CME 211: Extra Python Topics!

These notes contain some examples and pointers to various Python topics that I find myself using quite a bit.

Iterators

We've seen the `range` function in Python. In Python 2 `range` returned a list. In Python 3, `range` returns an "iterator", which avoids the memory allocation for a full list.

```
r = range(10)
print(r)
print(type(r))
```

We often write:

```
for i in range(10):
    print(i, end=" ")
print()
```

Python defines an interface for iterators. The key methods are `iter()` and `next()`:

```
r = iter(range(3)) # iter returns an iterator
print(next(r))
print(next(r))
print(next(r))
print(next(r))
```

Let's write our own simplified implementation of `range`:

```
class my_range:
    def __init__(self,n):
        self.i = 0
        self.n = n
    def __iter__(self):
        return self
    def __next__(self):
        if self.i == self.n:
            raise StopIteration
        t = self.i
        self.i += 1
        return t

for i in my_range(4):
    print(i,end=" ")
print()
```

An object with a `next()` method that behaves in the above manner is called an **iterator** in Python.

Generators

Defining a class for an iterator can be a bit verbose. Python has a keyword called `yield` which allows you to easily write a **generator**.

```
def my_range2(n):
    i = 0
    while i < n:
```

```
        yield i
        i += 1

for i in my_range2(4):
    print(i,end=" ")
print()
```

This allows you to create an iterator by just writing a function!

Application: iterating over words in a file

Let's say we wanted to count unique words in Shakespeare's entire body of work.

First, let's download it:

```
!wget https://ocw.mit.edu/ans7870/6/6.006/s08/lecturenotes/files/t8.shakespeare.txt
```

And inspect:

```
!head t8.shakespeare.txt
```

To loop over words, we could write code like:

```
with open("t8.shakespeare.txt","r") as f:
    for i, line in enumerate(f):
        # only loop over 10 lines
        if i > 10:
            break
        # loop over words
        for word in line.split():
            print(word)
```

Now let's say we have to loop over words of two different files. The code could get quite messy. Let's use a generator:

```
def words(filename):
    with open(filename,"r") as f:
        for line in f:
            for word in line.split():
                yield word
```

Now, the code that operates on words can be:

```
for i, word in enumerate(words("t8.shakespeare.txt")):
    if i > 10:
        break
    print(word)
```

Counting words

Let's use a dictionary to count words. Our first try might look like this:

```
word_count = {}
for word in words("t8.shakespeare.txt"):
    # let's clean the word up
    w = word.strip().lower()
    if w in word_count:
        # w is already in dict, so increment
        word_count[w] += 1
```

```
    else:
        # w is not in dict, do set to 1
        word_count[w] = 1
```

This is a common patters. The Python `collections` module has something called `defaultdict` that can help:

```
from collections import defaultdict
word_count = defaultdict(int)
for word in words("t8.shakespeare.txt"):
    w = word.strip().lower()
    word_count[w] += 1
```

Let's explore a little bit:

```
print(word_count['love'])
print(word_count['hate'])
print(word_count['king'])
print(word_count['queen'])
```

How might we get words with the highest count?

```
# create a list of tuples with (count, word)
word_count_list = [(c, w) for w, c in word_count.items()]
# sort will first sort by first element in the tuples
word_count_list.sort(reverse=True)
word_count_list[0:10]
```

Sweet!

Generator expressions

We've seen list comprehensions:

```
xs = list(range(10))
ys = [x*x for x in xs]
```

A list comprehension creates a entirely new list in memory and does all of the computation before the result (elements of `ys`) is used.

A generator expression creates an iterator that behaves like the list:

```
yg = (x*x for x in xs)
type(yg)
next(yg)
next(yg)
```

A generator expression does not create a list in memory. The computation is only performed when `next` is called.

You can pass a generator expression any where an iterator is expected:

```
xs = [1, 0, 0, 1, 0, 1]

# any returns True if any items of a collection (or iterator) are True
any(x == 1 for x in xs)

# all returns True only if all items in a collection (or iterator) are True
all(x == 1 for x in xs)
```

You can also use a generator expression in a `for` loop:

```
word_count = defaultdict(int)
for word in (w.strip().lower() for w in words("t8.shakespeare.txt")):
    word_count[w] += 1
```

Itertools

Have a look at the `itertools` module. It contains functions to help you write loops.

First, let's see `zip`, which is actually a Python built-in function:

```
letters = 'abcde'
numbers = [1,2,3,4,5]

# let's iterate over letters and numbers at same time
for l, n in zip(letters, numbers):
    print("{} {}".format(l, n))
```

Exercise: what does `zip` return? How can I get a list from it?

Now let's use the `product` function to iterate over all pairs:

```
from itertools import product
for l, n in product(letters, numbers):
    print("{} {}".format(l, n))
```