Computational Structures in Data Science

Iterators and Generators (Part 2)

Berkeley
UNIVERSITY OF CALIFORNIA

Today:

- Pick up where we left off!
- Iterators – the iter protocol
- Getitem protocol
- Is an object iterable?
- Lazy evaluation with iterators

Computational Structures in Data Science

Iterators

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org

What's an Iterator? [Docs]

iterator

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.

iterable

An object capable of returning its members one at a time. Examples of include all sequence types and objects of any classes you define with an `__iter__()` method or with a `__getitem__()` method that implements sequence semantics.

Next element in generator iterable

- Iterables work because they implement 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
  - Usually this is not necessary, but can be useful.

Iterators: The `iter` protocol [Docs]

- 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.

The Iter Protocol In Practice

- 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.
- Lists, Ranges, etc are *not* directly iterators
  - We cannot call next() on them.
  - We can all iter(list), iter(range), etc if needed.
  - However, they implement an `__iter__` method, and `list_iterator`, `range_iterator` class, etc.

Iterables

Demo

Computational Structures in Data Science

Building a Range Iterator

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org |

Making a Range Iterator

- What does a range need?
  - Start value
  - Stop
  - (We'll ignore step sizes)
- keep track of the current value
- An __iter__ method
- A __next__ method

Example

```
class myrange:
    def __init__(self, n):
        self.i = 0
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.i < self.n:
            current = self.i
            self.i += 1
            return current
        else:
            raise StopIteration()
```


Computational Structures in Data Science

The GetItem Protocol

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org |

Get Item protocol – Build a Sequene

- 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
```

Get Item Protocol

- When `__iter__` isn't defined, check if `__getitem__` exists
- `__getitem__` must accept integers as indices
  - Start at 0
  - Continue iterating until `IndexError` is raised

- This is an older way of making iterators.
- Why two ways?
  - Languages evolve over time!
  - There's often more than one valid design.

Get Item Protocol [Docs]

```
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
```

Computational Structures in Data Science

Iterators and Generators Review

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org |

Terms and Tools

- **Iterators:** Objects which we can use in a for loop
  - Anything that can be looped over!
  - Sometimes they're lazy, sometimes not!
- **Generators:** A shorthand way to make an iterator that uses yield
  - a function that uses `yield` is a *generator function*
  - a generator function returns a *generator object*
  - Generators do **not** use return
- **Sequences:** A particular type of iterable
  - They know they're length, support slicing
  - Are *not* lazy

What's the Big Picture?

- We have new tools for building data structures that behave sequences
- We can handle "infinite" streams of data.
- We can build our own for loops, perhaps custom for loops.

What can we do now?

- Build our own `for`-loop like functions!
- Python doesn't let us extend built in keywords
- So we can make a function like `doFor(sequence, action)`
  - Is the sequence already an iterator? → Use next()
  - Can we call `iter(sequence)`? → Use next()
  - Can we call `sequence[0]`? → Use Indexing
  - Now we can get items
  - We can call fn(some_item) until:
    - We catch `StopIteration` or `IndexError`
    - Other Errors we should probably not address

Computational Structures in Data Science

Type Checking

Berkeley
UNIVERSITY OF CALIFORNIA

Michael Ball | UC Berkeley | https://c88c.org |

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...