Iterators & Generators

Sequences

- A sequence is something that you can:
  - Index into
  - Get the length of

- What are some examples of sequences?

Sequences

- We’ve been working with sequences all semester!
- Examples:
  - Lists
  - Tuples
  - Strings

- Note: a dictionary is NOT a sequence

Demo

Iterables

- Any object that you can use a for loop over (more technical def. later)
- All sequences are iterable
- Examples:
  - Lists
  - Strings
  - Tuples
  - Dictionaries

Iterables

- So far, we have been treating iterables as sequences
- All sequences are iterables, but not at all iterables are sequences

- For example:
  - We can loop over elements of range(5) one at a time
  - What happens when we try to look at the whole range?

Iterables

- Functions that return objects that we can iterate over:
  - Range
  - Zip
  - Map
- 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

Demo

Motivating Questions

- How can we define things that work like any sequence without having to explicitly create these sequences?
  - The two implementations we will look at today are **iterators** and **generators**
- Why would we want to do this?

Iterators

- Classes define what it means to iterate over them
- In order to do this, the class must define an **iterator**

- **Iterator:** A special object that handles logic for iterating over another object
- An object can be its own iterator

Iterators

- In order to be iterable, a class must implement the `__iter__(self)` method
  - This method returns an **iterator** object
  - Iterator can be self
- An iterator must implement the `__next__(self)` method
- When doing a for loop over a sequence, python implicitly gets the iterator of the sequence and repeatedly calls `next` on it.

__next__(self)

- Accessed via the `next` method
- Returns the next element in the iteration
  - Must keep track of where it is in the sequence
- Once there are no more items left in the sequence, raise an exception:
  - `raise StopIteration`

- We'll learn more about exceptions later. They're like a special kind of return.

Demo

Generators

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

Generators

- Generators can often be easier to implement than iterators
- Example:
  - Enumerate pairs of elements in a sequence, s
  - How would you implement an iterator that outputs these pairs?

```
>>> list(all_pairs([1, 2, 3]))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Generators

- We can do this vary naturally using a generator function

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

```
>>> list(all_pairs([1, 2, 3]))
[(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3), (3, 1), (3, 2), (3, 3)]
```

Demo

What is the point?

- Iterators and generators let us capture potentially vast amounts of data without computing them right away
  - range(1,000,000,000,000) gives us one object, not a list of a trillion numbers!
- When we need the numbers, we can ask for them on demand - a concept called **lazy evaluation**
- Big implications for big data

Indexing

- There is another way an object can behave like a sequence: **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
- We (the makers of the class) get to decide what an index represents
  - Sequences: The item at a position in the sequence
  - Dictionaries: The value associated with a given key
  - Arrays: Index is a tuple representing the coordinate of the item

Indexing

- We are also responsible for deciding when an index is not valid for an object
  - `raise IndexError`

Demo

Magic methods

- Surrounded by double underscores
  - `__getitem__`, `__repr__`, `__str__`, `__next__`, `__iter__`

- Define behavior for special operators
  - len(x) → you are implicitly calling the `__len__()` method
  - x[5] → you are implicitly calling the `__getitem__()` method
  - list1 + list2 → you are implicitly calling the `__add__()` method

Magic methods

- Magic methods allow us to…
  - give meaning to operators (+, -, <, ==) …
  - so that we can use our own classes (Tree, VendingMachine) …
  - just like we use built-in types (lists, integers)
- Want to compare two Trees using ==?
  - Make a `__eq__(self, other)` method in your Tree class
- Want to multiply VendingMachines together using *?
  - Make a `__mul__(self, other)` method in your VendingMachine class

Magic methods

- Want to create your own iterators that can be used in a **for** loop?
  - Implement the `__iter__()` and `__next__()` methods
  - `__iter__(self)` should return an iterator
  - `__next__(self)` gets the next value in the iteration + updates the current position