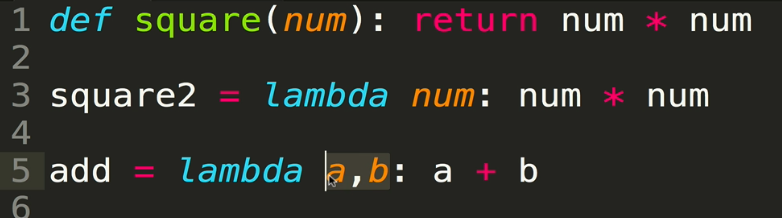
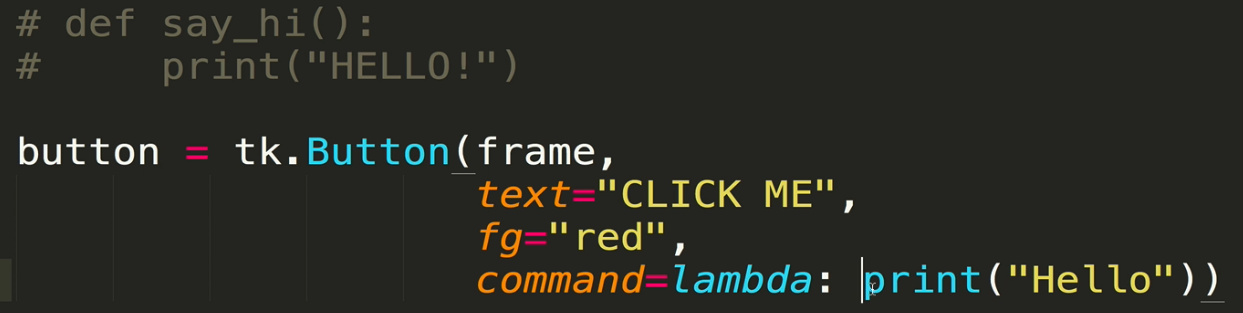
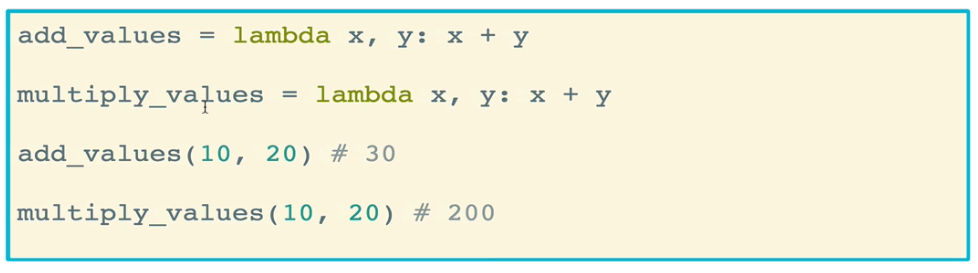
* Lambda functions
  + Lambda functions, also known as anonymous functions, can be thought of as functions that have no definition
  + The idea is that we can run a short, one-line procedure that automatically returns a single expression
  + Although they are not formally defined, they can be assigned a name. Then that name can be called with the appropriate parameters to execute the function
    - It is NOT typical to store a lambda function in a variable
  + Some simple examples



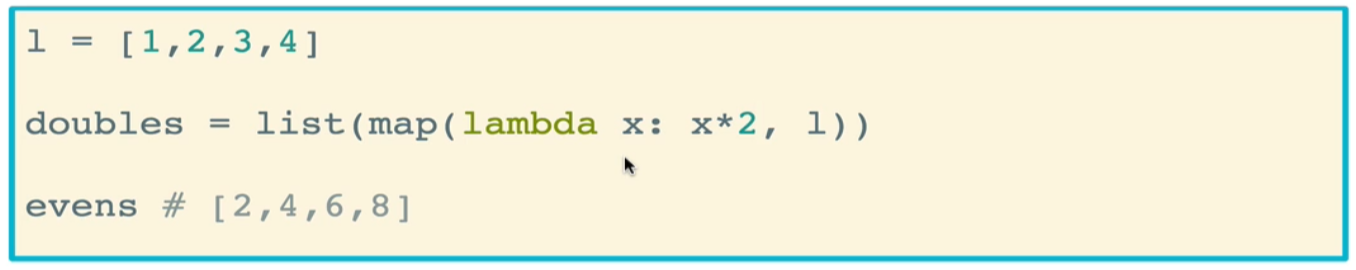
* + The common use case is when you need to pass a function into another function as a parameter, and that function will never be used again
    - In the example below, a GUI button reacts to button clicks by calling a lambda function that simply prints “Hello”. This is an alternative to calling an entirely separate function that prints “Hello”



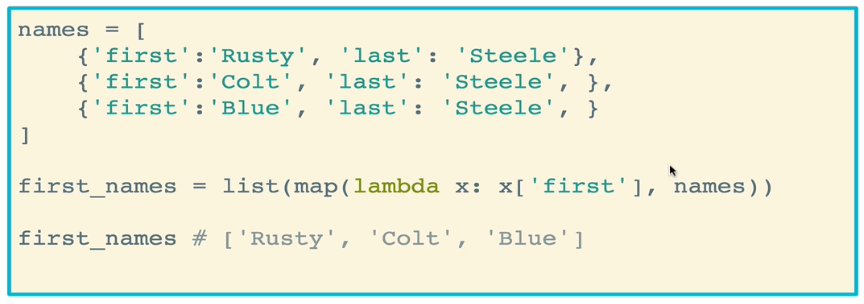
* + Generally speaking, lambda functions are not very commonly used. Some folks want them eliminated completely
  + More examples



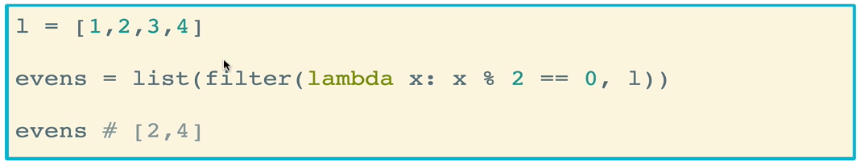
* **Map** is standard built-in function that takes at least two arguments: a function to map, and some iterable (list, string, dictionary, etc)
  + Whatever function is passed to map, it will run that function for each value in the iterable and return **a map object** collection that contains the “new” version of everything that was passed in from the iterable
  + The function passed in is usually a lambda function, but does not have to be!
  + Map objects can be iterated over, but can only be used once. To save it, you can simply turn it into a list
  + Examples:
    - A simple function that doubles each number in an iterable



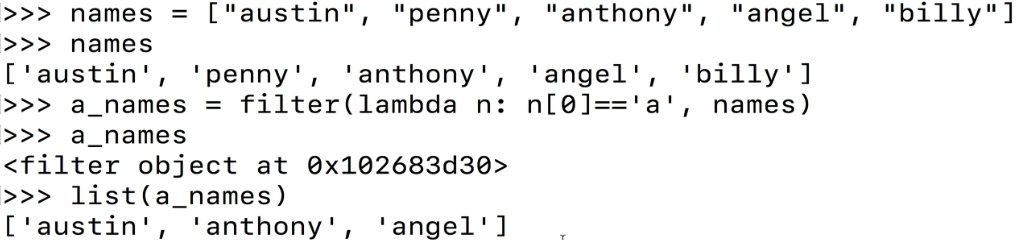
* + - A function that returns only the first names of the dictionary of names



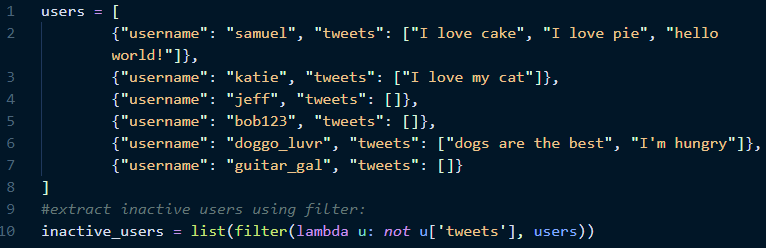
* The **filter** function allows you to take an iterable and filter out particular items based on an expression
  + It takes two arguments: a filtering function and an iterable to filter
  + Similarly to map, filter returns a **filter object**. But this filter object can be turned into a list
  + Simple example that filters out even numbers from a list and throws them into a new list, based on an expression that was passed in via a lambda function that tests whether the number is evenly divisible by 2



* + Another example: this function filters only for names that start with a



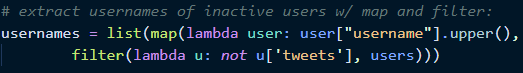
* + Sometimes you may also want to extract things from a list. In the below example, the *users* list contains a list of dictionaries, one for each user. The function will iterate over each dictionary, determine whether the value of the “tweets” keyword is, and throw that user into the list of non-tweeters
    - Remember that empty lists are inherently falsy, so to get non-tweeters, you have to filter for users that are *not* non-empty



* + - Of course, we could also have done this with list comprehension:



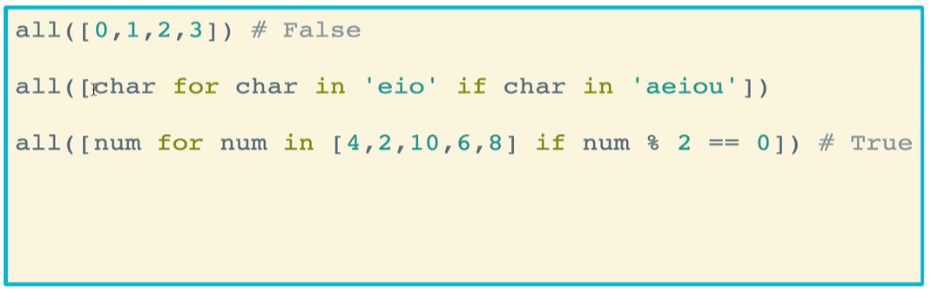
* + What if we just want the *names* of the inactive users? We can do that by combining filter and map
    - First, a list (or more precisely, a filter object) is created using the filter function that contains only inactive users
    - Next, that filter object is passed into the map function, which iterates through the filter object and pulls out the usernames



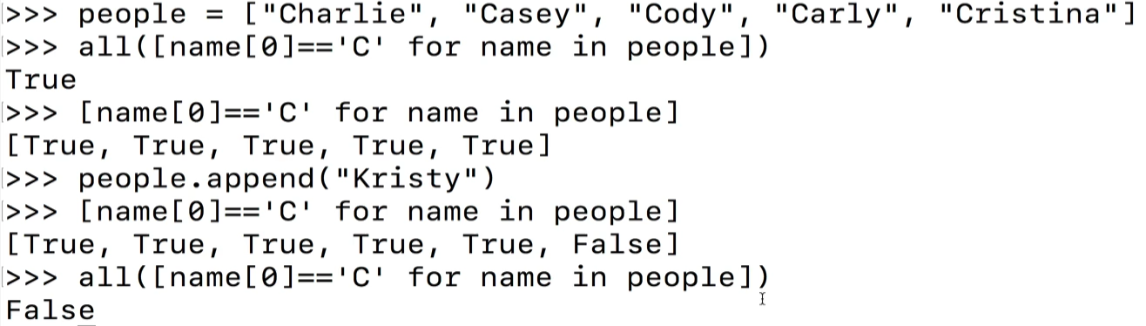
* + - List comprehension approach



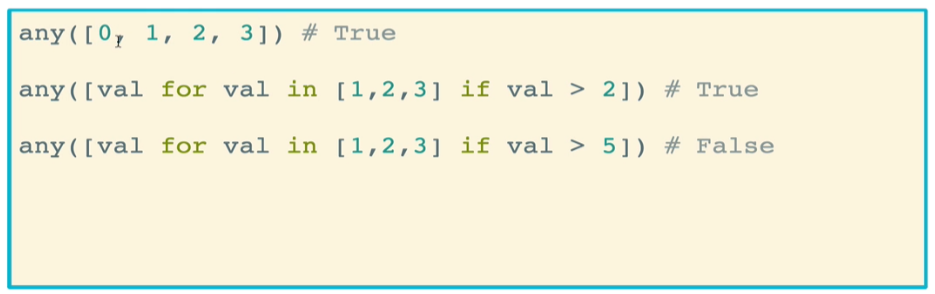
* The **all** function takes an iterable and returns True if all elements are truthy
  + Silly example where True will be returned



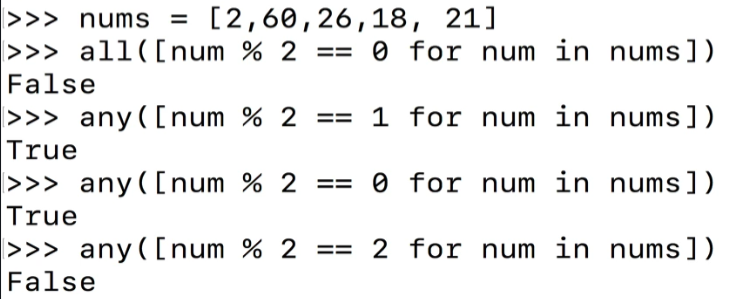
* + Another example testing whether the first letter of each of these words is C



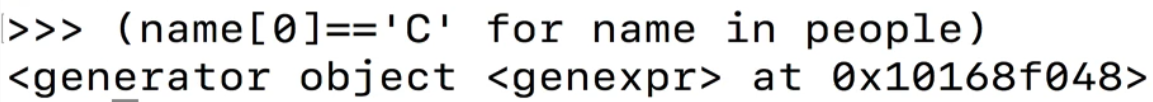
* The **any** function takes an iterable and returns True if *any* element in the iterable is truthy
  + Example



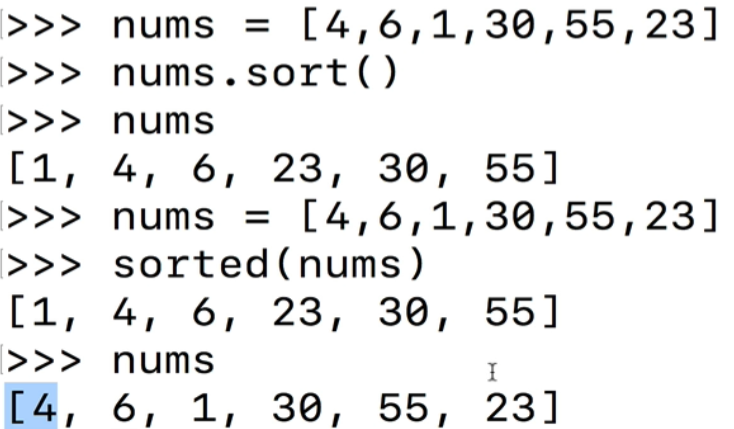
* + Another example



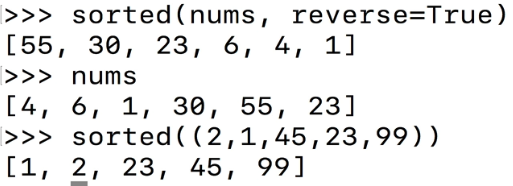
* You do not actually need to use list comprehension brackets when using *all* or *any*. You can remove the brackets and run commands as a **generator expression**, which can be thought of as a lightweight version of a list
  + This allows you to get your True or False statement without generating an intermediate list that you don’t need, saving memory and computational power
  + 



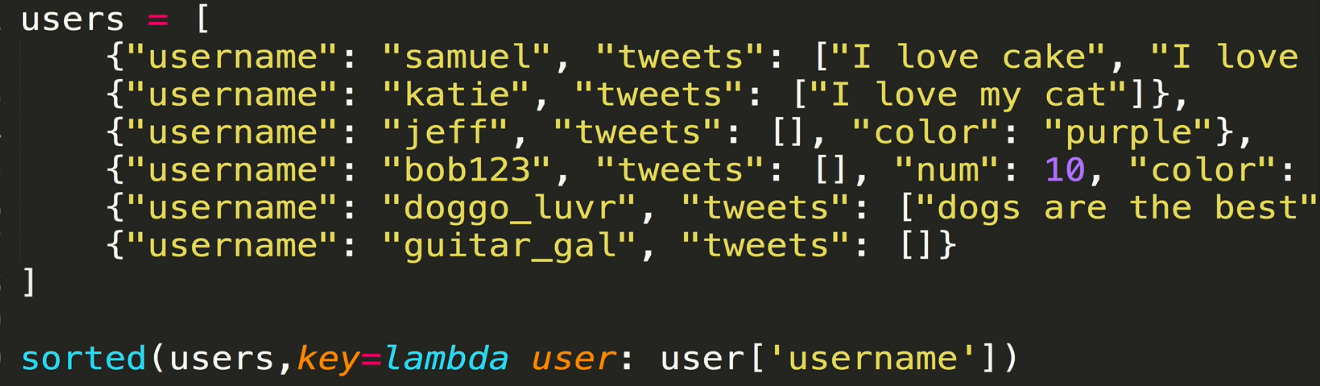
* The **sorted** function returns a new sorted list of items from an iterable
  + <https://www.w3schools.com/python/ref_func_sorted.asp>
  + The iterable can be a list or a tuple
  + It always *returns* a list
  + Unlike the built in sort() function for lists, sorted() does not change the original iterable unless the variable is overwritten



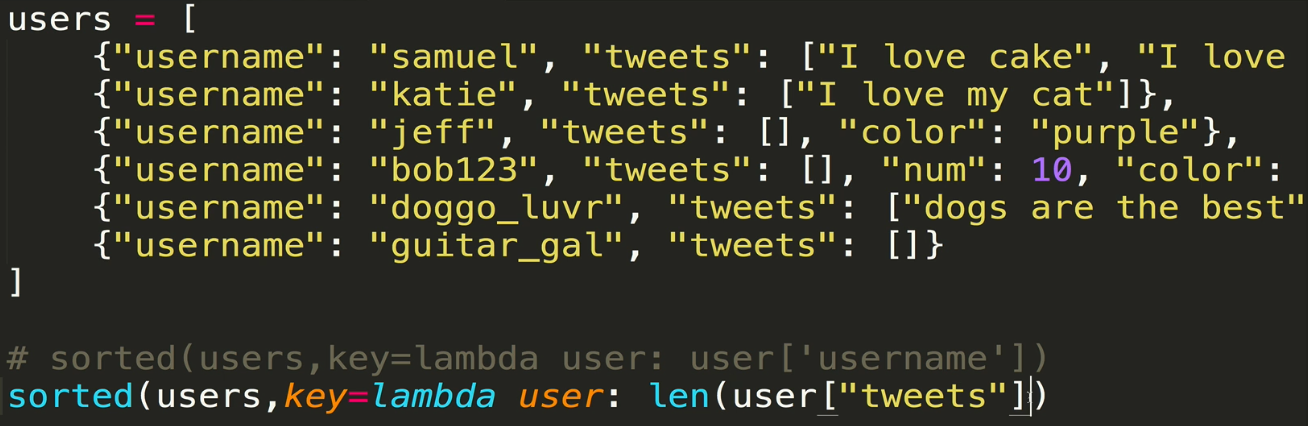
* + You can pass additional parameters into sorted aside from the iterable
    - The reverse parameter will reverse the sorting of the list if set to True



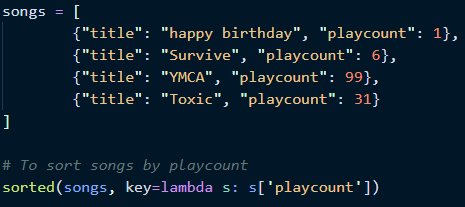
* + What about sorting more complicated things, such as sorting entries in a dictionary based on one of the keys of those entries. For this, we can determine how the function will perform sorting by using the *key* parameter
    - This is different from a dictionary key
    - Example: This uses a lambda function as a key for sorting a dictionary of users. The lambda function will iterate over each user in the dictionary and report the value of the username. Then the sorted function will proceed to sort the users in alphabetic order



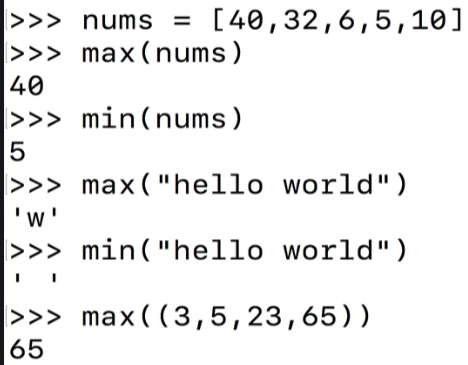
* + - Let’s instead sort according to who has the most tweets. Remember that *tweets* is a list, so we want to check the length of that list (i.e. the number of tweets) and sort the users accordingly



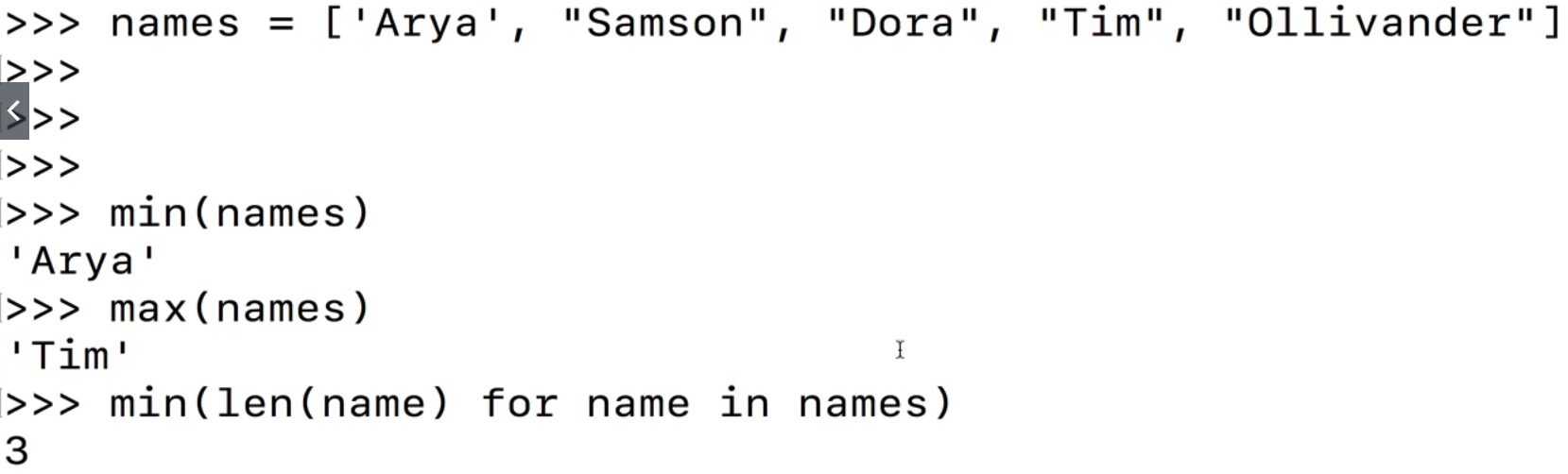
* + Another example: sorting a list of songs by play count. Each song is a dictionary, and we access the *playcount* key to get the value to sort by



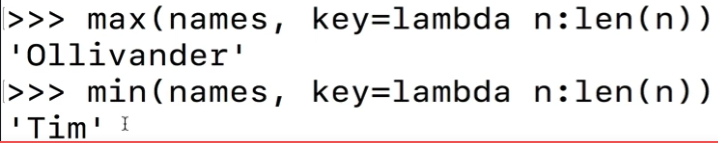
* The **max** function returns the largest item in an iterable, or the largest of two or more arguments passed in. The **min** function returns the smallest item in an iterable, or the smallest of two or more arguments passed in
  + https://www.geeksforgeeks.org/max-min-python/



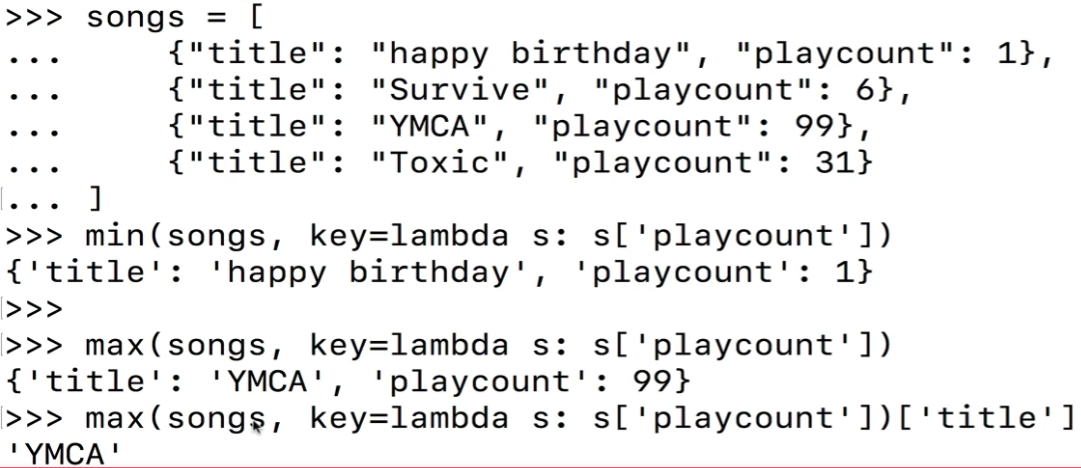
* + A more interesting example:
    - By default, min() and max() of strings are determined by their alphabetical order
    - We can also apply a test on the items of the iterable, such as the length of the string, and have the min() or max() functions report that length



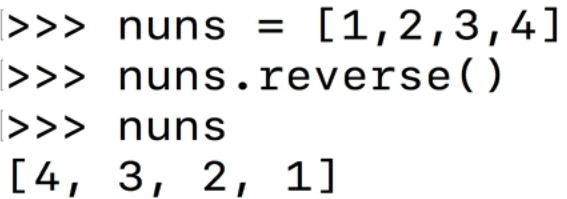
* + - What if we want the function to return the actual longest string in the list? We can’t use max() because remember, that’s alphabetical. Instead we use a lambda function. Just like with sorted(), we can pass in a key function to tell max() how to judge which item is the largest



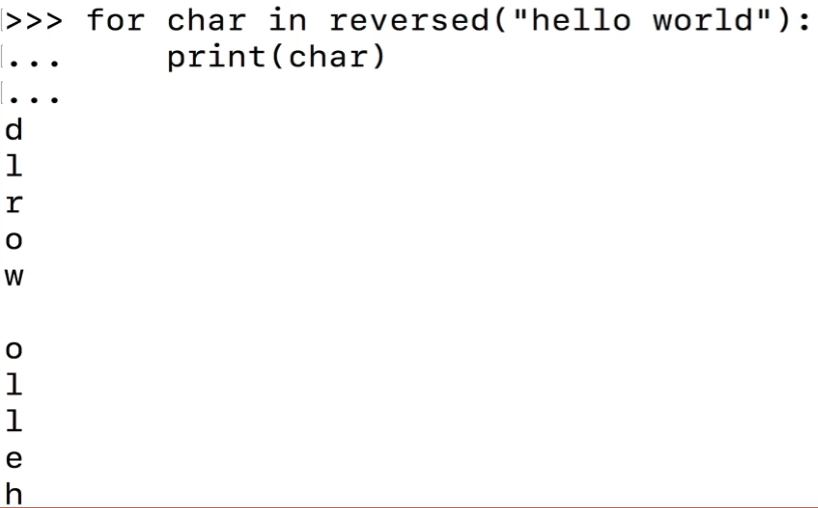
* + Final example: finding the item in the songs list that has the smallest playcount.
    - Remember that this returns a dictionary, so to get any particular value we simply need to tack on the key



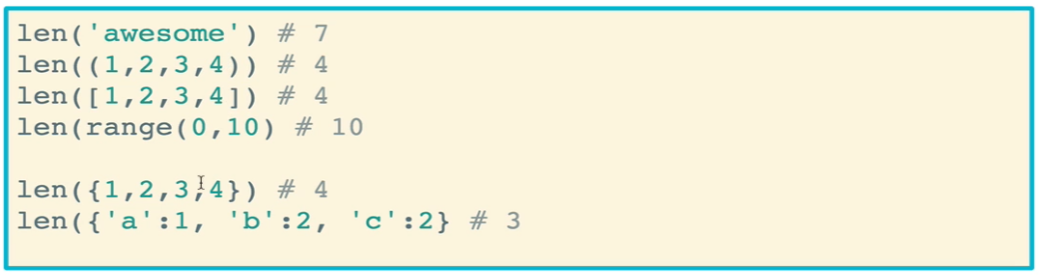
* The **reversed()** function returns a reverse iterator
  + I mean yeah you can write a loop to do it, but why do that when you can just use reversed()
  + Useful for iterating over something in reverse, without having to build a function to reverse the iterable
  + It is different from the list.reverse() method, which we revisit below



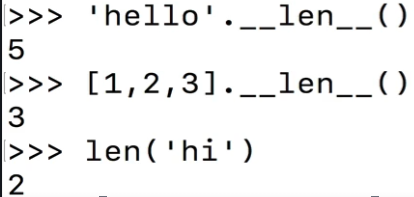
* + Instead, you pass the iterable directly into the reversed() function. It technically creates a reverseiterator object, and you can iterate over this object. You can also convert it into a list



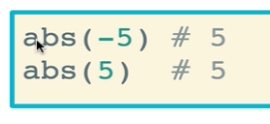
* As we’ve seen before, **len()** returns the length (number of items) of an object. The argument can be a sequence (such as a string, tuple, list, or range) or a collection (such as a dictionary or set)



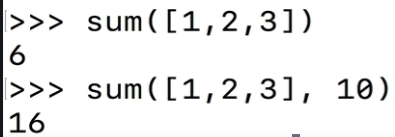
* + But how does it work? Behind the scenes, it uses the dunder \_\_len\_\_() method. Remember that all strings, lists, tuples, etc. are objects that have inherent methods, in this case \_\_len\_\_ is one of those methods



* + - We will see more of this in object-oriented programming where we develop our own classes and give them methods
* The **abs()** function returns the absolute value of a number. The argument may be an integer or a floating point number
  + Absolute value is the magnitude of the number without regard to its sign
  + It will not automatically convert a string to a number. You’ll need to do that conversion



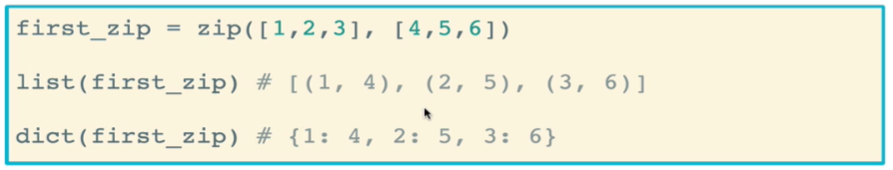
* + There is another method called **fabs**() that needs to be imported with the *math* module
    - It does the same thing, but converts everything to a float first (it probably stands for “float absolute value”
* The **sum()** function takes an iterable and, optionally, a *start*, then iterates from left to right, adding the values together
  + Start defaults at zero (0), meaning that the sum begins at zero
  + You can pass in a start value and the function will begin the addition at that value



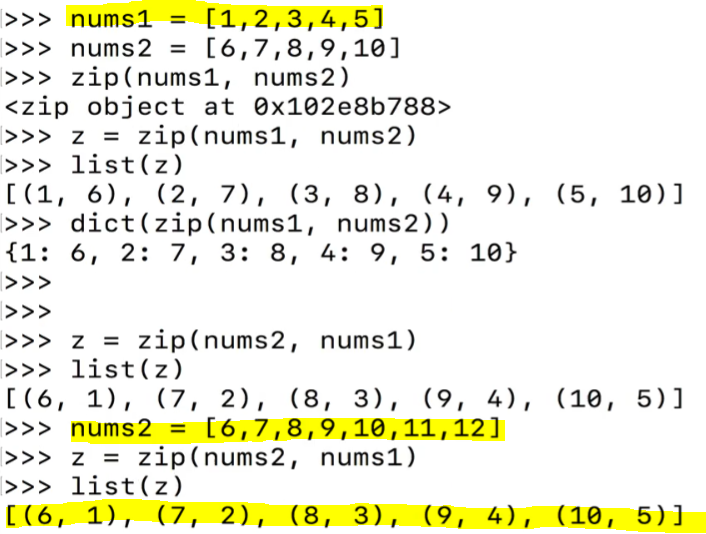
* + The iterable can be a list, tuple, or set
  + If you try to sum strings, you will get an error. The error will tell you to use ‘’.join() instead
* The **round()** function returns a *number* rounded to the number of digits *ndigits* of precision after the decimal point. If *ndigits* is omitted or is *None*, it returns the nearest integer to its input
  + Round is NOT the same as flooring, for example during integer division. It rounds according to appropriate rounding conventions



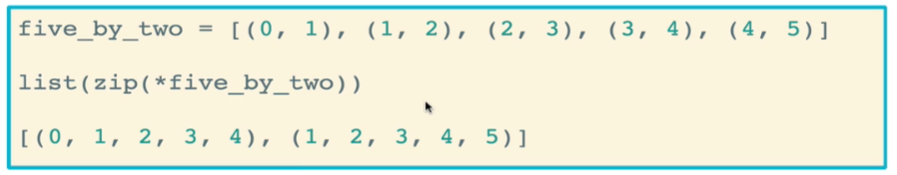
* The **zip()** function makes an iterator that aggregates elements from each of the iterables. It returns an iterator of tuples, where the i-th tuple contains the i-th element of each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted
  + Essentially, it zips up the 1st elements of the iterables, the 2nd elements of the iterabes, etc. Another way of saying this is that it maps the similar indices of multiple containers (iterables)
  + <https://www.geeksforgeeks.org/zip-in-python/>
  + By default it returns an iterator of types. You’ll need to manually convert it to a different type such as a dictionary or a list



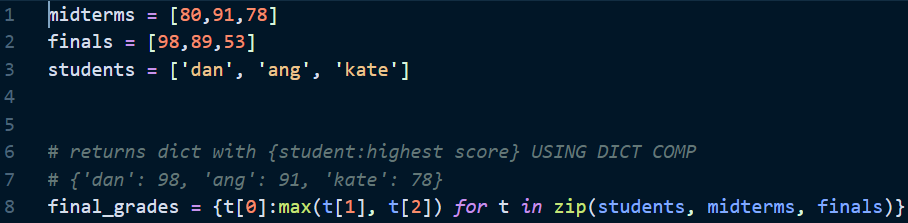
* + The items need not be the same lengths, but it the function will stop once the shortest iterable is exhausted



* + The star operator can be used to unpack or unzip a list of iterables, placing them in fewer tuples based on their indices
    - This is very common when working with complex data structures



* + A more practical use for zip() can be grouping together scores for students
    - This example matches each student name with their highest score



* + - Another approach using lambda functions

