CS61A Lecture 16

Friday, October 4th, 2019

Announcements

- Professor DeNero will be out of town Monday and Wednesday next week.
- New project next Monday.

Iterators

A container, like a list or a tuple, can provide an iterator that lets you access its elements in order.

You can choose one of two iterators:

- The `iter(iterable)` function returns an iterator over the elements of an iterable value.
- You can only do one thing on an iterator: you can call the `next(iterator)` function to get the next element in an iterator.

For example:

```
>>> s = [3,4,5]
>>> t = iter(s)
>>> t
<list_iterator object...>
>>> next(t)
3
>>> next(t)
4
```

The iterator is really only information about the position! It is not the list itself.

If you create a new iterator, it does not mess with the previous name.

```
>>> u = iter(s)
>>> next(u)
3
>>> next(t)
5
>>> next(t)
Error
```

`next()` is technically a mutable function because every time you call it, it returns a different value, but that information is really in `t`, not the `next` function!

If you change the values of the container after the iterator is created, the behavior cannot always be predicted. Don't do that!

You can also use an iterator in a for statement, to go through all the values in the iterator, but that uses up the iterator.

Views of a dictionary

An iterable value is any value that can be passed to `iter` to produce an iterator.

An iterator is returned from `iter` and can be passed to `next`; all iterators are mutable.

If you mutate a dictionary after an iterator is created, Python will error when the iterator is called.

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)

```
>>> d = {'one': 1, 'two': 2, 'three': 3}
>>> d['zero'] = 0
>>> k = iter(d) # or k = iter(d.keys())
>>> next(k)
'one'
>>> next(k)
'two'
>>> next(k)
'three'
>>> next(k)
'zero'
>>> v = iter(d.values())
>>> next(v)
1
>>> i = iter(d.items())
>>> next(i)
('one', 1)
```

Why is all this useful?

Many built-in Python sequence operations return iterators that compute the results lazily, which is to say they do as little work as necessary.

```
map(func,iterable) # Iterate over func(x) for x in a variable
filter(func,iterable) # Iterate over x 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
```

Let's say we do this:

```
>>> def double(x):
    print("double", x, "=", 2*x)
    return 2*x
>>> [double(x) for x in range(3,7)]
"double 3 is 6"
```

```
6
"double 4 is 8"
8
"double 5 is 10"
10
"double 6 is 12"
12
```

But if we were to use a map function:

```
>>> m = map(double,range(3,7))
>>> next(m)
"double 3 is 6"
6
```

It doesn't double the range until the `next` function is called!

You can do the following operations on an iterator to view the remaining elements:

```
list(iterable) #Create a list containing all x in iterable
tuple(iterable) #Create a tuple containing all x in iterable
sorted(iterable) #Create a sorted list containing x in iterable
```

Generators

A generator function is a function that yields values instead of returning them.

```
def plus_minus(x):
    yield x
    yield -x
```

A normal function returns once, but a generator function can yield multiple times. The generator is an iterator that is created automatically by the yield statement.

```
>>> t = plus_minus(3)
>>> next(t)
3
>>> next(t)
-3
>>> t
<generator object plus_minus...>
```

When the generator function is called, it returns a generator that iterates over its yields.

Generators and Iterators

A `yield from` statement yields all values from an iterator or iterable (Python 3.3+)

```
>>> list(a_then_b([3,4],[5,6]))
[3,4,5,6]
```

We could write `a_then_b` one of two ways:

```
def a_then_b(a,b):
    for x in a:
        yield x
    for x in b:
        yield x

def a_then_b(a,b):
    yield from a
    yield from b
```

So we could:

```
>>> list(countdown(5))
[5,4,3,2,1]
```

How would we write `countdown` ?

```
def countdown(k)
    if k > 0:
        yield k
        yield from countdown(k-1)
```