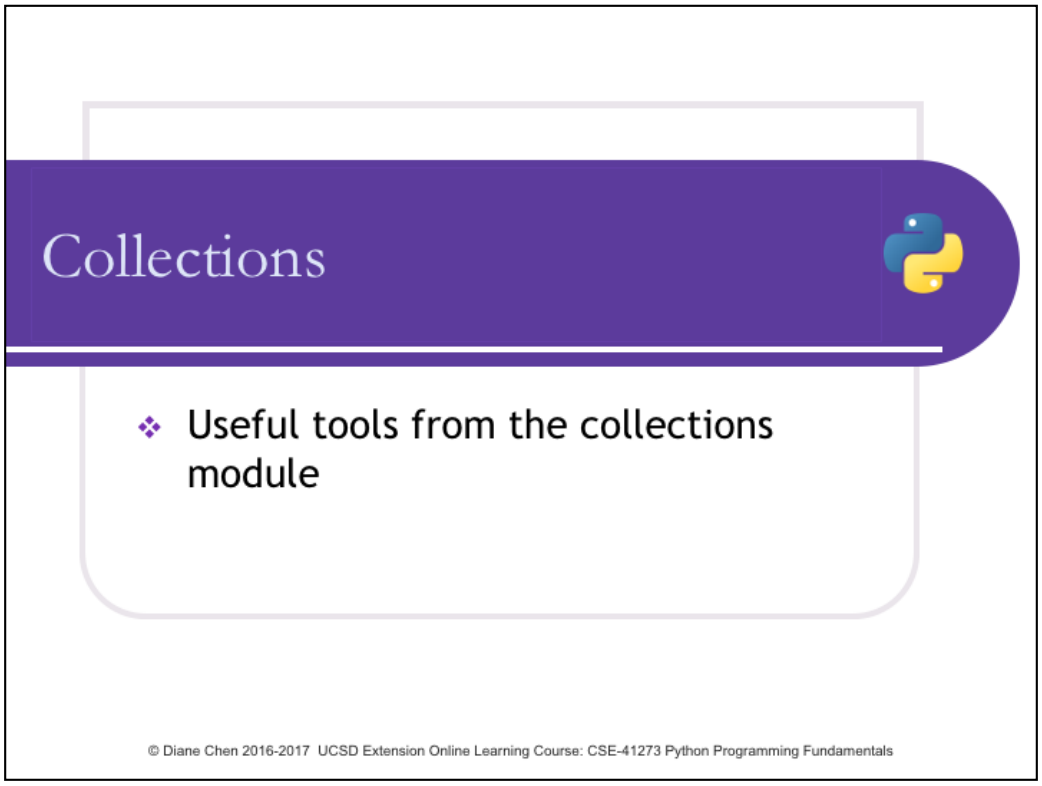
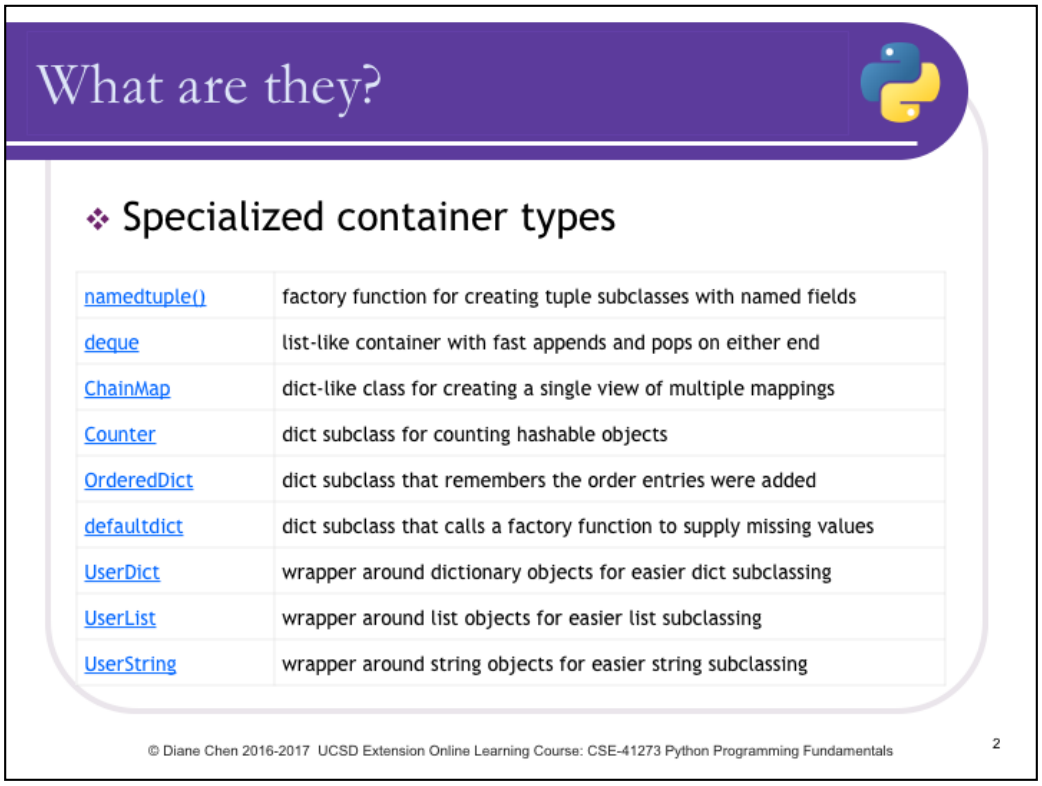


We are going to talk more about classes this week but first I want to show you the collections module because it has some very handy tools in it.

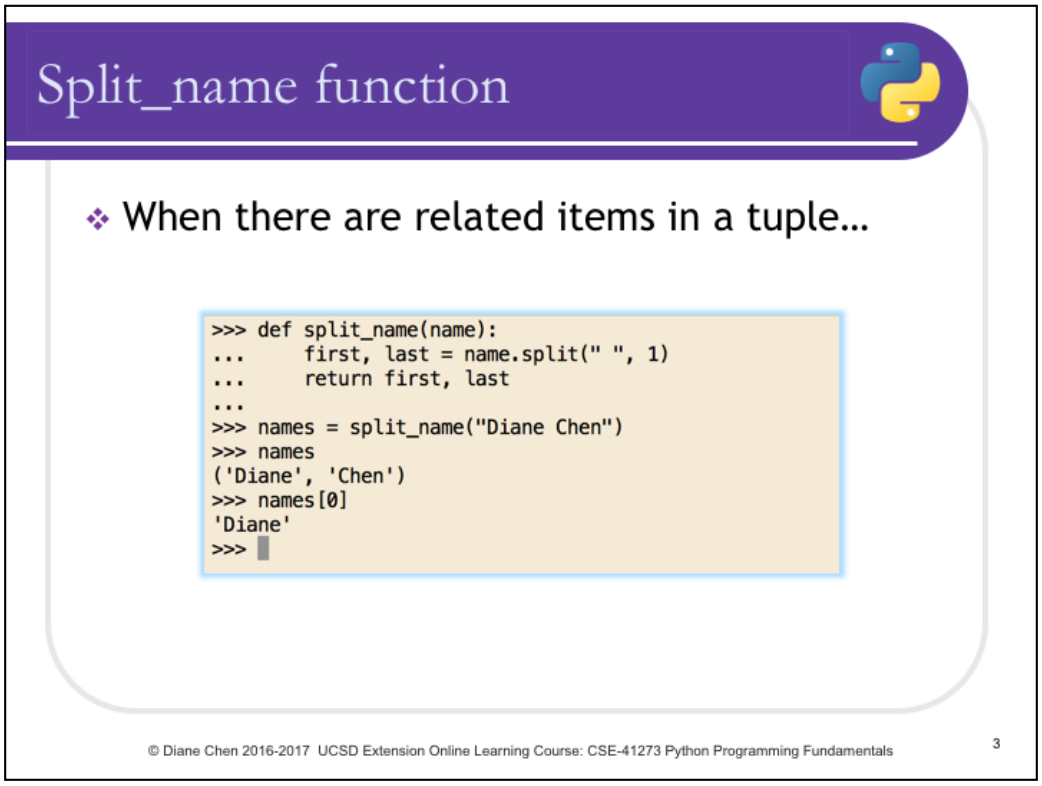


The collections module provides a number of specialized container types when the standard ones – list, set, dictionary, tuple – don't give the best functionality for your purposes.

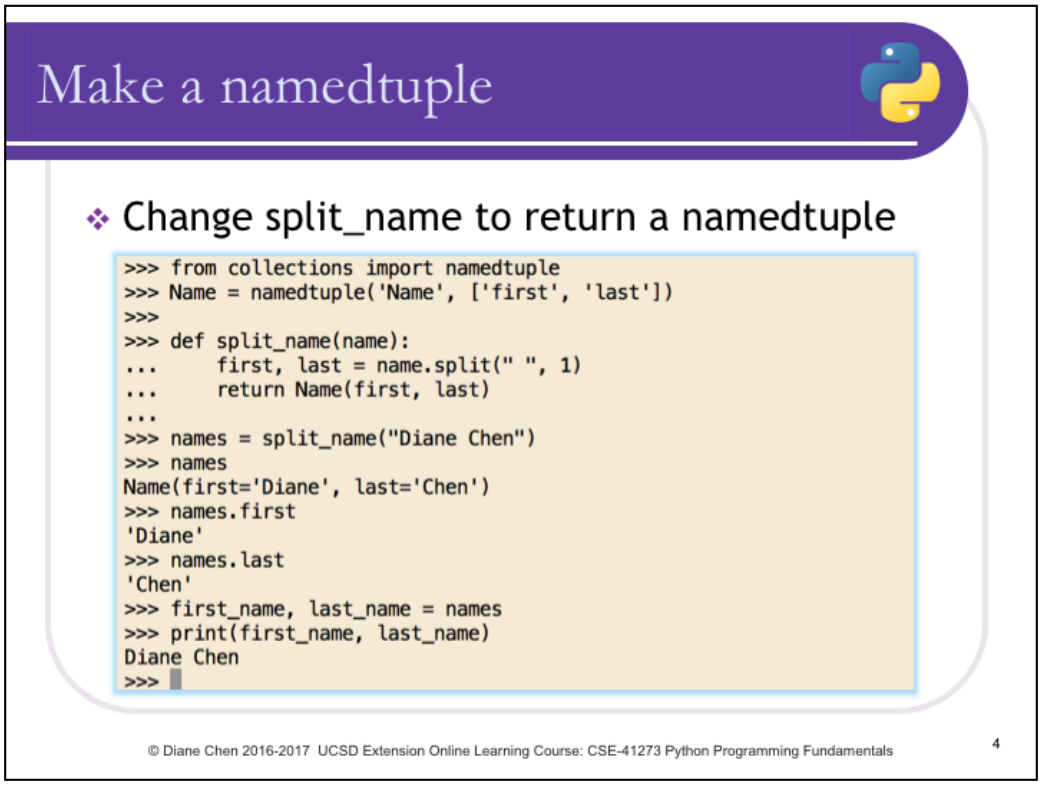
The namedtuple is a type that creates subclasses that have names attached. We will talk about namedtuples next.

A deque is a specialized form of list. The name "deque" stands for "double-ended queue". A queue is a first-in-first-out data structure where additions only happen at the end and removals only happen at the beginning. In reality, Python's list is a modified stack, which is a first-in-last-out data type, so appends (add to the end) and pops (remove from the end) are optimized. Python's list is indexable and you can add or remove from any part of the list, but it is much less efficient when it is not the end of the list. The deque object is useful if you are doing time-intensive operations and need a data structure that is more efficient for adds and removes that are not necessarily at the end.

ChainMap is a useful way of operating on multiple iterables. The Counter is very useful; we will look at that later also. OrderedDict is a dictionary that remembers the order things are added to it, similarly to how lists are ordered. Defaultdict is nice when you have operations on the dictionary where the index may or may not exist. Instead of raising an error when you try to access a key that doesn't exist, it will return a default value. We will see that later on in this lesson. UserDict, UserList, and UserString are useful for making your own subclasses of dictionaries, lists and strings respectively.



Now we'll go through an example of how namedtuples can be useful. Let's make a function to split a name into first and last name. Note that the split method on strings takes an optional argument of how many splits to do. We'll just split once. Normally it splits on whitespace and splits as many items as it can based on the whitespace. We're going to assume that first names are always one word, which isn't a good assumption, but for the sake of example, that's how this is going to work. So the result is a tuple where the first item is the first name and the second item is the last name. We know that we can get the first name with index zero, but what about other people? If someone calls our function, they'll have a tuple of two values. They could probably guess what the values are, but what if they weren't sure?



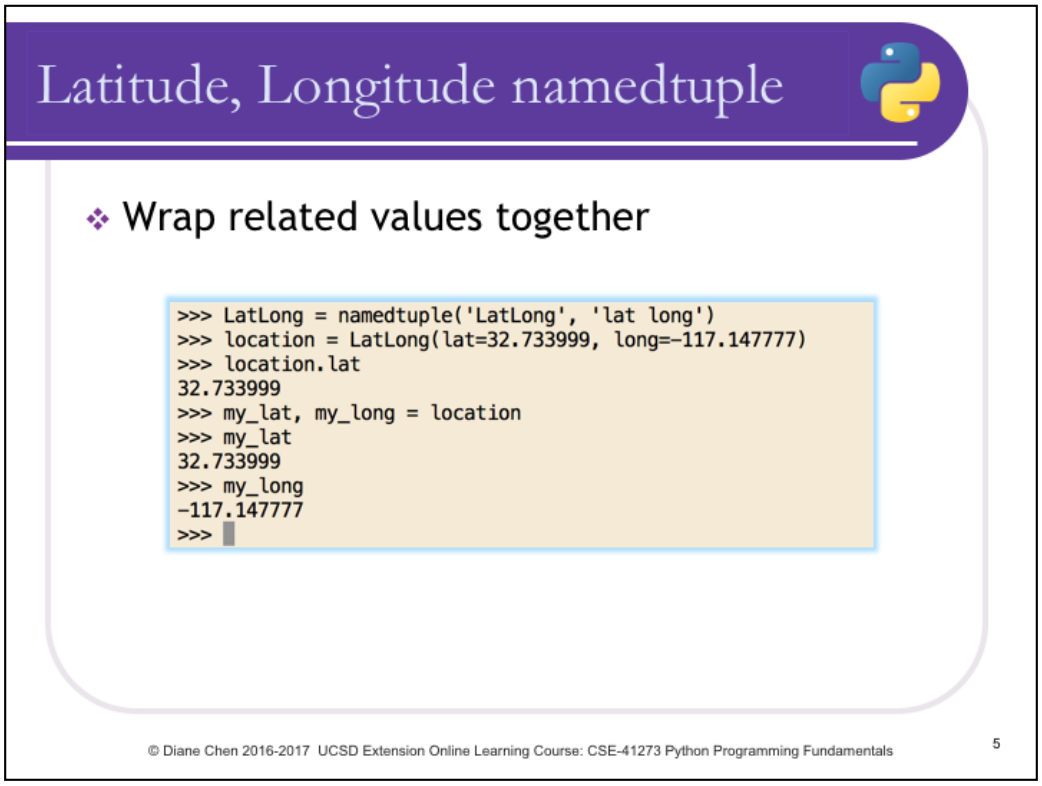
Python to the rescue! We can use namedtuple to make the returned value more informative. We import namedtuple from the collections module and define the namedtuple Name with a capital N.

Because functions are first-class objects, the call to namedtuple creates and returns a "factory function" to create namedtuples that we store in the variable Name with a capital N.

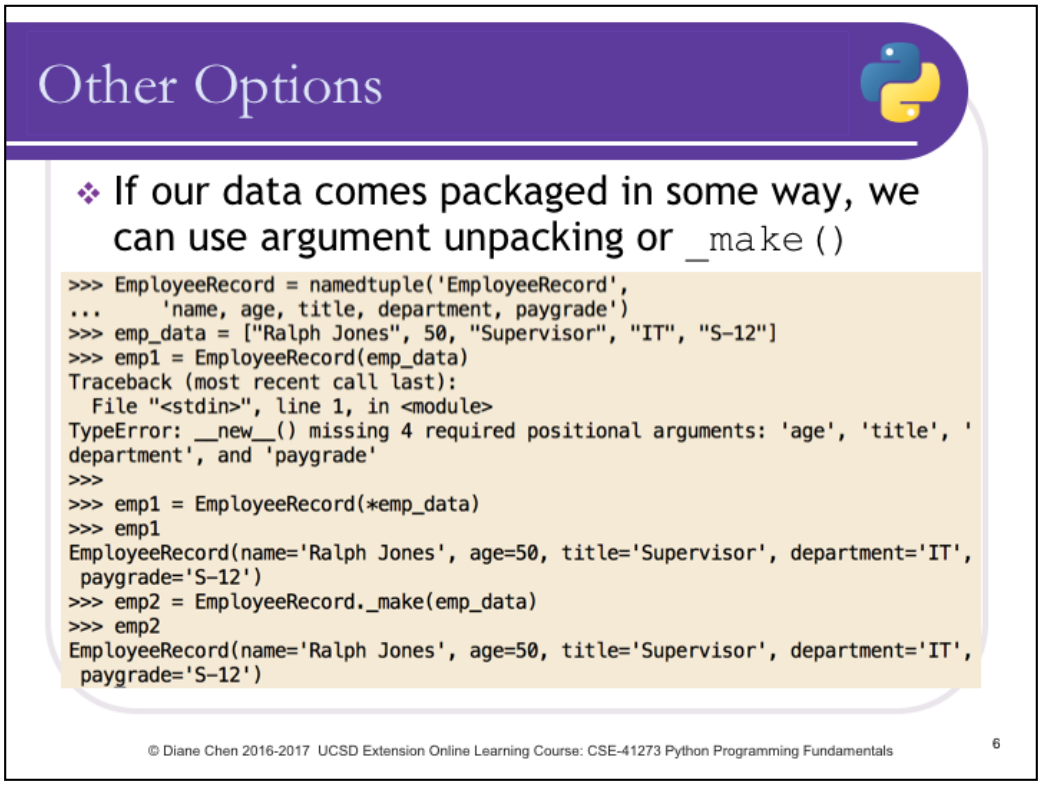
We could have used a different variable name for our factory function, but it is less confusing if they are the same name. We capitalize the namedtuple factory function because it is a new class that gets created for us by namedtuple. And I probably shouldn't have used a function dealing with names for the first namedtuple example. But, hopefully, it will all be clear.

Inside the function split_name, we split the names from the input string and return the result of calling our namedtuple factory function Name.

We can use this new object in the variable names just like a tuple but we can also use it like an object with attributes; treating the named values of the tuple as if they are properties or attributes, by referencing names.first and names.last. It behaves just like a tuple, so we can do tuple unpacking into the first_name and last_name variables as shown here. And we can print out first_name and last_name and have them shown together.



So, whenever you need to wrap some values together, think of namedtuple. It might just be what you need, to keep your code cleaner and more maintainable. Here is an example of tying latitude and longitude into a namedtuple. Note that we can also use keyword arguments when we create a new namedtuple object. Did you notice that the definition of the fields is different from the way we defined them in the previous slide? It is very flexible that way.



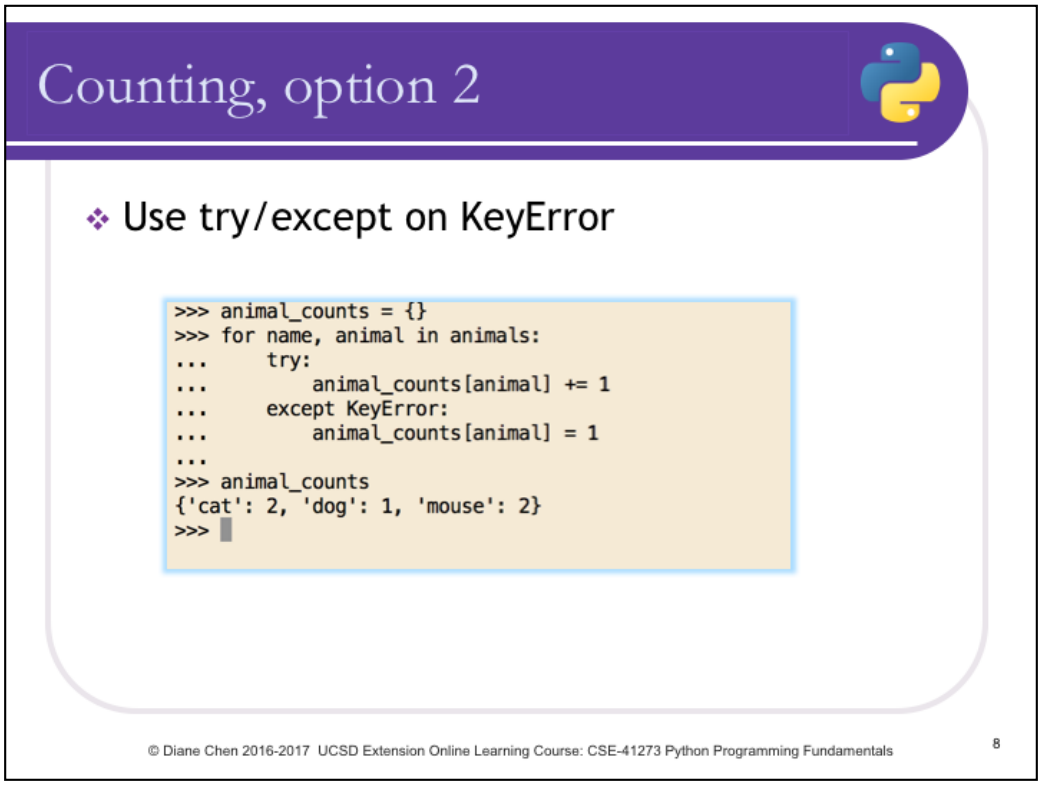
In real life, if we are using namedtuple in a program, we will probably be getting our data from some other source, perhaps a database, or CSV file. So each record that we want to put into a namedtuple will already be packaged up, for example in a list. I've typed it out here in a list called emp_data, but in a real life program, emp_data would be coming from somewhere else. We can't just input the emp_data list to the EmployeeRecord namedtuple factory, since it isn't the correct number of arguments, so we get an error. Fortunately we can use argument unpacking to separate the list into individual arguments. There is also a method, "underscore make" on the namedtuple object, that accepts an iterable of some kind – it doesn't matter what kind, as long as it is iterable and contains the correct number of items. "Underscore make" tells the EmployeeRecord namedtuple factory to iterate through the emp_data iterable (namely our list) to get the arguments to use to create our new namedtuple object. In our example here, both creation options produce the same namedtuple results.

Use of defaultdict

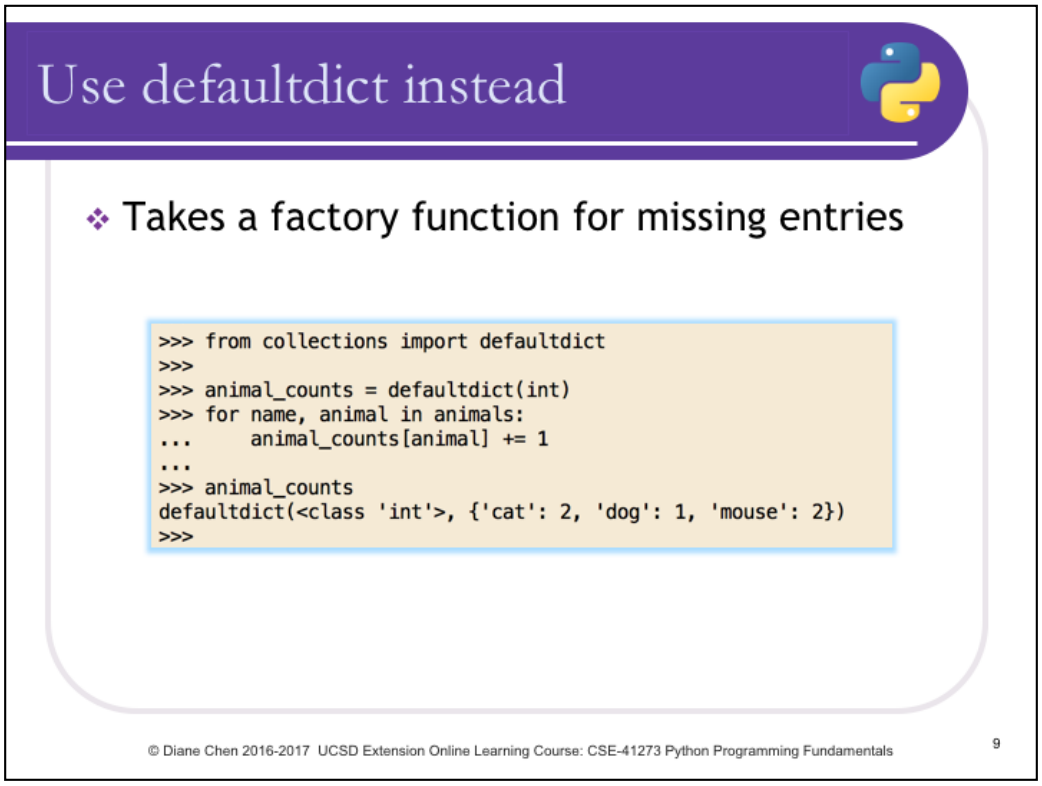
- Example: counting items from iterable

```
>>> animals = [("diane", "dog"), ("kurt", "cat"), ("mary", "mouse"),
...     ("cory", "cat"), ("margaret", "mouse")]
>>>
>>> animal_counts = {}
>>> for name, animal in animals:
...     if animal not in animal_counts:
...         animal_counts[animal] = 0
...     animal_counts[animal] += 1
...
>>> animal_counts
{'cat': 2, 'dog': 1, 'mouse': 2}
>>>
```

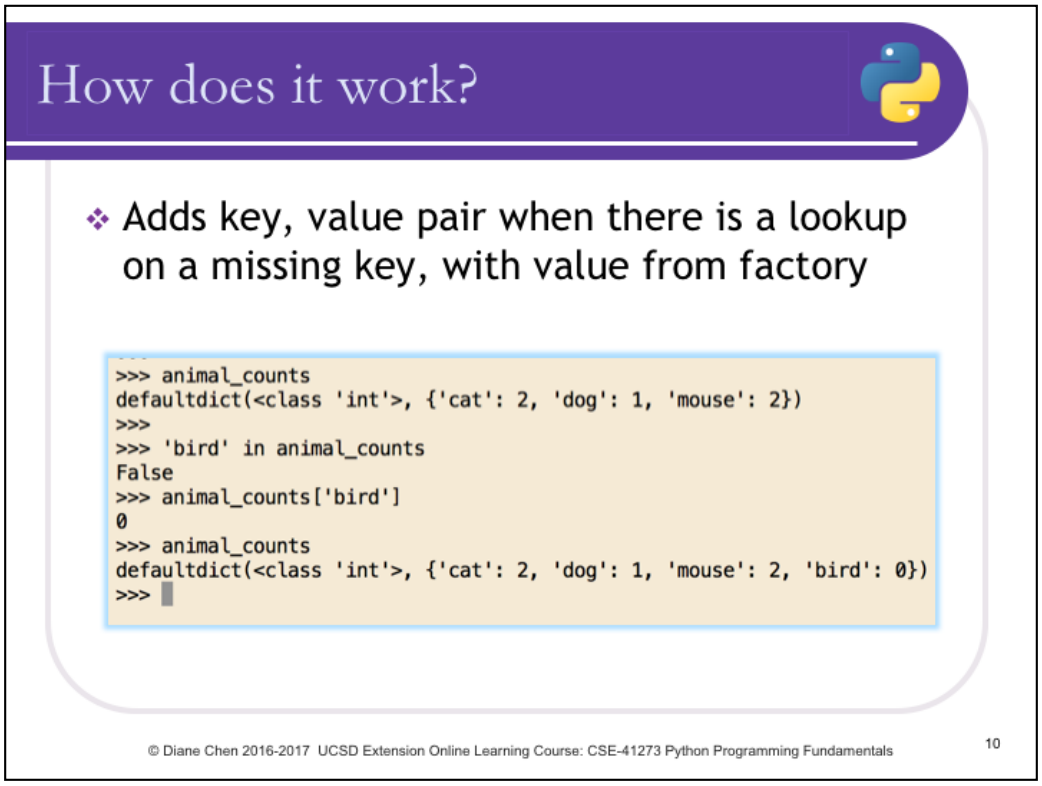

Now let's take a look at using defaultdict. Suppose we have a list of tuples where the the tuples contain a person's name and the type of animal they have. Say we want to count how many of each animal type our group of people has. We can use a dictionary for this, so we will have a dictionary of animals and the number of that animal type. Using a regular dictionary for counting, we have to check to see whether the animal is in the dictionary; if it is not, then we add it with a value of zero and then we increment the count value, whether it was new or not. If we didn't add it as zero, then when the system tried to access the value in order to increment it, we would get a KeyError.



We could also do a try/except and catch the key errors when we try to access a non-existent dictionary key. Both code examples work, but it's a little ugly no matter which way we do it. There must be a better way!



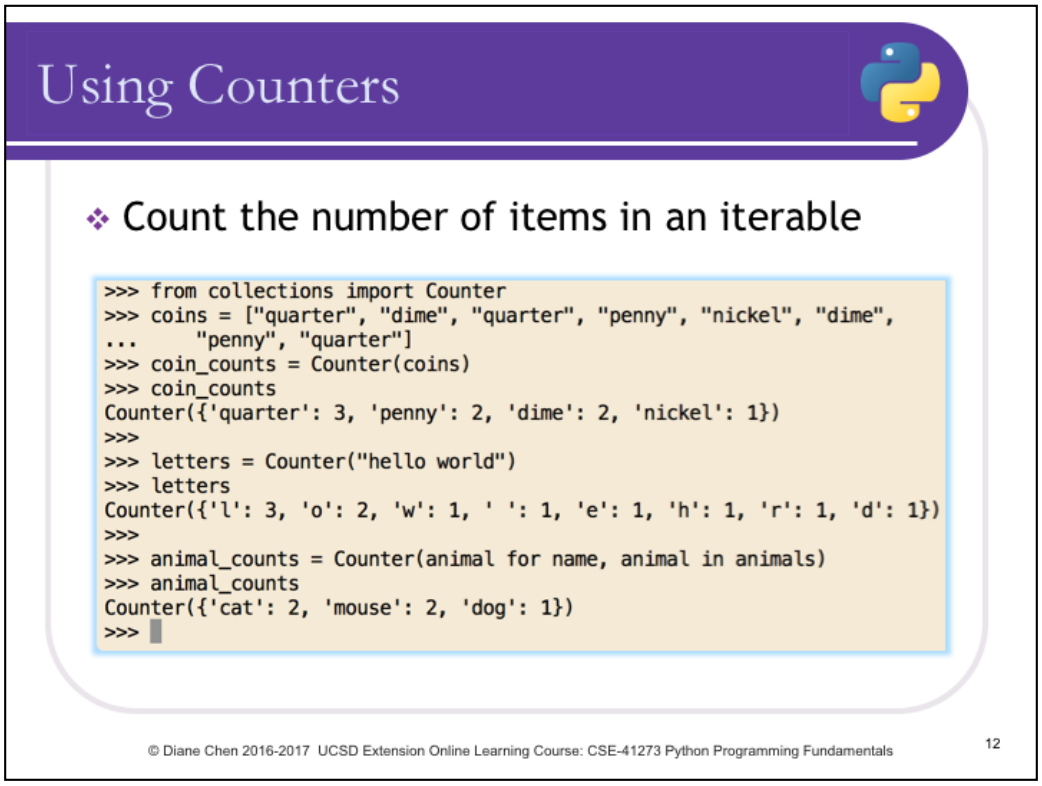
Of course there's a better way! Because situations like this are so common, the defaultdict was invented. When we create a defaultdict, we give it a factory function that supplies the value when a non-existent key is accessed. Because we are counting, we will use ”int” as our factory function. When ”int” is called without arguments it returns an integer of zero. This is perfect because we are incrementing the value and it will be 1 after the loop is done. Now our code is so much cleaner and more understandable!



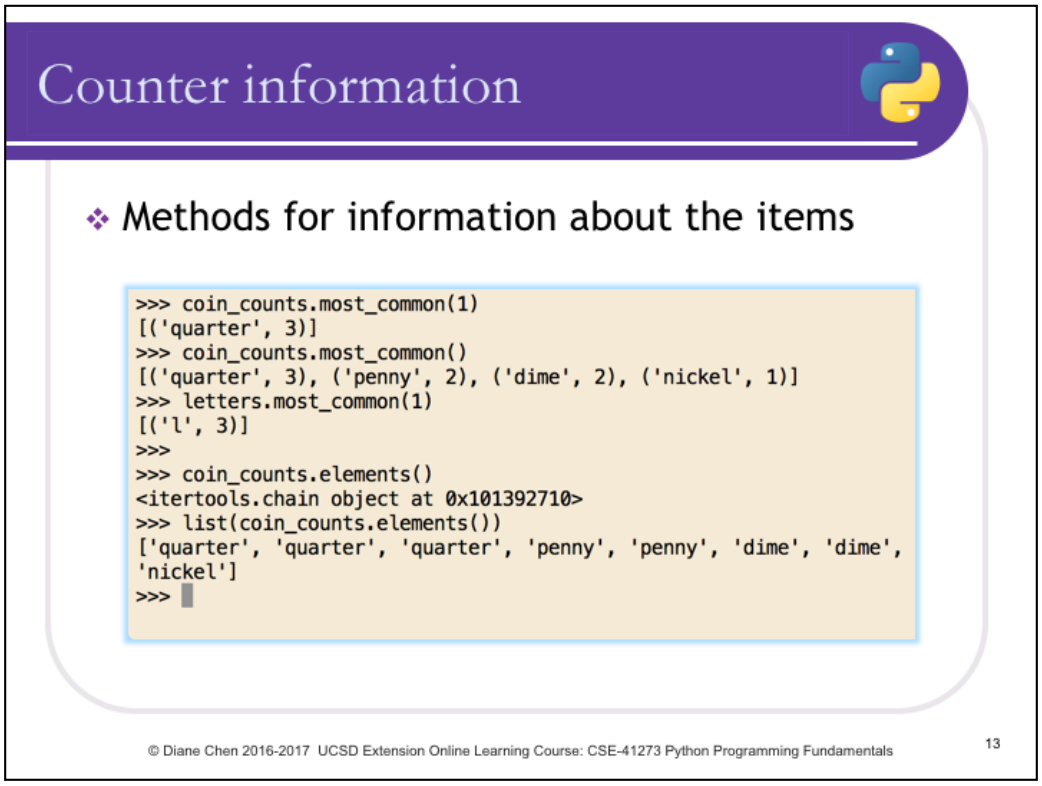
So how does it work? It only adds the key when we try to access a missing key's value. So in our example here, when we inquire if 'bird' is in animal_counts, the answer is False, because we haven't tried to access it yet, we only asked to see if it's already there. Once we try to access it, as here when we ask for the value of animal_counts for 'bird', then it is added using the factory function int(). In all other ways, a defaultdict behaves exactly like a regular dictionary. The important thing to remember is that any attempt to access a non-existent key's value will cause the key to be added to the dictionary with the default value.

Now let's see an example of using "list" as the factory function. It is very handy when you need to accumulate one-to-many data in a dictionary. Suppose we want to create a dictionary animal_owners, where the keys are the animal types, and the values are lists of the people who have that kind of animal. We will want to append the owner's name to the list corresponding to the animal in the key of the dictionary. By using "list" as our factory function, when it tries to access the list belonging to a non-existent animal key, the factory automatically makes an empty list and then we can append to it without error. We don't have to worry about if it exists or not. Here we see the code to create our animal_owners defaultdict. We still have the animals list available, as I've shown at the top. We create an animal_owners defaultdict with "list" as the factory function. Then all we have to do as we loop through the animals list is append the name to the list corresponding to the animal. Very easy!

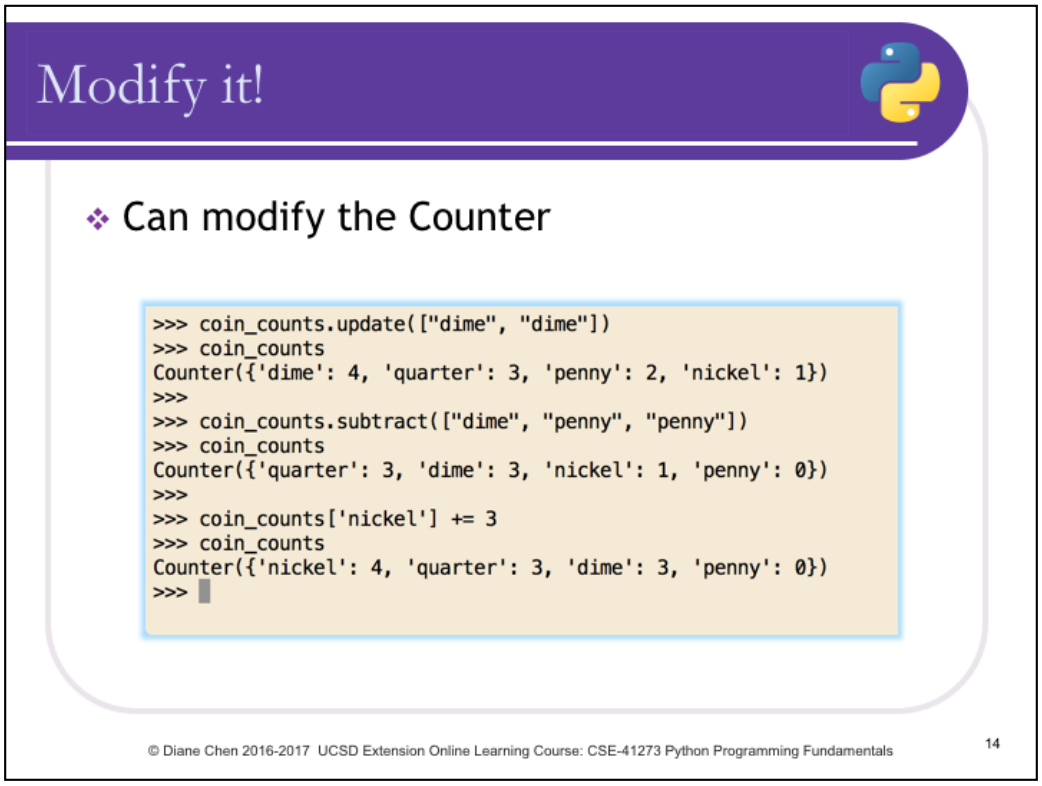
Note that the factory function can be any function. We can even define our own factory function or use a lambda function. That's getting a little beyond the scope of this class, but it's something to keep in mind for the future.



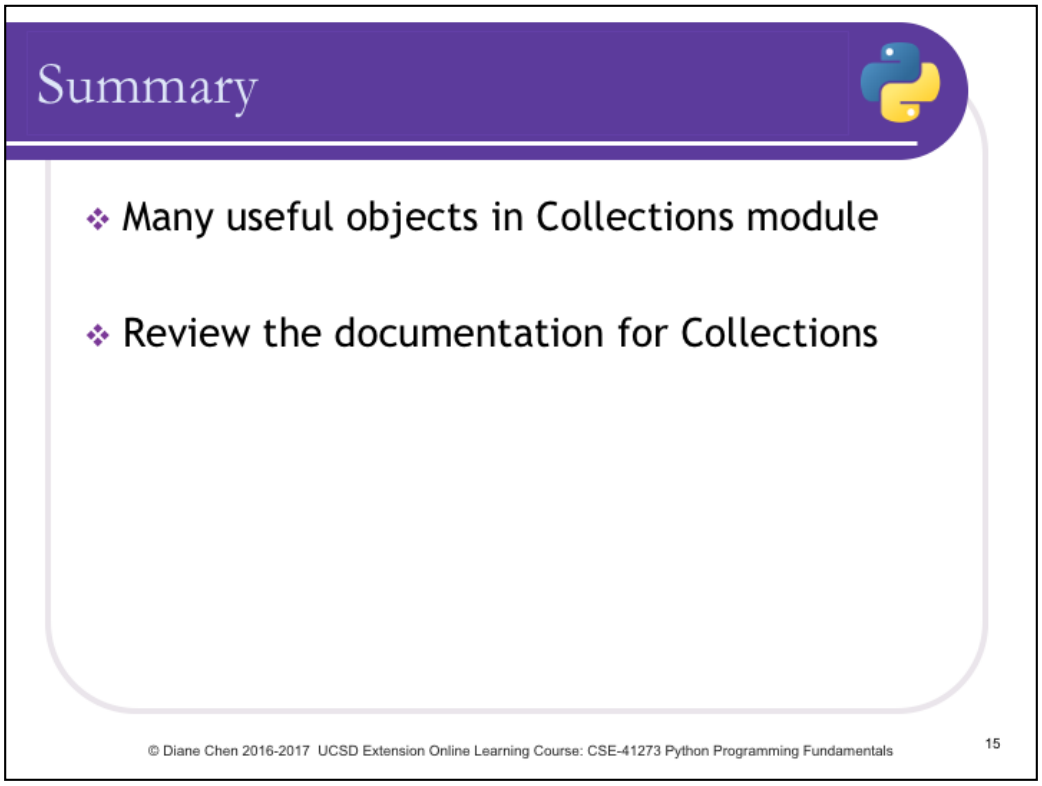
Counting the numbers of items of things in an iterable is very common, so it should come as no surprise that there is a Counter in the collections module. Say we have a list of coins and we want to count them. Easy! Just use a Counter object. It works on any iterable, even strings - you haven't forgotten that strings are also iterables, have you? And remember how we used a defaultdict to make counting animals easier? It's even easier with a Counter, as we can pass in a comprehension directly to the Counter when we create it. Take a close look at the comprehension used here. Because the animals list contains tuples, we have to unpack the names and animals, even though we only care about the animals.



From the Counter object, we can get information about the counter items. For example the most common items, optionally limited to a specific number of the most common. The elements() method produces an iterable object that we can use anywhere we need an iterable, such as making a list as shown here.



We can modify the counter as we get more data. The update method accepts an iterable and adds these objects to the Counter. We can subtract items from the counter too. Because it behaves just like a dictionary, we can directly modify the count value if we want, as here where we increment the number of nickels. We could even input another Counter object into the update() method, to add its values to the one.



As we've seen, there are many useful container objects in the Collections Module. We will see more of them in the future. Please review the documentation for the Collections module to get a feel for some of the other methods available on each container.