Iterables and Iterators

All Versions

Examples

Iterator vs Iterable vs Generator

An **iterable** is an object that can return an **iterator** . Any object with state that has an `__iter__` method and returns an iterator is an iterable. It may also be an object *without* state that implements a `__getitem__` method. - The method can take indices (starting from zero) and raise an `IndexError` when the indices are no longer valid.

Python's `str` class is an example of a `__getitem__` iterable.

An **Iterator** is an object that produces the next value in a sequence when you call `next(*object*)` on some object. Moreover, any object with a `__next__` method is an iterator. An iterator raises `StopIteration` after exhausting the iterator and *cannot* be re-used at this point.

Iterable classes:

Iterable classes define an `__iter__` and a `__next__` method. Example of an iterable class :

```
class MyIterable:

    def __iter__(self):

         return self

    def __next__(self):
         #code

#Classic iterable object in older versions of python, __getitem__ is still supported...
class MySequence:

    def __getitem__(self, index):

         if (condition):
             raise IndexError
         return (item)

 #Can produce a plain `iterator` instance by using iter(MySequence())
```

Trying to instantiate the abstract class from the `collections` module to better see this.

Example:

Python 2.x ≥ 2.3

```
import collections
>>> collections.Iterator()
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods next
```

Python 3.x ≥ 3.0

```
>>> TypeError: Cant instantiate abstract class Iterator with abstract methods __next__
```

Handle Python 3 compatibility for iterable classes in Python 2 by doing the following:

Python 2.x ≥ 2.3

```
class MyIterable(object): #or collections.Iterator, which I'd recommend....

     ....

     def __iter__(self):

          return self

     def next(self): #code

     __next__ = next
```

Both of these are now iterators and can be looped through:

```
ex1 = MyIterableClass()
ex2 = MySequence()

for (item) in (ex1): #code
for (item) in (ex2): #code
```

Generators are simple ways to create iterators. A generator *is* an iterator and an iterator is an iterable.

Extract values one by one

Start with `iter()` built-in to get **iterator** over iterable and use `next()` to get elements one by one until `StopIteration` is raised signifying the end:

```
s = {1, 2}   # or list or generator or even iterator
i = iter(s)  # get iterator
a = next(i)  # a = 1
b = next(i)  # b = 2
c = next(i)  # raises StopIteration
```

Iterating over entire iterable

```
s = {1, 2, 3}

# get every element in s
for a in s:
    print a  # prints 1, then 2, then 3

# copy into list
l1 = list(s)  # l1 = [1, 2, 3]

# use list comprehension
l2 = [a * 2 for a in s if a > 2]  # l2 = [6]
```

Verify only one element in iterable

Use unpacking to extract the first element and ensure it's the only one:

```
a, = iterable

def foo():
    yield 1

a, = foo()  # a = 1

nums = [1, 2, 3]
a, = nums  # ValueError: too many values to unpack
```

Iterator isn't reentrant!

```
def gen():
    yield 1

iterable = gen()
for a in iterable:
    print a

# What was the first item of iterable? No way to get it now.
# Only to get a new iterator
gen()
```

What can be iterable

Iterable can be anything for which items are received *one by one, forward only* . Built-in Python collections are iterable:

```
[1, 2, 3]     # list, iterate over items
(1, 2, 3)     # tuple
{1, 2, 3}     # set
{1: 2, 3: 4}  # dict, iterate over keys
```

Generators return iterables:

Generators return iterables.

```
def foo():  # foo isn't iterable yet...
    yield 1

res = foo()  # ...but res already is
```

Syntax

Parameters

Remarks