Lecture 11: Iterators & Generators

CS 61A - Summer 2024
Raymond Tan

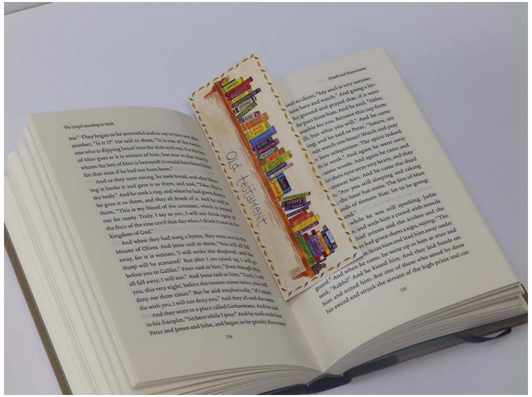
Iterators

- 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

Demo: Iterators

Bookmark Analogy

- Iterators can be thought of as “bookmarks” for a corresponding iterable
  - Pages of a book represent items in an iterable
- Calling `next` on the iterator gives us the next item in the sequence
  - Until the bookmark reaches the very end of the iterable, where calling `next` now returns an error



Iterable vs Iterator

- An *iterable* value is any value that can be passed to `iter` to produce an iterator
- An *iterator* is returned from calling `iter` on an *iterable*, and can be passed to `next`; all iterators are mutable
- **All iterators are iterable, not all iterables are iterators**

Iterators with for statements

- Reminder: Syntax of a `for` statement

```
for <var> in <iterable>:
    # Body of the loop
```

Since all iterators are iterable, we can iterate over any iterator using a `for` statement

Dictionary Iteration

- 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)

Demo: Dictionary Iteration

Why do we use 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.

Built-In Functions for Iteration

- Many Built-In Functions for Python return iterators that compute 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

map: Example

```
>>> f1 = lambda x : x ** 2
>>> s = [3, 4, 5]
>>> a = map(f1, s)
>>> a
<map object at 0x102812fb0>
>>> next(a)
9
>>> next(a)
16
>>> next(a)
25
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A map object is an iterator!

filter: Example

```
>>> f1 = lambda x : x % 2 == 0
>>> s = [3, 4, 5]
>>> a = filter(f1, s)
>>> a
<filter object at 0x102c33df0>
>>> next(a)
4
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A filter object is an iterator!

zip: Example

```
>>> s1 = ['a', 'b', 'c']
>>> s2 = [1, 2, 3]
>>> a = zip(s1, s2)
>>> a
<zip object at 0x102864ac0>
>>> next(a)
('a', 1)
>>> next(a)
('b', 2)
>>> next(a)
('c', 3)
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

A zip object is an iterator!

zip: Example

```
>>> s1 = ['a', 'b']
>>> s2 = [1, 2, 3]
>>> a = zip(s1, s2)
>>> next(a)
('a', 1)
>>> next(a)
('b', 2)
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Iterables passed to zip do not have to be the same length – the number of elements in the zip object is equal to the length of the shorter one

reversed: Example

```
>>> a = [1, 2, 3, 4]
>>> s1 = [1, 2, 3, 4]
>>> a = reversed(s1)
>>> a
<list_reverseiterator object at 0x102c33e50>
>>> next(a)
4
>>> next(a)
3
>>> next(a)
2
>>> next(a)
1
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

reversed: Example

```
>>> s2 = set()
>>> s2.add(1)
>>> s2.add(2)
>>> a = reversed(s2)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'set' object is not reversible
```

Not all iterators are reversible!

*sets are out of scope for CS 61A

Viewing contents of an Iterator

- To view the entire contents of an iterator, you can place the resulting elements into a container
  - `list(iterator)`: Create a list containing all x in `iterator`
  - `tuple(iterator)`: Create a tuple containing all x in `iterator`
  - `sorted(iterator)`: Create a sorted list containing x in `iterator`
- Note that doing so will “deplete” the iterator!
  - Calling next on the iterator afterwards would result in a StopIteration error
- Beware: Some iterators can be infinite. Using this method would cause your code to timeout when called on an infinite iterator

Mutating Underlying Iterables

```
>>> s = [1, 2, 3]
>>> a = iter(s)
>>> next(a)
1
>>> next(a)
2
>>> next(a)
3
>>> s.append(4)
>>> next(a)
4
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> s.append(5)
>>> next(a)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

We can mutate the underlying iterable object that an iterator is tracking, and the iterator will account for any extra items we add!

Break

Generators

Generators

- Generators allow us to define our own iterators over code we write

```
>>> def plus_minus(x):
...     yield x
...     yield -x
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus ...>
```

Generators & Generator Functions

- A ***generator function*** is a function that **yields** values instead of returning them
  - A normal function returns once; a generator function can yield multiple times
  - If a function has `yield` in its body, it is a generator function
- 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 `yield` statements
  - This is different from a normal function, where calling on the function immediately executes the body

Demo: Generators

Generators cont.

- When `next` is called on a generator, Python looks to the generator function and executes the body of the function until it hits a `yield` statement
    - At this point, this value is returned, and Python remembers where it left off
- Calling `next` again on the generator will pick back up at the place the previous call to `next` left off at
- If calling `next` on a generator that has no more `yield` statements in the body, then a StopIteration error occurs

yield from

- A `yield from` statement yields all values from an iterator or iterable

```python
def yield_odds(s):
    """Yield all the odd numbers in the iterable s."""
    for x in s:
        if x % 2 == 1:
            yield x
```

```python
def yield_odds(s):
    """Yield all the odd numbers in the iterable s."""
    yield from [x for x in s if x % 2 == 1]
```

Side note: What is wrong with this approach?

```python
def yield_odds(s):
    """Yield all the odd numbers in the iterable s."""
    for i in range(len(s)):
        element = s[i]
        if element % 2 == 1:
            yield element
```

The docstring states that `s` is an ***iterable***. We can't call `len` or index into all iterables!

Ex: `s` could be an iterator

Infinite Generators

- We can take advantage of the property of **lazy evaluation** of generators to create iterators over infinite sequences
- For example, we can create an iterator over the even natural numbers

```python
def evens():
    x = 0
    while True:
        yield x
        x += 2
```

Example: count_partitions revisited

- Definition: 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 to `m` in increasing order.
- We want to yield the string representation of all partitions possible

```
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
```

Example: count_partitions revisited

```python
def count_partitions(n, m):
    if n == m:
        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
```

```python
def yield_partitions(n, m):
    """List partitions.

    >>> for p in yield_partitions(6, 4): print(p)
    2 + 4
    1 + 1 + 4
    3 + 3
    1 + 2 + 3
    1 + 1 + 1 + 3
    2 + 2 + 2
    1 + 1 + 2 + 2
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 1 + 1 + 1
    """
    if n > 0 and m > 0:
        if _____:
            yield str(m)
        for p in __________:
            yield p + ' + ' + str(m)
        __________________________
```

count_partitions Solution

```
def yield_partitions(n, m):
    """List partitions.

    >>> for p in yield_partitions(6, 4): print(p)
    2 + 4
    1 + 1 + 4
    3 + 3
    1 + 2 + 3
    1 + 1 + 1 + 3
    2 + 2 + 2
    1 + 1 + 2 + 2
    1 + 1 + 1 + 1 + 2
    1 + 1 + 1 + 1 + 1 + 1
    """
    if n > 0 and m > 0:
        if n == m:
            yield str(m)
        for p in yield_partitions(n-m, m):
            yield p + ' + ' + str(m)
        yield from yield_partitions(n, m-1)
```

Summary

- A container can provide an ***iterator*** that provides access to its elements in order
  - Use `iter` to create the iterator, and `next` to get the next element in the iterator
- All iterators are iterable
- Generators allow us to create custom iterators
  - The lazy evaluation characteristic of generators allow us to create iterators over infinite sequences
  - The `yield` keyword is what distinguishes a generator function from a normal function to Python
- `yield from` allows us to yield directly from the items of an iterable