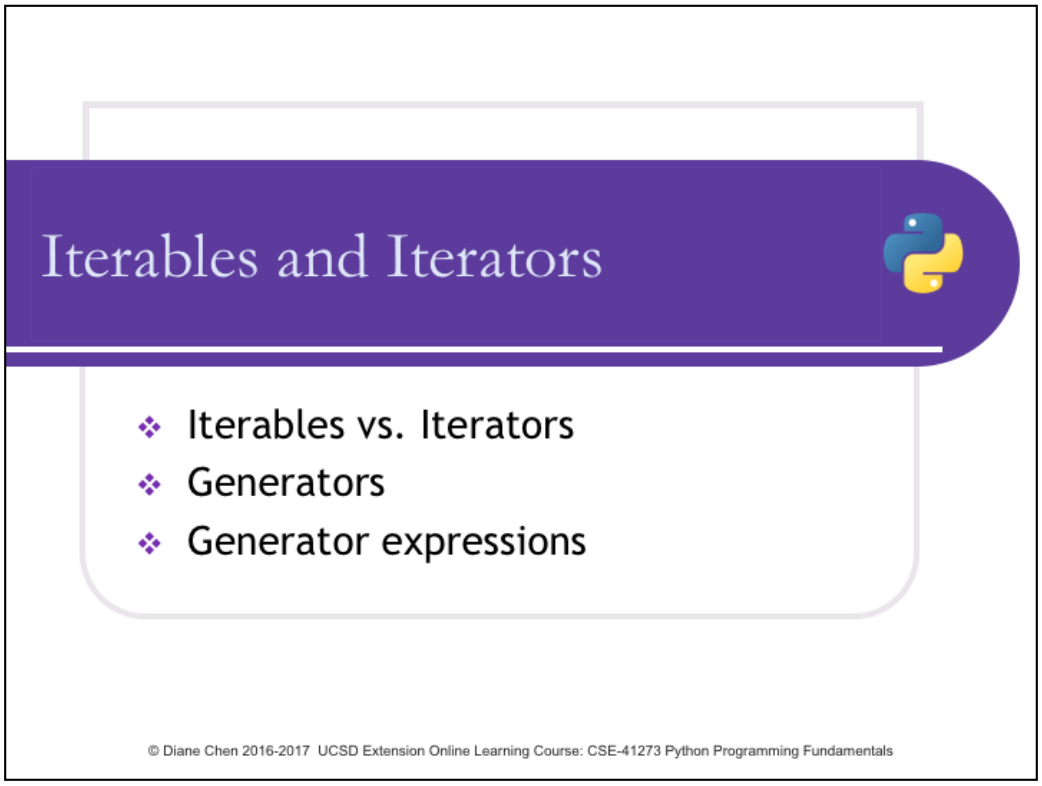
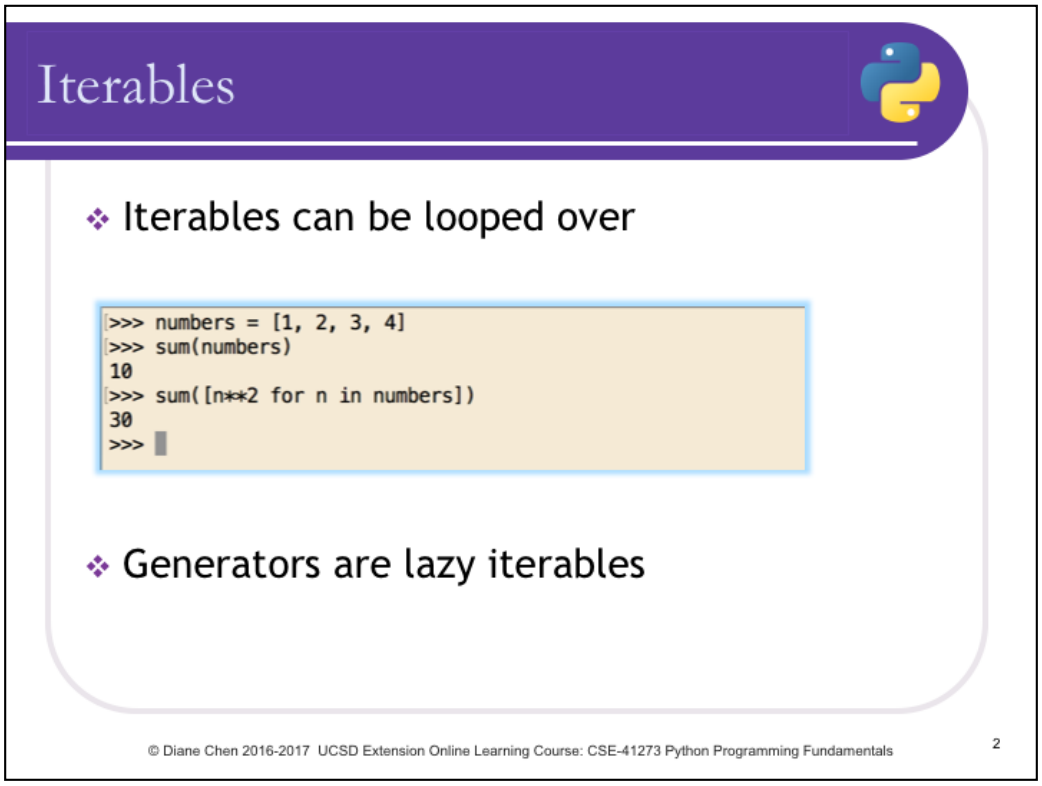


Now I'm going to talk about iterables and iterators. How to use them and what is the difference between the two. I'm also going to talk about generators and generator expressions. Generator expressions look just like comprehensions, but work a little differently because generators are a strange breed unto themselves.



As I mentioned previously, an iterable is something that can be looped over, like in a for loop. A list is an iterable. Python has functions that operate on iterables. List comprehensions return an iterable so we can pass list comprehensions straight into one of these functions. Let's use the function sum as an example. If we have a list of numbers, we can use it in sum. We can also pass a list comprehension to sum, for example to sum the squares of the list, as in our example here.

When we make a list comprehension, the entire list is created at one time, then passed into the sum function.

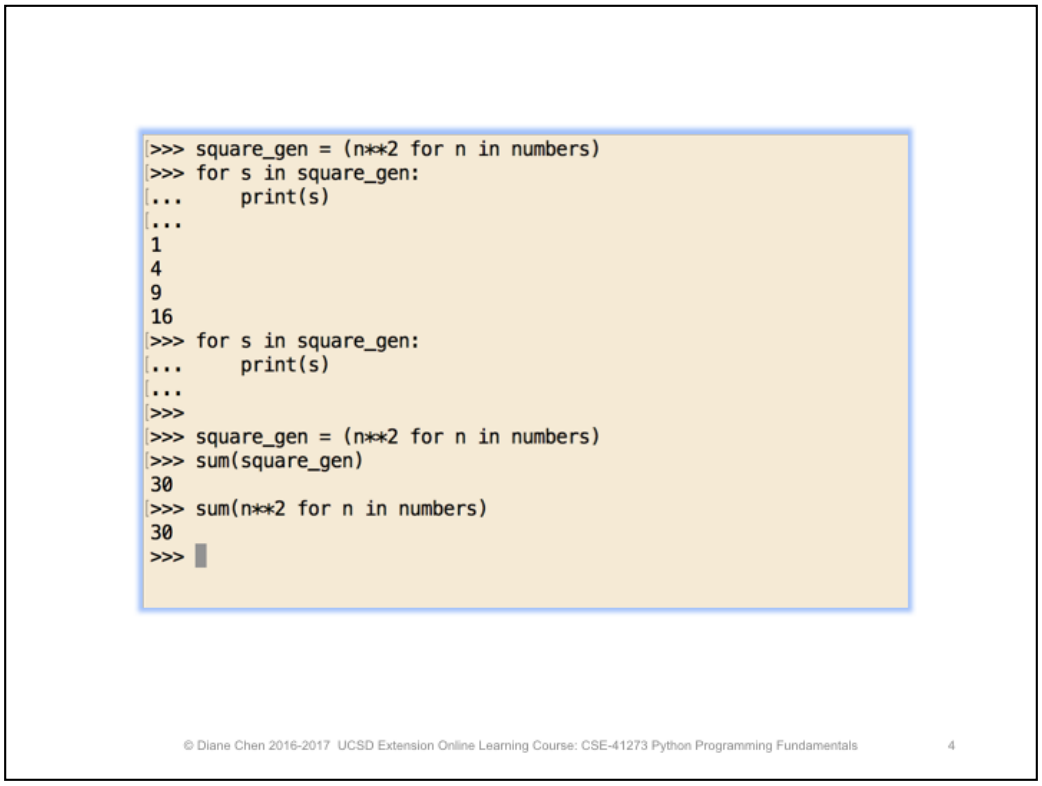
But we only needed to pass in an iterable, not necessarily a list. If the original list is large, we might be using up a lot of memory to create a list in the list comprehension, just to pass it to the sum() function.

We're going to be using a generator expression instead of a list comprehension. Generators are lazy iterables. Each item of the iterable isn't calculated or created until it's asked for by whatever is looping over it.

When we assign a list comprehension to a variable, the result is a list. We create a generator expression just like a list comprehension except we use parentheses instead of square brackets.

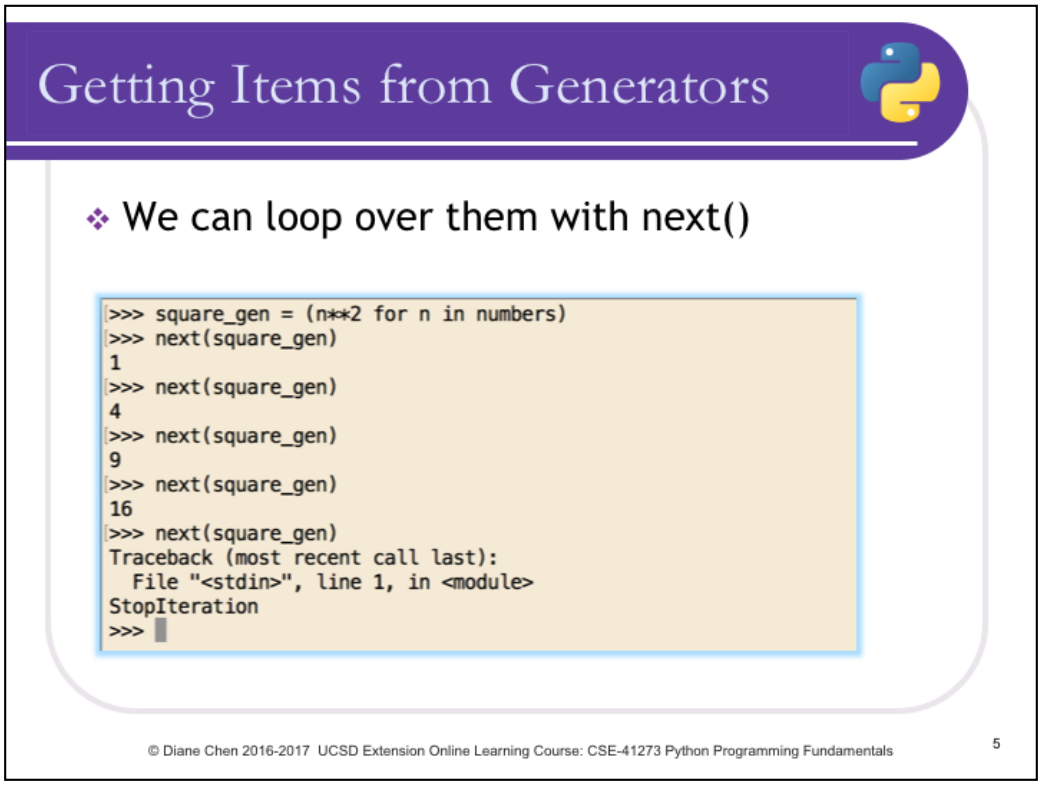
You might wonder why we say generator expression when it looks just like a comprehension. I don't know. It seems to be the convention, and probably there is a reason lost in the depths of time. Some people do say generator comprehension, and I think everyone understands it means the same thing.

Here we have assigned the result of the generator expression to the variable square_gen. But what is it? It is a special generator object. We can't index it like a list and we can't get it's length. That's because it is a lazy iterable – it only gives up the next value in the iterable sequence when it is requested, so the object has no public knowledge of anything other than the next item in the sequence. We can loop over it in a for loop just like any other iterable.

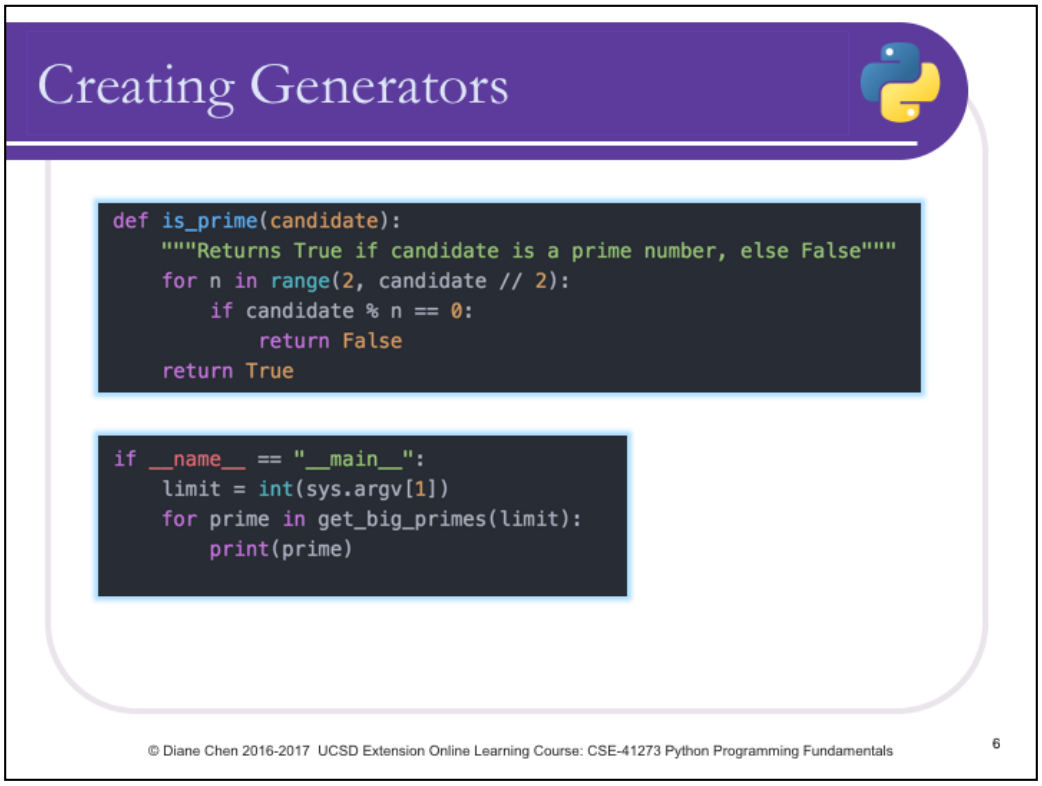


We saw that we can loop over a generator. However, we can only do it once. As I said before, generators are single-use iterables. Once we have looped over them, they are done - there's no reset button. The only way to start over is to create a new generator. This might not seem useful, but in most cases we really don't care about the permanence of the items, we just want to get them one by one. For example we can use a generator expression as input to the sum function. The sum function doesn't need to have a list, it only needs an iterable that it can loop over to compute the sum.

Note that I had to recreate the generator square_gen before calling sum() with it. If I had not done that, when I passed square_gen into sum, the result would be zero, since there were no more numbers left in square_gen. Most of the time we can just use a generator expression right in the function rather than creating a variable like square_gen.



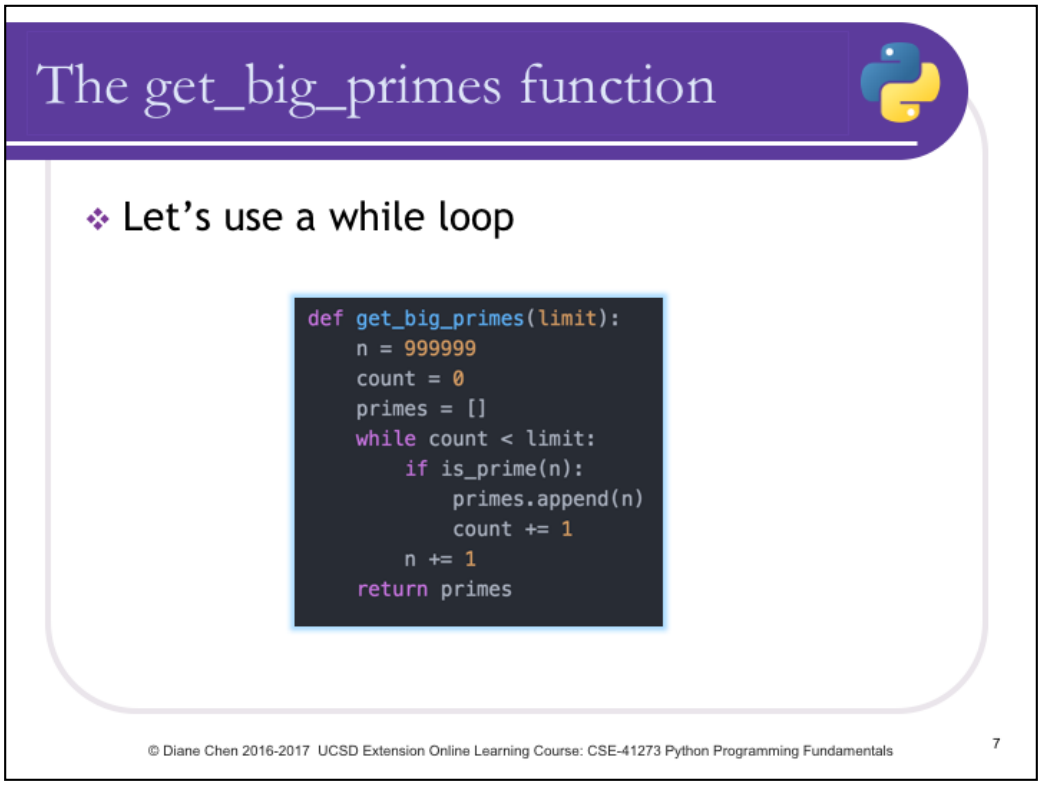
If we want, we can loop over the generator ourselves using the next() function. The error that occurs when the generator is used up is a StopIteration error. This error is the same error that is trapped in a for loop and causes the for loop to end.



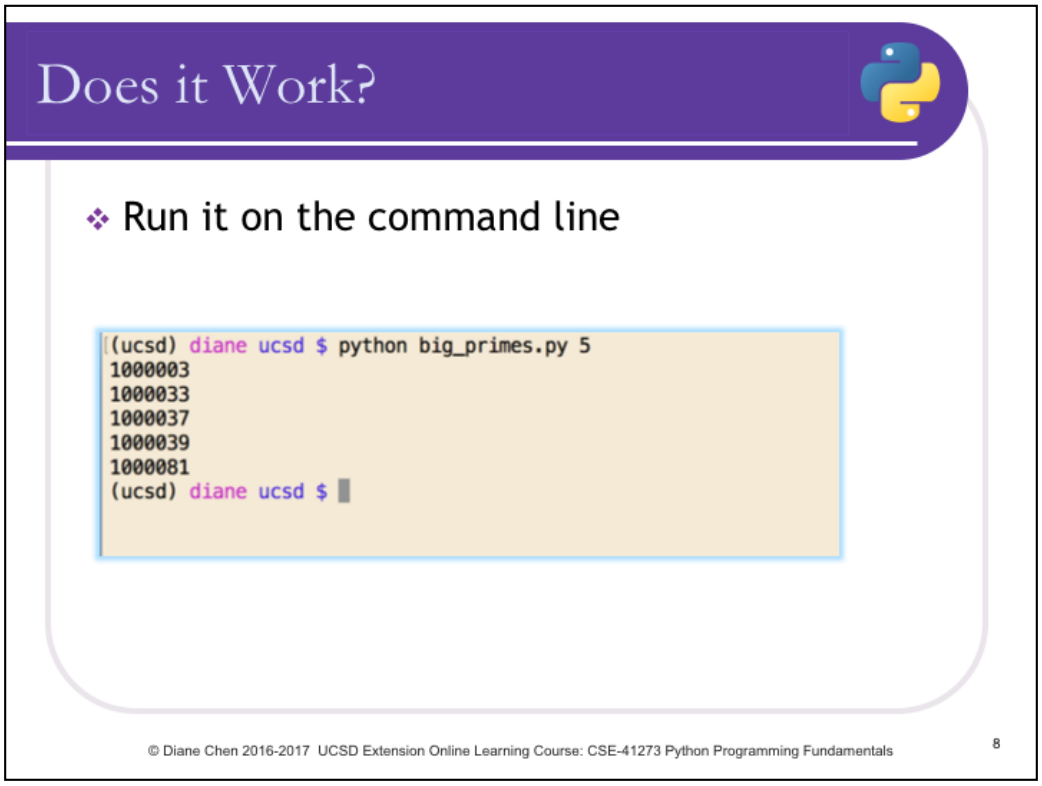
Here I'm going to show you how to create a generator function yourself. Say we want to write a program to generate a given number of prime numbers over 1,000,000.

We'll use this rather inefficient function to return True or False whether the input number is prime. We don't really care about efficiency at this point, we just want to get our answers. In fact, I want the inefficiency to show the advantage of generators later.

The number of primes we want to print is given on the command line. Don't forget to import the sys library into the program. We haven't defined our get_big_primes function yet. Think about how you might implement that using the is_prime function to test the primes.

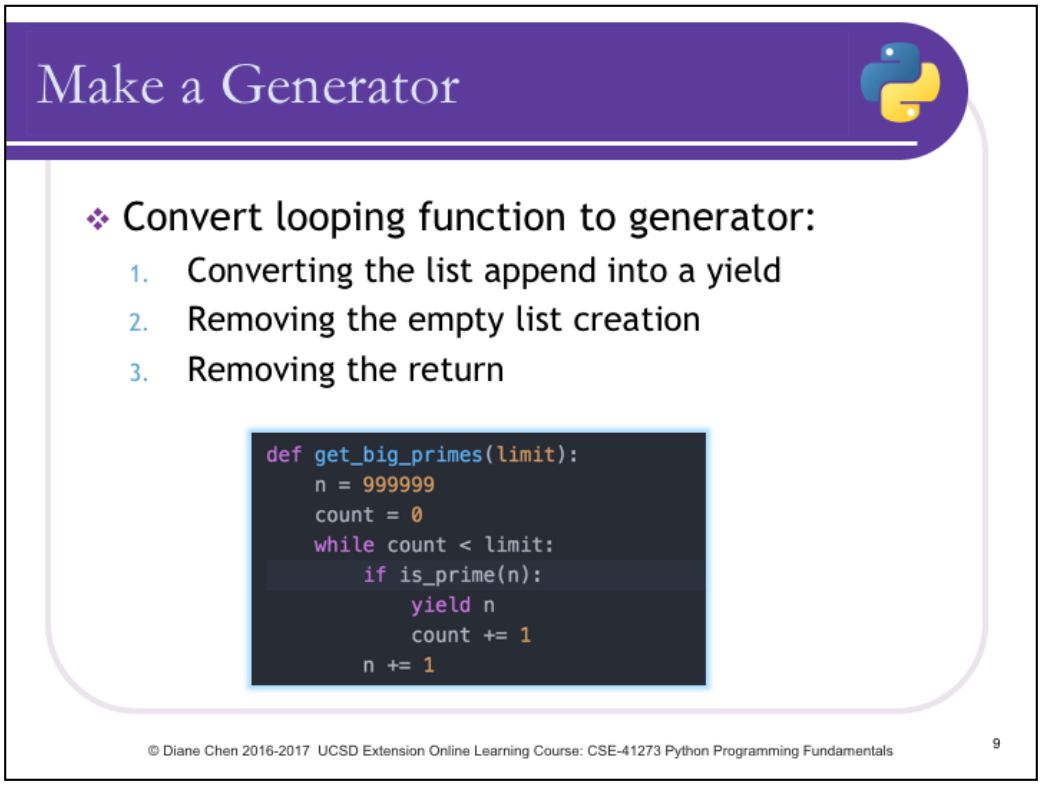


Let's make our get_big_primes function using a while loop. The variable n is the number we will be testing, and count is the number of primes we have found so far. We want to put put the prime numbers into a list as we find them, so we need to create an empty list to start with. When we have our required quantity of primes, we return the list.



Let's check if it works from the command line. It works, so we should be satisfied, right? But what if we requested a lot of numbers? We have to wait until it finds all the numbers and returns the list before we can see any of the numbers. While we're waiting we don't know for sure what is happening.

To really show the effect on my machine, I changed the program to find primes over 10,000,000 instead of just over 1 million. There is a significant delay when asking for 20 numbers. That's kind of boring to sit and wait until it spits out the list of numbers. What can we do to improve this? If you guessed that we're going to make a generator, you are correct!



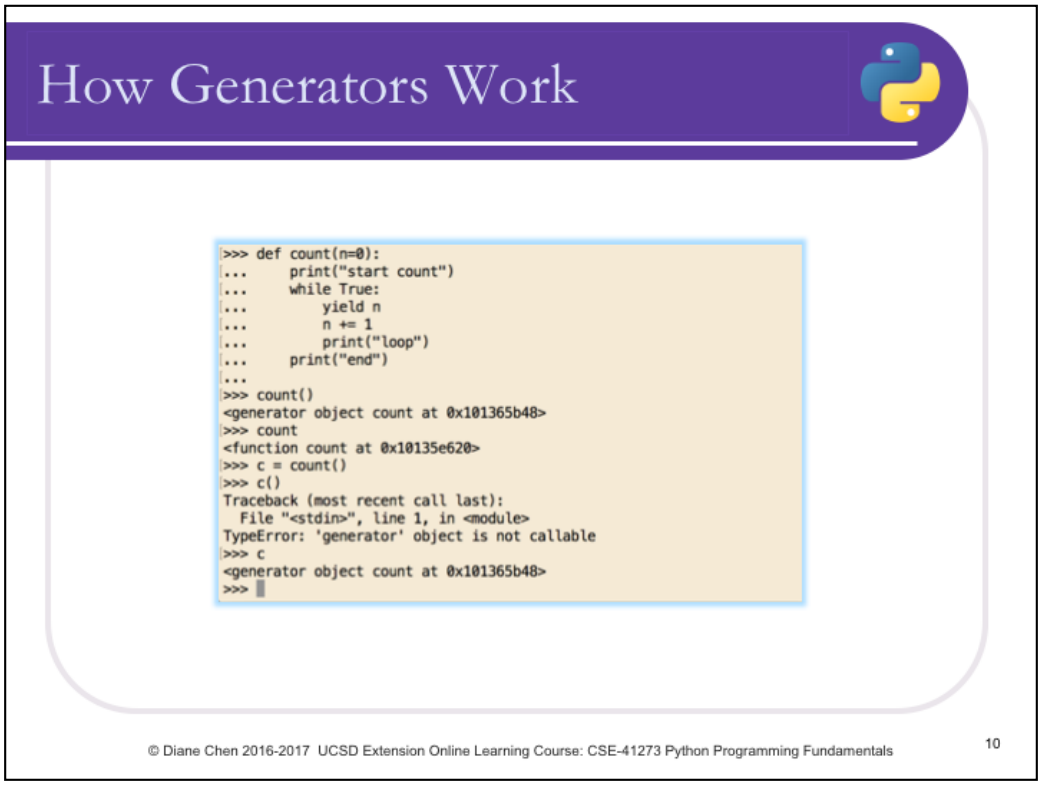
We're going to create our own generator function! Functions that construct a list or another iterable and return it can easily be turned into a generator by:

1. Converting the list append into a yield statement;
2. Removing the empty list creation; and
3. Removing the return.

Let's refactor that code to make it into a generator, as shown here. If we run it now on the command line, it will start printing numbers quickly and there is a tiny delay in between each number. What's going on here? How does this work? The "magic" of generators is in the "yield" keyword.

The get_big_primes function is "yielding" control back to the calling program on every loop iteration whenever it finds a prime number. When the next item is requested, the program continues immediately *after* the yield statement.

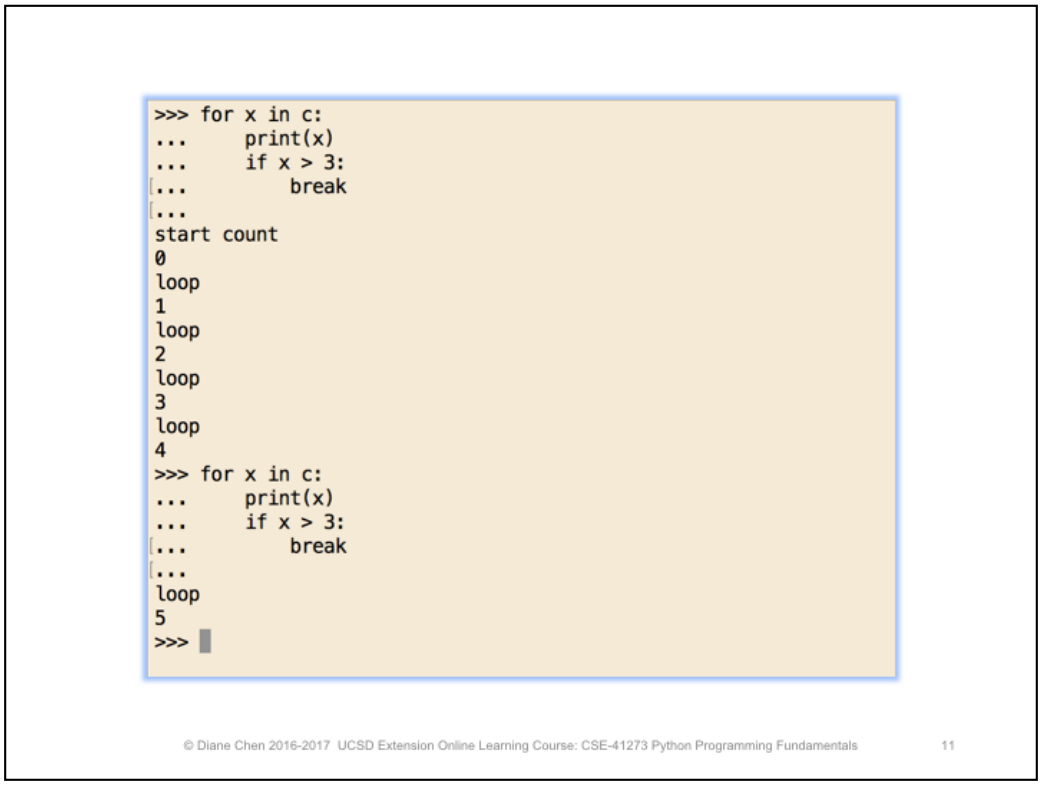
When the while loop finishes, the generator is finished and returns. Generators keep track of where they were so that when we loop over them they'll always start up where they left off.



Let's make another simple example to show how generators work.

We're going to make a generator that counts up from a given number that defaults to 0. It looks like a regular function except for the yield. I put some print statements in there so we can see what is going on with the execution of the generator.

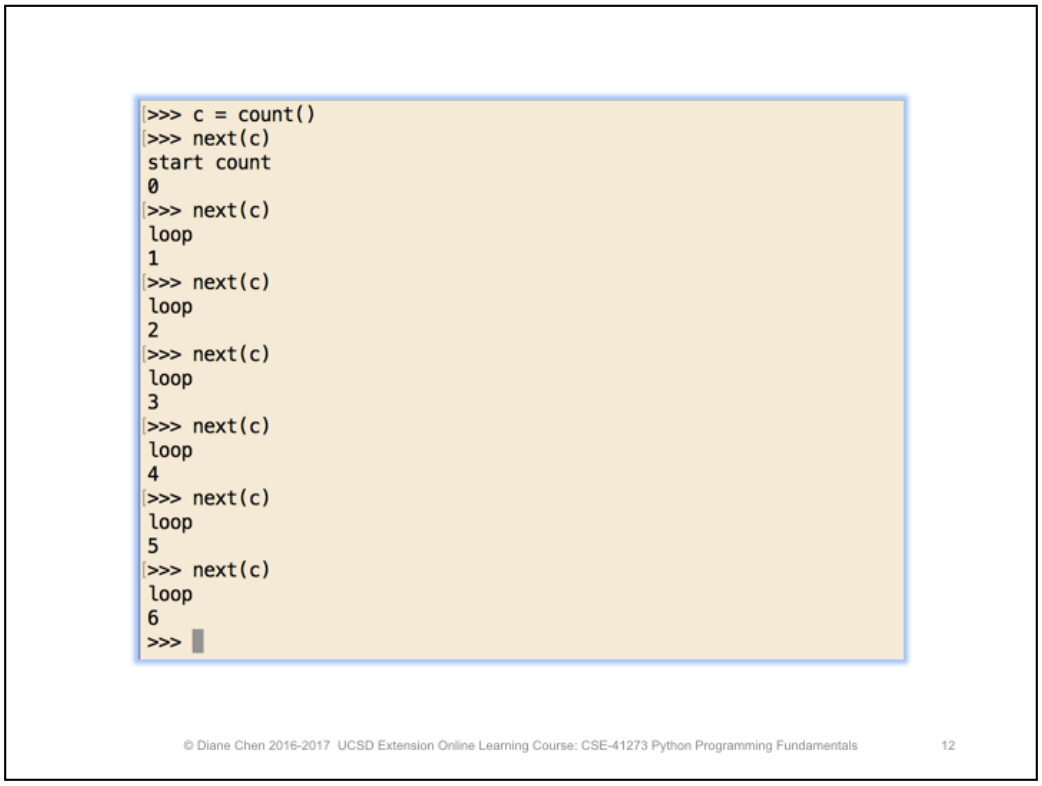
Note this is a generator *function* and not an actual generator. Let's create a generator object c from our generator function count(). What can we do with this object? If we try to call it we get an error because it is not a function. Well, you might guess that we can iterate over it! Let's try it!



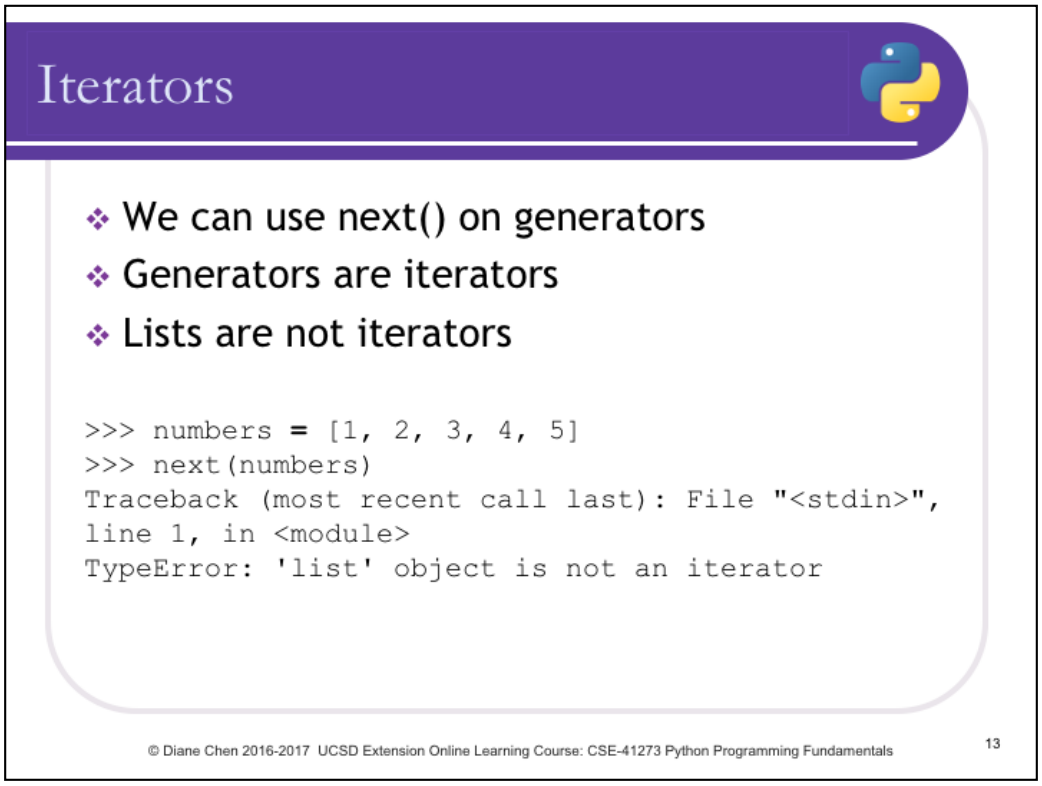
When we start looping over our generator c, the function is entered and our while loop starts executing. But look at the print output; it never returned out of the function because “end” was never printed!

When we loop over it again, it prints 5 and stops but it still hasn’t ended the generator. But if you think about it, it isn’t too strange. Our generator object c is essentially a single-use object that keeps yielding answers forever, due to the while True loop inside count().

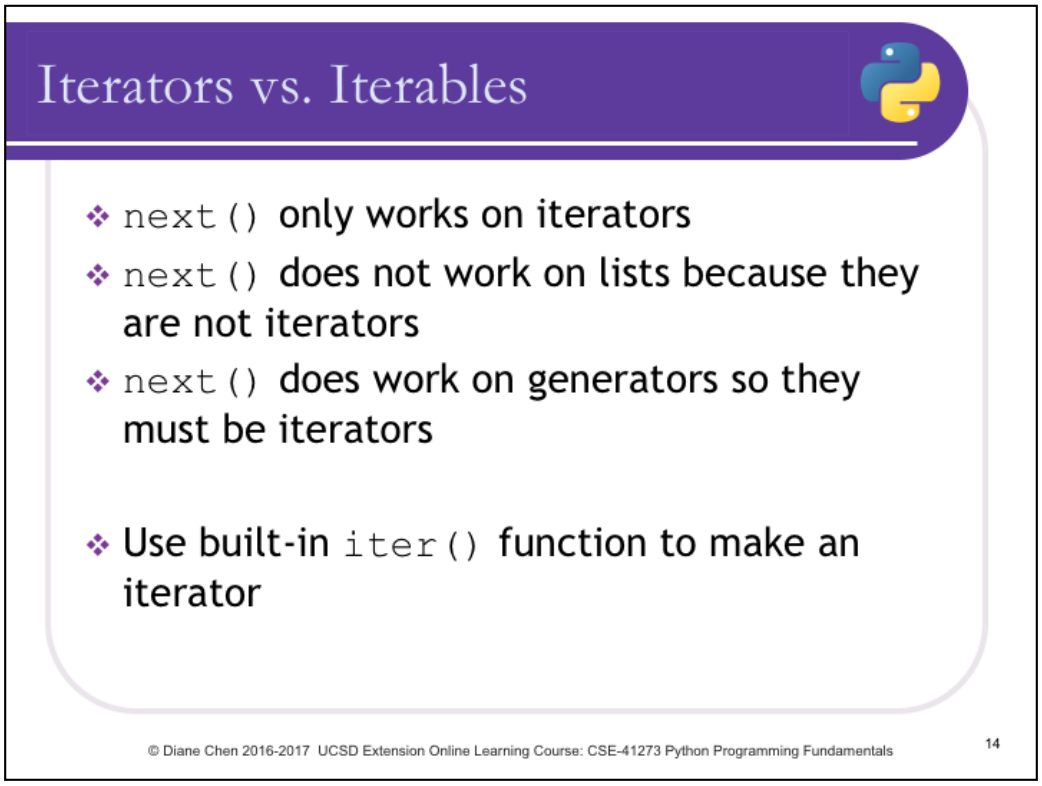
There is no logic to end the loop in the generator count(). In this example, the ending happens in the code that uses it. If we didn’t have that check to break out of our for loop here, it would go on forever! If you go back to look at the get_big_primes function, we see that it has logic in it to end itself when the criteria is met.



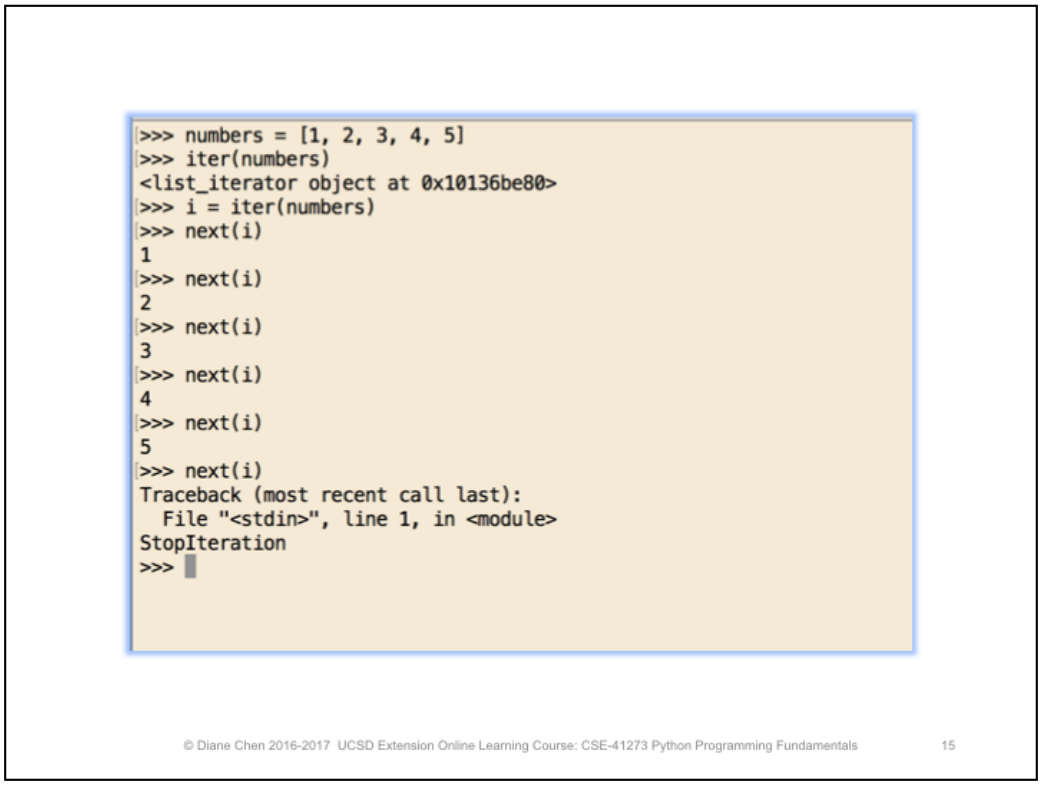
Being single-use iterables, the only way to restart a generator is to make a new one, like I've done here, creating a new variable c from count(). We can iterate over the generator using the next() function. It will keep giving us numbers as long as we want.



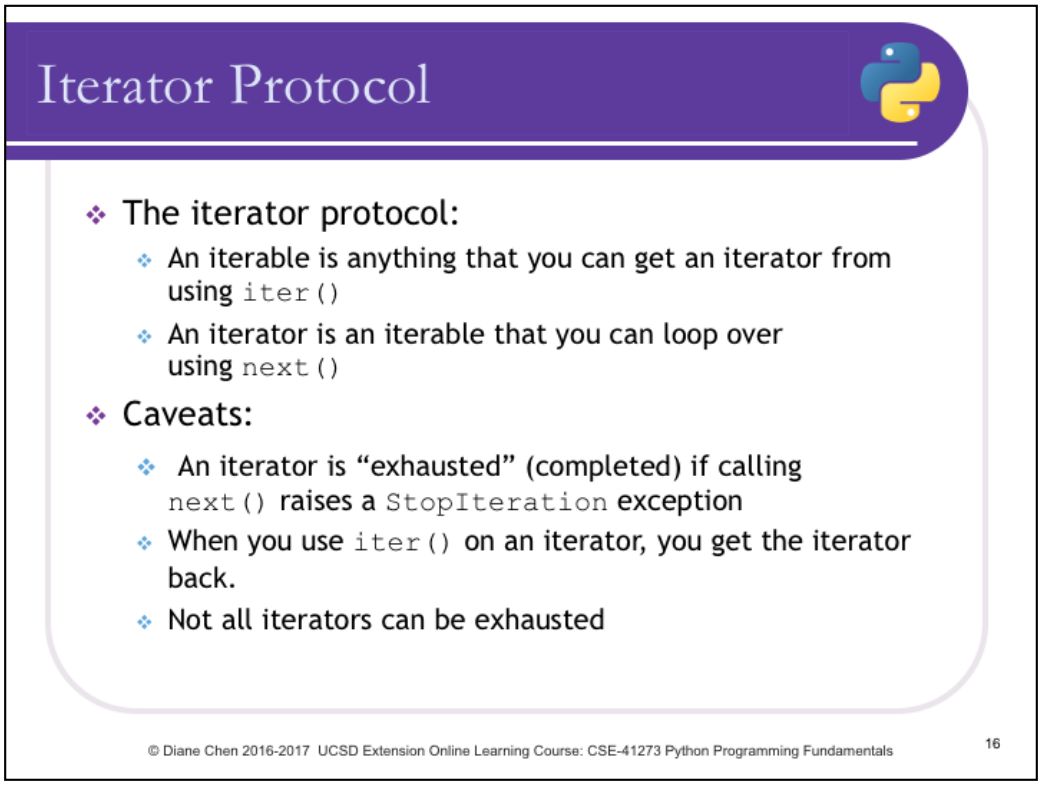
So we saw that we can use the built-in next() function to loop over a generator. This makes it an iterator. What would happen if we used next() on a list? We get an error. So Lists are not iterators because we cannot use next(). Lists are iterables, not iterators. As we saw with our generator expression earlier, when a generator is finished, if we call next() on it, we get a stopIteration error.



Let's review a minute here. Now we know that next() only works on iterators. And next() does not work on lists because lists are not iterators. But next() does work on generators so they are iterators. How do we loop over a list using next()? We can create an iterator from any iterable by using the built-in iter() function.



We can make an iterator from a list and call next() on it until it is done. As we saw with our generator expression earlier, when it is finished we get the StopIteration error. Since a generator is an iterator, if we call the iter() function on a generator, we get the generator back. If you want to iterate over something but you are not sure if it is an iterator or just an iterable, it is safe to call iter() on it, because if it is already an iterator, you just get the same iterator back.



Let's summarize some of what we covered. All this iterable, iterator and generator stuff can sometimes be a little confusing.

To really understand what's going on, we need to talk about the iterator protocol.

An iterable is anything that you can get an iterator from using the iter() function.

An iterator is an iterable that you can loop over using the next() function.

A couple of caveats:
An iterator is "exhausted" (completed) if calling the next() function on it raises a StopIteration exception.
When you use the iter() function on an iterator, you get the same iterator back.

Not all iterators can be exhausted (some can keep giving next values forever just like the example we showed with count).

So those generators we were working with are iterators. And all iterators are also iterables.