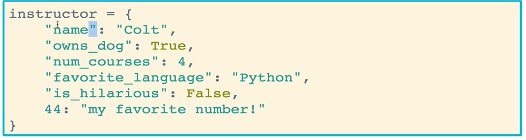
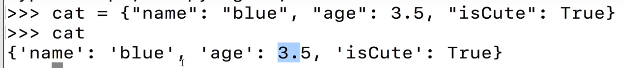
* Lists are great, but have some limitations
  + A collection of values, even if directly relevant to each other, sometimes does not offer enough information to make sense of it. What is the item at each index supposed to represent? Just looking at a list, it’s not immediately clear.



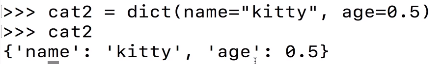
* + Dictionaries allow you to include additional information
* Enter **dictionaries**, which consist of pairs of information
  + Consist of key:value pairs. Keys describe the data, and values represent the data
  + Key: Name; Value: Jason
  + Check out the example dictionary below, called “instructor”, which has a bunch of key:value pairs describing Colt (the instructor). Notice also the curly braces and colons separated the key:value pairs
  + Keys are almost always numbers or strings
  + The values can be anything – a number, string, a list, or a dictionary



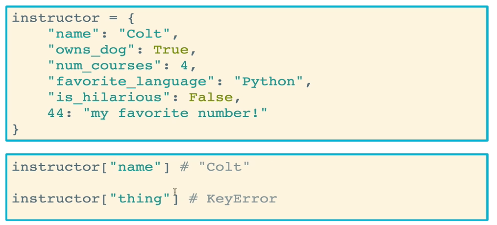
* One way to create dictionaries is to use the curly braces syntax



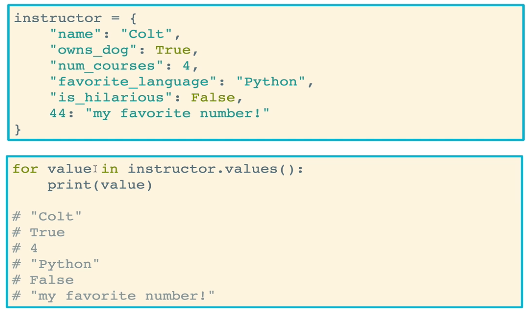
* Another way to create a dictionary is to use the dict() function, where keys and values are passed in as key = ‘value’



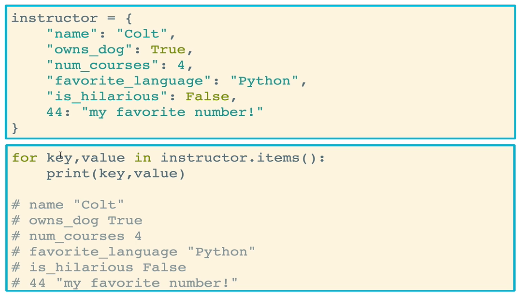
* To access individual values in a dictionary, we use a similar syntax as assigning values: we use the dictionary name and then pass in the key inside of square brackets
  + A KeyError is returned if the key does not exist
  + Later we will learn a method to figure out of a particular key exists in a dictionary



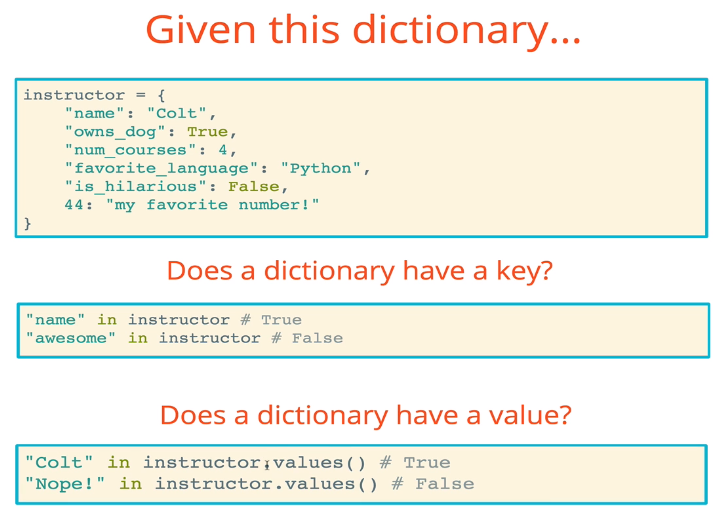
* How do we access ALL of the values in a dictionary using an iterable? It requires an extra step, and there are a few ways to do it.
  + We can actually loop through both keys and values
  + The **.values()** method gives you an iterable collection of *values* that you can run a for loop with. Note that it does not give you a key



* + The **.keys()** method gives you an iterable collection of *keys* that you can run a for loop over
  + Finally, you can use the **.items()** method to get both the keys and values. It gives you a list of tuples that, each tuple being a key:value pair



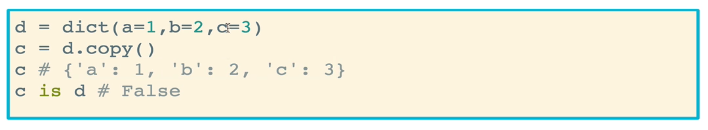
* You can do a logical test to see if a key or value is or is not in a dictionary
  + Remember that you can access the values using the .values() method
  + You can simply use the name of the dictionary to access the keys



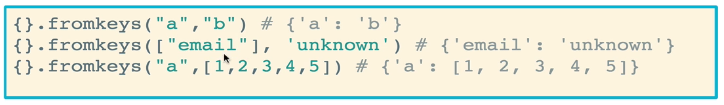
* Dictionaries, much like lists, have a lot of methods that can be used
  + *dict*.**clear**() empties out the dictionary

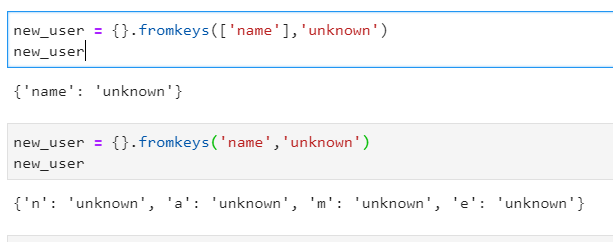


* + *dict*.**copy**() makes a copy of the dictionary. Save it to a variable
    - Just remember that the copy is a unique object in memory. So in the example below if you delete *d*, *c* will still exist
    - Conversely, the original and copy have the same values, and so the test for equality (==) will return true

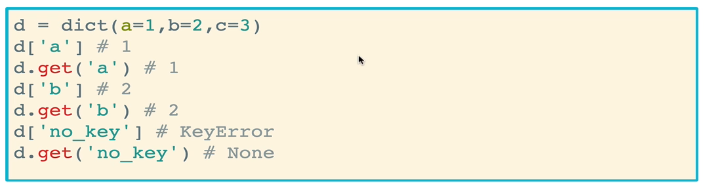


* + *Dict*.**fromkeys**() is usually called on an empty dictionary and creates key-value pairs from comma-separated values
    - Why would you want to do this? It becomes useful if you have a bunch of default keys that you want to set to an initial default value, until it is modified later on
    - Unless it is a single character, the keys must be passed in as a list or an iterable collection. Otherwise, the method will attempt to iterate over a string
    - Running .fromkeys() on an existing dictionary simply creates a new dictionary. It does not modify the existing dictionary

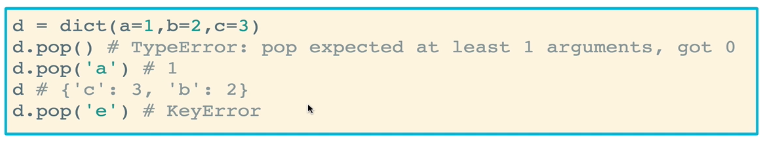




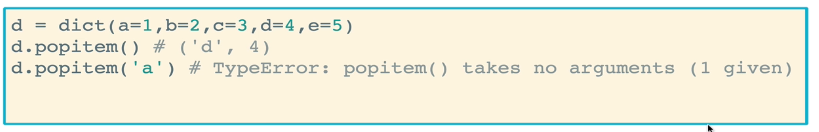
* .**get**() is the most useful method (according to instructor). It retrieves the value for a given key in an object. If the key does not exist, it returns None instead of KeyError
  + Similar to “in”, but without an error
  + You can try to access values using the parentheses approach, but if the key does not exist, the code will throw an error. If you use .get() instead, it will return None instead of an error



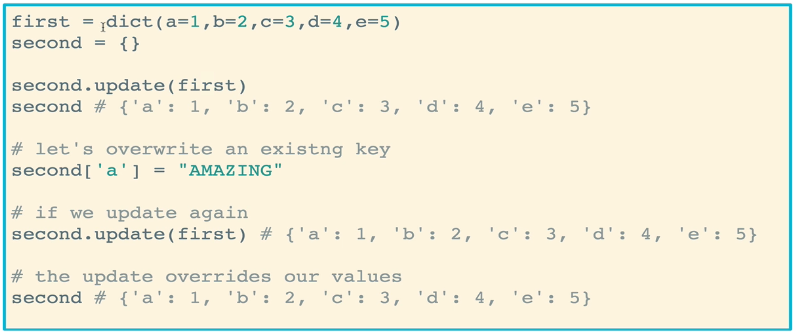
* .**pop**() is similar to the list method, but the key that you want to remove must be provided (as opposed to the index of the item in lists)
  + The method returns the value associated with the provided key, and remove it from the dictionary
  + If the key does not exist, a KeyError is returned



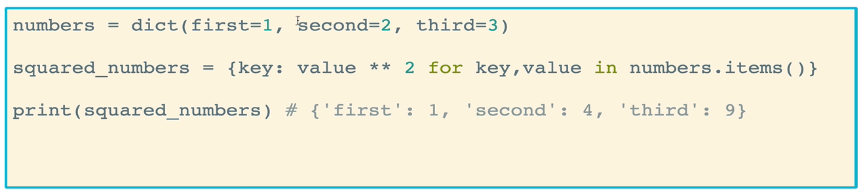
* .**popitem**() allows you to randomly remove something from a dictionary. No arguments needs to be provided. The method will remove the key and value, and return it as a key, value pair (in a tuple to be exact)
  + It will throw an error if you try to provide an argument



* .**update**() will update the keys and values in a dictionary with another set of key-value pairs
  + It will overwrite any key-value pairs that are already in the dictionary, but will not remove any key-value pairs that the incoming set does not include
  + Basically, it adds keys-values that are not in the dictionary already, and will overwrite the key-value pairs that are in the dictionary already



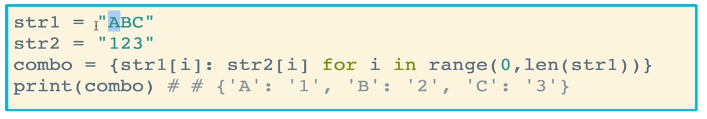
* Dictionary comprehension is a thing!
  + Syntax: {key:value for \_\_\_ in \_\_\_ }
  + Iterates over keys by default, and .items() can be used to iterate over keys *and* values
  + Example: This code iterates over each key:value pair in a dictionary called numbers and returns a new dictionary in which the value for each key is squared



* + The example below creates a new dictionary by iterating over each number a list, storing the original number as the key and the square of the number as the value



* + In the example below, two strings are zipped together to make a new dictionary. For the ith key:value pair in the dictionary, ith character from str1 is the key while the ith character from str2 is the value



* Conditional Logic with dictionary comprehension
  + This code will iterate over a list of numbers and create a new dictionary where the number is the key and the value is either “even” or “odd” depending on whether the value of the number is even or odd

