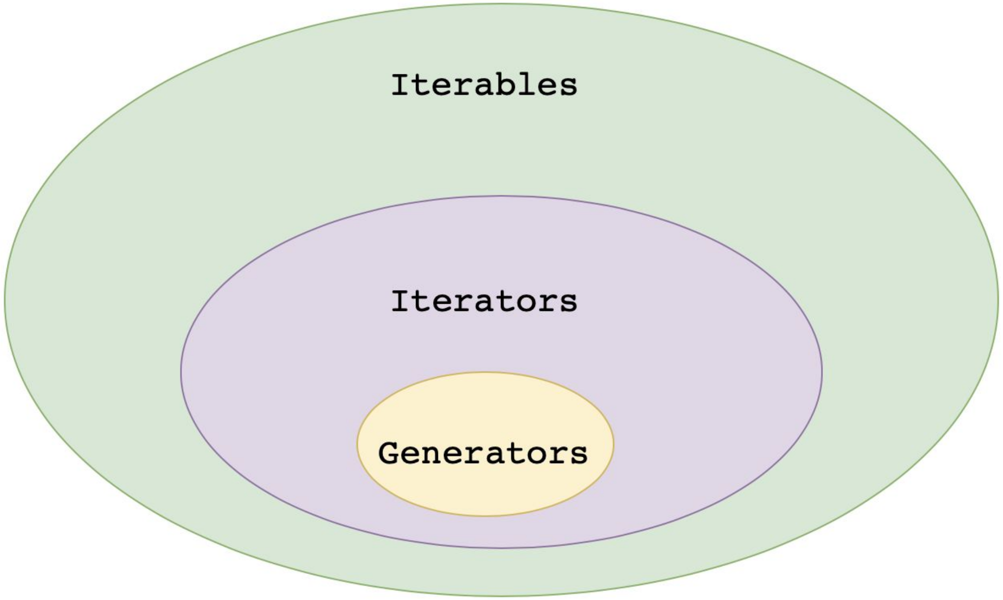
Iterators are Python objects that can produce a data value at a time using the `__next__()`

An object can be iterated over with "for" if it implements `__iter__()` or `__getitem__()`

Iteration protocol:

1. Check for an `__iter__` method. If it exists, use the new iteration protocol.
2. Otherwise, try calling `__getitem__` with successively larger integer values until it raises `IndexError`.

Generators



Generators

A Python *generator* is a function which returns a *generator iterator* (just an object we can iterate over) by calling `yield`

Generators

```
>>> def gen():
...     yield 1
...     yield 2
...     yield 3

>>> gen
<function gen at 0x7fb3801359d8>

>>> gen()
<generator object gen at 0x7fb381ab3c00>

>>> g = gen()
>>> next(g)
1
>>> next(g)
2
>>> next(g)
3
>>> next(g)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Generators

A generator function builds a generator object that wraps the body of the function.

When passing a generator object to `next()`,
execution continues until the next yield clause in the body of the function,
and calling `next()` returns the value that was generated before the function was suspended.

Generators

Difference:

```
>>> def range(start, end, step):
...     result = []
...     cursor = start
...     while cursor < end:
...         result.append(cursor)
...         cursor += step
...     return result


>>> for i in range(0, 10, 2):
...     print(i)
0
2
4
6
8
```

Generators

Difference:

```
>>> def gen_range(start, end, step):
...     cursor = start
...     while cursor < end:
...         yield cursor
...         cursor += step


>>> for i in gen_range(0, 10, 2):
...     print(i)
0
2
4
6
8
```

Suspended context

```
>>> def gen_context():
...     print('Starting context')
...     yield 1
...     print('Continue context')
...     yield 2
...     print('Finish context')

>>> g = gen_context()

>>> next(g)
Starting context
1
>>> next(g)
Continue context
2
>>> next(g)
Finish context
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Suspended context

Generator represents Lazy execution

Generator expression

Task: sum numbers from 0..100_000_000

```
>>> from timeit import timeit

>>> code1 = '''
... exp = [i for i in range(100_000_000)]
... sum(exp)
... '''

>>> code2 = '''
... exp = (i for i in range(100_000_000))
... sum(exp)
... '''

>>> timeit(code1, number=10)
85.0898126100008

>>> timeit(code2, number=10)
44.3151529070008
```

Generator expression

```
>>> gen_exp = (item for item in range(100_000_000))

>>> next(gen_exp)
0

>>> sum((item for item in range(100_000_000)))
4999999950000000

~

>>> sum(item for item in range(100_000_000))
4999999950000000
```

Generator expression exhausted

```
>>> gen = (item for item in range(10))

>>> list(gen)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

>>> list(gen)
[]
```

Examples

```
>>> def chain(*iterables):
...     for iterable in iterables:
...         for item in iterable:
...             yield item

>>> list(chain(range(10), range(5), [1, 2, 3]))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 1, 2, 3]
```

yield from

```
>>> def chain(*iterables):
...     for iterable in iterables:
...         yield from iterable

>>> list(chain(range(10), range(5), [1, 2, 3]))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0, 1, 2, 3, 4, 1, 2, 3]
```

Infinite generator

```
>>> def fib():
...     a, b = 1, 1
...     while True:
...         yield a
...         a, b = b, a + b

>>> fib_gen = fib()
>>> next(fib_gen)
1
>>> next(fib_gen)
1
>>> next(fib_gen)
2
>>> next(fib_gen)
3
>>> next(fib_gen)
5
```

Itertools

*itertools.***count***(start=0, step=1)*

Make an *iterator* that returns evenly spaced values starting with number *start*.

itertools.**cycle**(*iterable*)

Make an iterator returning elements from the iterable and saving a copy of each. When the *iterable* is exhausted, return elements from the saved copy. Repeats indefinitely

itertools.**repeat**(*object*[, *times*])

Make an iterator that returns *object* over and over again. Runs indefinitely unless the *times* argument is specified.

Itertools

```
>>> gen = itertools.count(1, .5)

... Infinity execution
```

Itertools

```
>>> gen = itertools.count(1, .5)
```

We can handle this using itertools!

```
>>> limit_gen = itertools.takewhile(lambda x: x < 10, gen)

>>> list(limit_gen)
[1, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0,
7.5, 8.0, 8.5, 9.0, 9.5]
```

Itertools

```
>>> gen = itertools.count(1, .5)
```

We can handle this using itertools!

```
>>> limit_gen = itertools.takewhile(lambda x: x < 10, gen)

>>> list(limit_gen)
[1, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0,
7.5, 8.0, 8.5, 9.0, 9.5]
```

Itertools

```
>>> cycle_gen = itertools.cycle([1, 2, 3]) # Infinite

>>> next(cycle_gen)
1

>>> next(cycle_gen)
2

>>> next(cycle_gen)
3

>>> next(cycle_gen)
1
```

Itertools

```
>>> list(itertools.product('ABCD', repeat=2))
[('A', 'A'), ('A', 'B'), ('A', 'C'), ('A', 'D'), ('B', 'A'),
('B', 'B'), ('B', 'C'), ('B', 'D'), ('C', 'A'), ('C', 'B'),
('C', 'C'), ('C', 'D'), ('D', 'A'), ('D', 'B'), ('D', 'C'),
('D', 'D')]
```

```
>>> list(itertools.product('ABCD', repeat=3))
[('A', 'A', 'A'), ('A', 'A', 'B'), ('A', 'A', 'C'), ('A', 'A',
'D'), ('A', 'B', 'A'), ('A', 'B', 'B'), ('A', 'B', 'C'), ('A',
'B', 'D'), ('A', 'C', 'A'), ('A', 'C', 'B'), ('A', 'C', 'C'),
('A', 'C', 'D'), … ]
```

Itertools

```
roundrobin('ABC', 'D', 'EF') --> A D E B F C
```

Itertools

```
>>> def roundrobin(*iterables):
...     "roundrobin('ABC', 'D', 'EF') --> A D E B F C"
... # Recipe credited to George Sakkis
...     num_active = len(iterables)
...     nexts = cycle(iter(it).__next__ for it in iterables)
...     while num_active:
...         try:
...             for next in nexts:
...                 yield next()
...         except StopIteration:
... # Remove the iterator we just exhausted from the cycle.
...             num_active -= 1
...             nexts = cycle(islice(nexts, num_active))
```

Context managers

```
>>> with open('text.txt', 'w+') as f:
...     f.write('hello\nworld')

>>> with open('text.txt', 'r') as f:
...     print(f.readlines())
['hello\n', 'world']
```

Context managers

```
>>> files = []
... for x in range(100000):
...     files.append(open('foo.txt', 'w'))

Traceback (most recent call last):
  File "<stdin>", line 3, in <module>
OSError: [Errno 24] Too many open files: 'foo.txt'

[Errno 24] Too many open files: 'foo.txt'
```

Context managers protocol

The simplest is to define a class that contains two special methods:
    `__enter__()` and `__exit__()`.

`__enter__()` returns the resource to be managed
(like a file object in the case of `open()`).

`__exit__()` does any cleanup work and returns nothing.

Context managers protocol

Let’s create our own context manager for write / read binary files

Context managers protocol

```
>>> class open_binary:
...     def __init__(self, filename):
...         self.filename = filename
...
...     def __enter__(self):
...         self.opened_file = open(self.filename, 'wb+')
...         return self.opened_file
...
...     def __exit__(self, *args):
...         self.opened_file.close()
```

Context managers protocol

```
>>> with open_binary('test.bin') as bf:
...     bf.write(b'hello')

>>> with open('test.bin', 'rb') as bf:
...     print(bf.read())
b'hello'
```

Context managers

@contextlib.**contextmanager**

This function is a decorator that can be used to define a factory function for with statement context managers, without needing to create a class or separate *__enter__()* and *__exit__()* methods.

Context managers

```
>>> @contextmanager
... def managed_resource(*args, **kwds):
...     # Code to acquire resource, e.g.:
...     resource = acquire_resource(*args, **kwds)
...     try:
...         yield resource
...     finally:
...         # Code to release resource, e.g.:
...         release_resource(resource)
...
... with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

Context managers

```
>>> @contextmanager
... def managed_resource(*args, **kwds):
...     # Code to acquire resource, e.g.:
...     resource = acquire_resource(*args, **kwds)
...     try:
...         yield resource
...     finally:
...         # Code to release resource, e.g.:
...         release_resource(resource)
...
... with managed_resource(timeout=3600) as resource:
...     # Resource is released at the end of this block,
...     # even if code in the block raises an exception
```

Context managers

The function being decorated must return a generator-iterator when called.

This iterator must yield exactly one value, which will be bound to the targets in the with statement’s as clause, if any.

At the point where the generator yields, the block nested in the with statement is executed.

The generator is then resumed after the block is exited. If an unhandled exception occurs in the block, it is reraised inside the generator at the point where the yield occurred. Thus, you can use a try…except…finally statement to trap the error (if any), or ensure that some cleanup takes place.