UC Berkeley EECS
Lecturer
Michael Ball

Iterators and Generators

Announcements & Strikes

- There will be a strike by Lecturers, Wednesday and Thursday
- Expect to have a lot of classes cancelled — including CS88
  - Hundreds of other professors will be cancelling lectures too
  - Lab sections *might* still continue. Your GSIs can choose to not hold class if they would like.
- Why?
  - In short, the Lecturer's Union and UCOP have been at the bargaining table for nearly 3 years, and there's been an expired contract for nearly 2 years.

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

Summary of last week

- 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

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
- range
- zip

- These objects are not sequences.
- If we want to see all of the elements at once, we need to explicitly call list() or tuple() on them

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)
```

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 an iterator

Iterators – 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, Iterator 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

Getitem 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...