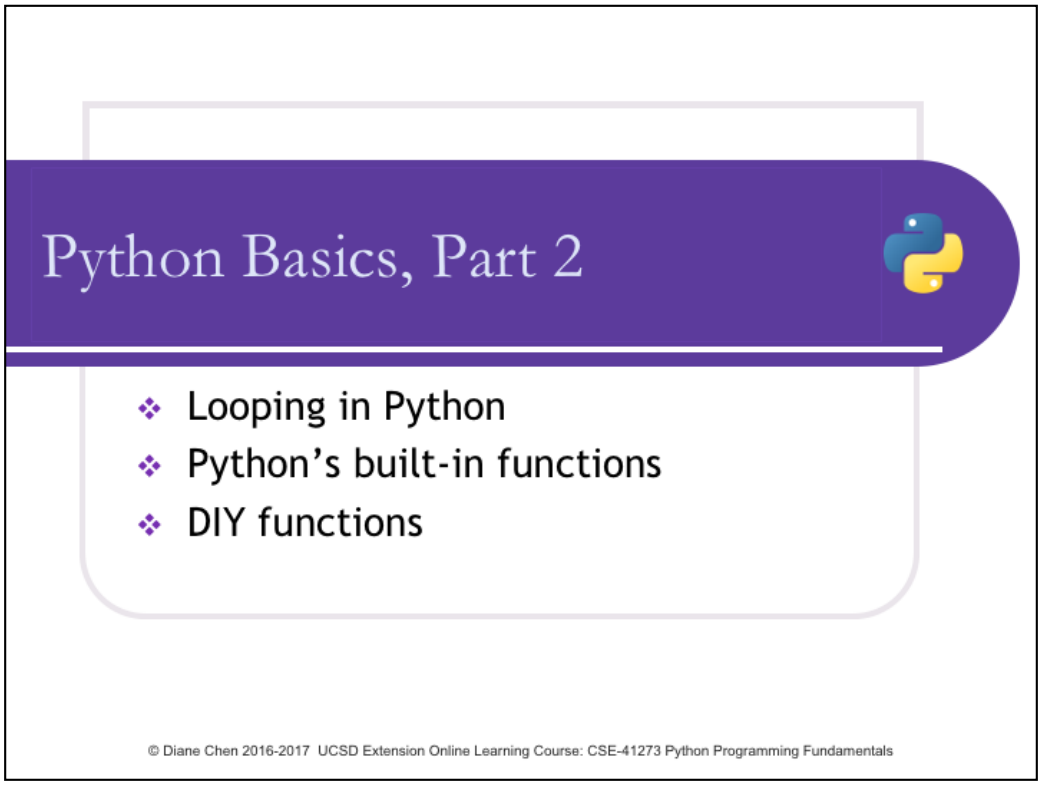
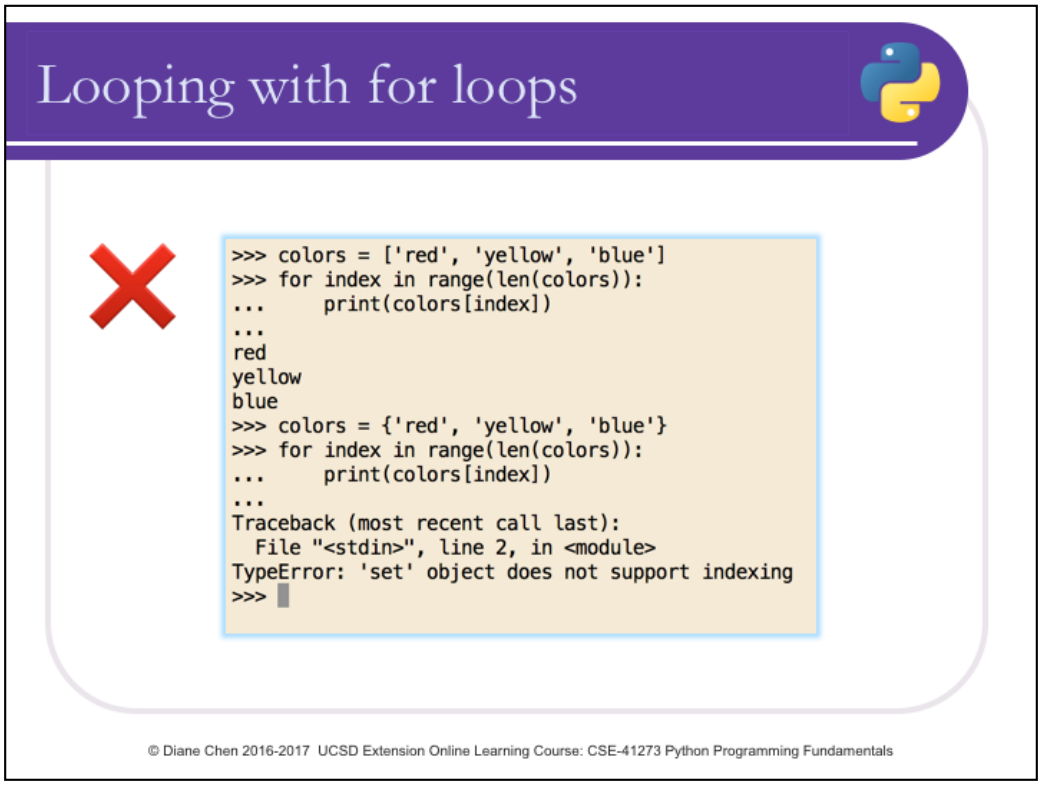


In this lecture, I will talk about looping in Python, some of Python's built-in functions, and a little bit about making your own functions.

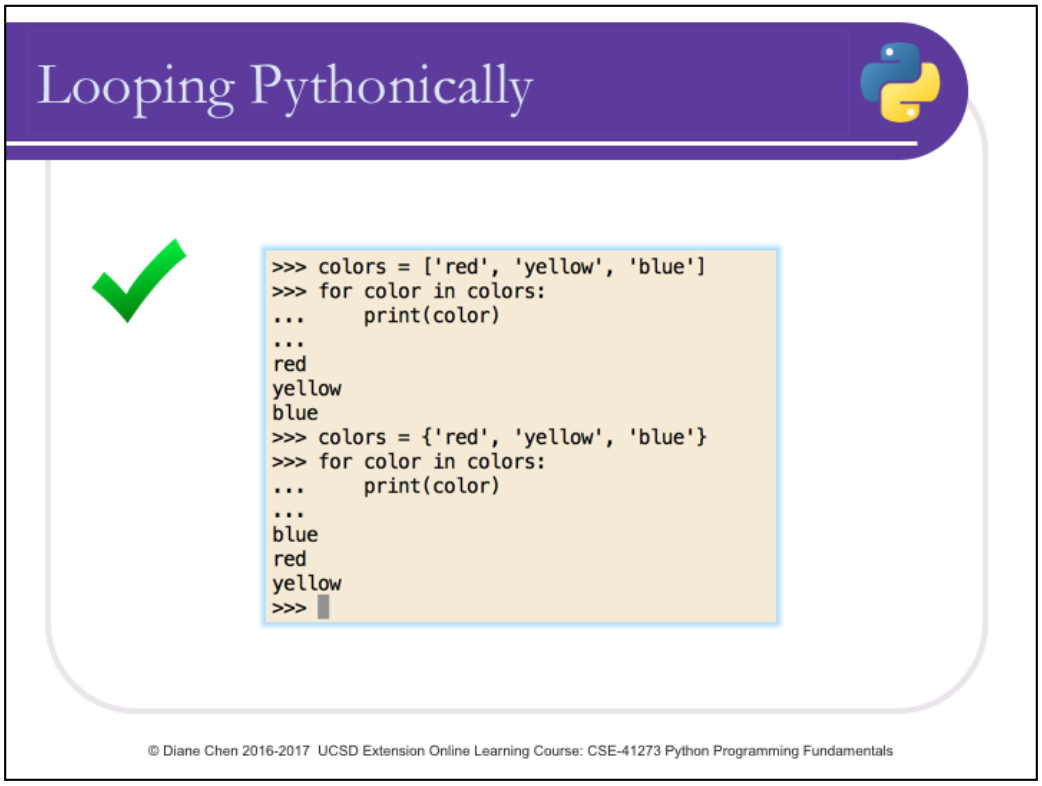


Python's for loops are different from for loops in say, Java, or C/C++. In fact, python's for loops are more like for each loops. We might be tempted to do our for loops like other languages with indexes, using Python's range() function that returns sequential integers.

Here we have a list of colors. If we want to print them out, we can loop over them sequentially and print them by index in the loop. But this doesn't work for sets, as they are not indexable. Similarly, dictionaries are not indexable.

Plus it isn't very Pythonic! The code is cluttered with excess bookkeeping stuff!

Let's see how we can improve that using Python's for loops the way they were intended!



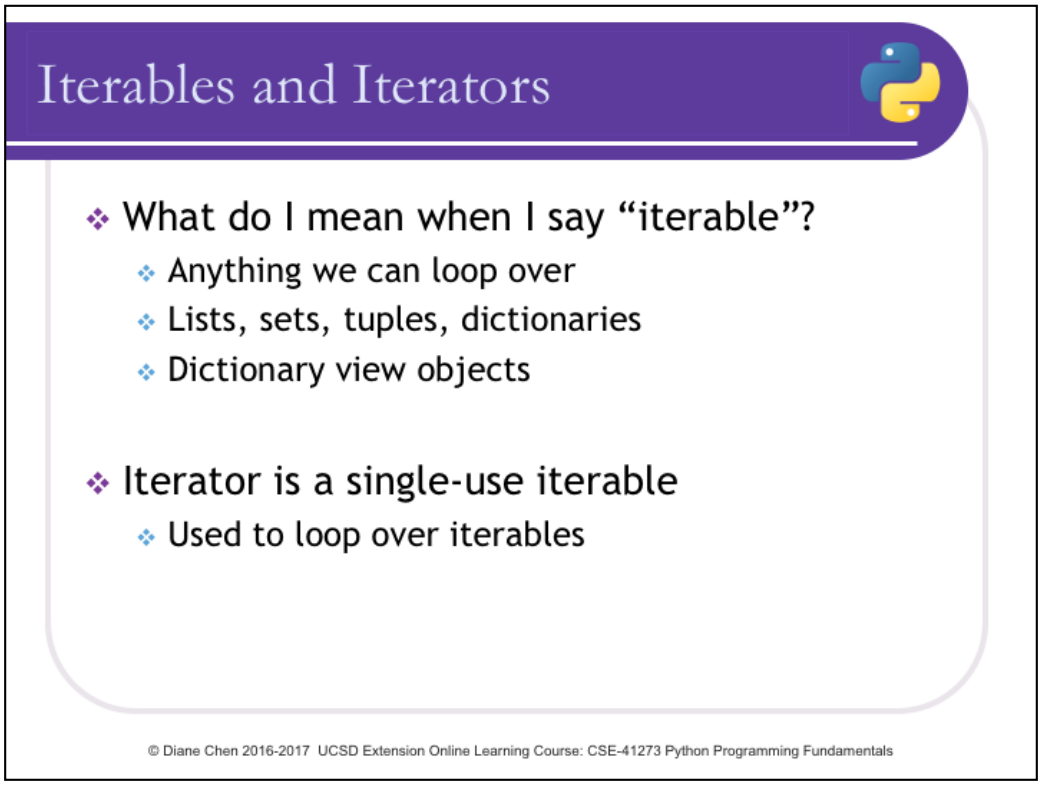
We generally don't need to use any indeximg when looping in Python. If there is a collection of things, for example our list of colors, then we can loop over it with the statement "for color in colors", and we get each element as we loop over the list. We can loop over any collection type.

Here we see that we can successfully loop over a set of colors just as if it were a list. We can't predict the order; it depends on the collection type. Sets and dictionaries are unordered collections. We can loop over them, and we will get all the elements from them in our loop, but we can't predict the order that we will get them in.

Here I've put the same colors in a set, and the looping works fine. When we print them out, they come out in a different order from the order we created them, but we still get all of the elements in the set.

Please, in your code, always use this form of looping in Python. I will take points off if I find your code looping using range of len of something!

If you really need the index, use the built-in function enumerate, which we will see when I talk about built-in functions.

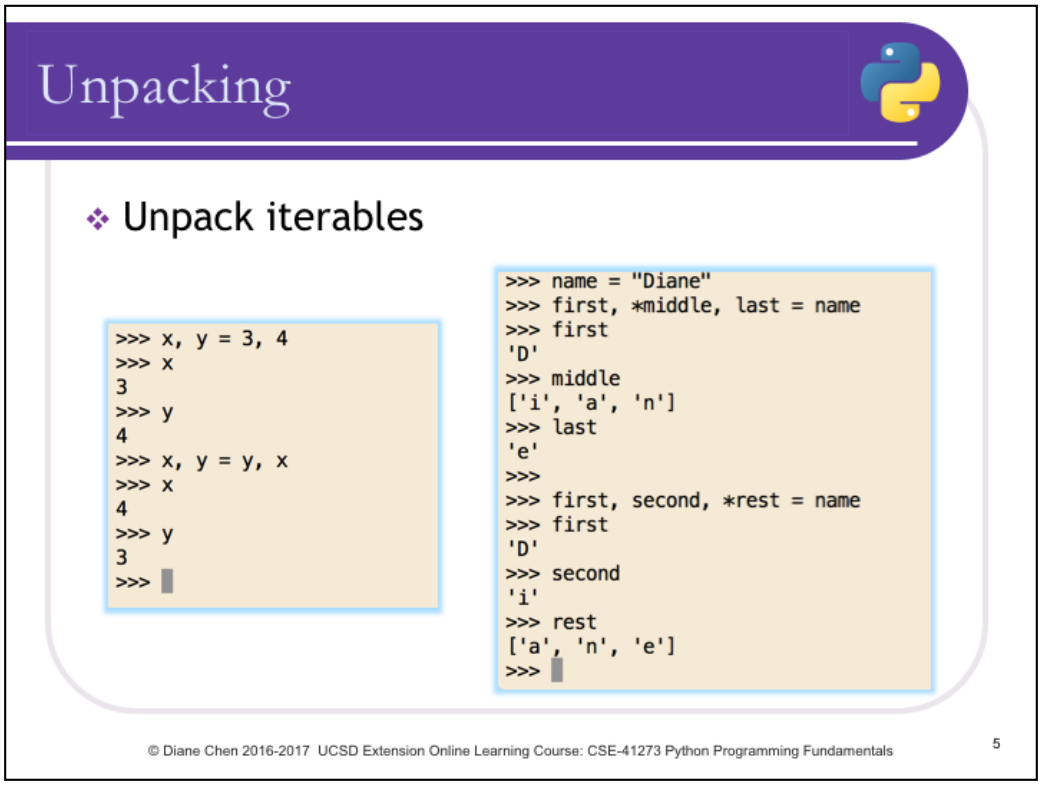


You will often hear me use the terms iterable and iterator. What do I mean by these terms? What is an iterable? What is an iterator?

An iterable is anything we can loop over. For example, lists, sets, tuples and dictionaries are all iterables, as are the dictionary view objects we saw in the previous lecture.

Iterators are a little harder to explain. Basically they are single-use iterables that we use to loop over the iterable from which the iterator was created. We will cover iterables and iterators in more detail in future lessons.

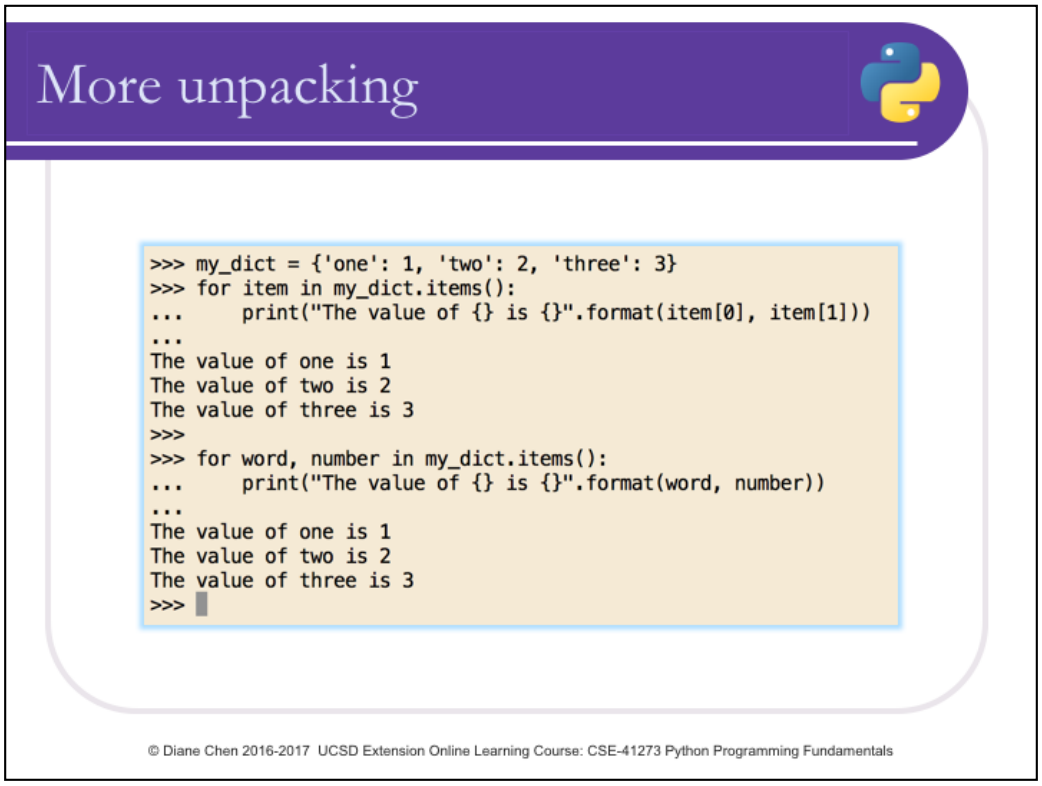
You will find as we go along that many things in Python are iterables or iterators. One of Python's strengths as a programming language is the power of the iteration capabilities.



Now I want to talk a bit about unpacking. Python has the capability of unpacking iterables into other variables. We can do simple assignments like the first line in the example on the left. Here we are taking advantage of "tuple unpacking", as the right side "3, 4" creates a tuple. Then the tuple is unpacked by the assignment into x and y, so x becomes 3 and y becomes 4. Since everything on the right-hand side of an assignment is evaluated before the left-hand side assignments are made, we can also do variable swapping, where I set x comma y to be the result of y comma x. What is happening here? The tuple created by "y, x" on the right-hand side of the assignment, is unpacked into x and y, so now x is 4 and y is 3.

In the right example, using the “star unpacking”, we can unpack an iterable into various “positional” variables, with the remaining ones going into the starred variable as a list. This works similarly to argument unpacking in functions, which we will see more of in a few minutes.

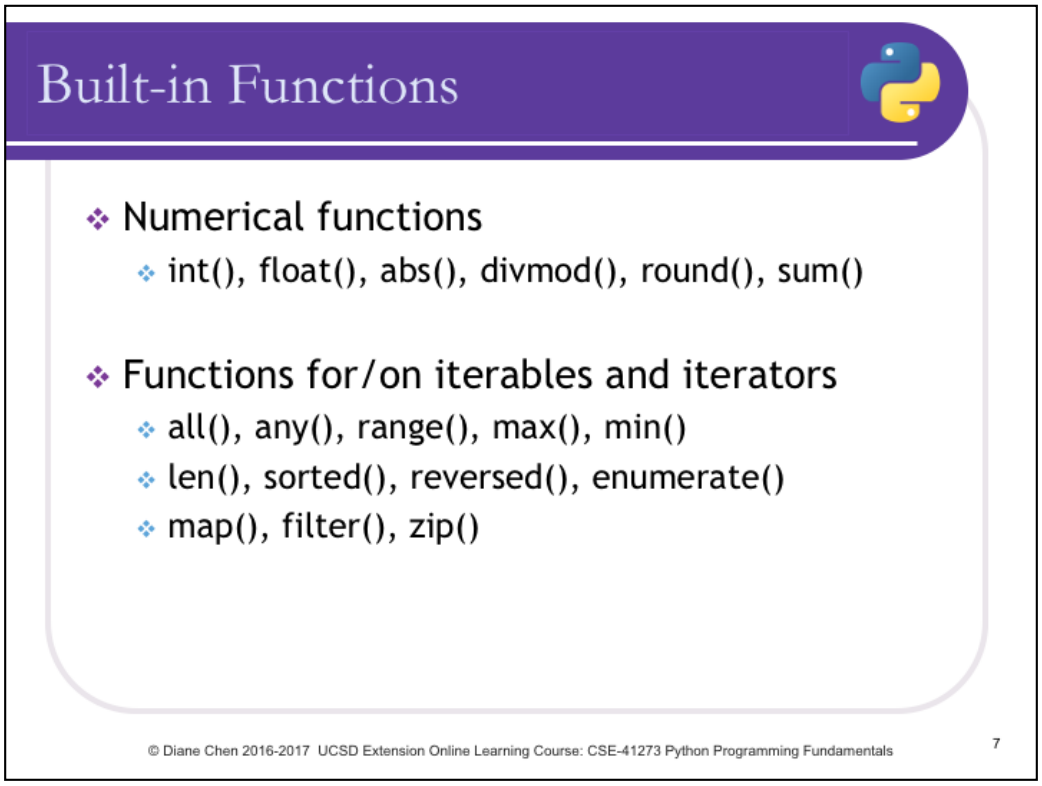
There must be enough elements in the iterable to fill all the positional variables given. The starred variable can be an empty list if there are no more elements left. An error occurs if there are not enough elements in the iterable to fill all the unstarred variables.



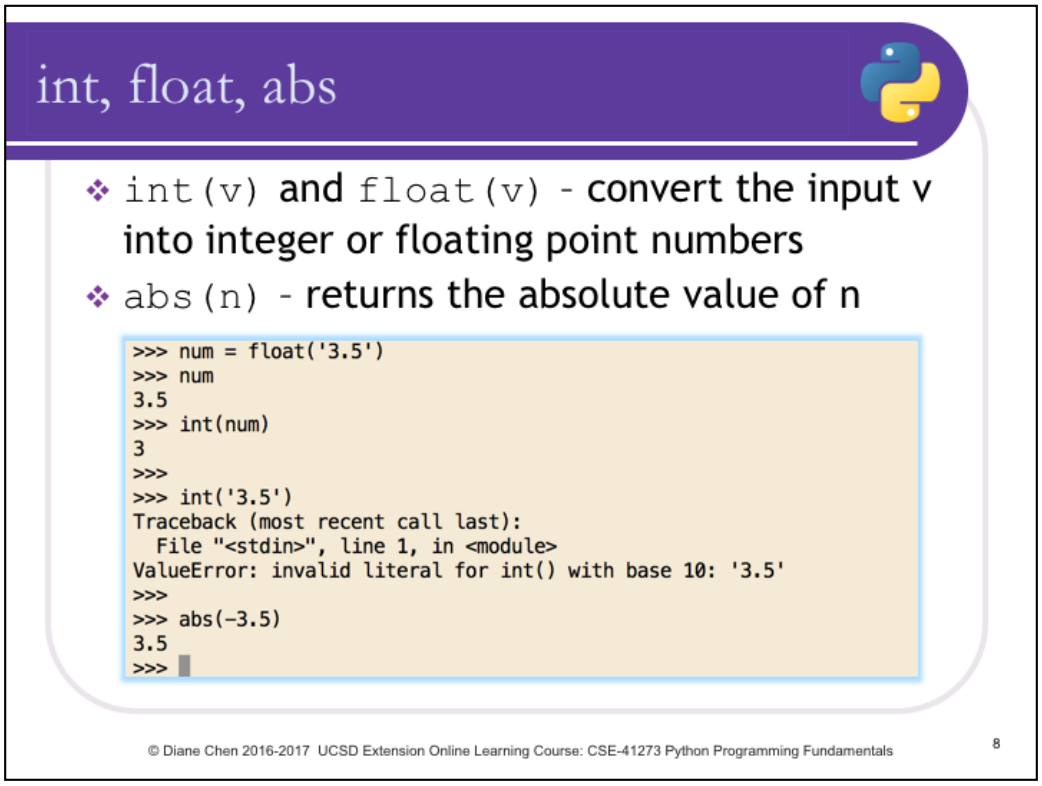
One common usage of unpacking is when we have something like the dictionary views we saw in the previous lecture. Here we have our dictionary with words and the numbers that correspond to the words. The dictionary method items() returns tuples containing the key, value pairs. In our first for loop, we loop over the tuples from the items() method. When we extract the information from the tuple for printing, we index the first and second element of the tuple.

Now look at the second loop. Here, in the ”for” statement, we unpack the tuple into variables called word and number. This makes the print statement much cleaner and easier to understand, because we have used meaningful variable names for the keys and values of the dictionary item.

This is a very simple, trivial example, but you can imagine how much more readable would be code that uses tuple unpacking like this when there is a lot more complicated stuff happening in the loop. Of course, you could have two lines at the beginning of the loop where we assign the variables “word” and “number” to the first and second tuple element, but again, that just adds clutter. If I haven’t already said it, I want to emphasize that one of the goals of Python is to provide a way to make readable code. After all, the compiler/interpreters don’t care about how the code looks or what the variables are named, it just does its job with the code it’s given. However, when code is read by people, clean code and meaningful variables names (among other things) go a long way to making the code more understandable.

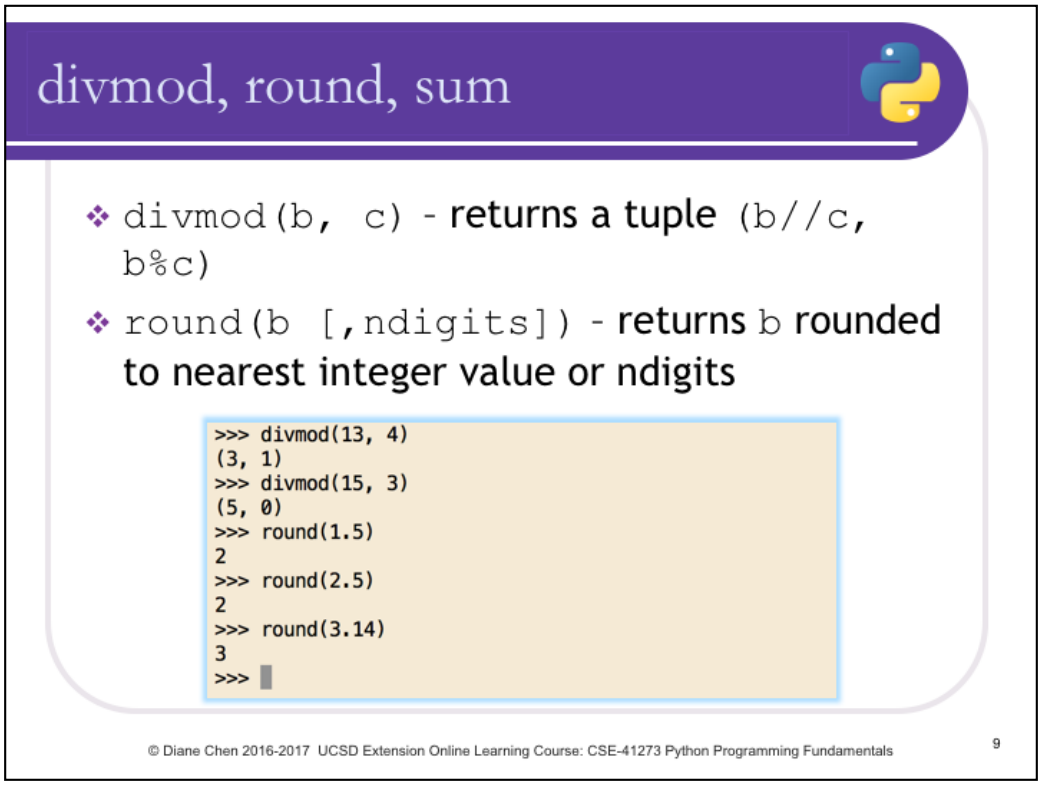


Let's review some of the built-in functions of Python. I'm just going to talk about 2 groups of functions; there are many other built-in functions, of course. We have numerical functions that operate on numerical variables. Then we have the functions that act on iterables and iterators or return them.



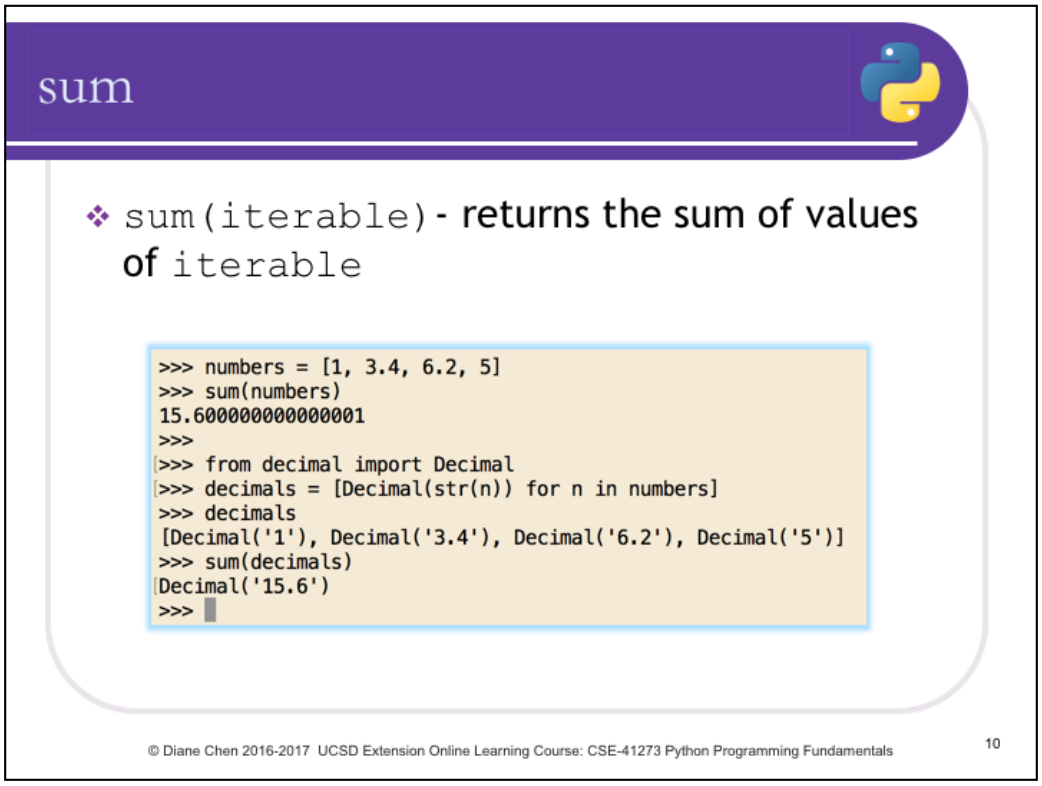
For numerical functions, Int() and float() convert their input to integer and float variables, respectively. We can convert float to int or vice versa. They also convert strings to int or float, so if you have a string representation of a number they will convert it.

The function Abs() returns the absolute value of the number.

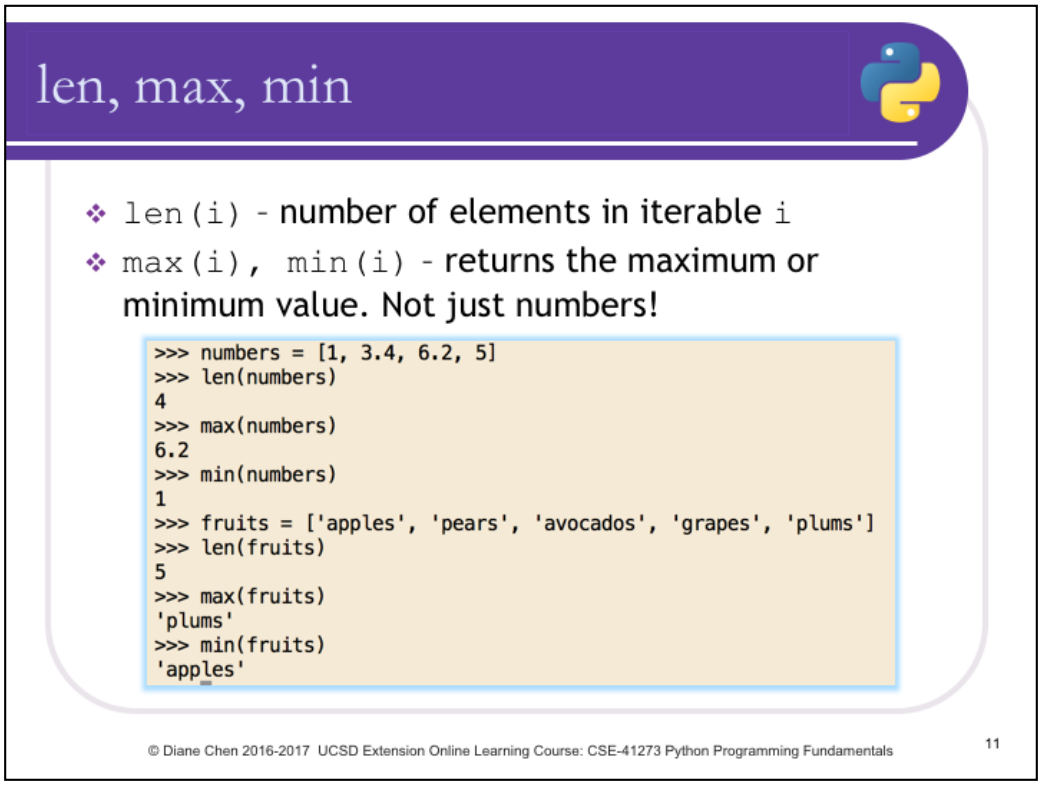


Divmod is a useful function, returning a tuple of the quotient and remainder using integer division.

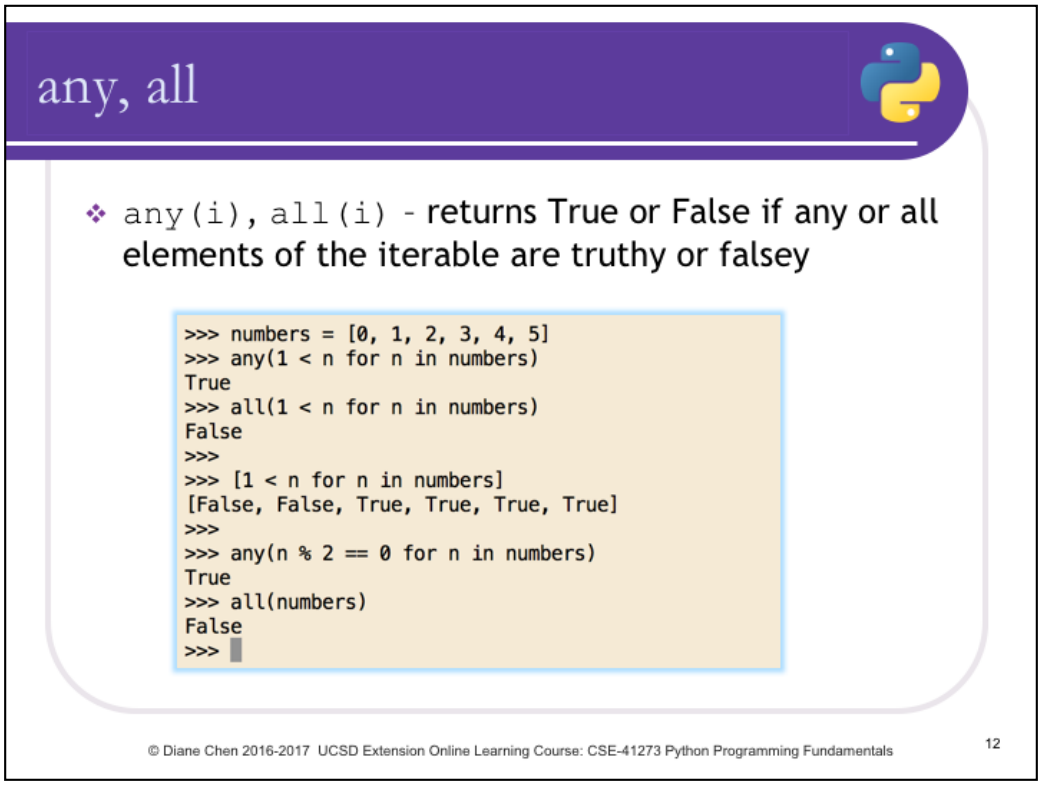
Round() returns the floating point number rounded to ndigits after the decimal. If ndigits is omitted, it returns the rounded integer. Note that round() uses what some call "banker's rounding", that is, if the value is equally close to 2 numbers, the rounding will favor the even answer.



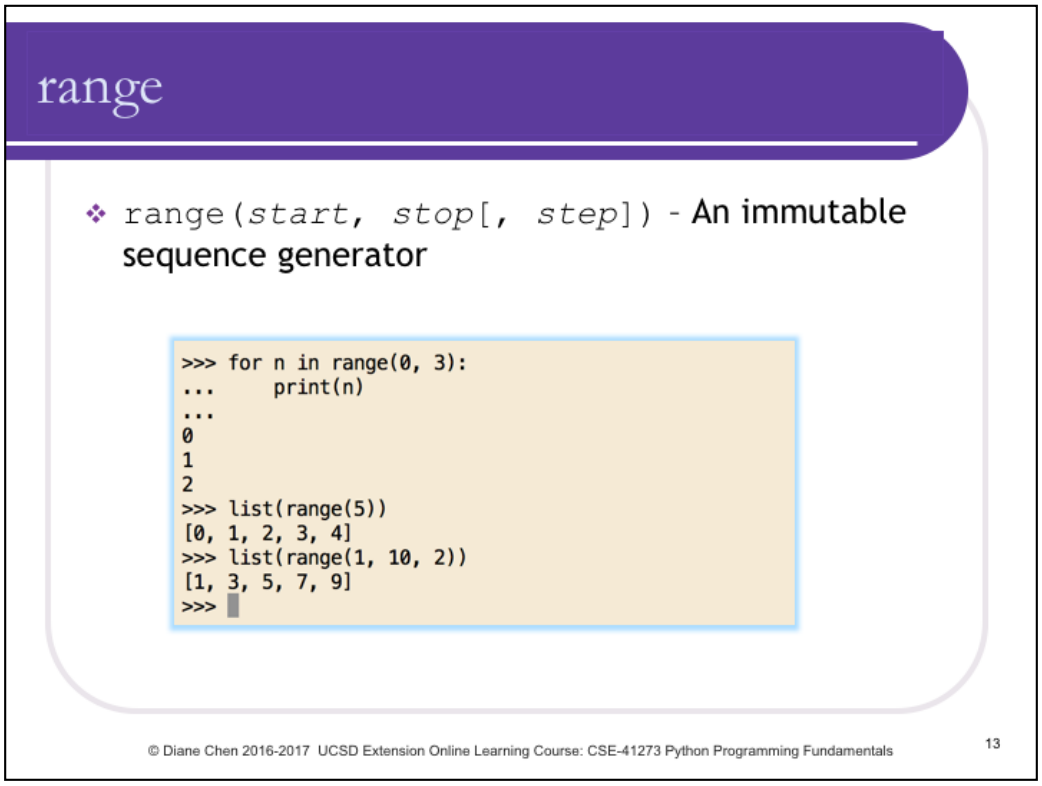
The sum() function returns the sum of the values in the input iterable. But what's going on here with the sum of the numbers? This "error" is due to the fact that the physical representation of a floating point number - that is, the actual bits used to represent a non-integer, cannot always represent the number exactly. When some arithmetic is performed on the numbers, sometimes there is a little bit of error that affects the result. So I'm going to digress a little bit here and point out that one of the Python standard library modules is a "decimal" library, where the numbers are guaranteed not to have any "floating point" error. As shown here, you can get the exact Decimal value of a number string. I could have created the decimals list by calling Decimal on just the n for n in numbers . BUT, the values would not be clean, as they are here, because Decimal would take the floating point "error" as part of the number. It doesn't know that the error is actually error, rather than the actual value of the number. You can read more about the decimal module in the Python Docs.



Now for functions on and for iterables and iterators. You've probably seen most of these before. len() returns the size of the iterable. max() and min() return the maximum and minimum value from the iterable, respectively. Note that these are not in with the numerical list of functions, as they work on any object that have comparisons defined.

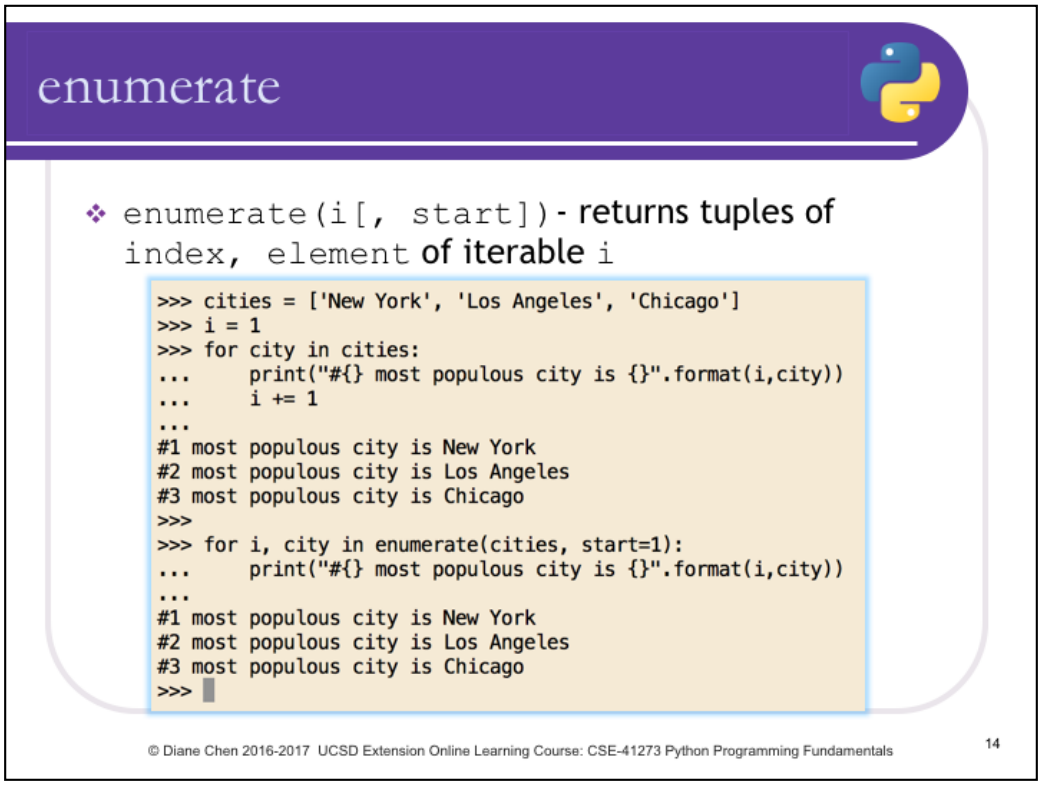


The functions any() and all() return boolean True or False if any or all of the elements of the iterable are truthy. In the first 2 examples, we input what looks like a list comprehension. The key is the comparison that returns a boolean; I've shown what the result is for a list comprehension like what we input to the any and all functions. We can have any comparison expression; here is an example where it asks if any of the numbers is even. Numbers have a truthiness value, namely that the number value of zero is considered falsey and any other value is truthy, so we can input the list of numbers into the all() function to see if they are all non-zero. In this case the answer is False because one of the elements of the list is zero. That brings up another point about truthiness of lists vs. the list elements. A list is truthy if it contains any elements, no matter what they are, and falsey only if it is an empty list. In the case of these functions any and all, they iterate over the elements of the input iterable, rather than check the truthiness of the iterable itself.



Python's range function is an immutable sequence generator. Sounds pretty pretentious, huh? You can think of it as being like another collection type. It allows us to get the integer numbers between start and stop. Note that the numbers contain the start value and go up to but not including the stop value. This is very common in Python; when there is some kind of list of start and stop values, the start value is included, but the stop value is excluded.

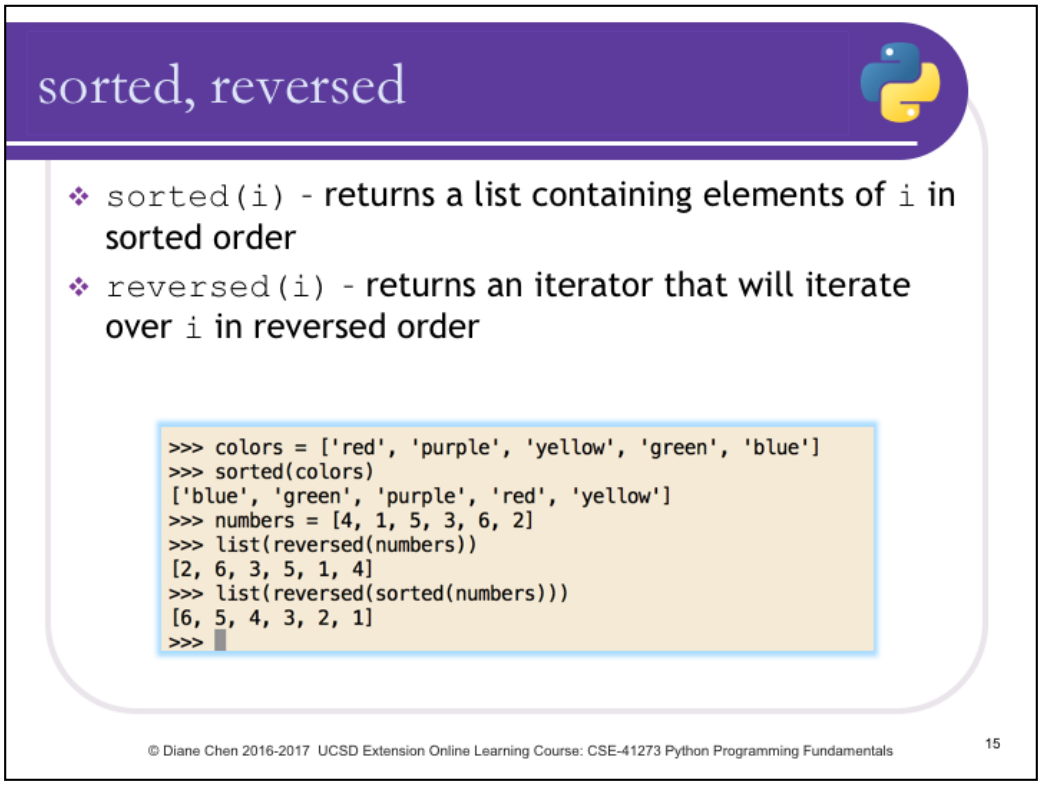
If the start argument is 0, we can leave it off. It also has an optional third argument as a step. Because it is a generator (we will talk about those later in more detail), in order to see the results, I've wrapped it into the list constructor. That also saves space on the slide instead of printing each one out in a for loop.



The function enumerate is an iterator that returns tuples of an index and corresponding element of the input iterable.

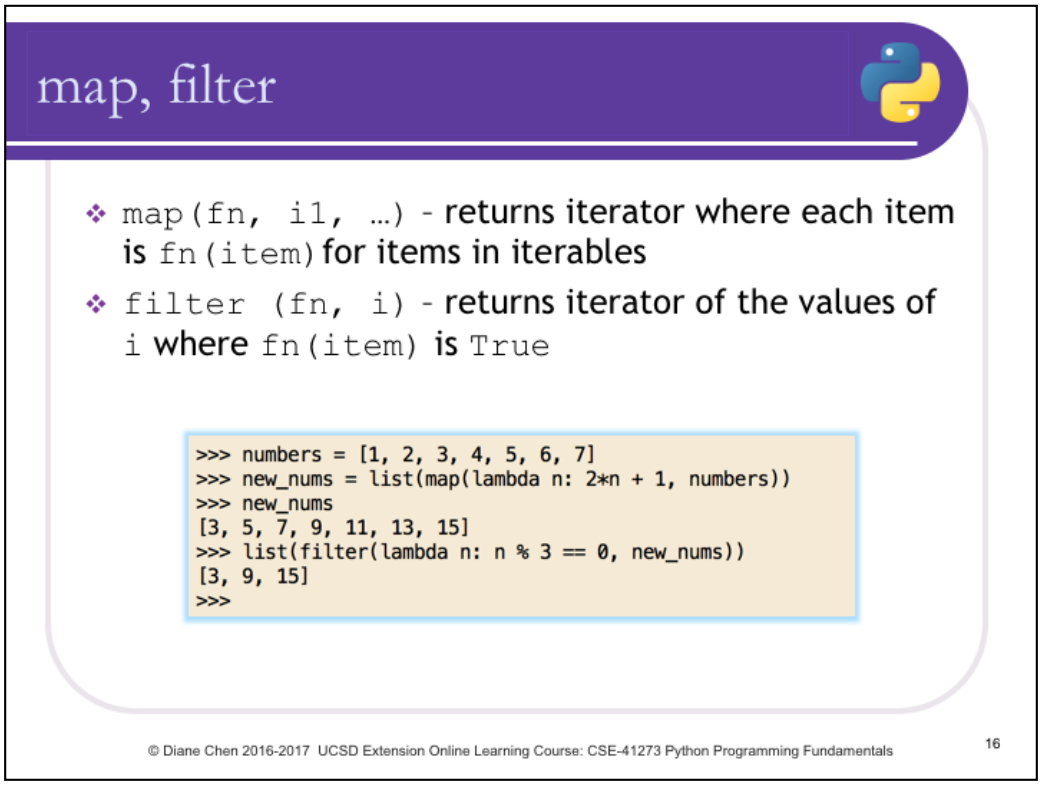
First I show what you would have to do if we wanted to print a list with an index and we didn't have the enumerate function. We would need to use the index (which we saw before doesn't even work on all iterable types), and manage the index ourselves.

Enumerate has an optional start parameter that defaults to zero. Here I show using enumerate with a start value of one, so we can number our list items. Note we are using tuple unpacking to get the index and value from the tuple that enumerate returns.

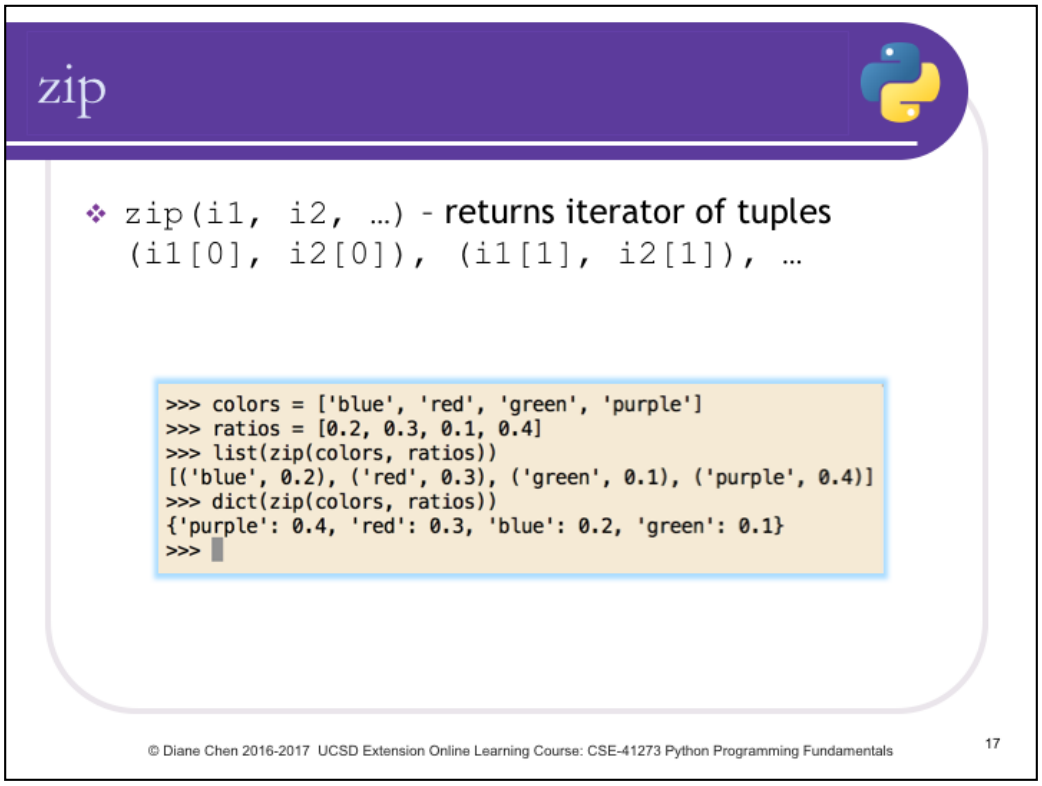


The function sorted() returns a list containing the elements of the input iterable in sorted order.

The reversed() function returns an iterator that iterates over the input iterable in reversed order. Because the reversed function returns an iterator, I wrap it in the list constructor so we can see the result.



map() takes a function and list of iterables as input and returns an iterator that yields the value of the function applied to each element of the iterators. The filter() function takes a function and iterable as input and returns an iterator containing the elements of the iterator where the function applied to the element is true. These are not commonly used anymore as they can be replaced with generator expressions and/or list comprehensions that are much more readable. In Python 2, these returned lists; in Python 3, they were changed to return iterators because it is more efficient. Note the use of lambda expressions, which I mainly did to save space on the slide. I could have used named functions and used them in place of the lambda expression.

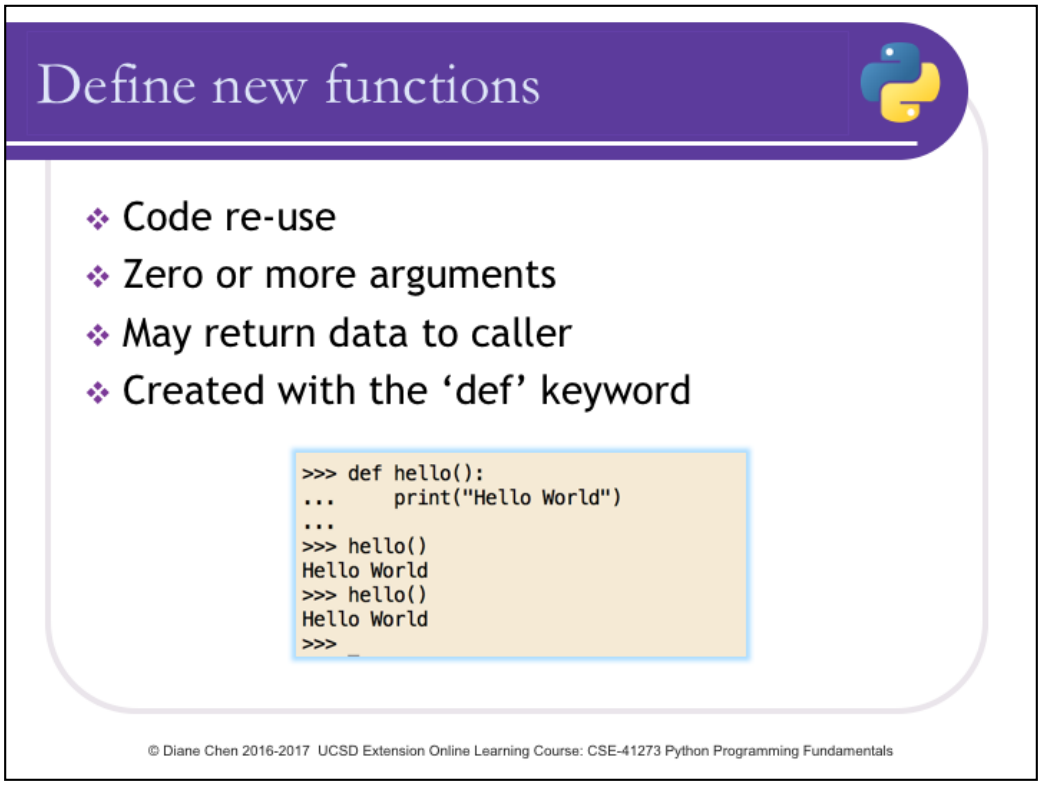


The zip function returns an iterator that yields tuples of corresponding elements of each of the input iterables. It stops when any of the iterables is exhausted.

One of the ways to create a dictionary is to give the dictionary constructor an iterable consisting of tuples of the key, value pairs.

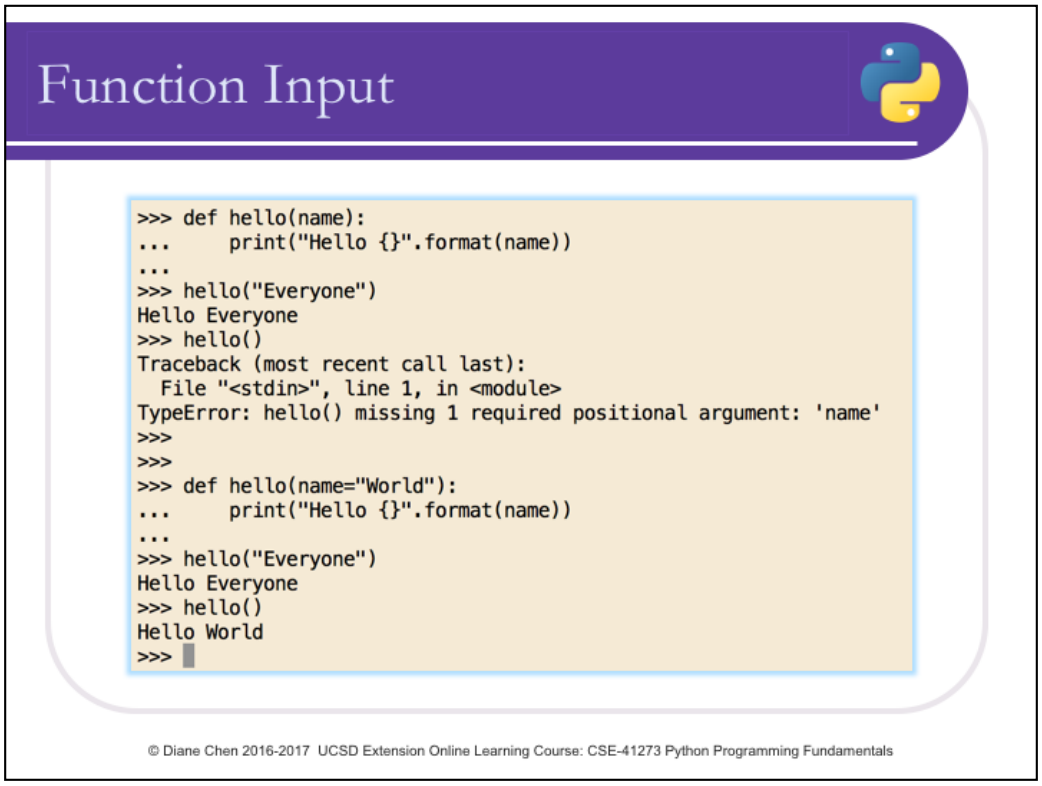
Here I show an example of creating a dictionary by using zip on the two iterables.

Zip is also handy if you have multiple iterables that you want to loop over together. We can say "for color, ratio in zip(colors, ratios)" and loop over the corresponding elements together without having to use an index. As the zip function returns tuples, we unpack the tuples in the for loop into the individual variables.



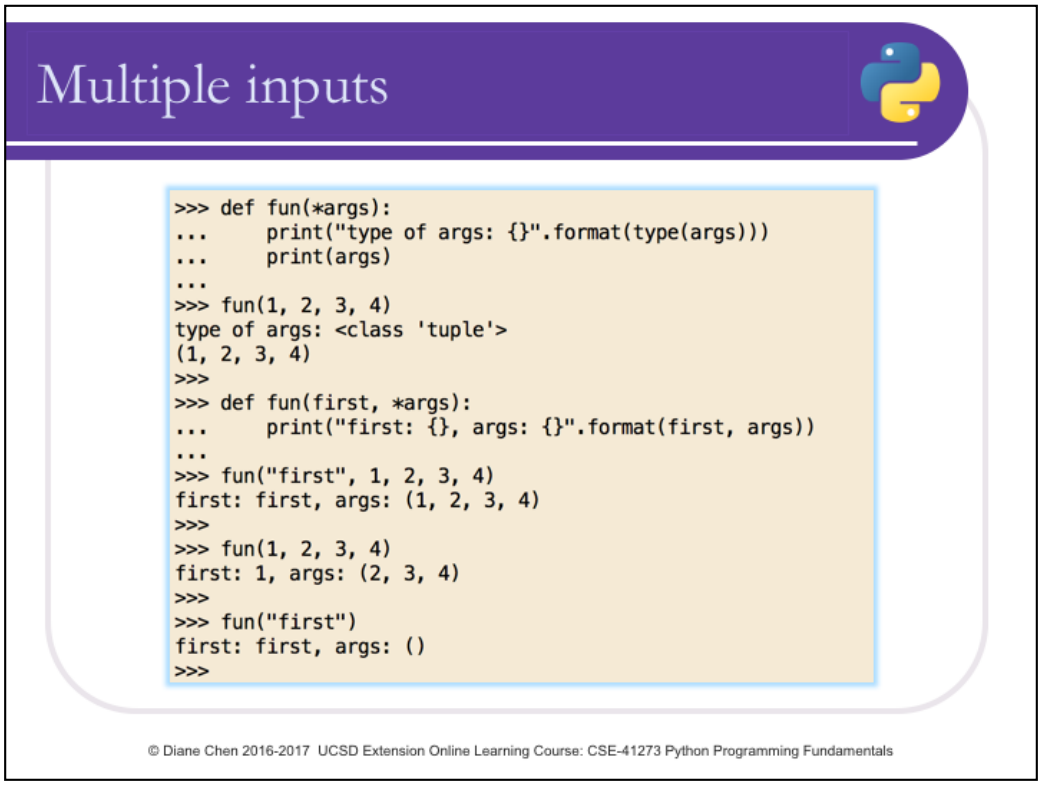
A function is a piece of re-usable code. Usually it takes one or more inputs (but sometimes no inputs), does some operations by itself or on the inputs, and may or may not return data back to the caller.

It is defined using the keyword “def” followed by parentheses containing the input arguments (if any), followed by a colon. The body of the function is an indented block following the definition.

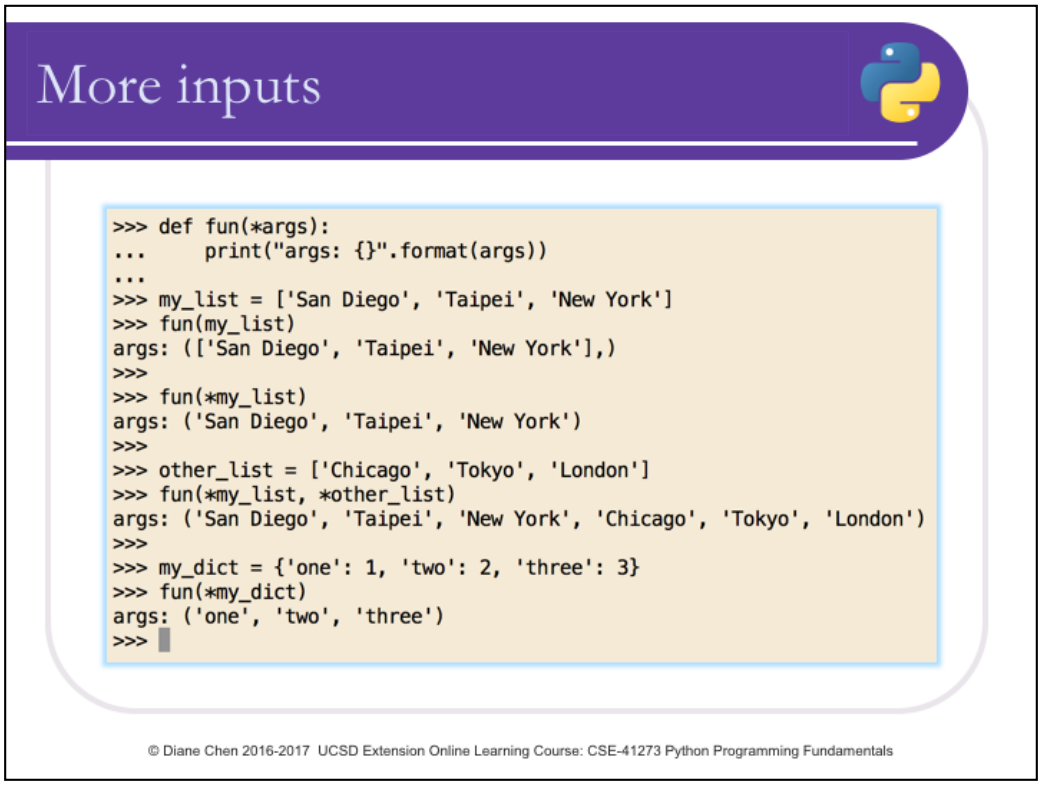


Let's change the previous hello function to add an input parameter to the function so we can modify the message printed out. In the top example, we just add name. We can print out the string that we input to the function. But if we call it without the input, we get an error that we are missing the argument.

In the next definition of the hello function, we add a default value to the name parameter. Now we can call it with or without an input argument without getting an error.



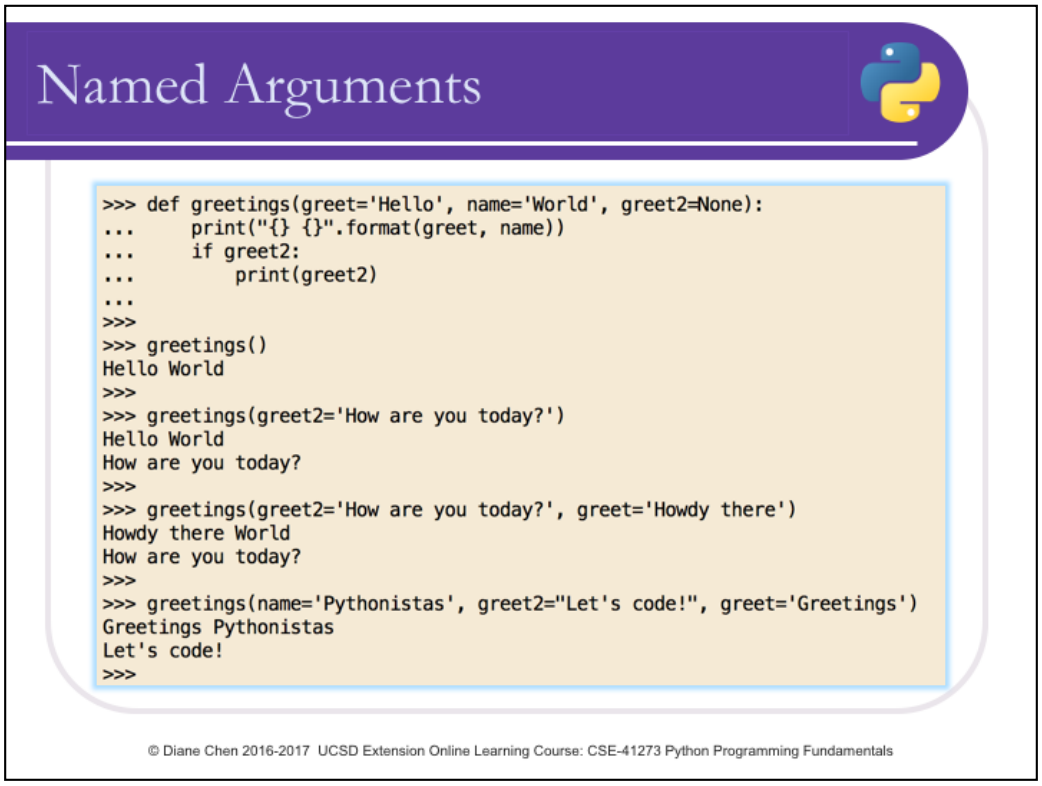
Let's look at more generalized examples. What if we want to allow the caller to give us an unspecified number of arguments? We can use ”star args”. It’s just a convention to use the variable name args, you can call it whatever you want. Since this is just an example and it has no meaning, ‘args’ is fine here – normally I would give it a name that has some meaning to the function using it. Inside the function the variable args is a tuple containing the inputs. We can combine positional arguments and star args in one function definition. There must be enough input parameters to fill up the positional arguments, or we get an error. But it’s OK for the *args variable to be empty.



We can put other iterables into the star args argument. Here is a list of 3 big cities I've lived in. When we put it into the function by itself, the list just becomes the first element of the input tuple in args. This is probably not what we want here. We probably want to have each list element as a separate input element in the tuple. How do we do that?

If we add the star before my_list in the input, the system unpacks the list and makes each element from the list into an element of the tuple in args. This allows us to use multiple lists. Here I added a list of cities I want to visit. By using the * for each list in the input, the args variable contains all the elements as if we had only put one big list in the input. It doesn't matter to the function itself.

Using a single *, we can unpack just the keys of a dictionary into the *args as shown here.



We can also have what we call named or keyword arguments, where the caller specifies the arguments by name, giving each one a value. There are multiple ways of implementing this. One way is just to have multiple arguments to the function that have default values. That way the caller can choose which ones (if any) to override. Not all of the arguments need to have defaults, just the ones that we don't want to force the user to have to enter.

In this case, I gave all the arguments default values so when greetings is called with no arguments it just prints out "hello world". We can override the default value for any of the arguments we want, and we can specify them in any order, since the values go with the names. Note that for this function, we don't have to put the names if we don't want to. I didn't have room on the slide, but if we called greeting with one string, the system would use the position of the input to determine which input argument it goes to. So one input string would go into the variable 'greet', two would go into 'greet' and 'name', and 3 would go into 'greet', 'name', and 'greet2' in the order given.

Note that when we have named or keyword arguments, we don't put spaces around the = sign. This is a strong convention – haha, that means I don't want to see spaces around the = sign in function definitions or function calls in any of your homework assignments! We'll talk more about coding style next week.

Keyword arguments

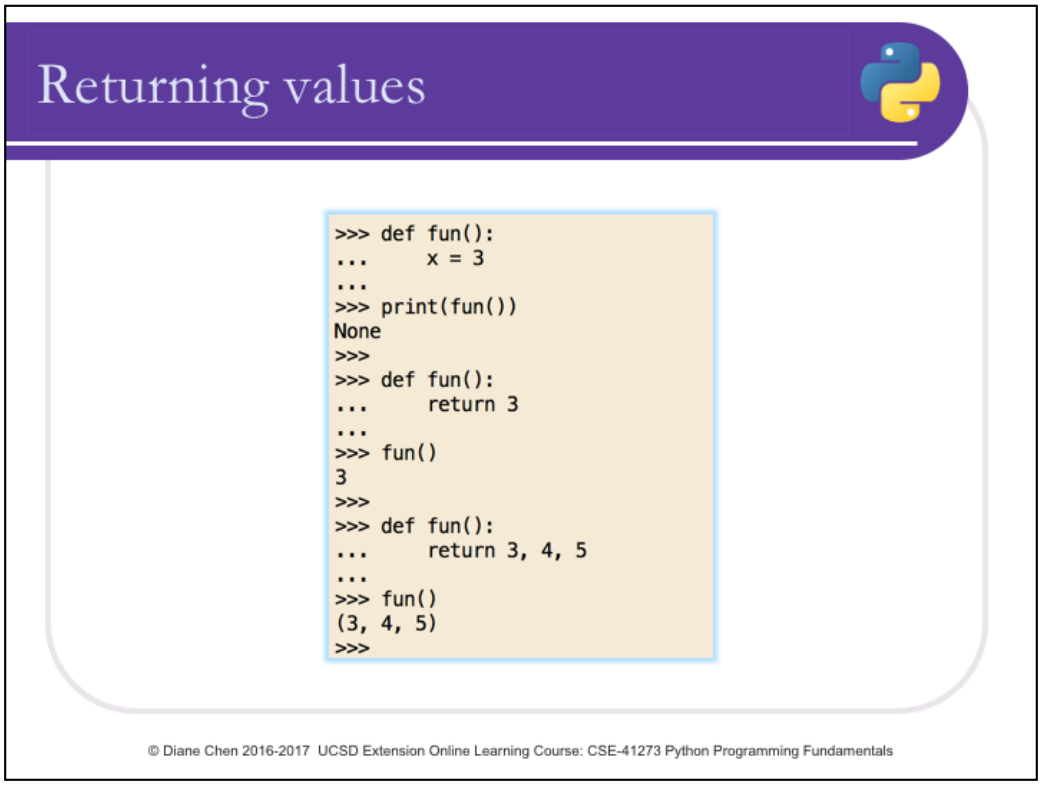
```
>>> def fun(**kwargs):
...     print(kwargs)
...
>>> fun(name='Pythonistas', greet2="Let's code!", greet='Greetings')
{'name': 'Pythonistas', 'greet2': "Let's code!", 'greet': 'Greetings'}
>>>
>>> my_dict = {'one': 1, 'two': 2, 'three': 3}
>>> fun(**my_dict)
{'one': 1, 'two': 2, 'three': 3}
>>>
>>>
>>> new_dict = {'four': 4, 'five': 5}
>>> combo = {**my_dict, **new_dict}
>>> combo
{'one': 1, 'two': 2, 'three': 3, 'four': 4, 'five': 5}
>>>
```


We can also allow the user to put in named arguments that don't match any of the defined input arguments. We use the **kwargs input to implement this. Kwargs is the conventional name to use for this, standing for "keyword args", but you can use any variable name, just like with *args. This is a special syntax for input of named arguments. Kwargs is implemented as a dictionary. Here the function just prints out what comes in kwargs. Just because I'm lazy, let's use the inputs we used for the greetings function previously. The function doesn't care what arguments it gets, it just puts them into the kwargs variable. Here you can see that it is a dictionary.

In the same way that we used * on the lists in our unpacking example a couple slides ago, we can use ** for dictionary unpacking into the kwargs argument if we want. The key of the dictionary becomes the argument name, and the dictionary value become the corresponding argument values.

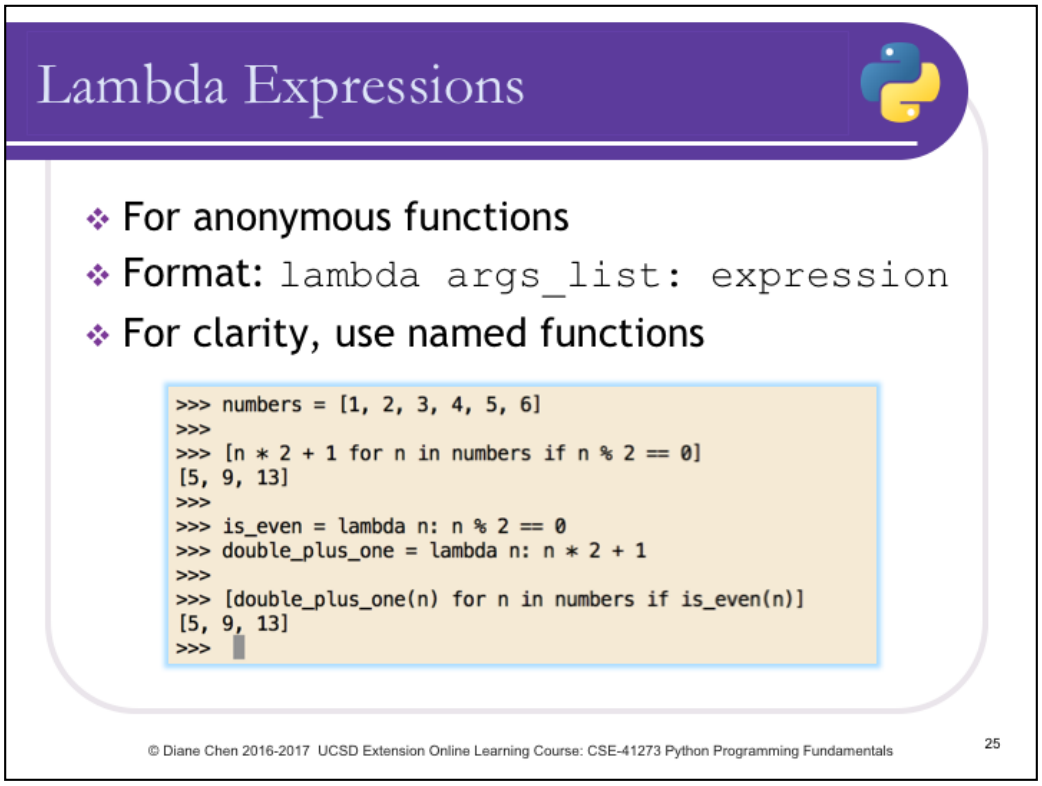
I've shown just the **kwargs by itself, but it can be combined with positional and *args together in one function if so desired.

Unrelated to functions, we can use the ** unpacking on dictionaries to create a new dictionary from more than one original dictionary. This only works since version 3.5.



We talked a lot about function arguments, but how do we return values to the calling program? As you might guess, we use the return statement. However, one thing to know is that all functions have a return whether it's written in the code or not. If there is no return statement, or in other words the code just falls through at the end of the function, the return value is None. Here we have the function fun() returning 3. We don't really need to print it, because the REPL will print it for us. We can return any object.

A cool thing about Python is that we can return more than one thing. The calling program will see the returned result as a tuple.



What are lambda expressions? When I first started learning Python, they were mysterious things and kind of confusing. Lambda expressions create anonymous functions, and sometimes people will say Lambda Functions. They are shorthand replacements for simple defined functions and are somewhat limited as they can only have one expression as a statement; the result of the expression is returned to the caller. They are functions and not really expressions by themselves! Here is an example of a list comprehension where we want to make a new list from the even numbers in the original list, doubling them and adding one. These are very simple expressions in the list comprehension, but at first glance, the meaning is not very clear. Imagine how confusing it might be with complicated expressions! Because functions are first-class objects in Python, we can create two functions, is_even and double_plus_one, by assigning a lambda expression to them. I just took the expressions used in the list comprehension and put them into the lambda expressions. Now it is obvious at first glance what is happening in our new list comprehension. A lambda expression can be used anywhere a function is needed. We will see more of them later.