CS 61A Fall 2023

Iterators, Generators

Discussion 6: October 4, 2023

Walkthrough Videos

Feel free to try these problems on the worksheet in discussion or on your own, and then come back to reference these walkthrough videos as you study.

To see these videos, you should be logged into your berkeley.edu email.

Iterators

An iterable is an object whose elements we can go through one at a time. Iterables can be used in for loops and list comprehensions. Iterables can also be converted into lists using the `list` function. Examples of iterables we have seen so far in Python include strings, lists, tuples, and ranges.

```
>>> for x in "cat":
...     print(x)
c
a
t
>>> [x*2 for x in (1, 2, 3)]
[2, 4, 6]
>>> list(range(4))
[0, 1, 2, 3]
```

Note the abstraction present here: given some iterable object, we have a set of defined actions that we can take with it. In this discussion, we will peak below the abstraction barrier and examine how iterables are implemented "under the hood".

In Python, **iterables** are formally implemented as objects that can be passed into the built-in `iter` function to produce an *iterator*. An **iterator** is another type of object that can produce elements one at a time with the `next` function.

- `iter(iterable)`: Returns an iterator over the elements of the given iterable.
- `next(iterator)`: Returns the next element in an iterator, or raises a `StopIteration` exception if there are no elements left.

For example, a list of numbers is an iterable, since `iter` gives us an iterator over the given sequence, which we can navigate using the `next` function:

```
>>> lst = [1, 2, 3]
>>> lst_iter = iter(lst)
>>> lst_iter
<list_iterator object ...>
>>> next(lst)
1
>>> next(lst)
2
>>> next(lst)
3
>>> next(lst)
StopIteration
```

Iterators are very simple. There is only a mechanism to get the next element in the iterator: `next`. There is no way to index into an iterator and there is no way to go backward. Once an iterator has produced an element, there is no way for us to get that element again unless we store it.

Note that iterators themselves are iterables: calling `iter` on an iterator simply returns the same iterator object.

For example, we can see what happens when we use `iter` and `next` with a list:

```
>>> lst = [1, 2, 3]
>>> next(lst)             # Calling next on an iterable
TypeError: 'list' object is not an iterator
>>> list_iter = iter(lst) # Creates an iterator for the list
>>> next(list_iter)       # Calling next on an iterator
1
>>> next(iter(list_iter)) # Calling iter on an iterator returns itself
2
>>> for e in list_iter:   # Exhausts remainder of list_iter
...     print(e)
3
>>> next(list_iter)       # No elements left!
StopIteration
>>> lst                   # Original iterable is unaffected
[1, 2, 3]
```

The `map` and `filter` functions we learned earlier in class return iterator objects.

Q1: WWPD: Iterators

What would Python display?

```
>>> s = "cs61a"
>>> s_iter = iter(s)
>>> next(s_iter)
```

'c'

```
>>> next(s_iter)
```

's'

```
>>> list(s_iter)
```

['6', '1', 'a']

```
>>> s = [[1, 2, 3, 4]]
>>> i = iter(s)
>>> j = iter(next(i))
>>> next(j)
```

1

```
>>> s.append(5)
>>> next(i)
```

5

```
>>> next(j)
```

2

```
>>> list(j)
```

[3, 4]

```
>>> next(i)
```

StopIteration

Generators

We can define custom iterators by writing a *generator function*, which returns a special type of iterator called a **generator**.

A generator function looks like a normal Python function, except that it has at least one `yield` statement. When we call a generator function, a ***generator** object* is returned without evaluating the body of the generator function itself. (Note that this is different from ordinary Python functions. While generator functions and normal functions look the same, their evaluation rules are very different!)

When we first call `next` on the returned generator, we will begin evaluating the body of the generator function until an element is yielded or the function otherwise stops (such as if we `return`). The generator remembers where we stopped, and will continue evaluating from that stopping point on the next time we call `next`.

As with other iterators, if there are no more elements to be generated, then calling `next` on the generator will give us a `StopIteration`.

For example, here's a generator function:

```
def countdown(n):
    print("Beginning countdown!")
    while n >= 0:
        yield n
        n -= 1
    print("Blastoff!")
```

To create a new generator object, we can call the generator function. Each returned generator object from a function call will separately keep track of where it is in terms of evaluating the body of the function. Like all other iterators, calling `iter` on an existing generator object returns the same generator object.

```
>>> c1, c2 = countdown(2), countdown(2)
>>> c1 is iter(c1)  # a generator is an iterator
True
>>> c1 is c2
False
>>> next(c1)
Beginning countdown!
2
>>> next(c2)
Beginning countdown!
2
```

In a generator function, we can also have a `yield from` statement, which will **yield** each element **from** an iterator or iterable.

```
>>> def gen_list(lst):
...     yield from lst
...
>>> g = gen_list([1, 2])
>>> next(g)
1
>>> next(g)
2
>>> next(g)
StopIteration
```

Since generators are themselves iterators, this means we can use `yield from` to create recursive generators!

```
>>> def rec_countdown(n):
...     if n < 0:
...         print("Blastoff!")
...     else:
...         yield n
...         yield from rec_countdown(n-1)
...
>>> r = rec_countdown(2)
>>> next(r)
2
>>> next(r)
1
>>> next(r)
0
>>> next(r)
Blastoff!
StopIteration
```

Q2: WWPD: Generators

What would Python display? If the command errors, input the specific error.

```
>>> def infinite_generator(n):
...     yield n
...     while True:
...         n += 1
...         yield n
>>> next(infinite_generator)
```

TypeError

```
>>> gen_obj = infinite_generator(1)
>>> next(gen_obj)
```

1

```
>>> next(gen_obj)
```

2

```
>>> list(gen_obj)
```

Infinite Loop

```
>>> def rev_str(s):
...     for i in range(len(s)):
...         yield from s[i::-1]
>>> hey = rev_str("hey")
>>> next(hey)
```

'h'

```
>>> next(hey)
```

'e'

```
>>> next(hey)
```

'h'

```
>>> list(hey)
```

['y', 'e', 'h']

```
>>> def add_prefix(s, pre):
...     if not pre:
...         return
...     yield pre[0] + s
...     yield from add_prefix(s, pre[1:])
>>> school = add_prefix("schooler", ["pre", "middle", "high"])
>>> next(school)
```

'preschooler'

```
>>> list(map(lambda x: x[:-2], school))
```

['middleschool', 'highschool']

Q3: Filter-Iter

Implement a generator function called `filter_iter(iterable, f)` that only yields elements of `iterable` for which `f` returns `True`.

Remember, `iterable` could be infinite!

```
def filter_iter(iterable, f):
    """
    Generates elements of iterable for which f returns True.
    >>> is_even = lambda x: x % 2 == 0
    >>> list(filter_iter(range(5), is_even)) # a list of the values yielded from the call
     to filter_iter
    [0, 2, 4]
    >>> all_odd = (2*y-1 for y in range(5))
    >>> list(filter_iter(all_odd, is_even))
    []
    >>> naturals = (n for n in range(1, 100))
    >>> s = filter_iter(naturals, is_even)
    >>> next(s)
    2
    >>> next(s)
    4
    """
    for elem in iterable:
        if f(elem):
            yield elem

# Alternate solution (only works if iterable is not an infinite iterator)
def filter_iter(iterable, f):
    yield from [elem for elem in iterable if f(elem)]
```

Q4: What's the Difference?

Implement `differences`, a generator function that takes an iterable `it` whose elements are numbers. `differences` should produce a generator that yield the differences between successive terms of `it`. If `it` has less than 2 values, `differences` should yield nothing.

```
def differences(it):
    """
    Yields the differences between successive terms of iterable it.

    >>> d = differences(iter([5, 2, -100, 103]))
    >>> [next(d) for _ in range(3)]
    [-3, -102, 203]
    >>> list(differences([1]))
    []
    """
    iterator = iter(it)
    prev_term = next(iterator)
    for term in iterator:
        yield term - prev_term
        prev_term = term
```

Q5: Primes Generator

Write a function `primes_gen` that takes a single argument `n` and yields all prime numbers less than or equal to `n` in decreasing order. Assume `n >= 1`. You may use the `is_prime` function included below, which we implemented in Discussion 3.

First approach this problem using a `for` loop and using `yield`.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    for num in range(n, 1, -1):
        if is_prime(num):
            yield num
```

Now that you've done it using a `for` loop and `yield`, try using `yield from`!

Optional Challenge: Now rewrite the generator so that it also prints the primes in *ascending order*.

```
def is_prime(n):
    """Returns True if n is a prime number and False otherwise.
    >>> is_prime(2)
    True
    >>> is_prime(16)
    False
    >>> is_prime(521)
    True
    """
    def helper(i):
        if i > (n ** 0.5): # Could replace with i == n
            return True
        elif n % i == 0:
            return False
        return helper(i + 1)
    return helper(2)

def primes_gen(n):
    """Generates primes in decreasing order.
    >>> pg = primes_gen(7)
    >>> list(pg)
    [7, 5, 3, 2]
    """
    if n == 1:
        return
    if is_prime(n):
        yield n
    yield from primes_gen(n-1)
```

Q6: Stair Ways

In discussion 4, we considered how many different ways there are to climb a flight of stairs with `n` steps if you are able to take 1 or 2 steps at a time. In this problem, you will write a generator function `stair_ways` that yields all the different ways you can climb such a staircase.

Each "way" of climbing a staircase is represented by a list of `1`s and `2`s, representing the sequence of step sizes a person should take to climb the flight.

For example, for a flight with 3 steps, there are three ways to climb it: * You can take one step each time: `[1, 1, 1]`. * You can take two steps then one step: `[2, 1]`. * You can take one step then two steps: `[1, 2]`..

Therefore, `stair_ways(3)` should yield `[1, 1, 1]`, `[2, 1]`, and `[1, 2]` in any order.

```
def stair_ways(n):
    """
    Yields all ways to climb a set of N stairs taking
    1 or 2 steps at a time.

    >>> list(stair_ways(0))
    [[]]
    >>> s_w = stair_ways(4)
    >>> sorted([next(s_w) for _ in range(5)])
    [[1, 1, 1, 1], [1, 1, 2], [1, 2, 1], [2, 1, 1], [2, 2]]
    >>> list(s_w) # Ensure you're not yielding extra
    []
    """
    if n == 0:
        yield []
    elif n == 1:
        yield [1]
    else:
        for way in stair_ways(n - 1):
            yield [1] + way
        for way in stair_ways(n - 2):
            yield [2] + way
```