

Welcome to Data C88C!

Lecture 17: Iterators

Wednesday, July 23rd, 2025

Week 5

Summer 2025

Instructor: Eric Kim (ekim555@berkeley.edu)

Announcements

- Ants project is out!
- Mid-semester survey feedback: [\[link\]](#)
 - If 75% of the class completes this form by Monday July 28th at 11:59 PM, everyone will receive 1 point of extra credit! If this goal is not met, nobody will receive the extra point.

Lecture Overview

- Tuples
- Iterators
 - Interfaces: Iterables, Iterators
- Map

Tuples

(Demo: 17.py:Demo00)

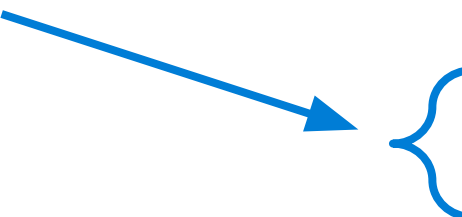
Iterators

Iterating in Python

- Recall that we can iterate through sequences like `list`, `tuple`, `str`, `dict` in a few ways:
 - For loops, list comprehensions
- What's going on under the hood?

```
my_nums = [1, 2, 3]
# iterate via for loop
for x in my_nums:
    print(x)
# iterate via list comprehension
[x ** 2 for x in my_nums]
```

...and how do we support
convenient iteration over our
user-defined classes like `Link`
(linked lists)?



```
>>> lnk1 = Link(1, Link(2, Link('meow')))
>>> for lnk in lnk1:
...     print(lnk.first)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'Link' object is not iterable
```

The answer: Python's **Iterator** and **Iterable** interfaces

Iterators

A container can provide an iterator that provides access to its elements in order

iter(iterable): Return an iterator over the elements of an iterable value

next(iterator): Return the next element in an iterator

```
>>> s = [3, 4, 5]
```

```
>>> t = iter(s)
```

```
>>> next(t)
```

```
3
>>> u = iter(s)
```

```
>>> next(t)
```

```
>>> next(u)
```

```
4
```

```
3
```

```
>>> next(t)
```

```
5
```

```
>>> next(u)
```

```
4
```

Python Iterator [\[source\]](#)

Iterator definition: "An object representing a stream of data. Repeated calls to the iterator's `__next__()` method (or passing it to the built-in function `next()`) return successive items in the stream. When no more data are available a `StopIteration` exception is raised instead..."

Observation: `nums_iterator` is an object that has state that keeps track of where in `nums` the iterator is currently at

This `nums_iterator` is now "exhausted", and subsequent calls to `next()` will keep raising `StopIteration` errors

```
class IteratorInterface:
    """An illustration of Python's Iterator interface"""
    def __next__(self):
        """Returns next item, or raise StopIteration if at end"""
        raise NotImplementedError("Override me!")

    def __iter__(self):
        """Return iter instance, often self"""
        return self

# tip: iter(thing) calls thing.__iter__()
>>> nums = [1, 2]
>>> nums_iterator = iter(nums)
>>> next(nums_iterator) # next() calls thing.__next__()
1
>>> next(nums_iterator)
2
>>> next(nums_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```


Python Iterable [\[source\]](#)

Iterable definition: "An object capable of returning its members one at a time. Examples of iterables include all sequence types (such as [list](#), [str](#), and [tuple](#)) and some non-sequence types like [dict](#), [file objects](#), and **objects of any classes you define with an `__iter__()` method** or with a `__getitem__()` method that implements [sequence](#) semantics."

```
class IterableInterface:
    """An illustration of Python's Iterable interface"""
    def __iter__(self):
        """Return an Iterator."""
        raise NotImplementedError("Override me!")
```

```
# Many of py built-in datatypes already
# implement the Iterable interface
>>> list_iter = iter([1, 2, 3])
>>> tuple_iter = iter((1, 2, 3))
>>> str_iter = iter('hi')
>>> dict_iter = iter({'a': 1, 'b': 2})
# Tip: dict iterators iterate over keys
>>> next(dict_iter)
a
>>> next(dict_iter)
b
```

Case study: `LinkIterable`

- Let's explore augmenting our `Link` (linked list) class to support Python's Iterable interface
- First, let's implement a `LinkIterator` that knows how to iterate over a `Link`


```
class IteratorInterface:
    """An illustration of Python's Iterator interface"""
    def __next__(self):
        """Returns next item, or raise StopIteration if at end"""
        raise NotImplementedError("Override me!")

    def __iter__(self):
        """Return iter instance, often self"""
        return self
```

```
>>> lnk1 = Link(1, Link(2, Link('meow')))
>>> lnk1_iterator = LinkIterator(lnk1)
>>> next(lnk1_iterator)
Link(1, Link(2, Link('meow')))
>>> next(lnk1_iterator)
Link(2, Link('meow'))
>>> next(lnk1_iterator)
Link('meow')
>>> next(lnk1_iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File
"C:\Users\Eric\teaching\data_c88c\s25\lectures\s25\c88c\17.py",
line 60, in __next__
    raise StopIteration()
StopIteration
```

```
class LinkIterator(IteratorInterface):
    """Iterator over linked lists (`LinkIterable`)"""
    def __init__(self, lnk_start):
        self.lnk_start = lnk_start
        self.lnk_cur = self.lnk_start

    def __next__(self):
        if self.lnk_cur == LinkIterable.empty:
            raise StopIteration()
        out = self.lnk_cur
        self.lnk_cur = self.lnk_cur.rest
        return out
```



This signals that
we've reached the
end of the linked list

(Demo: 17.py:Demo01)

Case study: `LinkIterable`

- Next, let's implement a `LinkIterable` that implements the Iterable interface

```
class IterableInterface:
    """An illustration of Python's Iterable interface"""
    def __iter__(self):
        """Return an Iterator."""
        raise NotImplementedError("Override me!")
```

```
class LinkIterable(IterableInterface):
    """Linked list that supports py Iterable interface"""
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __iter__(self):
        return LinkIterator(self)
```

```
>>> lnk1_iterable = LinkIterable(2, LinkIterable(3, LinkIterable('meow')))
>>> for lnk in lnk1_iterable:
...     print(lnk.first)
...
2
3
meow
```

```
# Under the hood, for loop is doing something like:
def for_loop_sim(some_iterable):
    some_iterator = iter(some_iterable)
    while True:
        try:
            thing = next(some_iterator)
        except StopIteration:
            return
        print(thing) # process thing
```

(Demo: 17.py:Demo02)

Interfaces

- A common way to organize software projects is to define interfaces (aka "contracts")
- Allows code to be generic and handle a wide variety of usecases
 - Ex: by providing the Iterator and Iterable interfaces, developers like you and me can take advantage of all of Python's iteration conveniences (ex: for loop, list comprehension) for my custom user-defined classes

```
class IteratorInterface:
    """An illustration of Python's Iterator interface"""
    def __next__(self):
        """Returns next item, or raise StopIteration if at end"""
        raise NotImplementedError("Override me!")

    def __iter__(self):
        """Return iter instance, often self"""
        return self

class IterableInterface:
    """An illustration of Python's Iterable interface"""
    def __iter__(self):
        """Return an Iterator."""
        raise NotImplementedError("Override me!")
```

Programming languages like Java/C++ support interface-like behavior via interface / abstract classes (covered in cs61B!)

```
class LinkIterator(IteratorInterface):
    """Iterator over linked lists (`LinkIterable`)"""
    def __init__(self, lnk_start):
        self.lnk_start = lnk_start
        self.lnk_cur = self.lnk_start

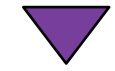
    def __next__(self):
        if self.lnk_cur == LinkIterable.empty:
            raise StopIteration()
        out = self.lnk_cur
        self.lnk_cur = self.lnk_cur.rest
        return out

class LinkIterable(IterableInterface):
    """Linked list that supports py Iterable interface"""
    empty = ()
    def __init__(self, first, rest=empty):
        self.first = first
        self.rest = rest

    def __iter__(self):
        return LinkIterator(self)
```

Discussion Question

What will be printed?



```
a = [1, 2, 3]
b = [a, 4]
c = iter(a)
d = c
print(next(c))
print(next(d))
print(b)
```

Answer:

```
>>> print(next(c))
1
>>> print(next(d))
2
>>> print(b)
[[1, 2, 3], 4]
```

Map Function

Map

`map(func, iterable)`: Make an iterator over the return values of calling `func` on each element of the iterable.

(Demo: 17.py:Demo03)

Discussion Question

all(s) iterates through s until a false value is found (or the end is reached).

What's printed when evaluating:

```
x = all(map(print, range(-3, 3)))
```

Why?

- print(-3) returns None after displaying -3
- None is a false value
- all([None, ...]) is False for any ...
- The map iterator never needs to advance beyond -3