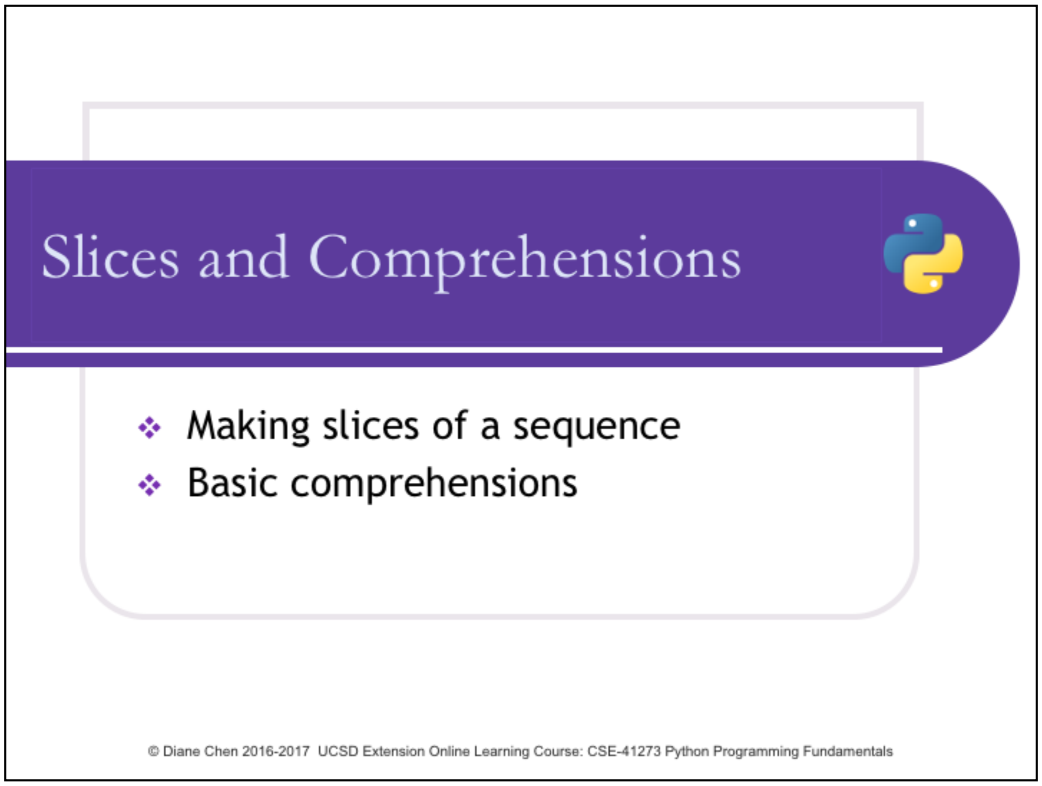
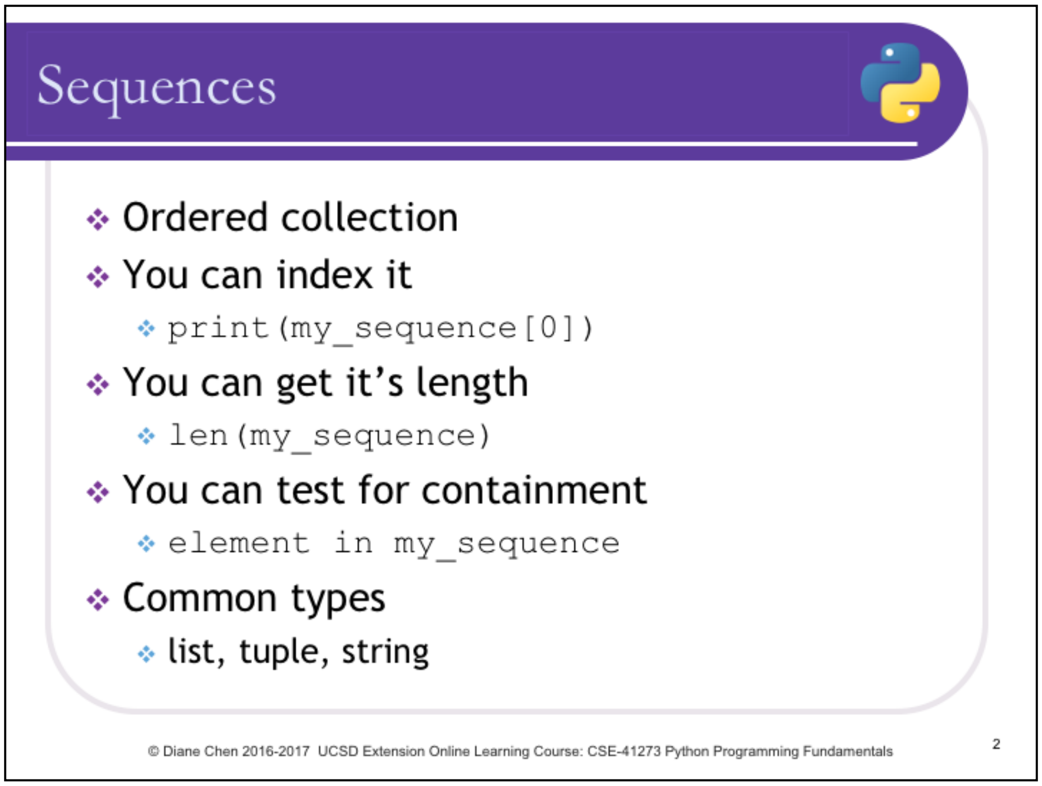


Slices and Comprehensions

- Making slices of a sequence
- Basic comprehensions

In this lecture I'm going to talk about slicing, a powerful way to get subsets of sequences. I'm also going to discuss basic list, set and dictionary comprehensions.

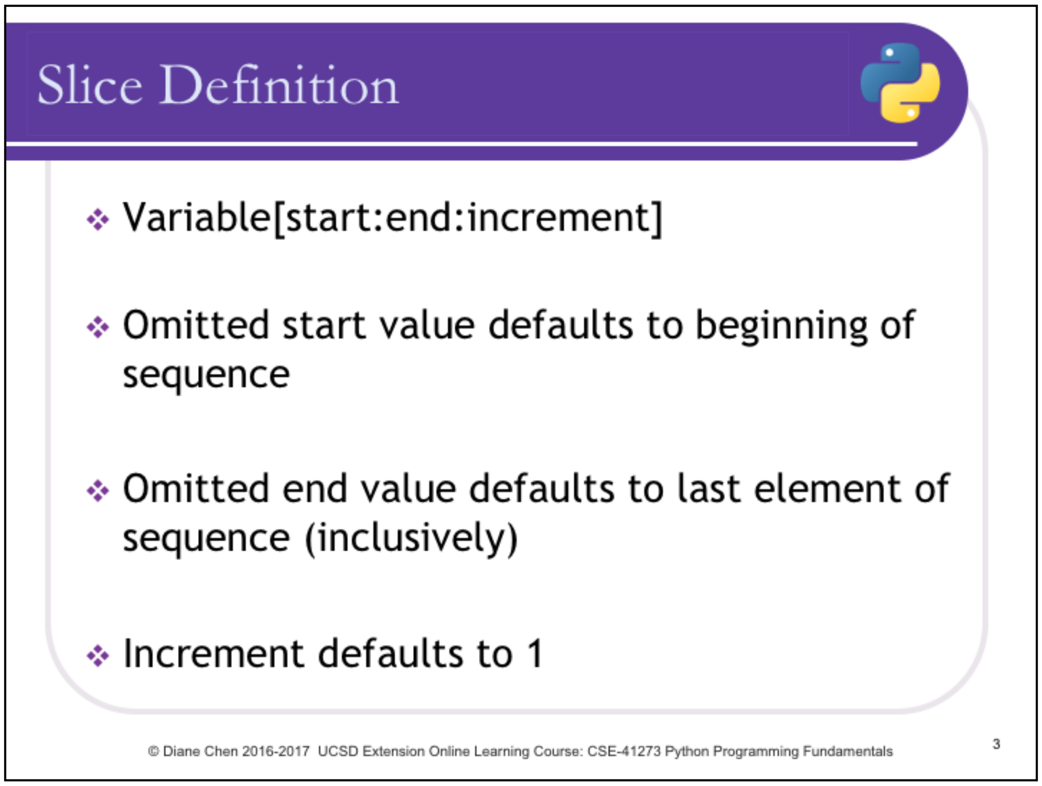


Sequences

- Ordered collection
- You can index it
  - `print(my_sequence[0])`
- You can get it’s length
  - `len(my_sequence)`
- You can test for containment
  - `element in my_sequence`
- Common types
  - list, tuple, string

2

What is a sequence in Python? A Sequence-style object is an ordered collection of objects. You can index elements of it, get its length, and test for containment. The common built-in types that support the sequence protocol are lists, tuples, and strings. Since dictionaries and sets are not ordered, they cannot be treated as sequences. You can think of sequences as ordered subsets of iterables.

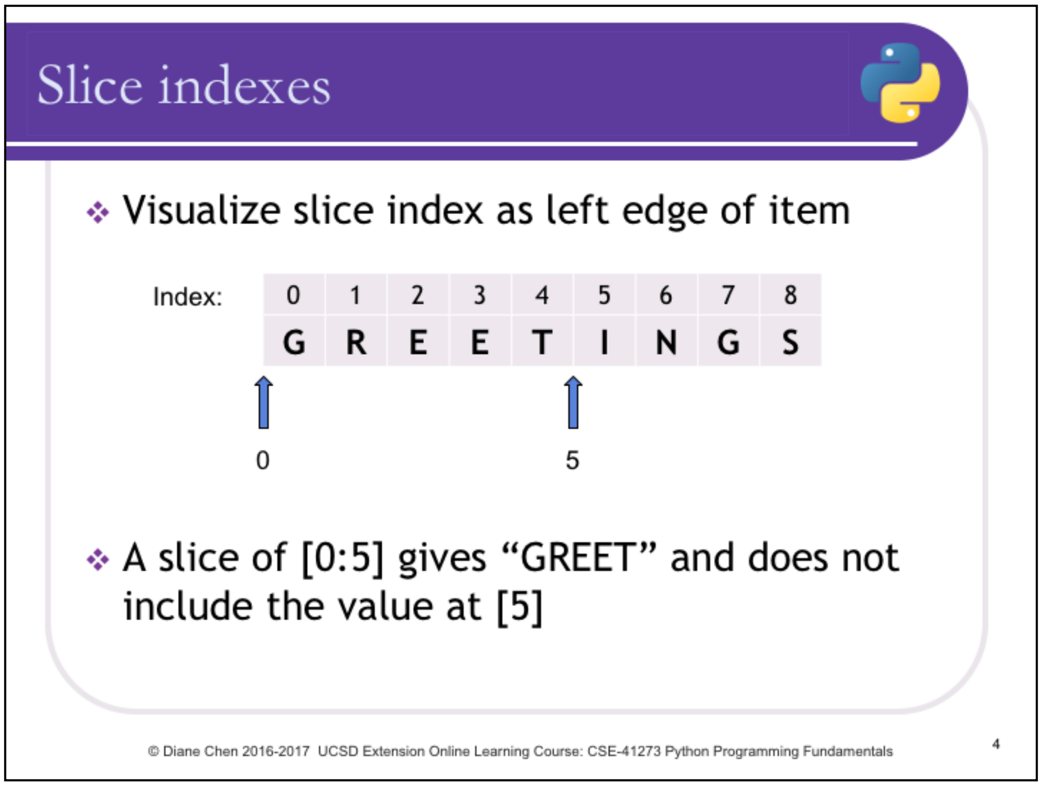


Slice Definition

- Variable[start:end:increment]
- Omitted start value defaults to beginning of sequence
- Omitted end value defaults to last element of sequence (inclusively)
- Increment defaults to 1

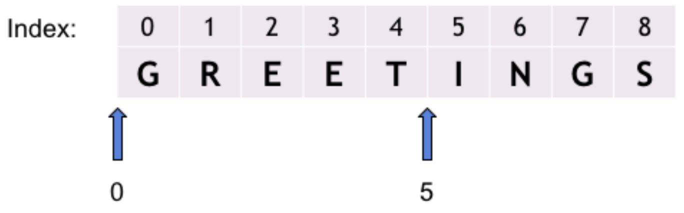
3

Slices are like subsections of sequences. We'll be using lists and strings in our examples. A slice is indicated with the start and end index of the slice, with an optional increment value that defaults to 1; separated by colons. It's actually the colon that tells Python we want a slice. Note that the end value is Non-inclusive, so if an end value is given, the slice will be up to but not including the end index. When the start value is omitted, it defaults to the first element and when the end value is omitted, it defaults to the end of the sequence.



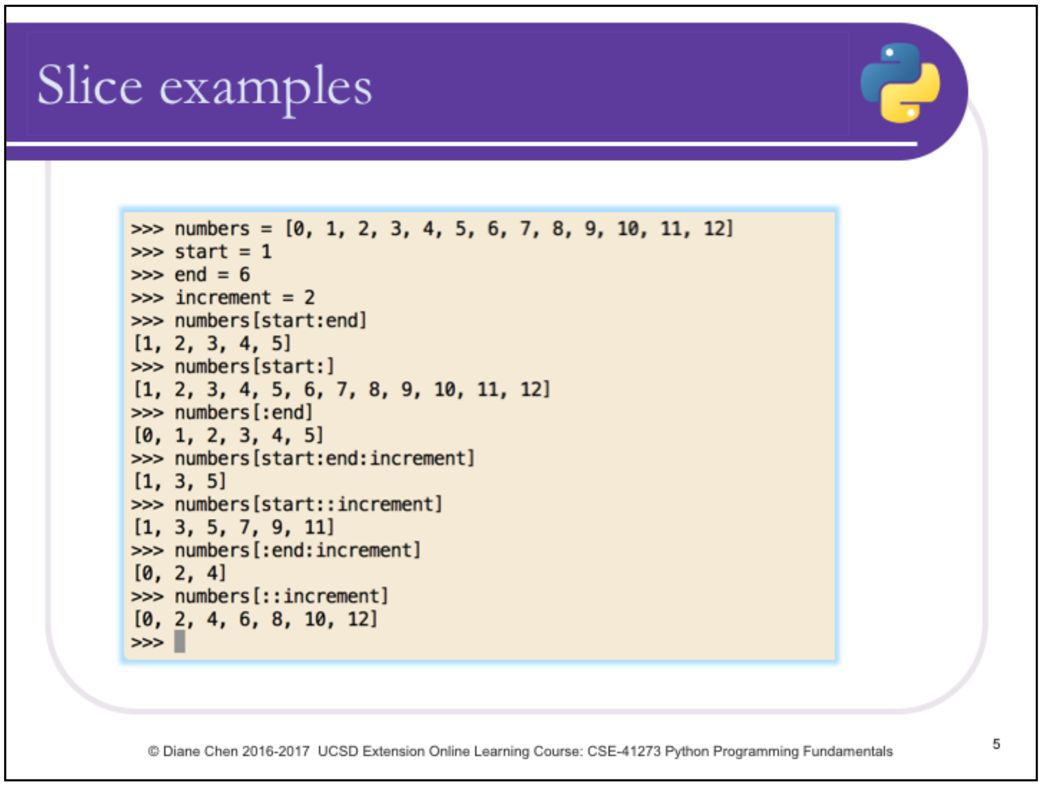
Slice indexes

- Visualize slice index as left edge of item



- A slice of [0:5] gives "GREET" and does not include the value at [5]

For slice indexing, I find it helps to visualize it as if the index is actually pointing to the left edge of the indexed location. So when the end index is 5, think of it as stopping "in between" the 4 and the 5. This works with other things like range(), where the "end" is not inclusive. Range of 10 gives numbers 0 through 9; it does not include the 10. Most things in Python are like this; where an "end" index means "up to but not including the index of end". One cool thing about slices is that no matter what values are used, it doesn't return an index error. It will return any elements that fall within the range given, otherwise it just returns an empty sequence.



Slice examples

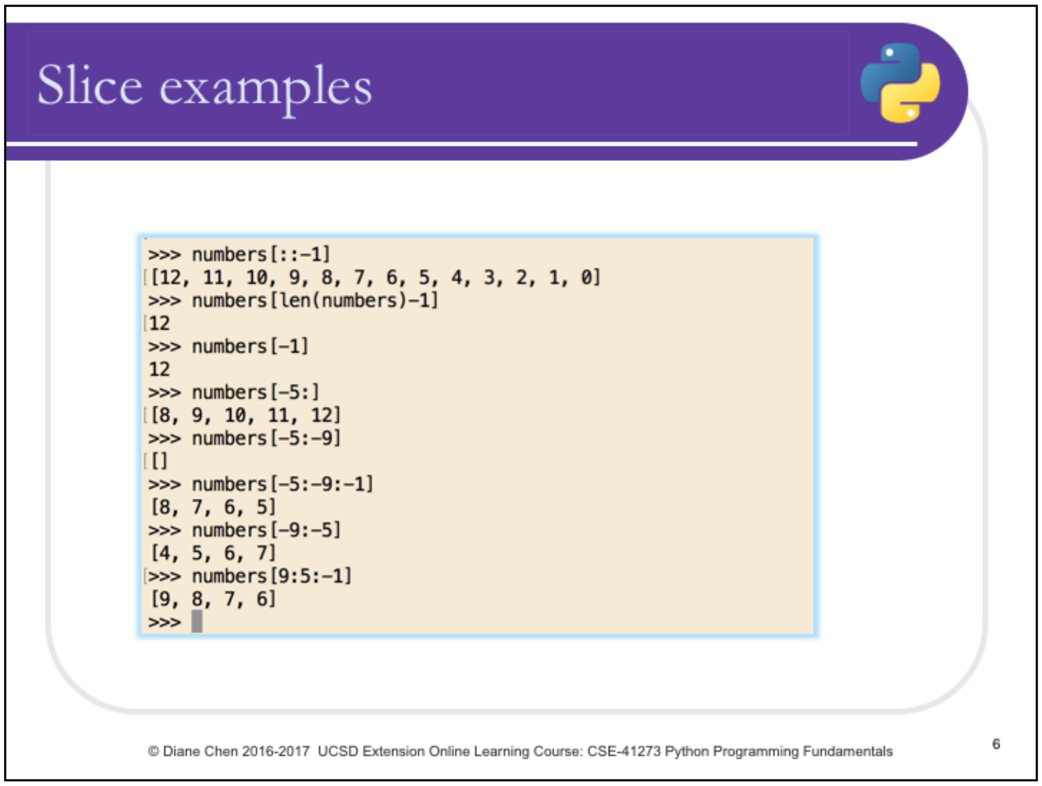
```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> start = 1
>>> end = 6
>>> increment = 2
>>> numbers[start:end]
[1, 2, 3, 4, 5]
>>> numbers[start:]
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>> numbers[:end]
[0, 1, 2, 3, 4, 5]
>>> numbers[start:end:increment]
[1, 3, 5]
>>> numbers[start::increment]
[1, 3, 5, 7, 9, 11]
>>> numbers[:end:increment]
[0, 2, 4]
>>> numbers[::increment]
[0, 2, 4, 6, 8, 10, 12]
>>> 
```

5

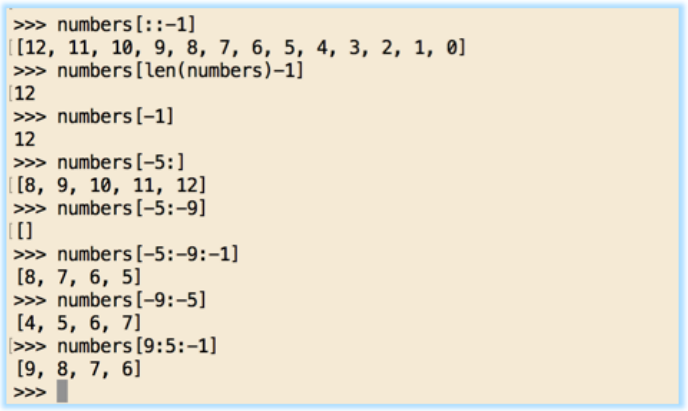
Let's see some examples in the Python REPL. If you want to follow along, don't forget to go to your class folder and activate your virtual environment before starting the Python REPL, so you get the correct version of Python.

Here we have a list of numbers from 0 through 12, so the list values correspond to their indices. A slice from 1 to 6 includes the indices 1, 2, 3, 4, 5 and does not include the number at index 6.

If you leave off the end value, it defaults to the end of the list. Similarly, if you leave off the start value, it defaults to the start of the list. We can also include an optional increment. Here are some various combinations of the start, end and increment. Take a closer look at them and make sure you understand what is going on. Play around with list slices in the REPL if it will help you to understand



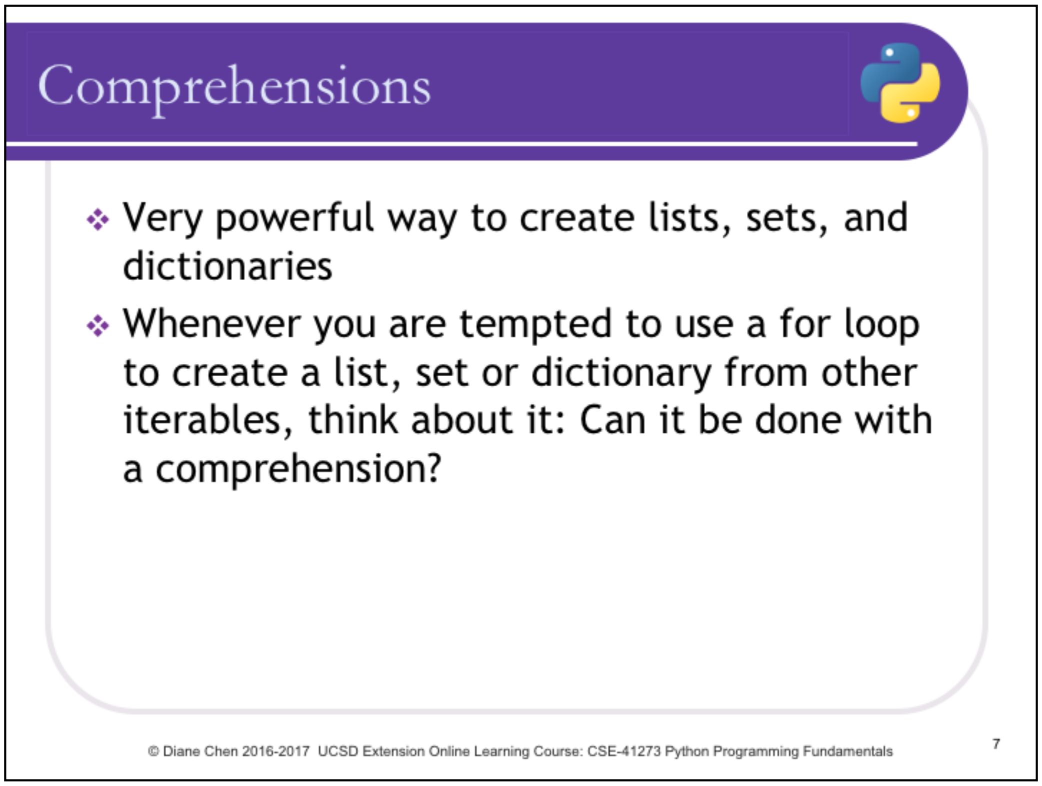
Slice examples



```
>>> numbers[::-1]
[12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
>>> numbers[len(numbers)-1]
12
>>> numbers[-1]
12
>>> numbers[-5:]
[8, 9, 10, 11, 12]
>>> numbers[-5:-9]
[]
>>> numbers[-5:-9:-1]
[8, 7, 6, 5]
>>> numbers[-9:-5]
[4, 5, 6, 7]
>>> numbers[9:5:-1]
[9, 8, 7, 6]
>>>
```

6

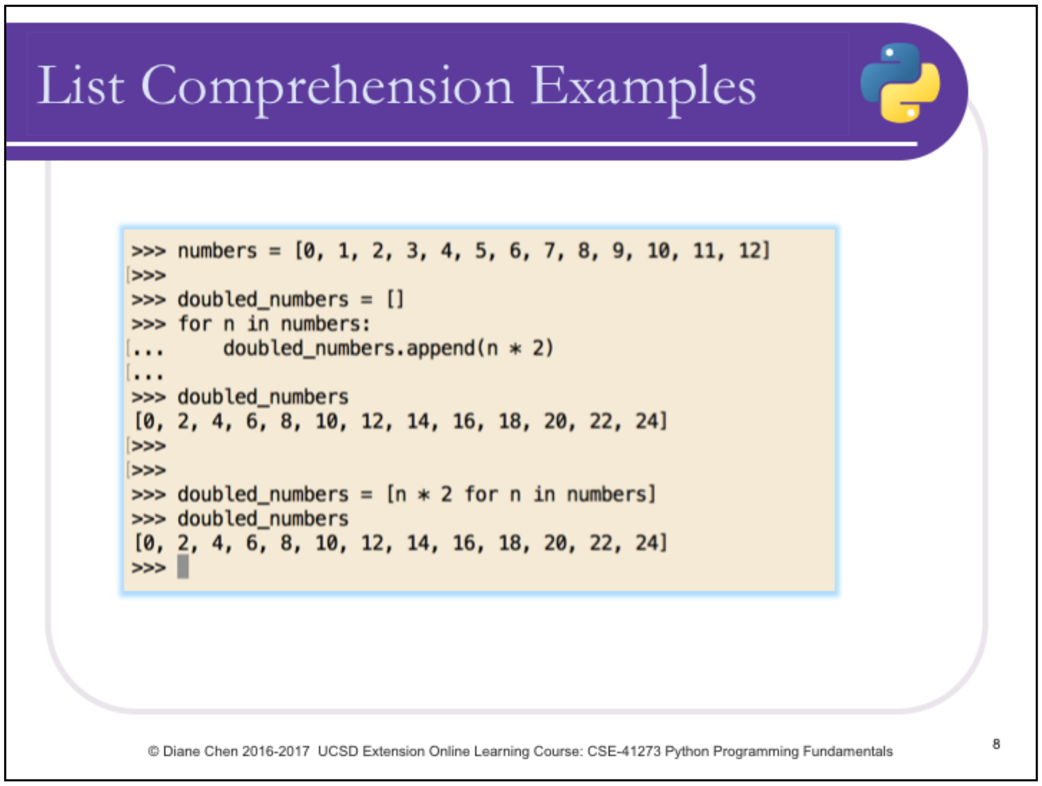
If we use minus one for the slice increment, it reverses the sequence. Remember how we get the last item index of a list or string? We can do it the ugly way, or Python has the handy negative indexing to get something relative to the end of the sequence. But it's a little tricky – you have to keep in mind that the default slice increment is positive 1 and it doesn't change the increment just because you use negative values for start and end index. If you need to, play around with slices in the Python REPL until you understand how they work.



Comprehensions

- Very powerful way to create lists, sets, and dictionaries
- Whenever you are tempted to use a for loop to create a list, set or dictionary from other iterables, think about it: Can it be done with a comprehension?

Now I'm going to talk about comprehensions. You may already have seen list comprehensions, but I want to make sure you understand them thoroughly. Comprehensions help make working with Python really fun and easy. They are very simple but powerful shorthand for creating lists, sets and dictionaries. Whenever you are using a for loop to construct a list, set or dictionary, stop and think about whether it might be possible to create it with a comprehension.



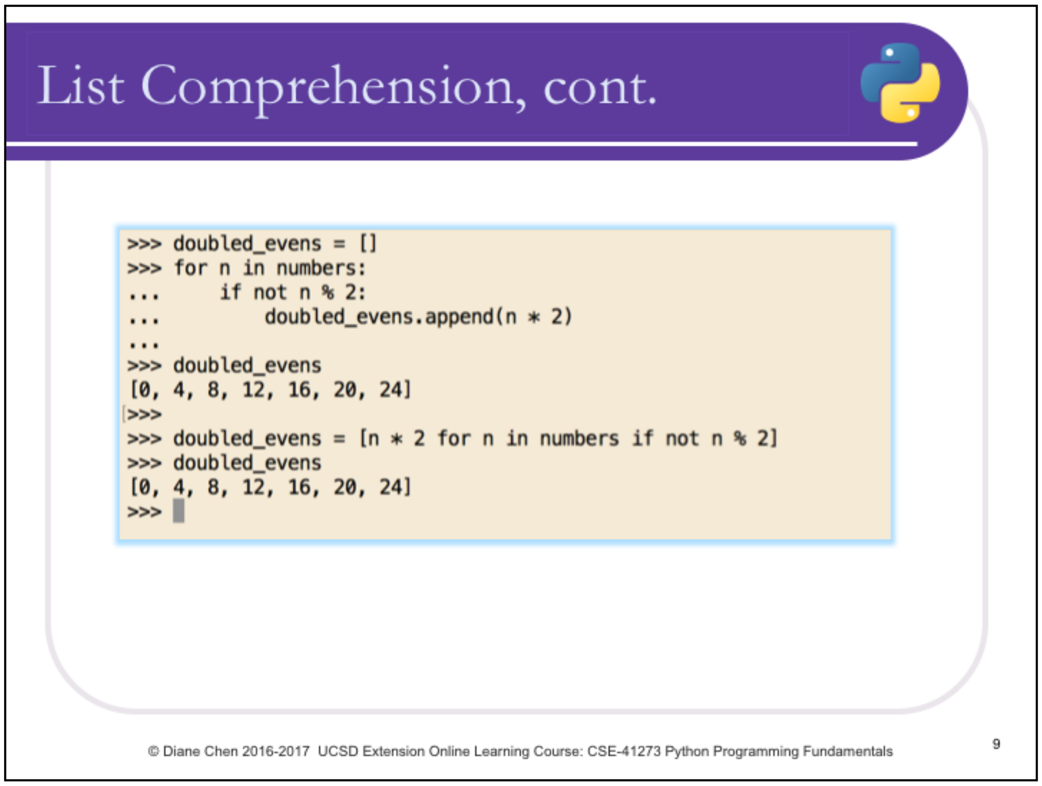
List Comprehension Examples

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
>>>
>>> doubled_numbers = []
>>> for n in numbers:
...     doubled_numbers.append(n * 2)
...
>>> doubled_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
>>>
>>>
>>> doubled_numbers = [n * 2 for n in numbers]
>>> doubled_numbers
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24]
>>>
```

Here's an example of making a list comprehension. We have our numbers list from 0 to 12.

Let's say we want to have another list of all the numbers doubled. To create it with a for loop, we first have to create an empty list, then loop through our original numbers list and append the doubled value to the new list. There's nothing wrong with it – it works just fine and everyone knows what it is doing.

… But it isn't very "Pythonic". Let's be Pythonic and make a list comprehension. Since lists use square brackets, we enclose our list comprehension in square brackets to tell Python we want a list. The first thing we put inside is the value to put into the list, n times 2. Then we define n with "for n in numbers" and close the bracket. That's it! We have our doubled_numbers list! Easy!



List Comprehension, cont.

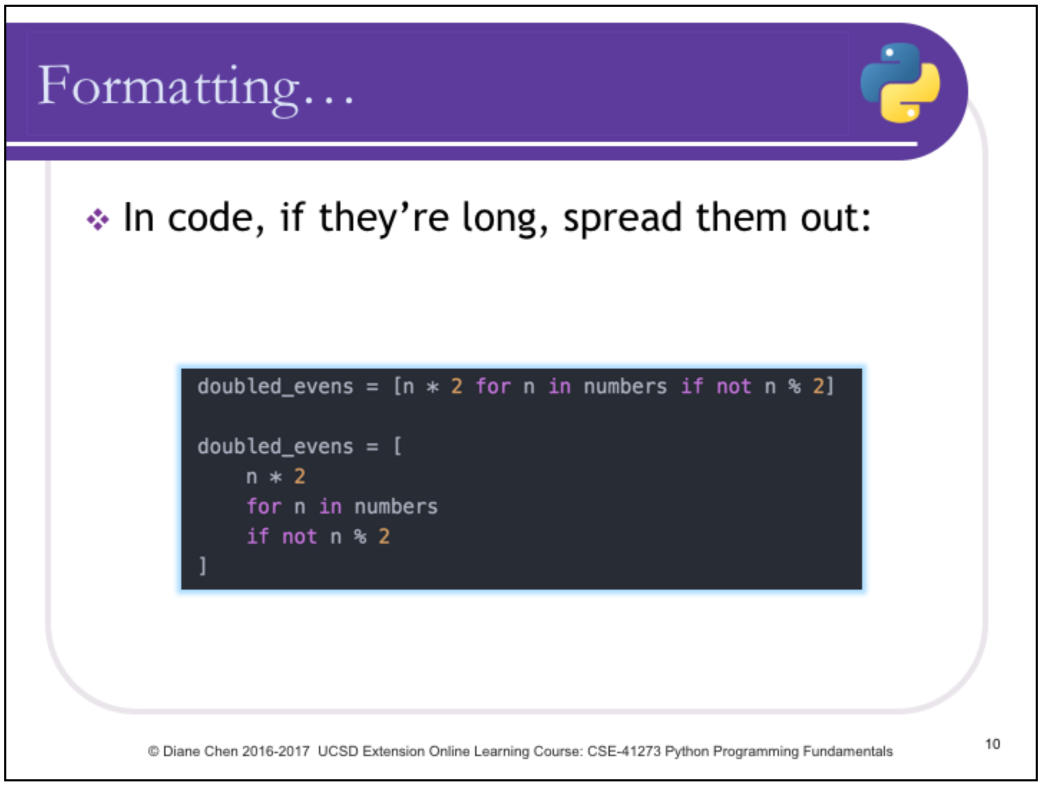
```
>>> doubled_evens = []
>>> for n in numbers:
...     if not n % 2:
...         doubled_evens.append(n * 2)
...
>>> doubled_evens
[0, 4, 8, 12, 16, 20, 24]
>>>
>>> doubled_evens = [n * 2 for n in numbers if not n % 2]
>>> doubled_evens
[0, 4, 8, 12, 16, 20, 24]
>>> 
```

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

9

What if we wanted our new list to only contain the doubles of the numbers that were even? Here we see how we would do it in a for loop with an extra if test inside the loop.

We can add the if clause to the list comprehension, by inserting it after the "for n in numbers". Now we only get the ones we want based on the expression.



Formatting…

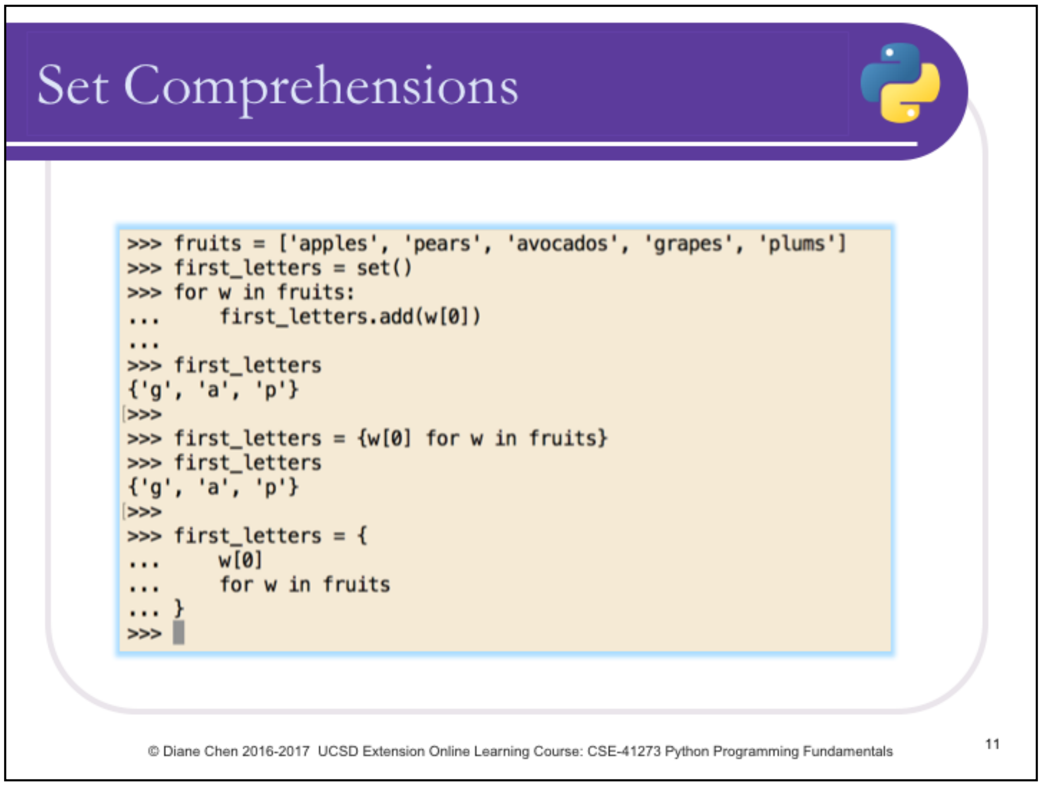
- In code, if they're long, spread them out:

```
doubled_evens = [n * 2 for n in numbers if not n % 2]

doubled_evens = [
    n * 2
    for n in numbers
    if not n % 2
]
```

10

Let me briefly talk about the layout of comprehensions. For simple comprehensions or working in the REPL, it's fine to put them all on one line. But when they're in code, I like to see them spread out as shown here. Note the alignment of the brackets and the indentation of the 3 lines of the comprehension. The first line contains the value to place in the list, the next line is the "for" clause, and the third line is the "if" expression. This layout makes it immediately obvious what is happening. I'm sure I'm repeating myself here when I say that one of the goals of Python is to make code easier for humans to read and understand it. These kinds of shorthand or "syntactic sugar" like list comprehensions help improve readability of the code. We'll cover coding style a little bit later in the lesson. Please make a note of this format as I will be expecting your homework to follow this.

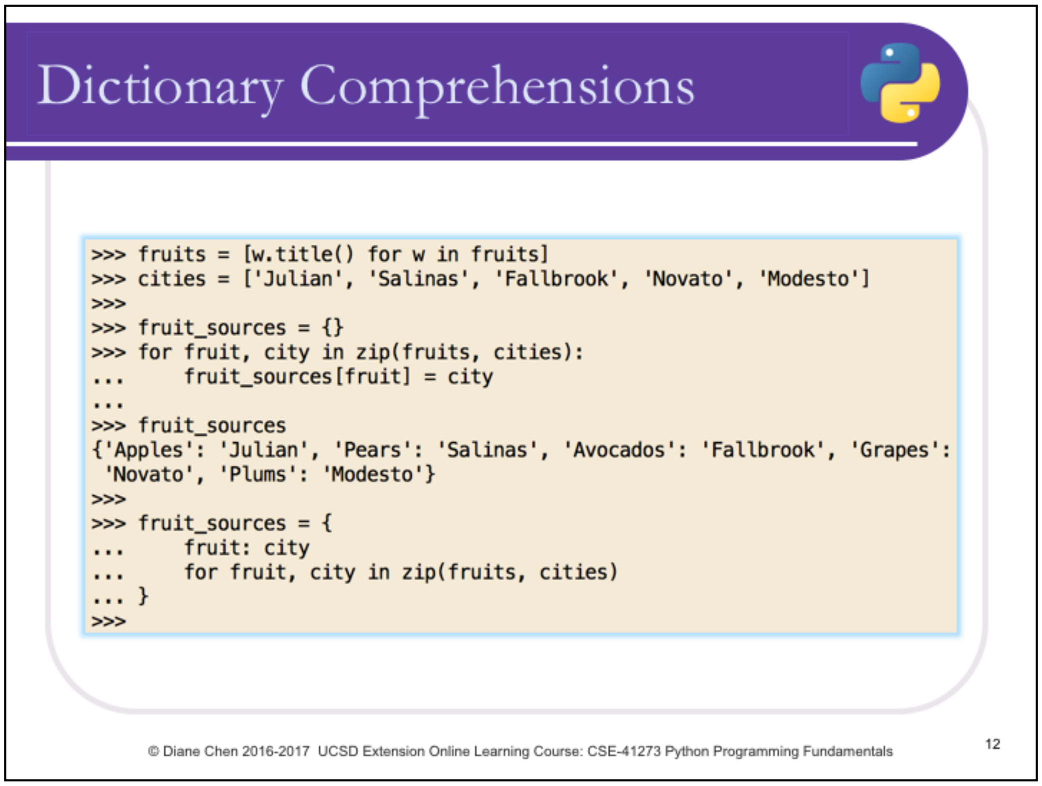


Set Comprehensions

```
>>> fruits = ['apples', 'pears', 'avocados', 'grapes', 'plums']
>>> first_letters = set()
>>> for w in fruits:
...     first_letters.add(w[0])
...
>>> first_letters
{'g', 'a', 'p'}
>>>
>>> first_letters = {w[0] for w in fruits}
>>> first_letters
{'g', 'a', 'p'}
>>>
>>> first_letters = {
...     w[0]
...     for w in fruits
... }
>>>
```

Now let's talk about set comprehensions. They are similar to list comprehensions except we will be creating a set instead of a list. Say we have a list of fruits: apples, pears, avocados, grapes and plums. We want to know how many first letters there are. Of course this is a silly contrived example, but it is a good illustration of creating sets, because sets do not allow duplicates.

To make the set with a for loop, we start by initializing the empty set. Looping over the list for fruits, we add the first letter of each fruit to the set called first_letters. And we find that the set contains a, g and p. Note how the system prints out the set looking like a list but with curly brackets instead of the square brackets. We make a set comprehension just like a list comprehension, except with curly brackets. This is how the system knows we want to make a set comprehension instead of a list comprehension. And just like a list comprehension, we can spread it out over multiple lines to make it more readable.

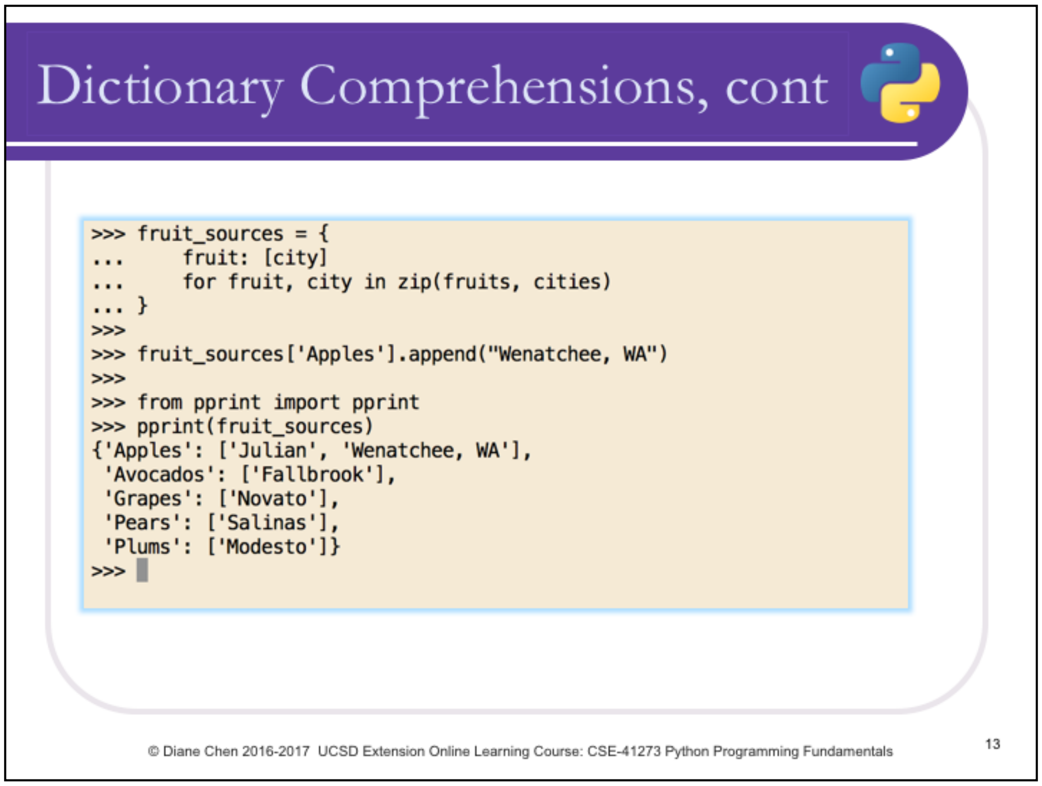

Dictionary Comprehensions

```
>>> fruits = [w.title() for w in fruits]
>>> cities = ['Julian', 'Salinas', 'Fallbrook', 'Novato', 'Modesto']
>>>
>>> fruit_sources = {}
>>> for fruit, city in zip(fruits, cities):
...     fruit_sources[fruit] = city
...
>>> fruit_sources
{'Apples': 'Julian', 'Pears': 'Salinas', 'Avocados': 'Fallbrook', 'Grapes':
 'Novato', 'Plums': 'Modesto'}
>>>
>>> fruit_sources = {
...     fruit: city
...     for fruit, city in zip(fruits, cities)
... }
>>>
```

Now we're going to make a dictionary comprehension. We have our list of fruits and, just for fun and to make it look nicer, let's capitalize our list of fruits. We make a new fruits list with a list comprehension, using "w.title() for w in fruits".

Title is a method on strings that capitalizes each word in the string. In this case, all the fruits are just one word so we could have used the capitalize method, but if we had any fruits of more than one word, capitalize would only do the first word. But I digress. Let's get back to the dictionary comprehension.

We also have a corresponding list of cities where the fruit comes from. We want to make a dictionary with the fruits as keys and the city where the fruits come from as the values. So to do it with a for loop, we first create an empty dictionary. This is a perfect example of when the zip function comes in handy. Zip() returns tuples of items, one from each input iterable. We could mess with indexing or use the enumerate function, but by using zip along with tuple unpacking, we get the corresponding fruit and city for each iteration of the loop. We loop over the fruit and city, and set the value of the dictionary fruit_sources for the key of fruit to be the city. So let's be Pythonic again and make a dictionary comprehension. We use curly brackets, just like a set. Python can tell from our context that we are creating a dictionary and not a set because we give a key/value pair using the colon to indicate it. We get the corresponding fruit and city from each list from the zip function. And we're done! We have our dictionary.



Dictionary Comprehensions, cont

```
>>> fruit_sources = {
...     fruit: [city]
...     for fruit, city in zip(fruits, cities)
... }
>>>
>>> fruit_sources['Apples'].append("Wenatchee, WA")
>>>
>>> from pprint import pprint
>>> pprint(fruit_sources)
{'Apples': ['Julian', 'Wenatchee, WA'],
 'Avocados': ['Fallbrook'],
 'Grapes': ['Novato'],
 'Pears': ['Salinas'],
 'Plums': ['Modesto']}
>>> 
```

13

We've tried to source our fruits from areas not too far from southern California, but Julian just can't provide enough apples for our needs. So let's change our dictionary to store a list of cities, rather than just one city per fruit. That way, we will know where to go for apples when Julian runs out of apples for us.

We make our dictionary comprehension almost the same as before, but we make the city be in a list, instead of just the single city as the value of the key/value pair. Even though it only contains one item when we first create it, because it is a list, we can modify it easily. So after we are done creating our fruit_sources dictionary, we are able to append the apple capital of the world, Wenatchee, Washington, to our list of cities where apples come from.

And just in case you haven't seen it before, I've shown "pretty print", a function pprint, in a module named pprint. If we use it instead of print, it prints out our dictionary nicely formatted.