UC Berkeley EECS
Lecturer
Michael Ball

Iterators and Generators

Today:

- Sequences vs Iterables
- Using iterators without generating all the data
- Generator concept
  - Generating an iterator from iteration with yield
- Magic methods
  - next
  - iter
- Iterators – the iter protocol
- Getitem protocol
- Is an object iterable?
- Lazy evaluation with iterators

Review: Why Object-Oriented Design?

- Approach creation of a class as a design problem
  - Meaningful behavior => methods [& attributes]
  - ADT methodology
  - What's private and hidden? vs What's public?
- Design for inheritance
  - Clean general case as foundation for specialized subclasses
- Use it to streamline development

- Anticipate exceptional cases and unforeseen problems
  - try … catch
  - raise / assert

Review: What is a **sequence?** [Docs]

- Sequence is an "ordered set"
  - list
  - tuples
  - ranges
  - strings
- Some common operations:
  - Slicing syntax: `data[1:4]`
  - Membership: `'cs88' in courses`
  - Concatenation: `breakfast_foods + lunch_foods + dinner_foods`
  - Count Items: `'cs88'.count('8')`

Iterable - an object you can iterate over

- iterable: An object capable of yielding its members one at a time.
- iterator: An object representing a stream of data.
- We have worked with many iterables as if they were sequences

Functions that return iterables

- map
- filter
- zip

- These objects are **not** sequences.
- They are *generators,* or iterables. A "stream" of data we can iterate over.
- Why?
  - Can't directly slice into them.
  - Don't know their length
- If we want to see all the elements at once, we need to explicitly call list() or tuple() on them

Using a Generator

- Calling list() works, but it builds the result in one go.
  - This loses the benefits when we have large data!
- Generators allow us to successively *generate* (get it?) the next result!

```
data = map(lambda x: x*x, range(5))
# Iterate with for loops
for point in data:
    print(point)

data = map(lambda x: x*x, range(5))
next(data) # returns 0
next(data) # returns 1 …
next(data) # eventually raises StopIteration error
```

Generators: turning iteration into an iterable

- *Generator* functions use iteration (for loops, while loops) and the yield keyword
- Generator functions have no return statement, but they don't return None
- They implicitly return a generator object
- Generator objects are just iterators

```
def squares(n):
    for i in range(n):
        yield (i*i)
```

Nest iteration

```
def all_pairs(x):
    for item1 in x:
        for item2 in x:
            yield(item1, item2)
```

Iterables

Demo

Next element in generator iterable

- Iterables work because they have some "magic methods" on them. We saw magic methods when we learned about classes,
- e.g., __init__, __repr__ and __str__.
- The first one we see for iterables is `__next__`

- `iter( )` – transforms a sequence into asn iterator

Iterators: The iter protocol

- In order to be *iterable*, a class must implement the iter protocol
- The iterator objects themselves are required to support the following two methods, which together form the iterator protocol:
  - `__iter__()` : Return the iterator object itself. This is required to allow both containers and iterators to be used with the for and in statements.
    - » This method returns an iterator object (which can be `self`)
  - `__next__()` : Return the next item from the container. If there are no further items, raise the `StopIteration` exception.
- Classes get to define how they are iterated over by defining these methods
  - containers (objects like lists, tuples, etc) typically define a Container class and a separate `ContainterIterator` class.

Get Item protocol

- Another way an object can behave like a sequence is indexing: Using square brackets "[ ]" to access specific items in an object.
- Defined by special method: __getitem__(self, i)
  - Method returns the item at a given index

```
class myrange2:
    def __init__(self, n):
        self.n = n

    def __getitem__(self, i):
        if i >= 0 and i < self.n:
            return i
        else:
            raise IndexError

    def __len__(self):
        return self.n
```

Determining if an object is iterable

- from collections.abc import Iterable
- isinstance([1,2,3], Iterable)

- This is more general than checking for any list of particular type, e.g., list, tuple, string...