Lab 08: Iterators and Generators

AUTHOR
Christian Osendorfer

Topics

Iterators

An iterable is an object whose elements we can go through one at a time. Iterables can be used in for loops and list comprehensions. Iterables can also be converted into lists using the list function. Examples of iterables we have seen so far in Python include strings, lists, tuples, and ranges.

```
>>> for x in "cat":
...     print(x)
c
a
t
>>> [x*2 for x in (1, 2, 3)]
[2, 4, 6]
>>> list(range(4))
[0, 1, 2, 3]
```

Note the abstraction present here: given some iterable object, we have a set of defined actions that we can take with it. In this discussion, we will peak below the abstraction barrier and examine how iterables are implemented "under the hood".

In Python, iterables are formally implemented as objects that can be passed into the built-in iter function to produce an iterator. An iterator is another type of object that can produce elements one at a time with the next function.

- `iter(iterable)`: Returns an iterator over the elements of the given iterable.
- `next(iterator)`: Returns the next element in an iterator, or raises a `StopIteration` exception if there are no elements left.

For example, a list of numbers is an iterable, because `iter` gives us an iterator over the given sequence, which we can navigate using the `next` function:

```
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
>>> next(lst)
1
>>> next(lst)
```

```
2
>>> next(lst)
3
>>> next(lst)
StopIteration
```

Iterators are very simple. There is only one mechanism to get the next element in the iterator: `next`. There is no way to index into an iterator and there is no way to go backward. Once an iterator has produced an element, there is no way for us to get that element again *unless we store it*.

Note that iterators themselves are iterables: calling `iter` on an iterator simply returns the same iterator object.

For example, we can see what happens when we use `iter` and `next` with a list:

```
>>> lst = [1, 2, 3]
>>> next(lst)             # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> next(list_iter)       # Calling next on an iterator
1
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
2
>>> for e in list_iter:   # Exhausts remainder of list_iter
...     print(e)
3
>>> next(list_iter)       # No elements left!
StopIteration
>>> lst                   # Original iterable is unaffected
[1, 2, 3]
```

The `map` and `filter` functions return iterator objects.

Generators

We can define custom iterators by writing a generator function, which returns a special type of iterator called a generator.

A generator function looks like a normal Python function, except that it has at least one `yield` statement. When we call a generator function, a generator object is returned without evaluating the body of the generator function itself. (Note that this is different from ordinary Python functions. While generator functions and normal functions look the same, their evaluation rules are very different!)

When we first call `next` on the returned generator, we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we return). The generator remembers where we stopped, and will continue evaluating from that stopping point on the next time we call next.

As with other iterators, if there are no more elements to be generated, then calling `next` on the generator will give us a `StopIteration`.

For example, here's a generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

To create a new generator object, we can call the generator function. Each returned generator object from a function call will separately keep track of where it is in terms of evaluating the body of the function. Like all other iterators, calling `iter` on an existing generator object returns the same generator object.

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1)  # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

In a generator function, we can also have a `yield from` statement, which will yield each element from an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
StopIteration
```

Since generators are themselves iterators, this means we can use `yield from` to create recursive generators!

```
>>> def rec_countdown(n):
...     if n < 0:
...         print("Blastoff!")
...     else:
...         yield n
```

```
...         yield from rec_countdown(n-1)
...
>>> r = rec_countdown(2)
>>> next(r)
2
>>> next(r)
1
>>> next(r)
0
>>> next(r)
Blastoff!
StopIteration
```

Implement in Python

Unless otherwise stated, don't use any modules that implement a solution to the questions asked. Come up with your doctests and compare these with those from other students. Don't use list/dictionary comprehension in this lab!

Count Occurrences

Implement `count_occurrences`, which takes an iterator `t` and a value `x`. It returns the number of elements equal to `x` that appear in the first `n` elements of `t`.

> **Important**
>
> Call `next` on `t` exactly `n` times. Assume there are at least `n` elements in `t`.

> **Tip**
>
> When the same iterator is passed into a function a second time, it should pick up where it left off after the first call, as with `s` from the doctest below.

```
def count_occurrences(t, n, x):
    """Return the number of times that x is equal to one of the
    first n elements of iterator t.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count_occurrences(s, 10, 9)
    3
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> count_occurrences(s2, 3, 10)
    2
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> count_occurrences(s, 1, 3)  # Only iterate over 3
    1
    >>> count_occurrences(s, 3, 2)  # Only iterate over 2, 2, 2
    3
    >>> list(s) # Ensure that the iterator has advanced the right amount
    [1, 2, 1, 4, 4, 5, 5, 5]
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 6, 6, 2, 2, 2, 5])
    >>> count_occurrences(s2, 6, 6)
```

```
    2
    """
    # Your code here.
```

Repeated

Implement `repeated`, which takes in an iterator `t` and an integer `k` greater than 1. It returns the first value in `t` that appears `k` times in a row.

> **Important**
>
> Call next on `t` only the minimum number of times required. Assume that there is an element of `t` repeated at least `k` times in a row.

> **Tip**
>
> If you are receiving a StopIteration exception, your repeated function is likely not identifying the correct value.

```
def repeated(t, k):
    """Return the first value in iterator t that appears k times in a row,
    calling next on t as few times as possible.

    >>> s = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s, 2)
    9
    >>> s2 = iter([10, 9, 10, 9, 9, 10, 8, 8, 8, 7])
    >>> repeated(s2, 3)
    8
    >>> s = iter([3, 2, 2, 2, 1, 2, 1, 4, 4, 5, 5, 5])
    >>> repeated(s, 3)
    2
    >>> repeated(s, 3)
    5
    >>> s2 = iter([4, 1, 6, 6, 7, 7, 8, 8, 2, 2, 2, 5])
    >>> repeated(s2, 3)
    2
    """
    assert k > 1
    # Your code here.
```

Zip

Use `zip` to implement the following algorithms.

```
def matches(a, b):
    """Return the number of values k such that A[k] == B[k].
    >>> matches([1, 2, 3, 4, 5], [3, 2, 3, 0, 5])
    3
    >>> matches("abdomens", "indolence")
    4
    >>> matches("abcd", "dcba")
    0
```

```
    >>> matches("abcde", "edcba")
    1
    >>> matches("abcdefg", "edcba")
    1
    """
```

```
def palindrome(s):
    """Return whether s is the same sequence backward and forward.
    >>> palindrome([3, 1, 4, 1, 5])
    False
    >>> palindrome([3, 1, 4, 1, 3])
    True
    >>> palindrome('seveneves')
    True
    >>> palindrome('seven eves')
    False
```

Generate constant

Write `generate_constant`, a generator function that repeatedly yields the same value forever.

```
def generate_constant(x):
"""A generator function that repeats the same value x forever.
>>> two = generate_constant(2)
>>> next(two)
2
>>> next(two)
2
>>> sum([next(two) for _ in range(100)])
200
"""
```

Black hole

Implement `black_hole`, a generator that yields items in `seq` until one of them matches `trap`, in which case that value should be yielded forever. You may assume that `generate_constant` works. You may not index into or slice `seq`.

```
def black_hole(seq, trap):
    """A generator that yields items in seq until one of them matches trap, in
    which case that value should be repeatedly yielded forever.
    >>> trapped = black_hole([1, 2, 3], 2)
    >>> [next(trapped) for _ in range(6)]
    [1, 2, 2, 2, 2, 2]
    >>> list(black_hole(range(5), 7))
    [0, 1, 2, 3, 4]
    """
    # Your code here
```

Filter-Iter

Implement a generator function called `filter_iter(iterable, f)` that only yields elements of iterable for which `f` returns True. Note that `iterable` can be infinite!

```
def filter_iter(iterable, f):
    """
    Generates elements of iterable for which f returns True.
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even))
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    # Your code here.
```

Differences

Implement `differences`, a generator function that takes an iterable `it` whose elements are numbers. `differences` should produce a generator that yields the differences between successive terms of it. If `it` has less than 2 values, differences should yield nothing.

```
def differences(it):
    """
    Yields the differences between successive terms of iterable it.

    >>> d = differences(iter([5, 2, -100, 103]))
    >>> [next(d) for _ in range(3)]
    [-3, -102, 203]
    >>> list(differences([1]))
    []
    """
    # Your code here.
```

Compute without a computer

What would Python display? If the command errors, input the specific error.

```
>>> def infinite_generator(n):
...     yield n
...     while True:
...         n += 1
...         yield n
>>> next(infinite_generator)

>>> gen_obj = infinite_generator(1)
```

```
>>> next(gen_obj)
>>> next(gen_obj)
>>> list(gen_obj)
>>> def rev_str(s):
...     for i in range(len(s)):
...         yield from s[i::-1]
>>> hey = rev_str("hey")
>>> next(hey)
>>> next(hey)
>>> next(hey)
>>> list(hey)
>>> def add_prefix(s, pre):
...     if not pre:
...         return
...     yield pre[0] + s
...     yield from add_prefix(s, pre[1:])
>>> school = add_prefix("schooler", ["pre", "middle", "high"])
>>> next(school)
>>> list(map(lambda x: x[:-2], school))
```