Iterators and Generators

Announcements

Midterm this Thursday 7/13 7–9pm

- Alteration request form is still open until the day of the exam for last minute/emergencies

Topical Review Sessions start today. Check out Ed

HW 2 recovery is released and due next Monday 7/17

Lab 5 is released and due Tuesday 7/11

HW 3 is released and due Friday 7/14. We highly recommend completing before the midterm.

Cats is released and checkpoint is due tomorrow, Tuesday 7/11. Whole project is due Tuesday 7/18

Lab 6 was released early for extra practice opportunities, due on 7/13

Check discussion participation on Gradescope, instructions on Ed

As a reminder be kind to staff and other students, citizenship policy in the syllabus

Iterables

Iterables

Iterables are objects (or data) that can be iterated over

This means it contains some elements in some order that can be kept track of by going from one element to the next; looped over

EG: can use inside for loop or list comprehension

list

0	1	2	3	4
1	2	3	4	5

"hello this is a string"

?

list_iterator instance

range instance

range(0, 6)

dict

"k1"	"v1"
"k2"	"v2"
"k3"	"v3"

tuple

0	1	2	3
0	9	8	7

Iterators

Iterators

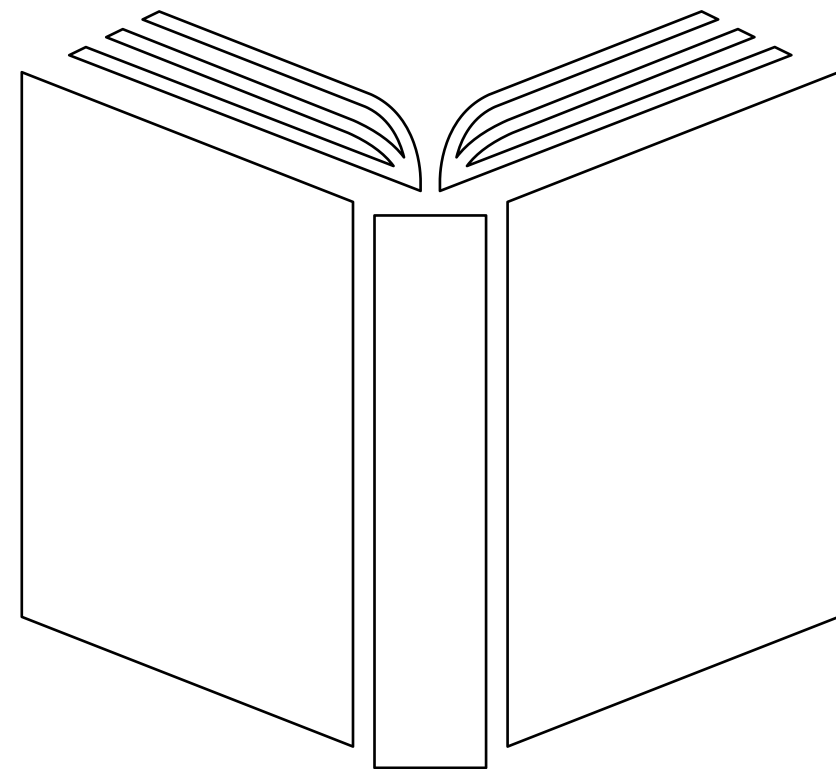
A container can provide an iterator that provides access to its elements in order

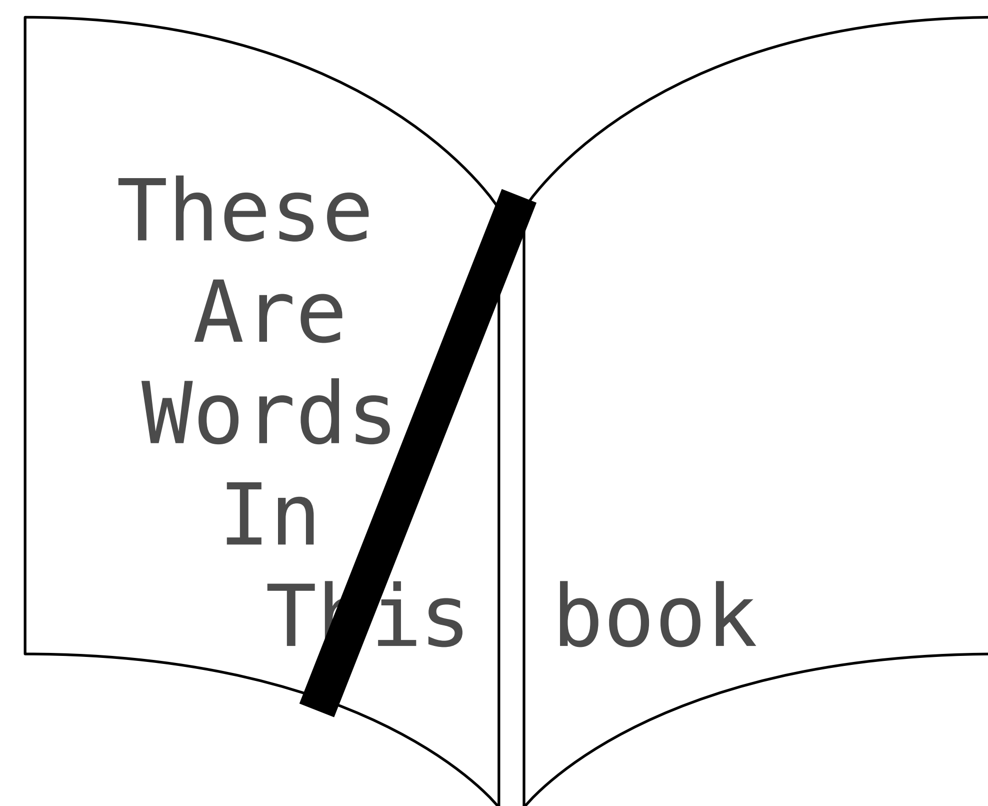
```
iter(iterable): Return an iterator over the elements
                of an iterable value

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

```
>>> s = [3, 4, 5]
>>> t = iter(s)
>>> next(t)
3
>>> next(t)
4
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(u)
4
```

(Demo)

Iterators are like bookmarks





All iterables

[1, 2, 3, 4]

{‘k’: ‘v’}

“Hello world”

All iterators

iter([1, 2, 3, 4])

Objects with “___ is a(n) ___” relationship

Dictionary Iteration

Views of a Dictionary

An *iterable* value is any value that can be passed to **iter** to produce an iterator

An *iterator* is returned from **iter** and can be passed to **next**; all iterators are mutable

A dictionary, its keys, its values, and its items are all iterable values

- The order of items in a dictionary is the order in which they were added (Python 3.6+)
- Historically, items appeared in an arbitrary order (Python 3.5 and earlier)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d.keys())  # or iter(d)
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
```

```
>>> v = iter(d.values())
>>> next(v)
1
>>> next(v)
2
>>> next(v)
3
>>> next(v)
0
```

```
>>> i = iter(d.items())
>>> next(i)
('one', 1)
>>> next(i)
('two', 2)
>>> next(i)
('three', 3)
>>> next(i)
('zero', 0)
```

(Demo)

For Statements

(Demo)

Built-In Iterator Functions

Built-in Functions for Iteration

Many built-in Python sequence operations return iterators that compute results lazily

`map(func, iterable):`	Iterate over func(x) for x in iterable
`filter(func, iterable):`	Iterate over x in iterable if func(x)
`zip(first_iter, second_iter):`	Iterate over co-indexed (x, y) pairs
`reversed(sequence):`	Iterate over x in a sequence in reverse order

To view the contents of an iterator, place the resulting elements into a container

`list(iterable):`	Create a list containing all x in iterable
`tuple(iterable):`	Create a tuple containing all x in iterable
`sorted(iterable):`	Create a sorted list containing x in iterable

(Demo)

Zip

The Zip Function

The built-in **zip** function returns an iterator over co-indexed tuples.

```
>>> list(zip([1, 2], [3, 4]))
[(1, 3), (2, 4)]
```

If one iterable is longer than the other, **zip** only iterates over matches and skips extras.

```
>>> list(zip([1, 2], [3, 4, 5]))
[(1, 3), (2, 4)]
```

More than two iterables can be passed to **zip**.

```
>>> list(zip([1, 2], [3, 4, 5], [6, 7]))
[(1, 3, 6), (2, 4, 7)]
```

Implement **palindrome**, which returns whether s is the same forward and backward.

```
>>> palindrome([3, 1, 4, 1, 3])
True
>>> palindrome([3, 1, 4, 1, 5])
False
```

```
>>> palindrome('seveneves')
True
>>> palindrome('seven eves')
False
```

Using Iterators

Reasons for Using Iterators

Code that processes an iterator (via **next**) or iterable (via **for** or **iter**) makes few assumptions about the data itself.

- Changing the data representation from a **list** to a **tuple**, **map object**, or **dict_keys** doesn't require rewriting code.
- Others are more likely to be able to use your code on their data.

An iterator bundles together a sequence and a position within that sequence as one object.

- Passing that object to another function always retains the position.
- Useful for ensuring that each element of a sequence is processed only once.
- Limits the operations that can be performed on the sequence to only requesting **next**.

Example: Casino Blackjack

Walkthrough video: https://youtu.be/p0acHiRp44M

Generators

Generators and Generator Functions

```
>>> def plus_minus(x):
...     yield x
...     yield -x

>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

A *generator function* is a function that **yield**s values instead of **return**ing them

A normal function **return**s once; a *generator function* can **yield** multiple times

A *generator* is an iterator created automatically by calling a *generator function*

When a *generator function* is called, it returns a *generator* that iterates over its yields

(Demo)

Objects with "___ is a(n) ___" relationship

Iterators are like bookmarks

All iterables

{'k': 'v'}

[1, 2, 3, 4]

All iterators

iter([1, 2, 3, 4])

All generators

"Hello world"

t = plus_minus(3)

Generators & Iterators

Generator Functions can Yield from Iterables

A **yield from** statement yields all values from an iterator or iterable (Python 3.3)

```
>>> list(a_then_b([3, 4], [5, 6]))
[3, 4, 5, 6]
```

```
def a_then_b(a, b):
    for x in a:
        yield x
    for x in b:
        yield x
```

```
def a_then_b(a, b):
    yield from a
    yield from b
```

```
>>> list(countdown(5))
[5, 4, 3, 2, 1]
```

```
def countdown(k):
    if k > 0:
        yield k
        yield from countdown(k-1)
```

(Demo)

Example: Partitions

Yielding Partitions

A partition of a positive integer n, using parts up to size m, is a way in which n can be expressed as the sum of positive integer parts up to m in increasing order.

partitions(6, 4)

```
2 + 4 = 6
1 + 1 + 4 = 6
3 + 3 = 6
1 + 2 + 3 = 6
1 + 1 + 1 + 3 = 6
2 + 2 + 2 = 6
1 + 1 + 2 + 2 = 6
1 + 1 + 1 + 1 + 2 = 6
1 + 1 + 1 + 1 + 1 + 1 = 6
```

```
def count_partitions(n, m):
    if n == 0:
        return 1
    elif n < 0:
        return 0
    elif m == 0:
        return 0
    else:
        with_m = count_partitions(n-m, m)
        without_m = count_partitions(n, m-1)
        return with_m + without_m
```

(Demo)