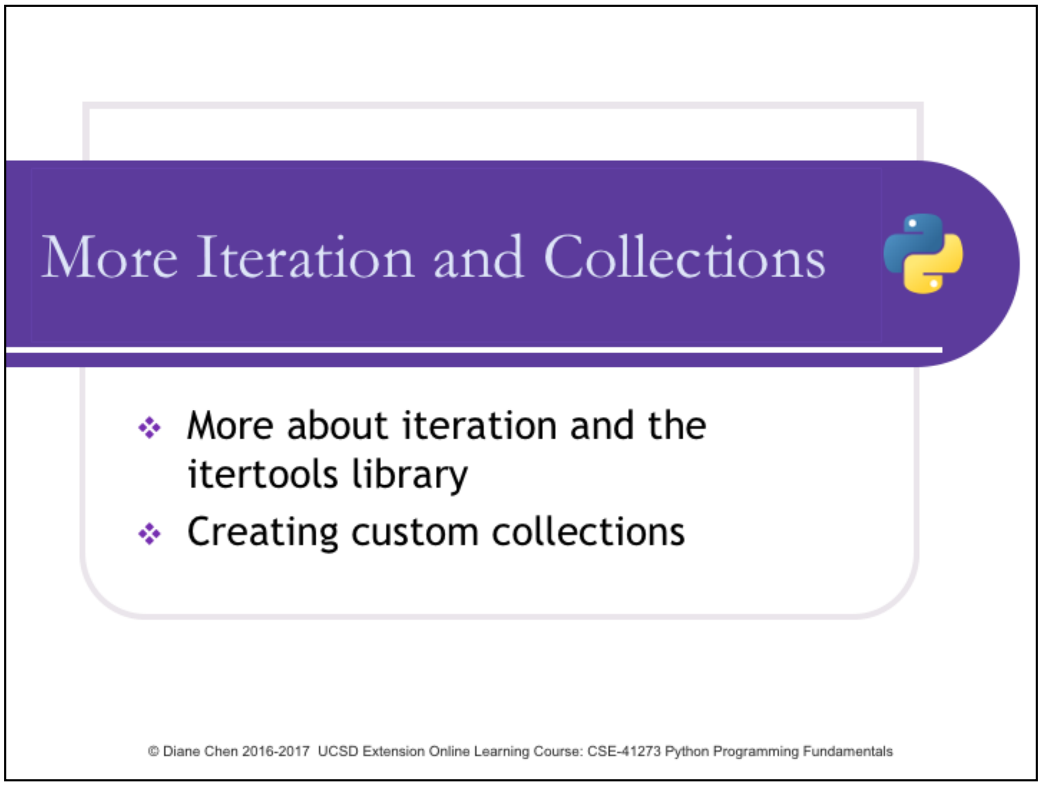
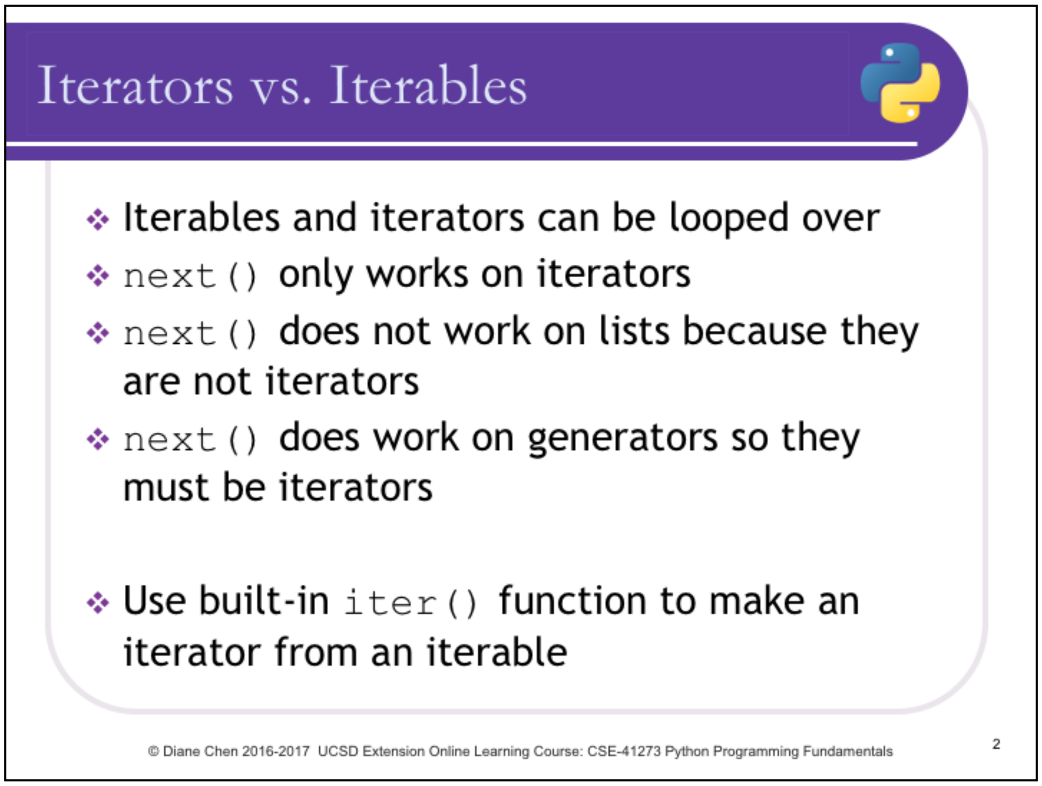


More Iteration and Collections

- More about iteration and the itertools library
- Creating custom collections

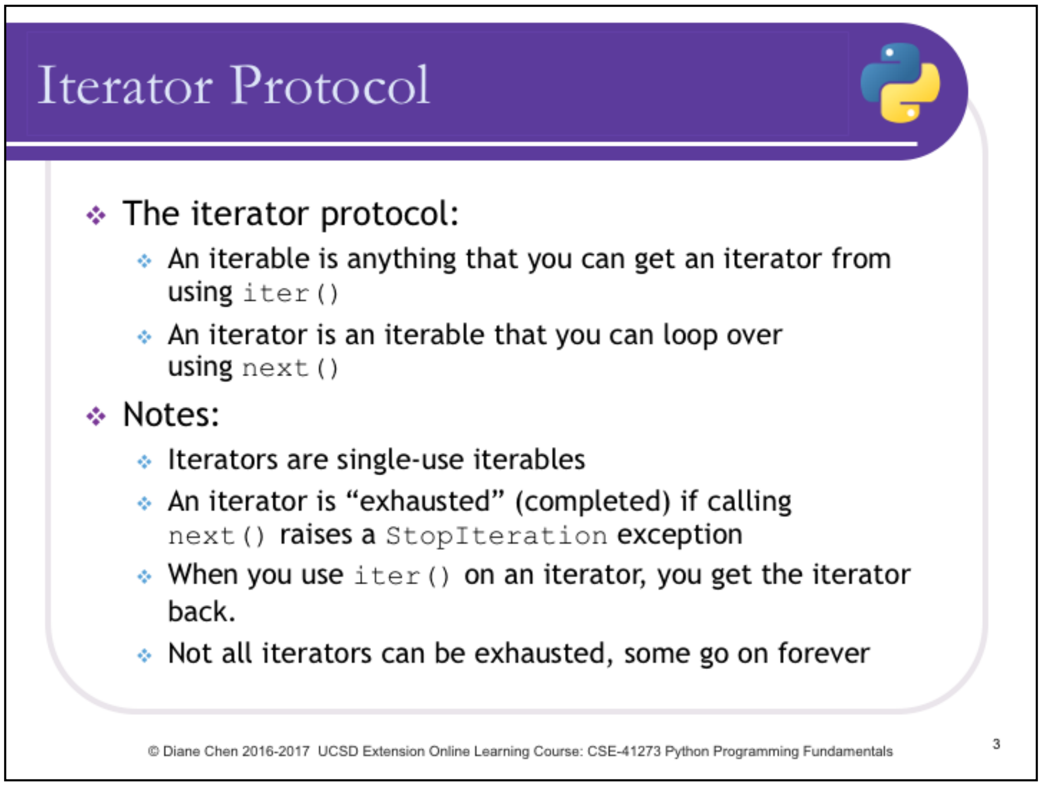
In this lecture I'm going to cover more about iteration, making an iterable class, and the Itertools module. These things make iteration in Python easy, clean and beautiful. Then we're going to look at making custom collections.



Iterators vs. Iterables

- Iterables and iterators can be looped over
- `next()` only works on iterators
- `next()` does not work on lists because they are not iterators
- `next()` does work on generators so they must be iterators
- Use built-in `iter()` function to make an iterator from an iterable

I'm going to discuss iterators and iterables some more. I'm repeating myself here because this concept is very important in Python. Python has wonderful iteration tools and if you understand them well, you will be able to write code that is cleaner and more efficient than otherwise. So, let's review from what we covered before. Iterables and iterators can be looped over, for example, in places like for loops, or inside list comprehensions. Lists, strings and tuples are examples of iterables. Dictionaries and sets can also be iterables, but they are "unordered" so, while you will get every item when looping over them, the order is not guaranteed. The function next() only works on iterators; it doesn't work on lists because lists are not iterators; they are only iterables. The next() function works on generators, so that means they are iterators, too. When you want an iterator for an iterable, you can call the built-in iter() function to create an iterator. For example, if you want a manual iterator to iterate over a list, you can create one by calling the iter function on the list.


Iterator Protocol

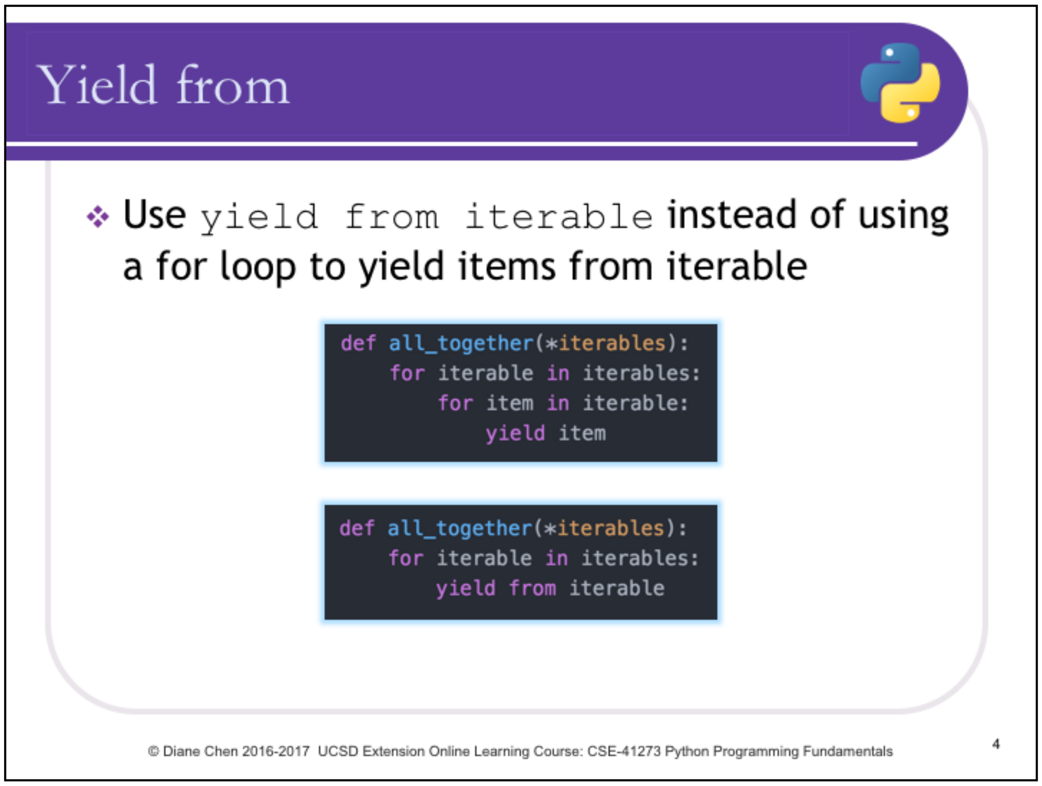
- The iterator protocol:
  - An iterable is anything that you can get an iterator from using `iter()`
  - An iterator is an iterable that you can loop over using `next()`
- Notes:
  - Iterators are single-use iterables
  - An iterator is "exhausted" (completed) if calling `next()` raises a `StopIteration` exception
  - When you use `iter()` on an iterator, you get the iterator back.
  - Not all iterators can be exhausted, some go on forever

3

Let's review the Iterator protocol. You can get an iterator from any iterable using the iter() function. You can loop over iterators using the next() function, but next() does not work on simple iterables. Some notes again. Iterators are single-use iterables. An iterator is finished when a StopIteration Error is raised from calling the next() function on it. At this point, you cannot use the iterator any more, it will just keep giving you the stopIteration error every time you call next() on it.

When you call iter() on something that is an iterator, it returns the iterator back. Not all iterators can be used up, some can just go on forever.

This iterator protocol is what happens internally in a for loop; Python calls the iter function on whatever you are looping over in the for loop. Then for each loop, it calls the next() function on the iterator. When there are no more items in the iterator, then a stopIteration error is raised and the for loop finishes.



Yield from

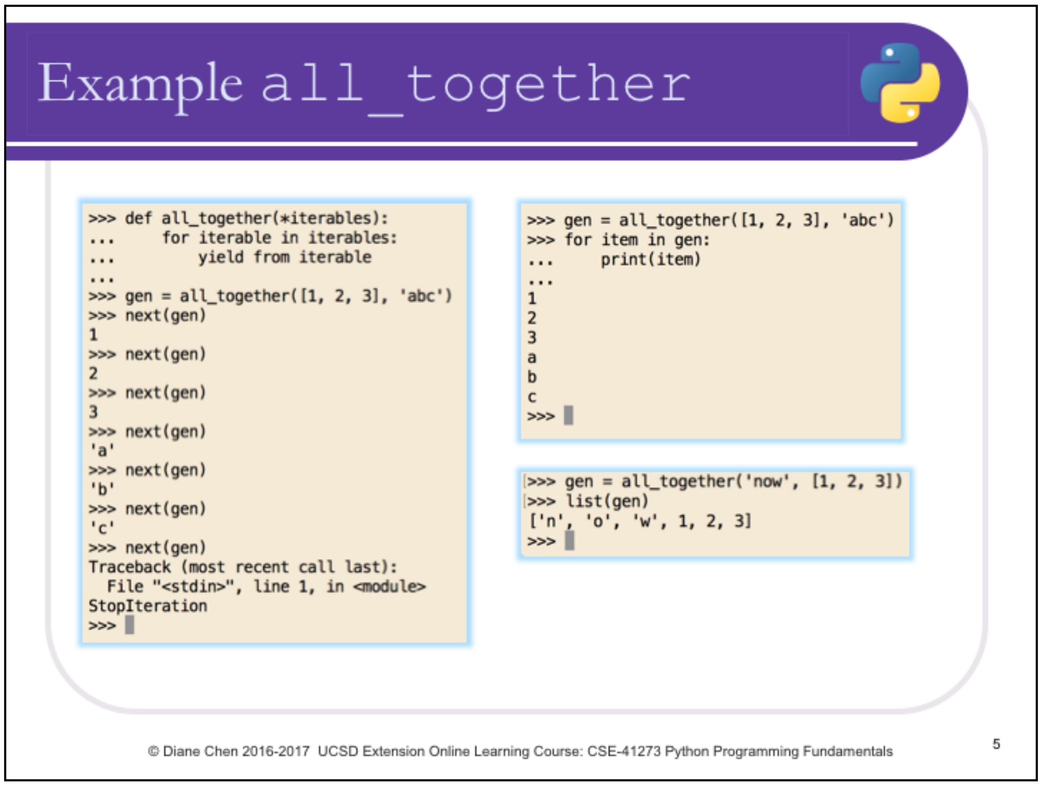
- Use `yield from iterable` instead of using a for loop to yield items from iterable

```python
def all_together(*iterables):
    for iterable in iterables:
        for item in iterable:
            yield item
```

```python
def all_together(*iterables):
    for iterable in iterables:
        yield from iterable
```

4

We saw generators in an earlier lesson. Here's a handy shortcut for generators. The "yield from" expression can be used instead of a for loop iterating over an iterable and yielding each element. For example, let's make a generator function "all together" that takes a variable number of iterables as inputs and yields items from each one sequentially. In the first code example here, we have two nested for loops. The inner loop yields each item from the current iterable of the input list of iterables. In the second code snippet, the inner loop from the first example is replaced with the "yield from" expression. The "yield from" expression is what we call "syntactic sugar" that wraps the for loop with the yield together in one statement. It's clean and clear what is happening in the code.



Example all_together

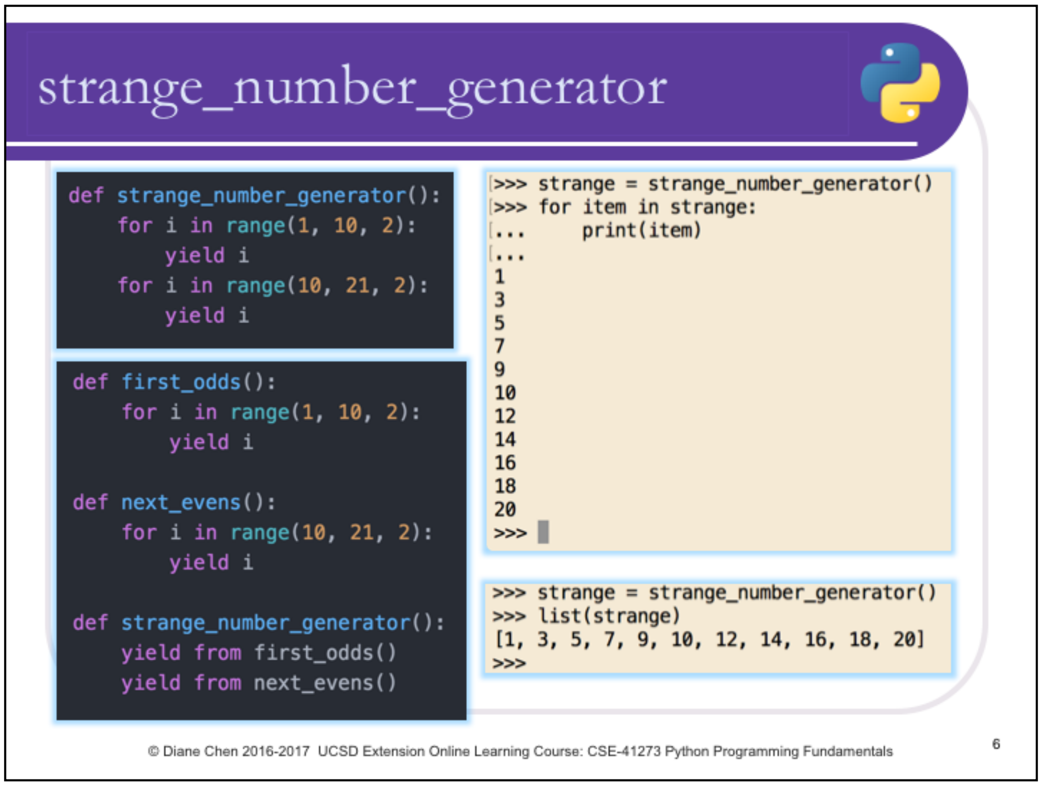
```
>>> def all_together(*iterables):
...     for iterable in iterables:
...         yield from iterable
...
>>> gen = all_together([1, 2, 3], 'abc')
>>> next(gen)
1
>>> next(gen)
2
>>> next(gen)
3
>>> next(gen)
'a'
>>> next(gen)
'b'
>>> next(gen)
'c'
>>> next(gen)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>>
```

```
>>> gen = all_together([1, 2, 3], 'abc')
>>> for item in gen:
...     print(item)
...
1
2
3
a
b
c
>>>
```

```
>>> gen = all_together('now', [1, 2, 3])
>>> list(gen)
['n', 'o', 'w', 1, 2, 3]
>>>
```

5

Here we see the use of our generator function all_together. We create a generator with inputs of a list of numbers 1, 2, 3, and a string “abc”. When we repeatedly call next on it, we get all the elements of both iterables in order. Or we can iterate over it and print them out or put them into a list, and they all come out in the correct order.



strange_number_generator

```python
def strange_number_generator():
    for i in range(1, 10, 2):
        yield i
    for i in range(10, 21, 2):
        yield i
```

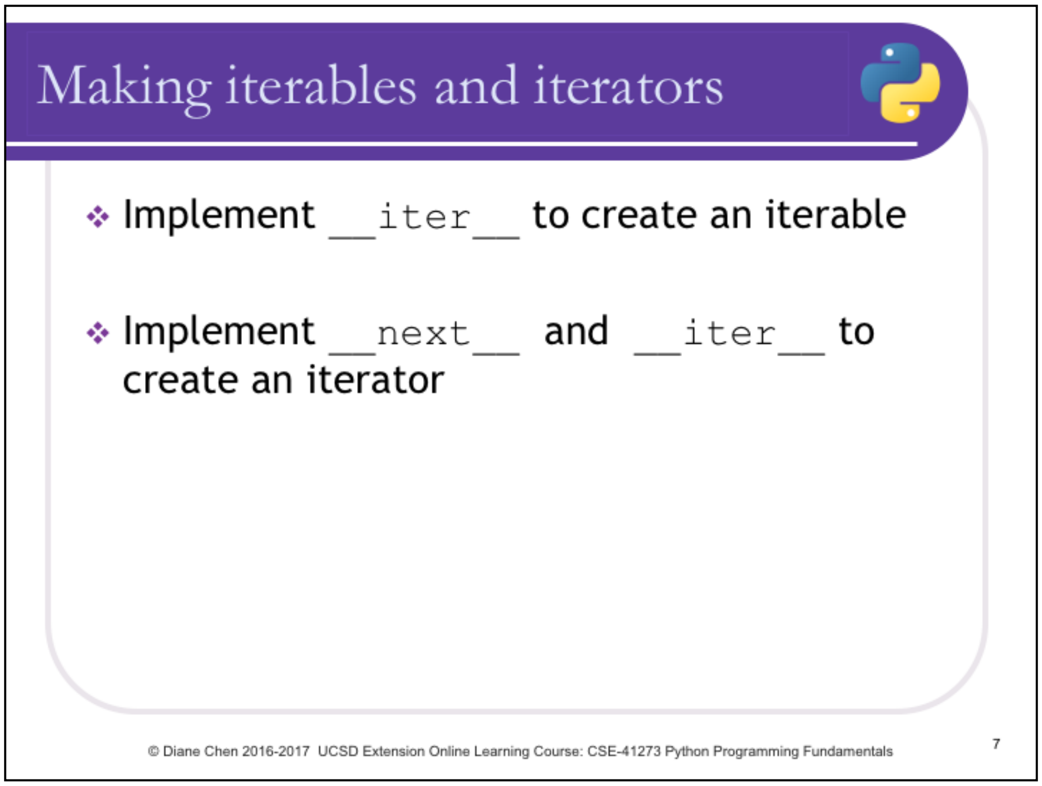
```python
def first_odds():
    for i in range(1, 10, 2):
        yield i

def next_evens():
    for i in range(10, 21, 2):
        yield i

def strange_number_generator():
    yield from first_odds()
    yield from next_evens()
```

```
>>> strange = strange_number_generator()
>>> for item in strange:
...     print(item)
...
1
3
5
7
9
10
12
14
16
18
20
>>> 
```

```
>>> strange = strange_number_generator()
>>> list(strange)
[1, 3, 5, 7, 9, 10, 12, 14, 16, 18, 20]
>>>
```

6

Here is a rather silly example that shows how code for a generator may be refactored into using ”yield from”. In the first version of the strange number generator, we yield from two different sets of numbers, one that yields some odd numbers, then another that yields some even numbers. In the second code snippet, the number generators are factored out into their own generator functions, each yielding their own set of numbers. Then the code in the strange number generator can just “yield from” each generator function. Of course, as I said, this is a silly example. But imagine that the code for each set of calculations was complicated before each yield, rather than just the simple range call here, or if there were many sequential sets of calculations, it might make sense to factor it out into separate generator functions to keep the overall code in strange number generator cleaner.

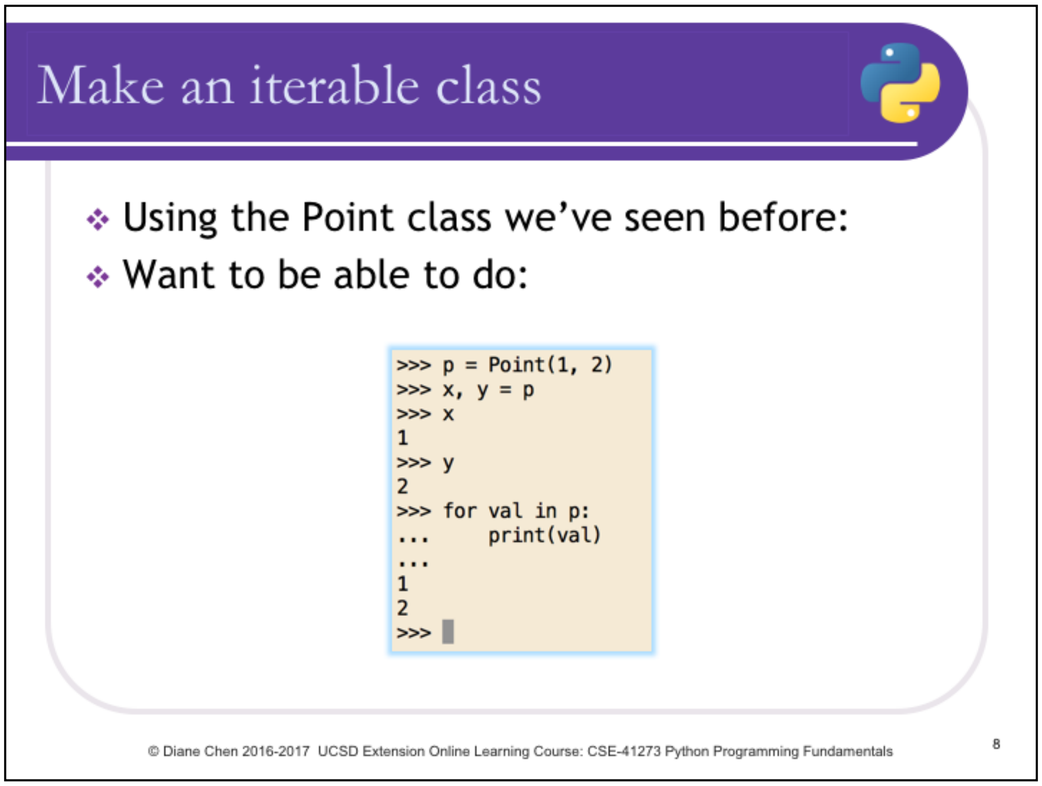


Making iterables and iterators

- Implement `__iter__` to create an iterable
- Implement `__next__` and `__iter__` to create an iterator

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 7

It should come as no surprise that you can create your own iterable or iterator objects! An iterable needs to have the dunder iter method defined and for an iterator, you need to define both dunder iter and dunder next. For example, say we have a class we have defined that contains a bunch of data including a list. We might want to be able to iterate over the instance object and get the list items without having to know about the internal list. If we implement the dunder iter method to iterate over the list, then our instance objects are iterable. To make an iterator, the dunder iter method should return self, and then the dunder next method needs to be implemented to return the items. It's surprisingly easy to make your own iterables or iterators.



Make an iterable class

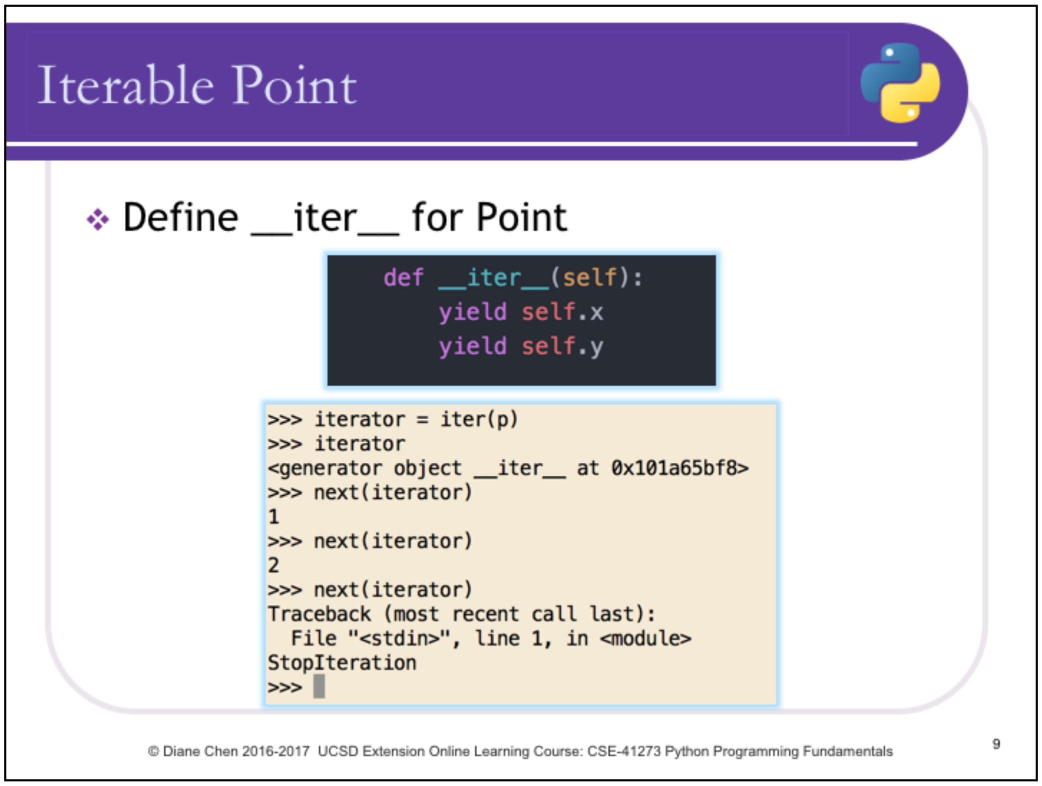
- Using the Point class we've seen before:
- Want to be able to do:

```
>>> p = Point(1, 2)
>>> x, y = p
>>> x
1
>>> y
2
>>> for val in p:
...     print(val)
...
1
2
>>>
```

8

Here's an example of making a class iterable. Let's go back to our Point class and make our points iterable. What does this mean, making the point iterable? We want to be able to take an instance of the Point class and assign the x and y values from the Point instance object without having to reference them directly. This will also enable the use of a Point instance object in a for loop.

Note that the line where we get x and y from the instance object uses "tuple unpacking" because we are requesting a tuple, namely x and y together, which triggers the iterator protocol. If we just said "x = p" on one line and "y = p" on the next line, we would have two new variables that each point to the instance object p.



Iterable Point

- Define __iter__ for Point

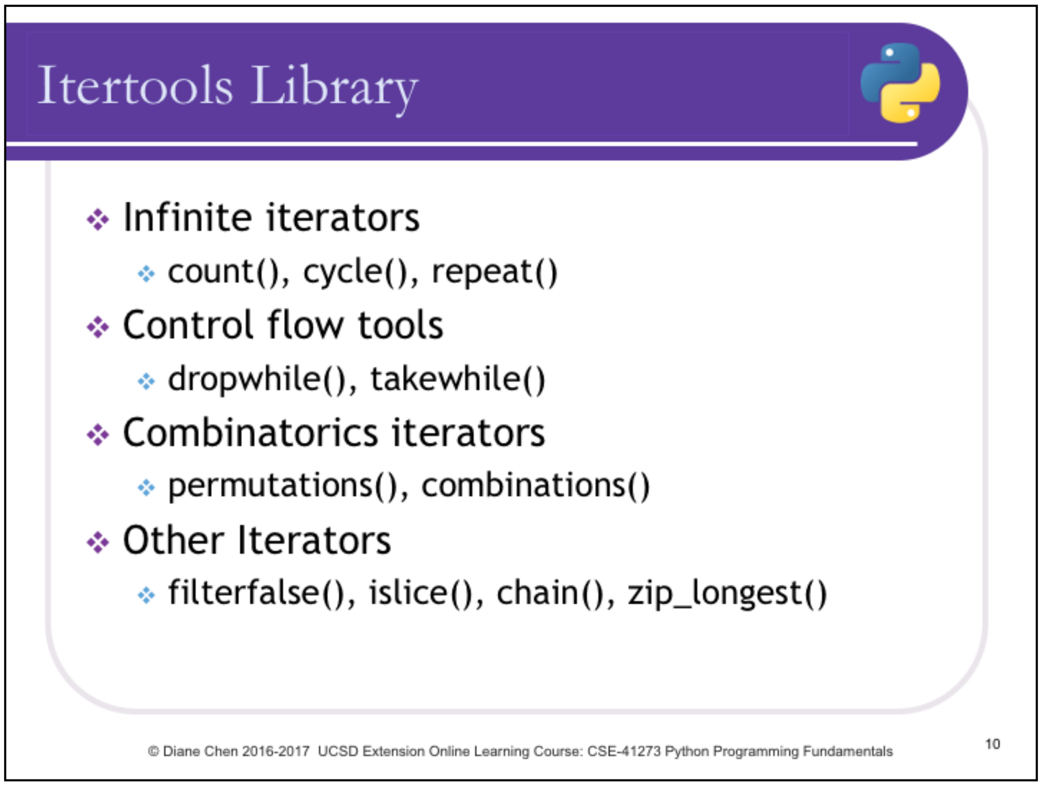
```
def __iter__(self):
    yield self.x
    yield self.y
```

```
>>> iterator = iter(p)
>>> iterator
<generator object __iter__ at 0x101a65bf8>
>>> next(iterator)
1
>>> next(iterator)
2
>>> next(iterator)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
>>> 
```

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 9

All we need to make our point object iterable is to define the __iter__ method. And here it is! It first yields self.x, then yields self.y. That's all you need to do to make the point object iterable and work like shown on the previous slide. Easy, peasy!

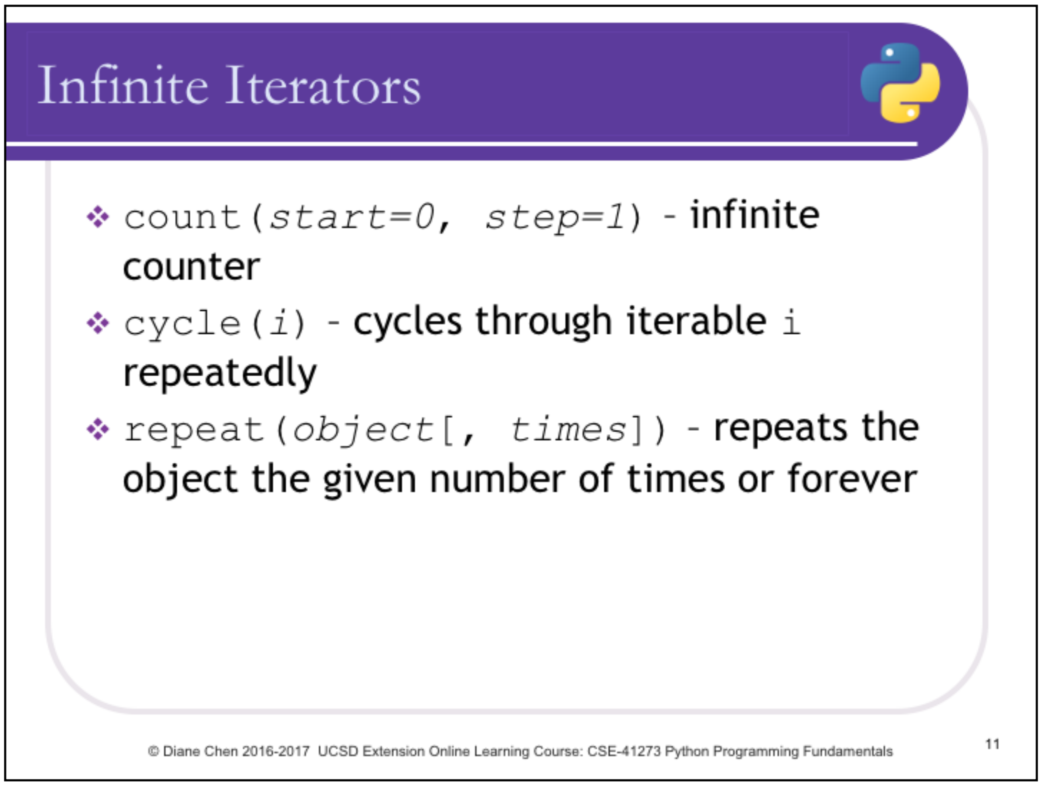
We can do our own iteration of a point instance as shown here. We just use the iter() function to create a new iterator object, then call next() on it to get the values. A StopIteration error is raised after we get both the x and y values.



Itertools Library

- Infinite iterators
  - count(), cycle(), repeat()
- Control flow tools
  - dropwhile(), takewhile()
- Combinatorics iterators
  - permutations(), combinations()
- Other Iterators
  - filterfalse(), islice(), chain(), zip_longest()

Now we'll cover the itertools library. the itertools library contains lots of different tools. We have infinite iteratrors that can go on forever. Control flow tools that determine whether an iterable's items are yielded. Combinatorics iterators giving us permutations and combinations. And then there are some other useful iterator tools.



Infinite Iterators

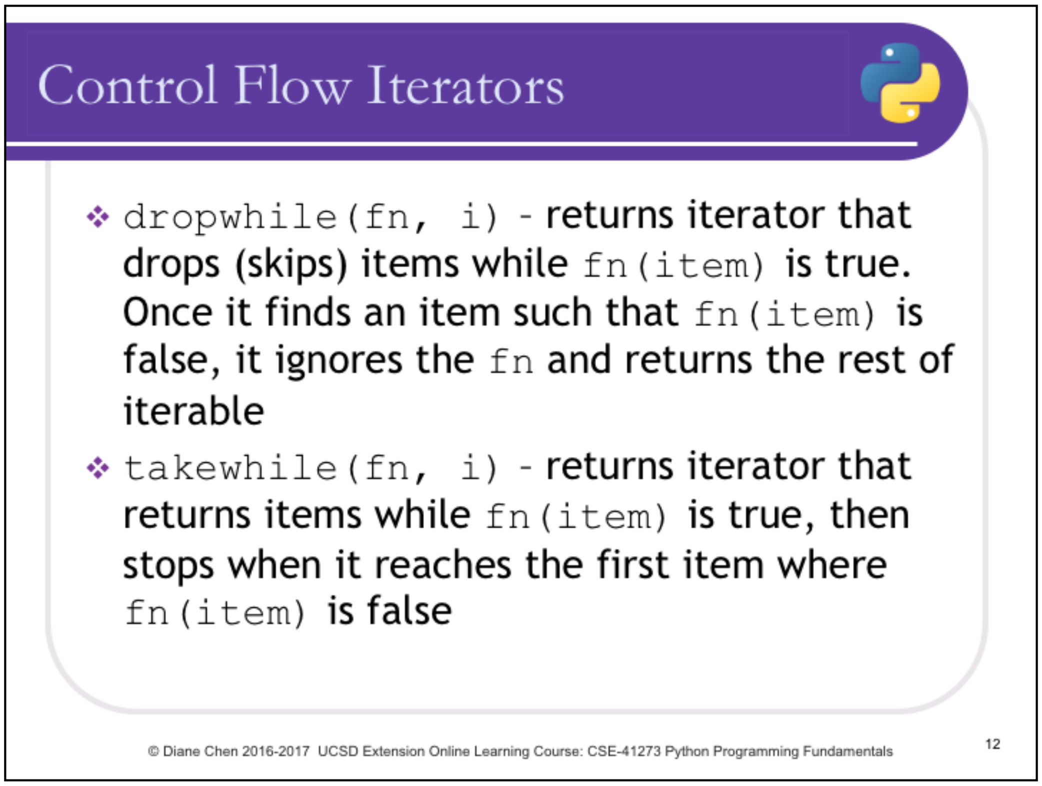
- `count(start=0, step=1)` - infinite counter
- `cycle(i)` - cycles through iterable `i` repeatedly
- `repeat(object[, times])` - repeats the object the given number of times or forever

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals

11

Note that to use any of the functions from the itertools library, you must import them from the itertools module. You can either use "import itertools" and refer to them with the itertools namespace, such as itertools.count. Or you can use "from itertools import count". For the sake of space on the slides, I am leaving off the namespace definition.

The infinite iterators count, cycle and repeat should be used with caution. You need to make sure that you have some way for them to end. These 3 are pretty obvious what they do. Count() is an infinite counter, with optional start and step values. cycle() cycles through the given iterable repeatedly. Repeat() repeats the given object forever unless a number of times is also given.

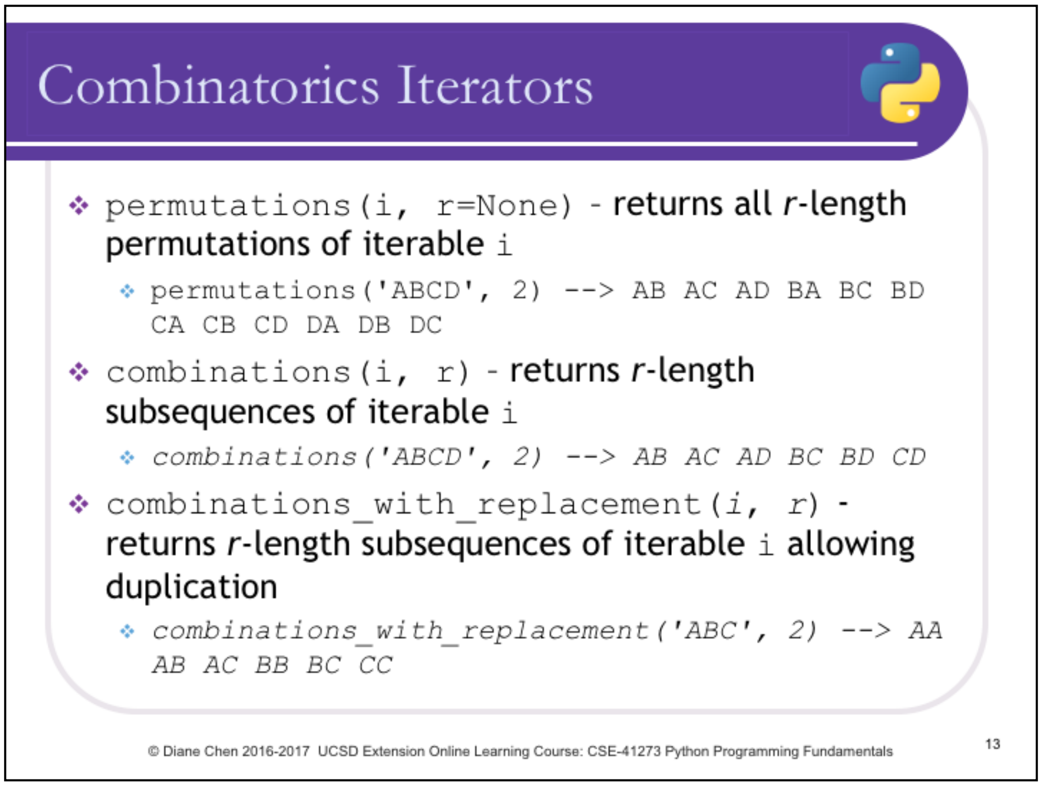


Control Flow Iterators

- `dropwhile(fn, i)` - returns iterator that drops (skips) items while `fn(item)` is true. Once it finds an item such that `fn(item)` is false, it ignores the `fn` and returns the rest of iterable
- `takewhile(fn, i)` - returns iterator that returns items while `fn(item)` is true, then stops when it reaches the first item where `fn(item)` is false

12

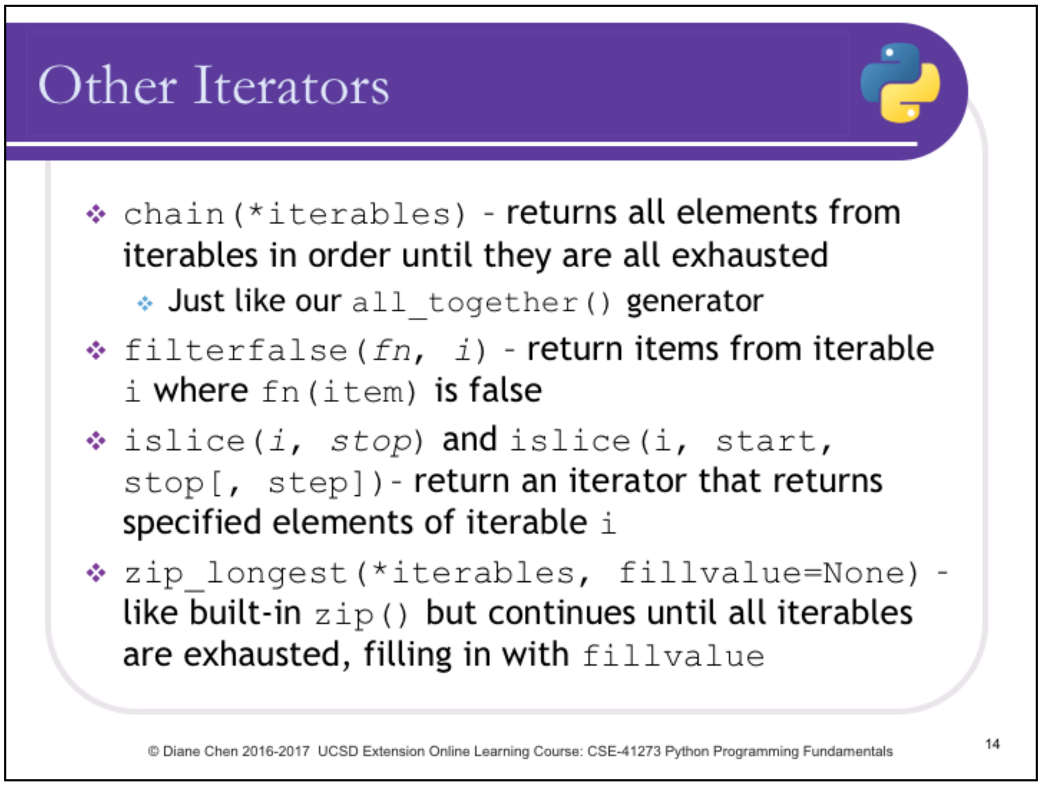
I call these two Control Flow Iterators. They are maybe a little confusing at first, but can come in handy when you need them. Dropwhile() takes a function and an iterable. When it starts it applies the function to each element of the iterable. If the result of this is True, it skips or drops the item and goes to the next one. It keeps dropping the elements as long as the results are true. Once it finds an element such that the function applied to it returns false, it returns that element and from then on, it ignores the function and continues to return the rest of the iterable. An example of using dropwhile might be when you have a huge log file with dated entries and want to look at entries after a certain date/time. The function you pass to dropwhile can check the date/time of the log entry and return True if it is before the date you want to check. Similarly, an iterator from takewhile returns items from the iterable as long as the function applied to the item is true. Once it finds an element where the function applied to it returns false, it stops returning values. As an example, if you had the same log file and wanted to look at all the entries up to a certain date/time, you could use a similar function that returns True as long as the date is not past the date/time you want to stop.



Combinatorics Iterators

- `permutations(i, r=None)` - returns all *r*-length permutations of iterable `i`
  - `permutations('ABCD', 2) --> AB AC AD BA BC BD CA CB CD DA DB DC`
- `combinations(i, r)` - returns *r*-length subsequences of iterable `i`
  - *`combinations('ABCD', 2) --> AB AC AD BC BD CD`*
- `combinations_with_replacement(i, r)` - returns *r*-length subsequences of iterable `i` allowing duplication
  - *`combinations_with_replacement('ABC', 2) --> AA AB AC BB BC CC`*

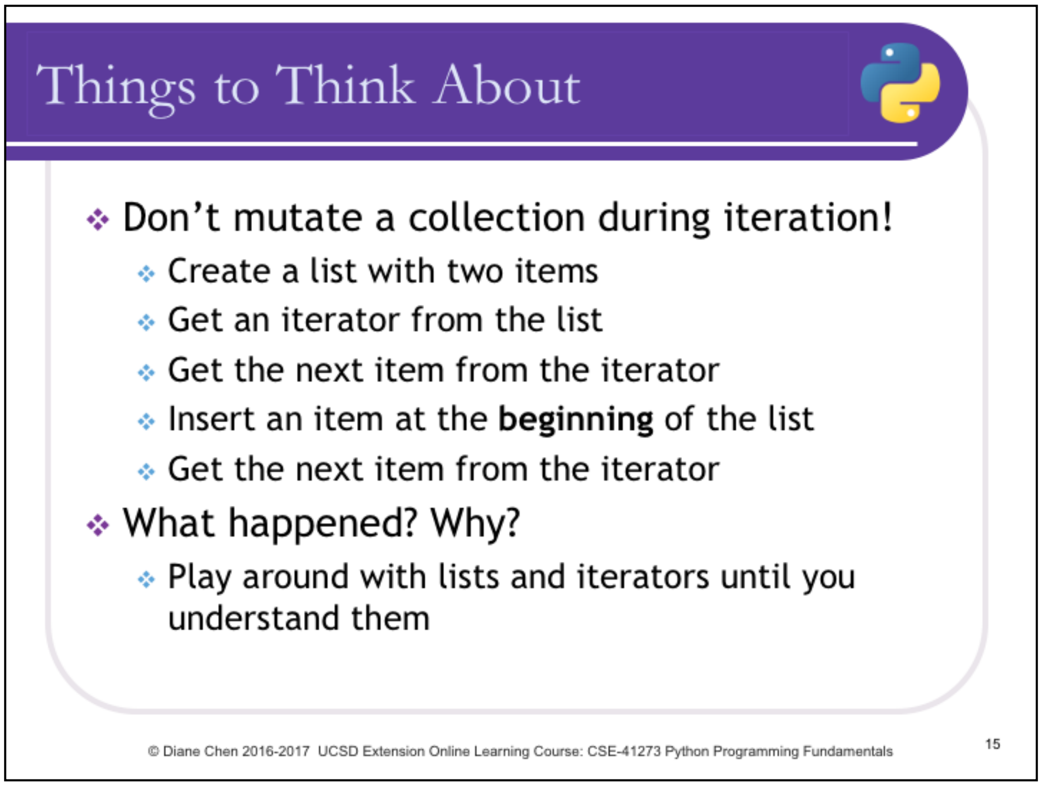
The combinatorics iterators. Permutations() takes an iterable and a size and returns an iterator that yields all the r-length permutations of the iterable. Note they are returned in tuples, not exactly as shown here; this is for brevity on the slides. Combinations() returns r-length subsequences of the iterable's items. Combinations_with_replacement returns r-length subsequences including duplication.



Other Iterators

- `chain(*iterables)` - returns all elements from iterables in order until they are all exhausted
  - Just like our `all_together()` generator
- `filterfalse(fn, i)` - return items from iterable `i` where `fn(item)` is false
- `islice(i, stop)` and `islice(i, start, stop[, step])` - return an iterator that returns specified elements of iterable `i`
- `zip_longest(*iterables, fillvalue=None)` - like built-in `zip()` but continues until all iterables are exhausted, filling in with `fillvalue`

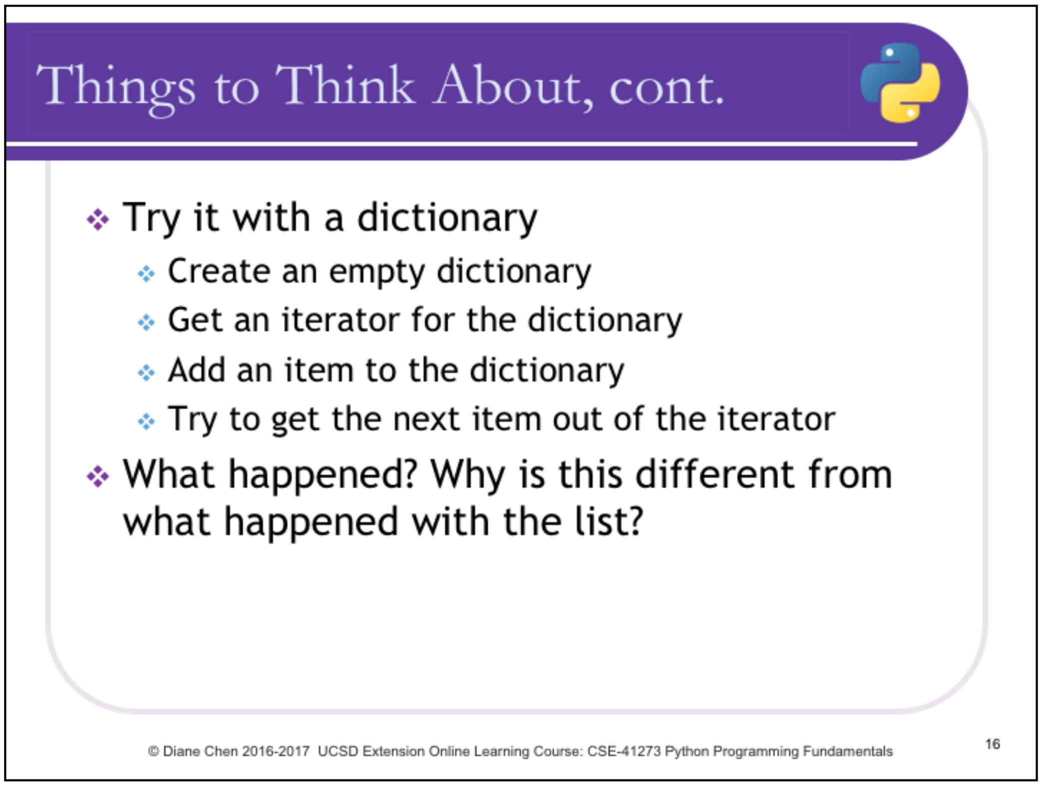
Here are some other useful functions from itertools. Chain() takes a list of iterables and returns all elements from each iterable in the list until they are all exhausted. Sound familiar? This is just like what our all_together generator function did. Filterfalse() is similar to the built-in filter() function, except it returns all the items from the iterable for which the function applied to it returns false. Islice() can be used to make an iterator that returns specific elements of the input iterable. Zip_longest() is just like built-in zip in that it returns an iterator that yields tuples from all the given iterables, but it continues to return tuples until all of the iterables are exhausted. It fills in the tuple values for the missing iterables with the given fillvalue.



Things to Think About

- Don't mutate a collection during iteration!
  - Create a list with two items
  - Get an iterator from the list
  - Get the next item from the iterator
  - Insert an item at the **beginning** of the list
  - Get the next item from the iterator
- What happened? Why?
  - Play around with lists and iterators until you understand them

Don't mutate a collection or iterable during iteration. Try this in the REPL. Create a list with 2 items in it. Create an iterator from the list and call next() on it. Now insert an item at the *beginning* of the list – look it up with help on list. Now get the next item from the iterator. What happened? Can you think of why? Play around with this until you understand what is going on here.

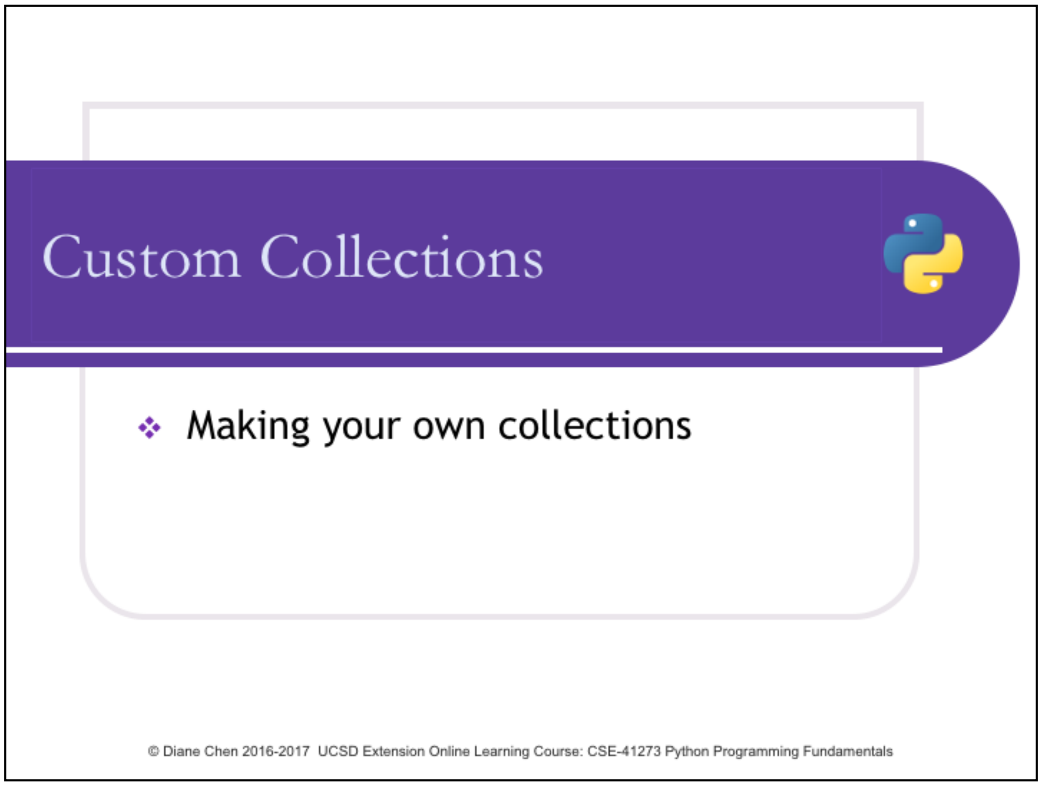


Things to Think About, cont.

- Try it with a dictionary
  - Create an empty dictionary
  - Get an iterator for the dictionary
  - Add an item to the dictionary
  - Try to get the next item out of the iterator
- What happened? Why is this different from what happened with the list?

16

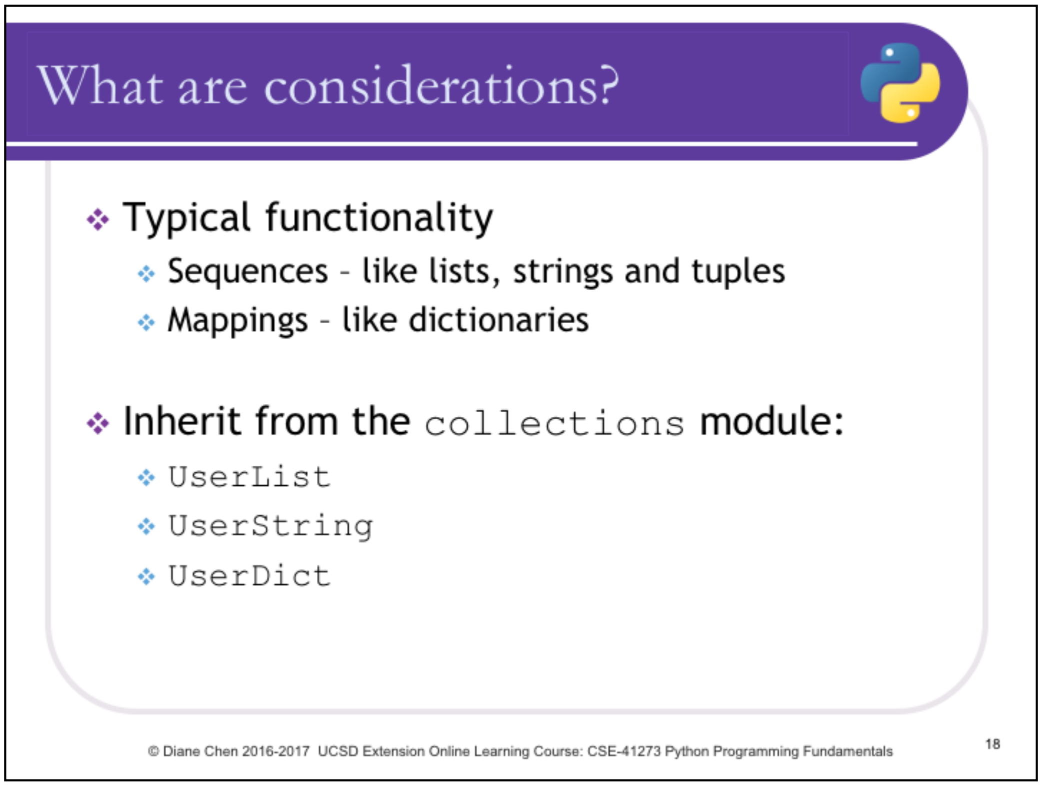
Now try a similar thing with a dictionary. Create an empty dictionary and get an iterator for the dictionary. Add an item to the dictionary. Try to get the next item from the dictionary. What happened this time? Why do you think this is different from what happened with the list? Think about why this happened – what is different between lists and dictionaries?



Custom Collections

- Making your own collections

I'm going to talk about making custom collections, for the times you need a little extra special functionality for strings, lists, or dictionaries.



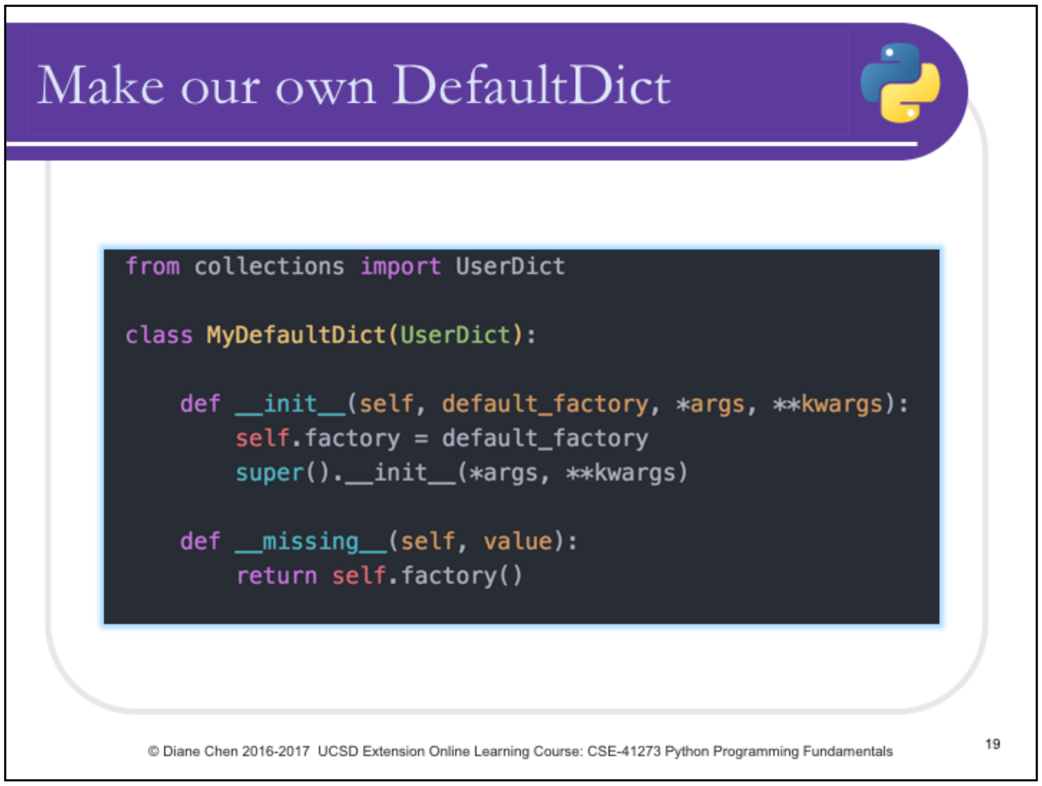
What are considerations?

- Typical functionality
  - Sequences - like lists, strings and tuples
  - Mappings - like dictionaries
- Inherit from the `collections` module:
  - `UserList`
  - `UserString`
  - `UserDict`

© Diane Chen 2016-2017 UCSD Extension Online Learning Course: CSE-41273 Python Programming Fundamentals 18

We have to think about what kind of functionality do we want our collection to have. Typically, we have 2 kinds of collections. Sequences behave something like lists, strings and tuples, where they are indexable, have an order and length. Then we have mappings, like dictionaries, that are indexed by a hashable key. We can inherit from anything, really, but Python's collections module provides some classes just for this purpose, that make custom collections easy.

In the collections module, we have base classes that we can inherit from to create a custom collection object. There are UserList for list-like objects, UserString for string-like objects, and UserDict for dictionary-like objects. Each one provides all of the existing functionality of its respective type. When we inherit from them, we inherit all of the methods that each type comes with. Then we only need to override and implement the methods that we want to behave differently.

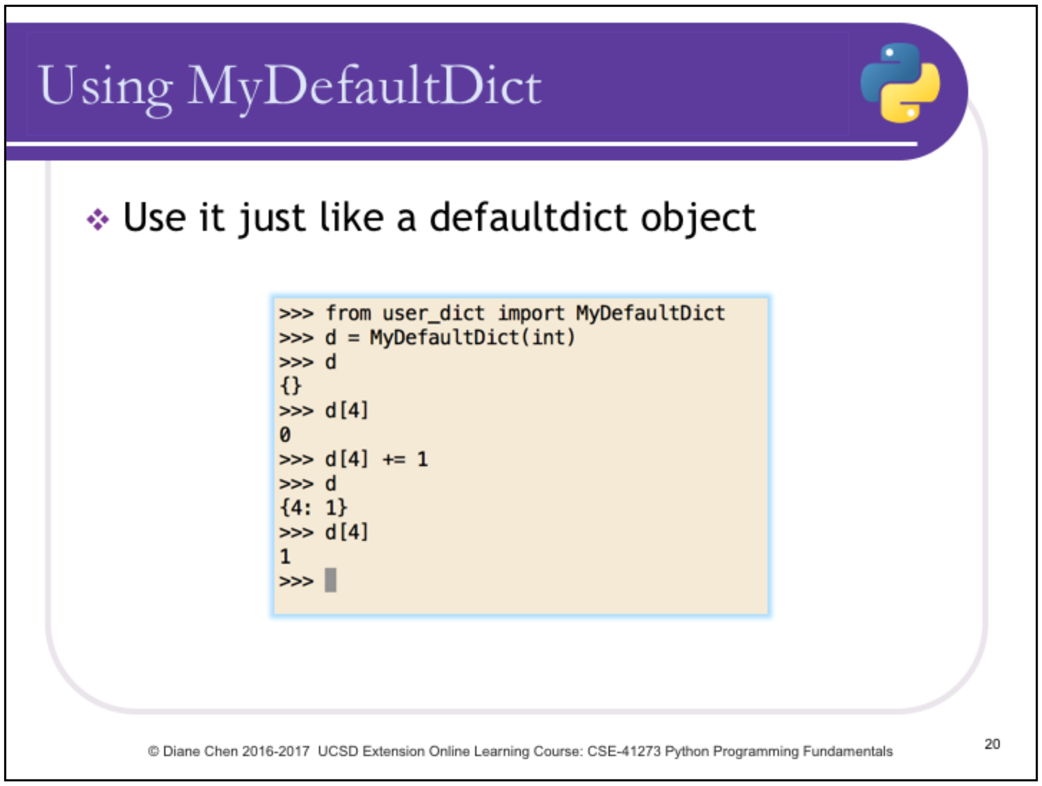


Make our own DefaultDict

```python
from collections import UserDict

class MyDefaultDict(UserDict):

    def __init__(self, default_factory, *args, **kwargs):
        self.factory = default_factory
        super().__init__(*args, **kwargs)

    def __missing__(self, value):
        return self.factory()
```

Here's an example of how we can implement our own version of defaultdict. By inheriting from UserDict in the collections module, we get all the regular dictionary methods for free. We create our own __init__() method so we can save the default factory function and then we call super with the remaining arguments, to handle whether there is an initializer given. Then, we implement the __missing__() method. This method is called automatically when there is an attempt to access the value of a non-existent key. In the __missing__() method, we return the result of calling the factory function that we stored from the initialization.

Normally in a regular dictionary, the __missing__() method is not implemented. When a non-existent key is accessed, the __missing__() method is called. Because it is not implemented, a KeyError is raised. So, by implementing the __missing__() method ourselves, we prevent the KeyError and get a default value back.



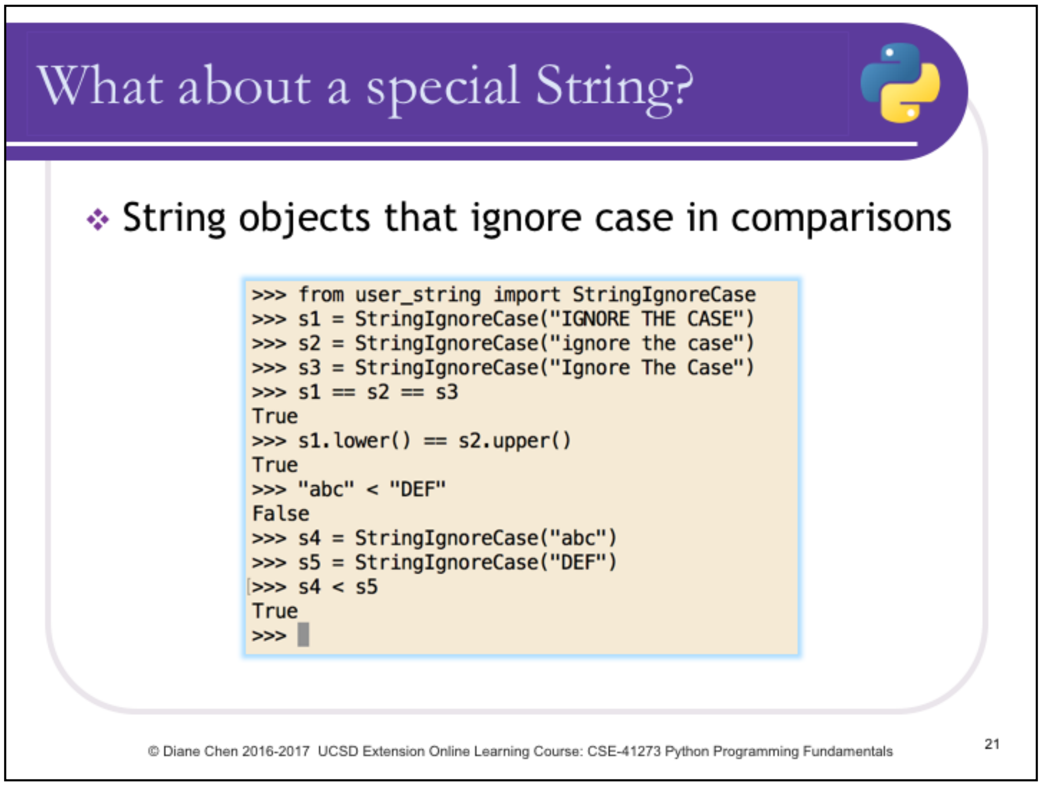
Using MyDefaultDict

- Use it just like a defaultdict object

```
>>> from user_dict import MyDefaultDict
>>> d = MyDefaultDict(int)
>>> d
{}
>>> d[4]
0
>>> d[4] += 1
>>> d
{4: 1}
>>> d[4]
1
>>>
```

20

We can use our new object just like a regular defaultdict. It has all the functionality of a regular dictionary, but we don't have to worry if the key is in the dictionary or not. Because we implemented the __missing__ method, we will not get key errors when attempting to access a value for a key that does not exist in the dictionary.



What about a special String?

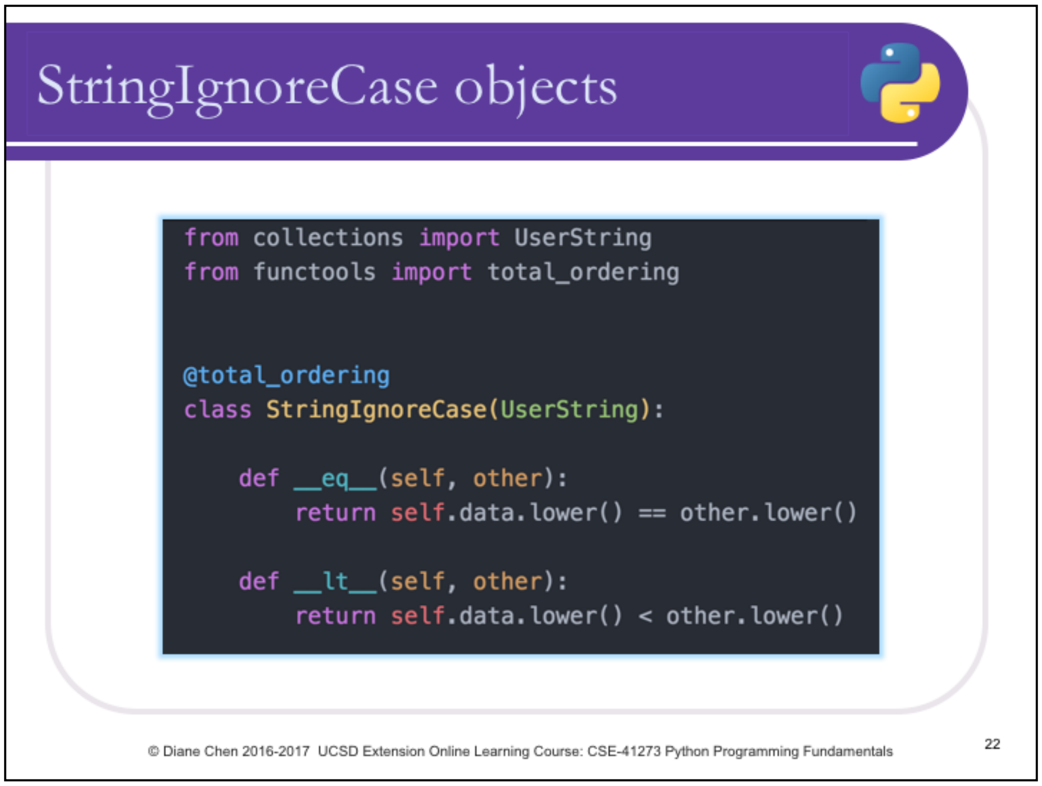
- String objects that ignore case in comparisons

```
>>> from user_string import StringIgnoreCase
>>> s1 = StringIgnoreCase("IGNORE THE CASE")
>>> s2 = StringIgnoreCase("ignore the case")
>>> s3 = StringIgnoreCase("Ignore The Case")
>>> s1 == s2 == s3
True
>>> s1.lower() == s2.upper()
True
>>> "abc" < "DEF"
False
>>> s4 = StringIgnoreCase("abc")
>>> s5 = StringIgnoreCase("DEF")
>>> s4 < s5
True
>>>
```

21

Now let's go through an example of making a custom string object that has its own special behavior. Say we want to make strings that we can compare, ignoring the case of the string. I've given the class the rather unwieldy name of "StringIgnoreCase". We can create the strings s1, s2, and s3, where the characters are the same, other than the case. So, even though the strings are not truly identical, when we compare them, Python tells us they are all equal. Even if we apply the string methods upper() and lower(), they still compare as equal. This is because of the behavior that we have put into it.

With "normal" strings, lowercase letters come after uppercase, so lowercase will always compare as being greater than uppercase. But, with our new strings, because the case is ignored in comparisons, our lowercase "abc" will compare as less than uppercase "DEF".

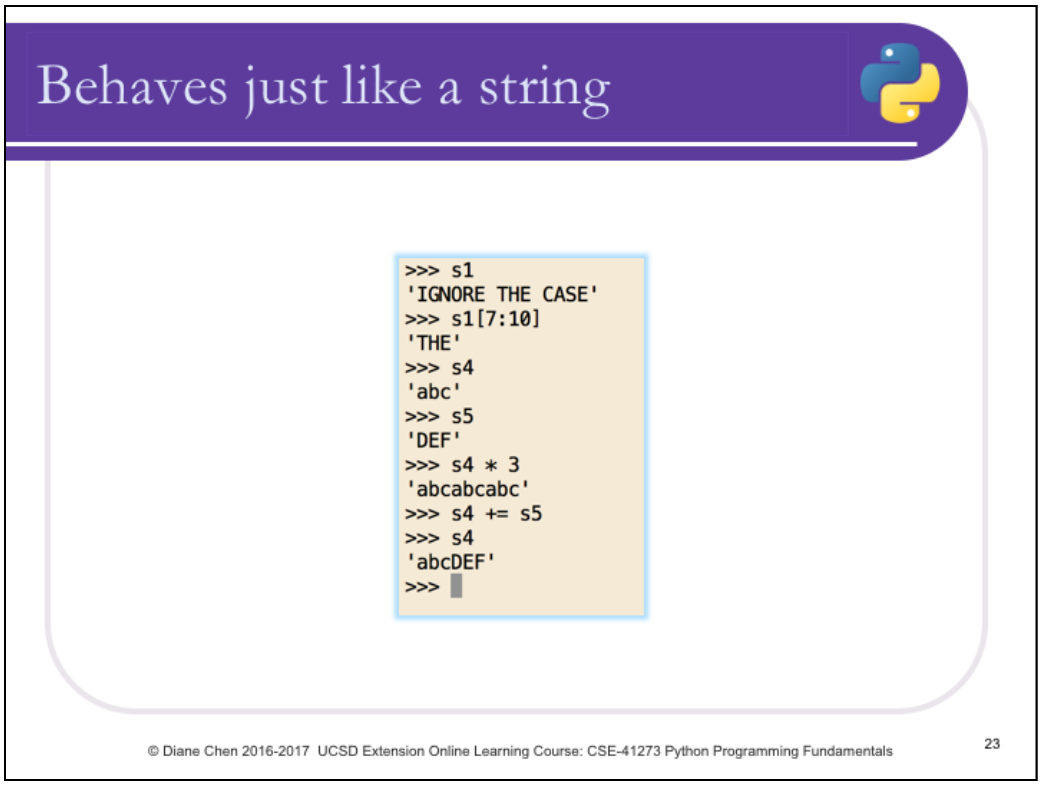


StringIgnoreCase objects

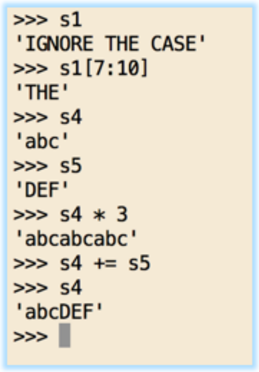
```python
from collections import UserString
from functools import total_ordering


@total_ordering
class StringIgnoreCase(UserString):

    def __eq__(self, other):
        return self.data.lower() == other.lower()

    def __lt__(self, other):
        return self.data.lower() < other.lower()
```

22

To create our special StringIgnoreCase class, we inherit from the UserString class in collections. We add the @total_ordering decorator to the class so Python will handle the comparison operators we don't define ourselves. Then, we only need to implement the __eq__() and __lt__() methods. By forcing the string, stored in the data attribute, to be lowercase before comparison, we can make it behave the way we want.



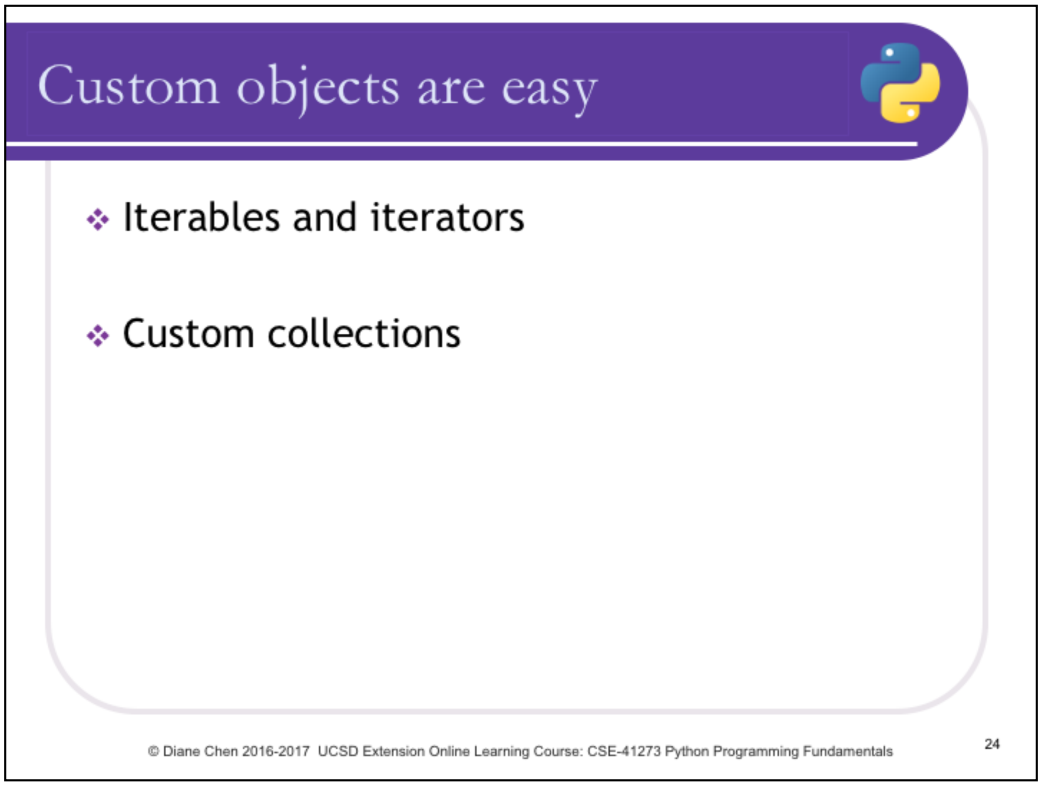
Behaves just like a string



```
>>> s1
'IGNORE THE CASE'
>>> s1[7:10]
'THE'
>>> s4
'abc'
>>> s5
'DEF'
>>> s4 * 3
'abcabcabc'
>>> s4 += s5
>>> s4
'abcDEF'
>>> 
```

23

The objects created behave just like strings in every other way. They are stored as they were defined, so the strings, when displayed, show the case they had when they were created. We can slice them, multiply and add them as if they were normal strings. The only thing that is different is when we compare them.



Custom objects are easy

- Iterables and iterators
- Custom collections

You can see from these examples that inheriting from existing objects and defining our own special behavior is really pretty easy in Python. By taking advantage of the code re-use we get from inheriting from the right objects, we save ourselves a lot of time and aggravation in our custom classes. If we had to implement the special StringIgnoreCase class entirely by ourselves, it would be a *lot* of work. We would have to implement indexing, slicing, adding, multiplying, and many other things that we got for free from the UserString class. And, in addition to implementing it all, we would have to write tests for all that other functionality!