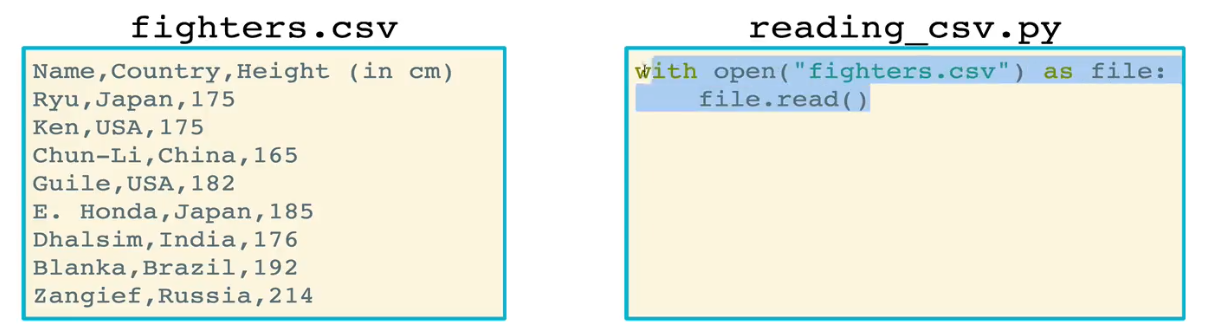
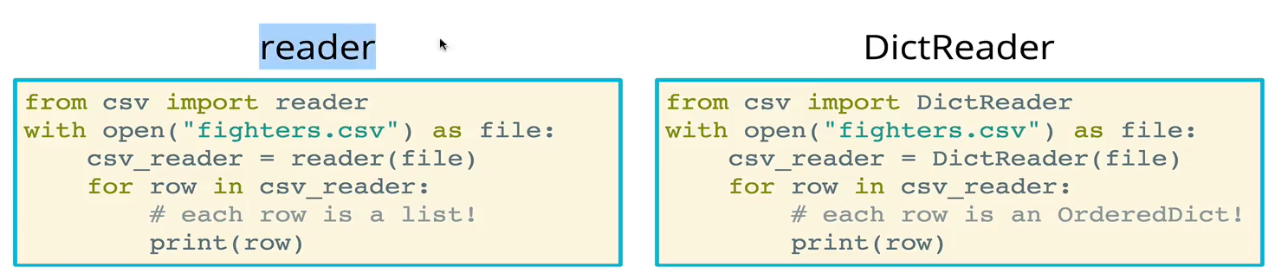
* Comma separated values (CSV) is a common and popular way to store data, usually tabular data
  + We can read CSV files just like other text files, but that’s not a great idea because it will just give you a giant string that you then need to go and parse
    - Don’t do this!

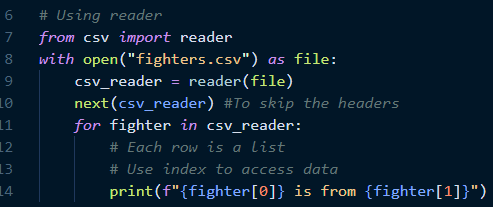


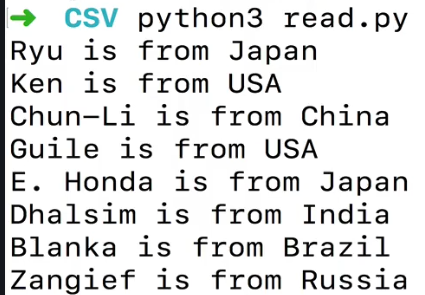
* There are built-in CSV modules to read/write CSVs more easily. Let’s start with the readers.
  + The CSV ***reader*** lets you iterate over rows of the CSV as lists
    - Calling reader() on your file will create a csv.reader object. This is an iterator. This will then need to be iterated over with a *for* loop
    - Each row of the CSV will be a list in this csv.reader object. The items in the list will be the items in the different columns of that row



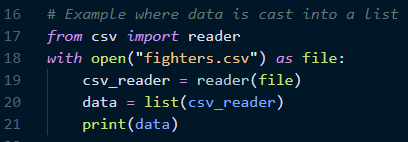


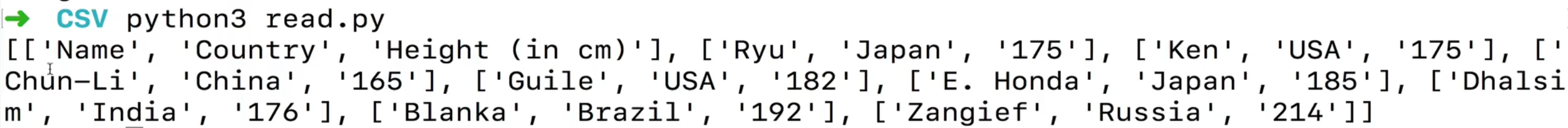
* + - You can access these items in each list by using index references. This is one of the downsides of the *reader* – you need to use indices instead of column (header) names (see example below)



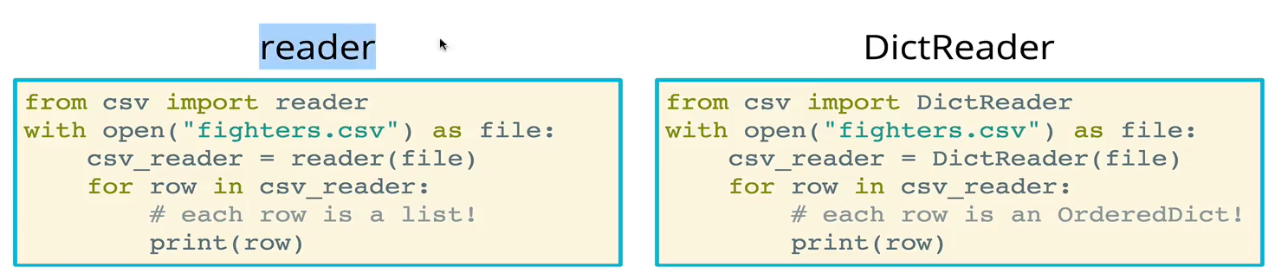


* + - Sometimes it is helpful to convert the csv.reader object into a list. Each item in this list will be a list of items in the columns for that row. It depends on what you want to do with the data



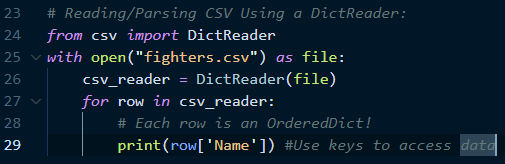


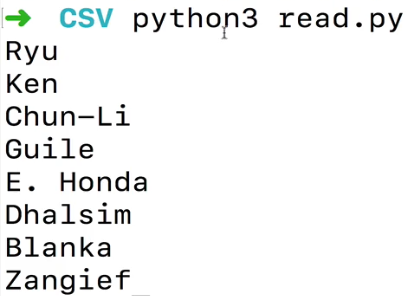
* + *DictReader* lets you iterate over rows of the CSV as OrderedDicts
    - Calling DictReader on your file returns another iterator object, and each row is an OrderedDict object, a special collection in Python
    - Think of it as a dictionary with order (remember that a regular dictionary is not ordered)



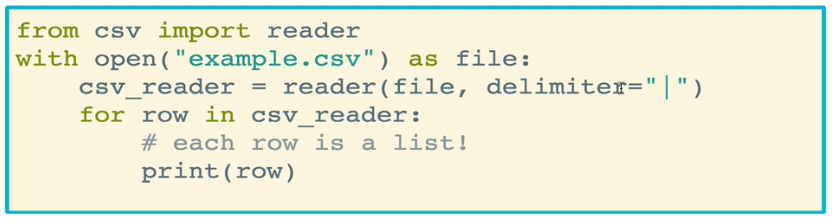


* + - What’s nice about this representation is that it behaves just like a dictionary. You can access each item using a key, and the keys are automatically set up to be the headers from the CSV

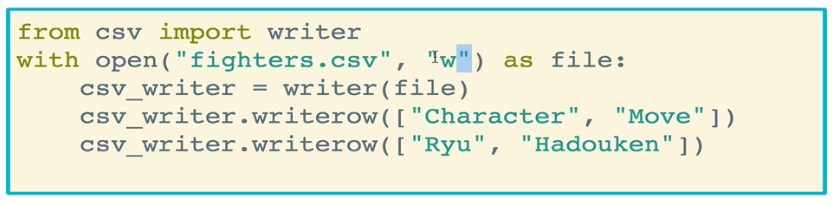




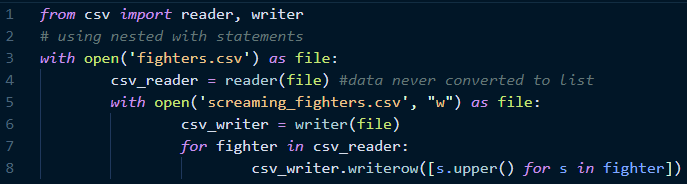
* + Last thing is that these readers can also accept a delimiter keyword argument (kwarg) in case your data is separated by something other than a comma



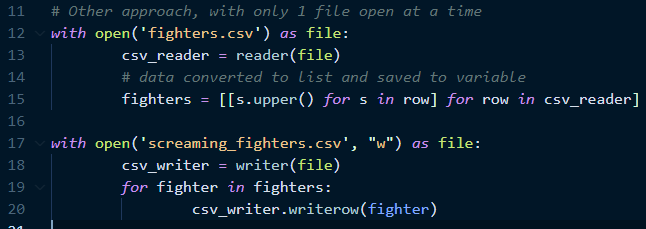
* Writing to CSV files using lists
  + We use the ***writer*** method within the csv package to create a writer object for writing to CSV using lists of data. It’s the writing equivalent to the CSV reader.
  + The method we will use to perform the actual writing is **writerow()**
    - Using writerow will write a line to the csv file
    - To make a header, just user another writerow() command
  + Remember to use the *open* function in “w” mode since we’re writing to it



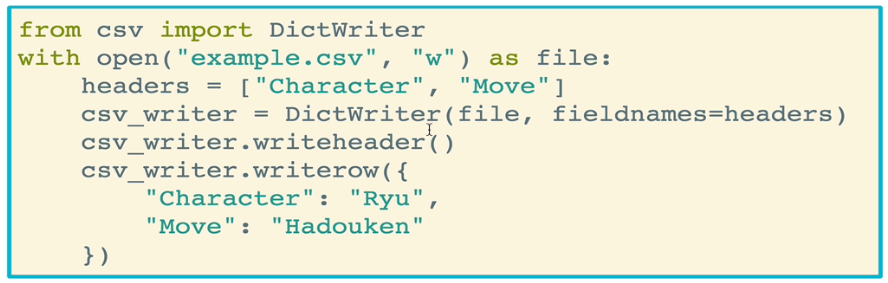
* + In this example, we’ll read in data from an existing CSV, make some changes, and write those changes to a new CSV file.



* + - The above can also be accomplished using two independent open() calls, which ensure only one file is open at a time. Remember that capitalizing the items for each row requires nested list comprehension because each row in the csv\_reader object is a list, and we need to iterate through each item in each of those lists



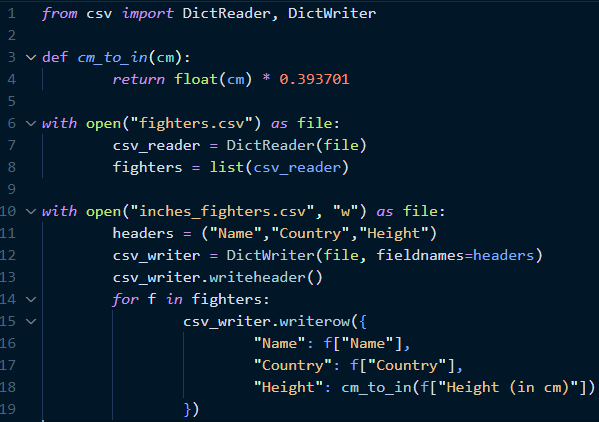
* Writing to CSV files with **DictWriter**
  + DictWriter creates a writer object for writing using dictionaries (as opposed to *writer* which uses lists)
  + The keyword argument ***fieldnames*** is used for the DictWriter specifying headers
  + The ***writeheader***method is called on a writer to actually write the header row
  + Finally, the ***writerow*** method is called on a writer to write a row based on a dictionary
    - Note that since you’re using a dictionary, the keys don’t need to be in order. You just need to make sure the key names match the headers.



* + Another example:

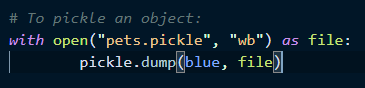


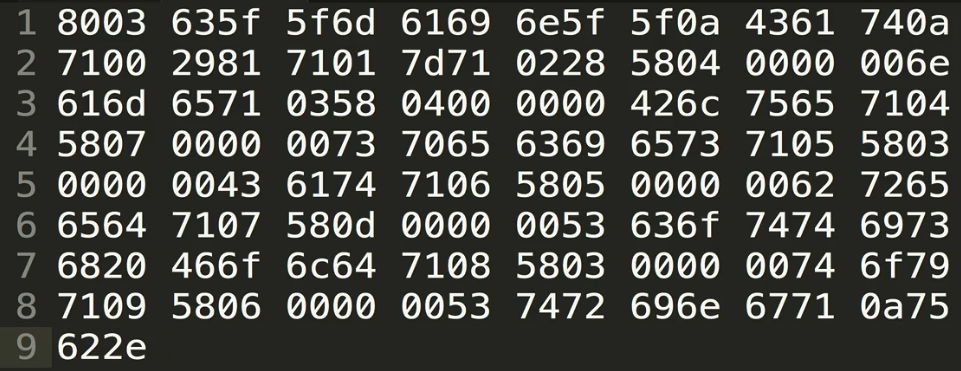
* Let’s use DictReader and DictWriter together! In this example, we are converting each fighter’s height from centimeters to inches and writing to a new file
  + Remember that the keys of the dictionary passed into writerow() must match the header names. In this case, this means we cannot pass in the stored dictionary from DictReader directly



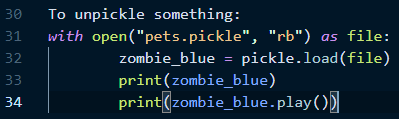


* **Pickling** is a Python-specific way of working with files. Named after the idea of pickling food to make it last longer
  + Documentation: <https://docs.python.org/3/library/pickle.html>
  + In Python, we can pickle something, put it in a pickle file, and Python will serialize the data into a byte stream. At some later time we can pull the data back out of the byte stream
  + The pickle module needs to be imported
  + To pickle an object, use the open() method with the “wb” option to write in binary mode. Then use the **pickle.dump()** method to push your object (blue the cat in this example) into the file

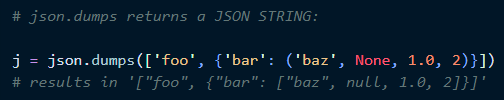




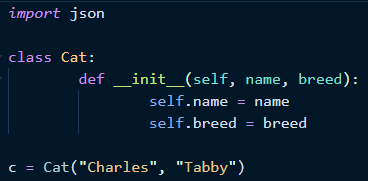
* + To unpickle the object, use the open() method to open the pickle file in “rb” (read binary) mode. Then use the **pickle.load()** method on the file to get your data back.



* + You can also pickle and unpickle multiple things at a time. You just pass in the things to pickle as a tuple to dump(). When you load() from that pickle, the objects will also come back as a tuple
  + Pickling is NOT
    - Readable
    - Useful for when you need to load your data in another format or another language
* Pickling and JSON. Reminder: **JSON** is JavaScript object notation. It is simply a way to send data in an efficient manner. Python has a ***json*** module to encode Python into JSON, and decode JSON into Python
  + The **json.dumps()** method takes python and turns it into JSON formatting. Note that it returns a JavaScript string
    - In this example, a list consisting of a string and a dictionary are converted to a JSON string



* + How do we handle this with arbitrary objects that aren’t lists, dictionaries, etc.?
    - One approach is to use the .\_\_dict\_\_ method on your object, which takes all of the method’s attributes that loads them into a dictionary. Then we can run json.dumps() on that dictionary

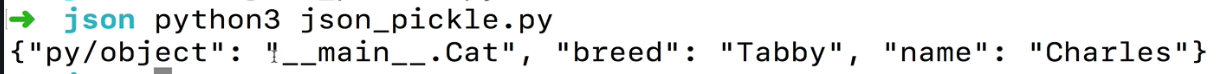


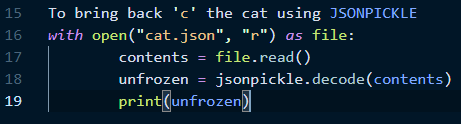


* Alternatively, we can use **jsonpickle**! It serializes and deserializes complex Python objects to and from JSON
  + Documentation: <https://jsonpickle.github.io/>
  + It’s essentially pickling that works with JSON
  + Must be imported
  + To use it:
    - Create an object
    - Call jsonpickle.encode() on the object to pickle it
    - Write the object to a file to be stored (usually you want that so you can transfer the file elsewhere)
    - To get retrieve your data, open the file and use the .read() method. Probably best to store this as a variable
    - Then call jsonpickle.decode() on the file that was read in
    - Done!



* + - * Below are the contents of *frozen*





* + - * Below are the contents of *unfrozen*. Notice how we have returned our cat object! We can now do any of the standard things we would normally be able to do with this object

