**Introduction to iterators**

Welcome to the course! My name is Hugo Bowne-Anderson and I am a Data Scientist at DataCamp. In this course, you'll learn all about iterables, iterators, list comprehensions and generators - all essential components in the Pythonista's toolbox. We will conclude with an entire chapter devoted to a case study. Let's talk about iterators.

**Iterating with a for loop**

There's no reason to be scared of iterators because you have actually been working with them for some time now! When you use a for loop to print out each element of a list, you're iterating over the list.

You can also use a for loop to iterate over characters in a string such as you see here. You can also use a for loop to iterate a over

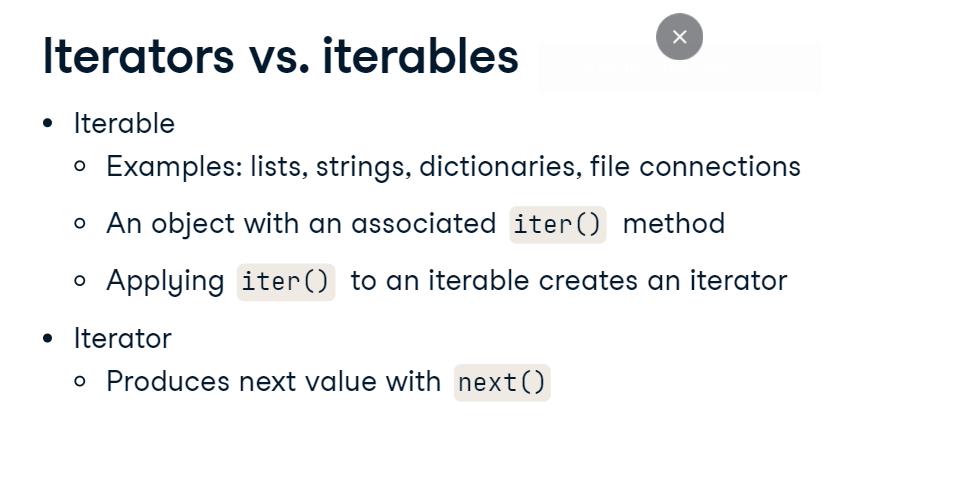
**Iterating with a for loop**

a sequence of numbers produced by a special range object. The reason that we can loop over such objects is that they are special objects



**Iterators vs. iterables**

called iterables: lists, strings and range objects are all iterables, as are many other Python objects, such as dictionaries and file connections! The actual definition of an iterable is an object that has an associated iter method. Once this iter method is applied to an iterable, an iterator object is created. Under the hood, this is actually what a for loop is doing: it takes an iterable, creates the associated iterator object, and iterates over it! An iterator is defined as an object that has an associated next method that produces the consecutive values. To create an iterator from an iterable,



**Iterating over iterables: next()**

all we need to do is use the function iter and pass it the iterable. Once we have the iterator defined, we pass it to the function next and this returns the first value. Calling next again on the iterator returns the next value until there are no values left to return and then it throws us a StopIteration error. 

**Iterating at once with \***

You can also print all values of an iterator in one fell swoop using the star operator, referred to as the splat operator in some circles. This star operator unpacks all elements of an iterator or an iterable. Be warned, however, once you do so, you cannot do it again as there are no more values to iterate through! We would have to redefine our iterator to do so.



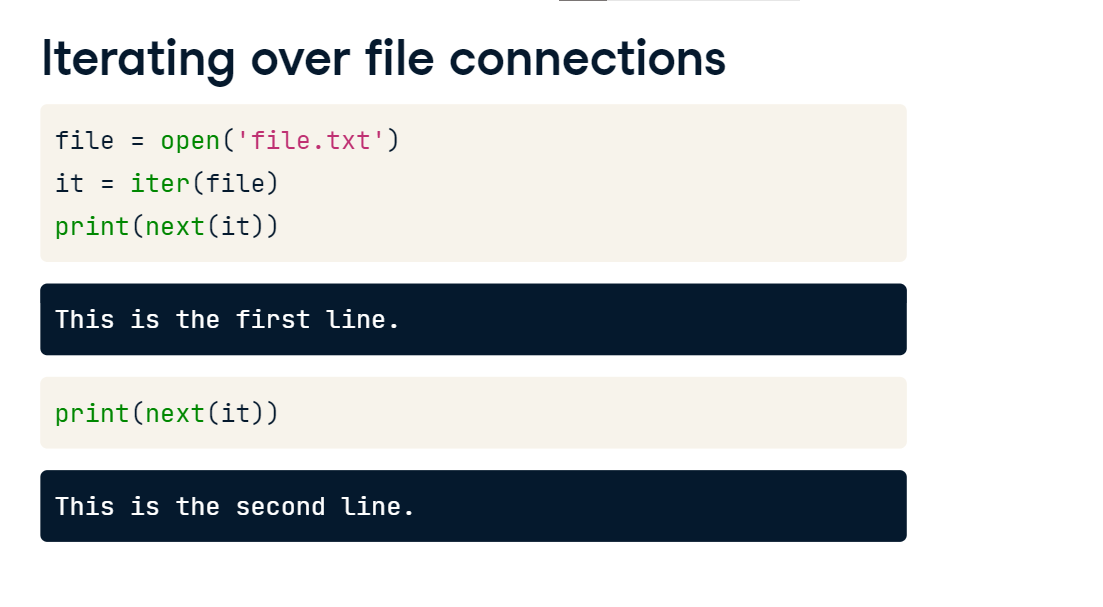
**Iterating over dictionaries**

We mentioned before that dictionaries and file connections are iterables as well. To iterate over the key-value pairs of a Python dictionary, we need to unpack them by applying the items method to the dictionary as you can see here.



**Iterating over file connections**

With respect to file connections, here you can see how to use the iter and next methods to return the lines from a file, file dot txt. This has been your crash course in the fundamentals of iterables and iterators.

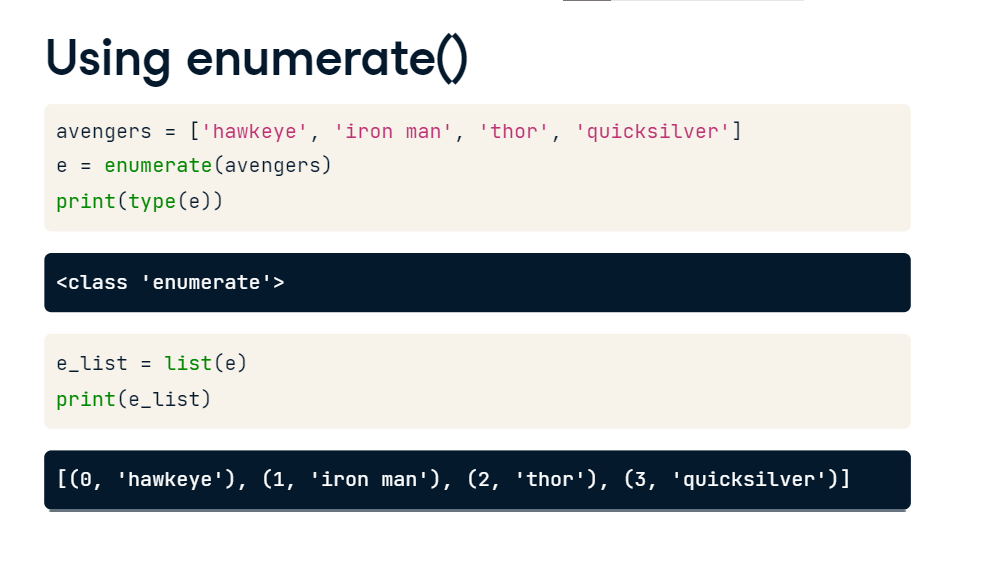


**Playing with iterators**

We're now going to dive a bit deeper into the world of iterables and iterators by checking out some very cool, very useful functions. The first function, enumerate, will allow us to add a counter to any iterable while the second function, zip, will allow us to stitch together an arbitrary number of iterables. Let's begin:

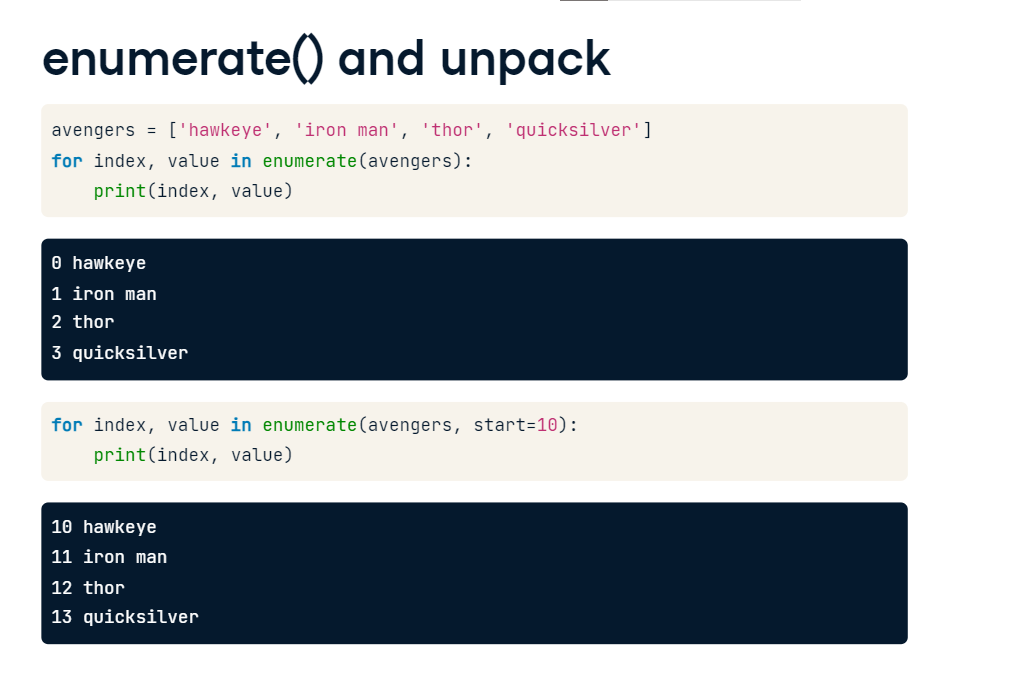
**Using enumerate()**

enumerate is a function that takes any iterable as argument, such as a list, and returns a special enumerate object, which consists of pairs containing the elements of the original iterable, along with their index within the iterable. We can use the function list to turn this enumerate object into a list of tuples and print it to see what it contains. The enumerate object itself



**enumerate() and unpack**

is also an iterable and we can loop over it while unpacking its elements using the clause for index, value in enumerate(avengers). It is the default behavior of enumerate to begin indexing at 0. However, you can alter this with a second argument, start, which you can see here.



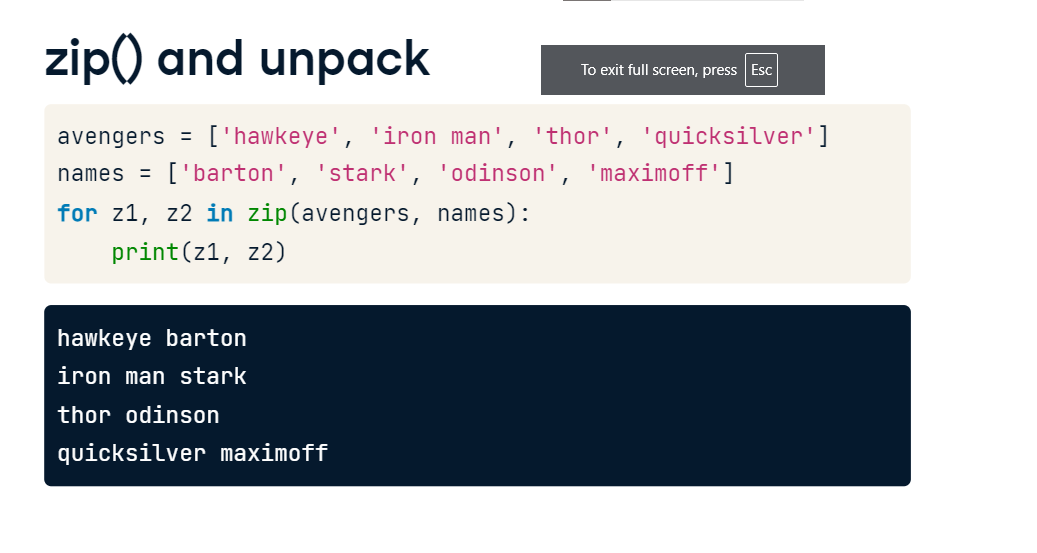
**Using zip()**

Now let's move on to zip, which accepts an arbitrary number of iterables and returns an iterator of tuples. Here we have two lists, one of the avengers, the other of their names. Zipping them together creates a zip object which is an iterator of tuples. We can turn this zip object into a list and print the list. The first element is a tuple containing the first elements of each list that was zipped. The second element is a tuple containing the second elements of each list that was zipped and so on.



**zip() and unpack**

Alternatively, we could use a for loop to iterate over the zip object and print the tuples.



We could also

**Print zip with \***

have used the splat operator to print all the elements! Now it's your turn to play with these useful functions.



**Using iterators to load large files into memory**

Now that you're more comfortable with iterables, iterators and how they work, we're going to check out a particular use case:

**Loading data in chunks**

dealing with large amounts of data. Let's say that you are pulling data from a file, database or API and there's so much of it, just so much data, that you can't hold it in memory. One solution is to load the data in chunks, perform the desired operation or operations on each chuck, store the result, discard the chunk and then load the next chunk; this sounds like a place where an iterator could be useful!To surmount this challenge, we are going to use the pandas function read\_csv, which provides a wonderful option whereby you can load data in chunks and iterate over them. All we need to do is to specify the chunk using the argument yep, you guessed it: chunksize. As with much of what we do in Data Science, this is best illustrated by an example.

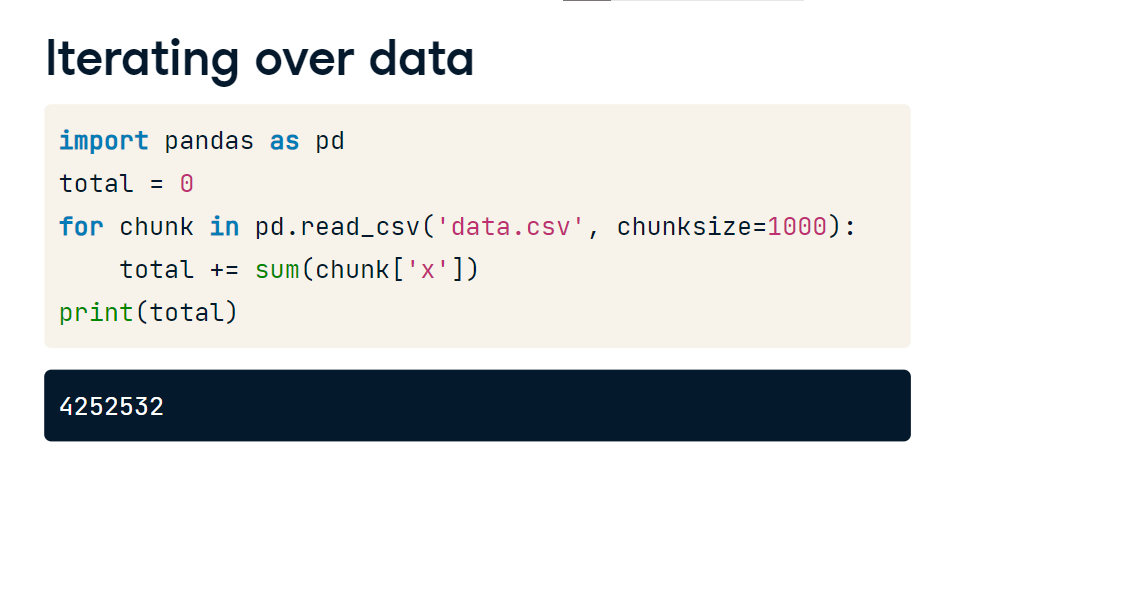
**Iterating over data**

Let's say that we have a csv with a column called 'x' of numbers and I want to compute the sum of all the numbers in that column. However, the file is too large to store in memory. We first import pandas and then initialize an empty list result to hold the result of each iteration. We then use the read\_csv function, utilizing the argument chunksize, setting it to the size of the chunks I want to read in. In this example, we use a chunk size of 1,000. You can play around with it. The object created by the read\_csv call is an iterable so I can can iterate over it, using a for loop, in which each chunk will be a DataFrame. Within the for loop, that is, on each iteration, we compute the sum of the column of interest and we append it to the list result. Once this is executed, we can take the sum of the list result and this gives us our total sum of the column of interest. Iterators to the rescue!



**Iterating over data**

Also note that we need not have used a list to store each result - we could have initialized total to zero before iterating over the file and added each sum during the iteration procedure, as you see here.



What's up next, I hear you ask? Well, that's a great question. We've got some of the most important Pythonista tools in store: comprehensions and generators. These terms may not mean much to you yet, but soon you won't be able to forget them. In fact, you may even want more because they are that useful. List comprehensions, for example, allow us to create lists from other lists or from columns of DataFrames, among many other objects. Why would we want to do this? Check this out, for example. Let's say that we have a DataFrame with a column of time-stamped data and we want to extract the year from it. We could extract the column from the DataFrame, loop over its entries using, for example, a for loop, and within the for loop, extract the desired information. In that case, we would have had to initialize an empty list in which to store the retrieved data. A list comprehension, however, allows us to perform this entire operation, in a single line of code and it's waaaaaaay more efficient than the equivalent for loop code. This is one of a number of Pythonista Data Science chops that you'll learn in the rest of this course.