Formal Definitions:

Iterable (data type) -- an object with fixed representation that can be processed sequentially
Iterator (object type) -- an object that can be used to retrieve values contained in an iterable

Iterables & Iterators Explained:

An iterable is just an object that can be iterated or looped over. A good check to keep in the back of your mind is to see whether it would make sense to create a for loop ('for x in _____') and place our object in the blank. If it makes sense, then this object is an iterable; if not, then it is isn't. So, some examples of iterables we have seen so far would be lists, dictionaries, and tuples. An iterator is an object that allows us to traverse through an iterable. A good way to think about iterators and iterables are using the book and bookmark analogy. Think of an iterable as a book and an iterator as a bookmark. The sole purpose of a bookmark is to keep track of someone's current place in a book and have access to the next pages in the book. Similarly, the sole purpose of an iterator is to act as a bookmark over an iterable, to be able to keep track of its current place and have access to the next elements in the iterable. The iterator is just a bookmark that always starts at the front of an iterable, always knows its current position, and continues to move to the next element when we tell it to until we cannot go forward anymore (at which point a StopIterationError is thrown to indicate the iterator has reached the end of its 'book' and has no more 'next' elements to give).

Useful Functions:

iter(x) -- creates an iterator over/for the inputted iterable x; x must be an iterable
next(y) -- gives the value of the iterable at the iterator's current position and moves the iterator's position to the next value; y must be an iterator
StopIterationError -- returned by next(y) after all items have already been visited; signals the end of an iterable sequentially
list(y) -- puts item of iterator y into a list; y must be an iterator

Example/Functionality Walk Thru:

Start with an iterable, and name it so that we can access it again later.
test = [1, 2, 3, 4, 5]
This list is an iterable because we can loop over it and because it would make sense to place it in a 'for' loop.
Now, we will create an iterator(bookmark) for this iterable and once again name it so that it can be stored for later use.
test_iterator = iter(test)
Now, test_iterator is our iterator and acts as our bookmark for the list test (our book). We know that an iterator starts at the very first element of its iterable, so now test_iterator stores the value 1 (first element of test). We know that test is an iterable so it is a valid input to the iter function which will create an iterator for test. Now, we can get the next element of the list very easily.
next(test_iterator)
The next function returns the current value of the iterator (1), test_iterator, and then updates the bookmark location by moving it to the next element of the list.
We can continue to call next to the rest of the elements of the list (until the last one is reached).
next(test_iterator)
This will return 2 and move the iterator to the next element in the test list.
test[1] = 1;

This updates our list, test, so that now the number at position 1 is 1, however, it doesn't really affect our iterator at all because our iterator has already passed this value. If, however, we had changed the value of the element at position 4, then the iterator would reflect this change because that element hasn't been reached by the iterator yet at this point.

next(test_iterator)

This will return 3 and move the iterator to the next element in the test list.

next(test_iterator)

This will return 4 and move the iterator to the next element in the test list.

test_iterator2 = iter(test)

Even though this iterator is created when the original iterator, test_iterator, is at 5, a new iterator always starts at the beginning of its iterable, so this iterator, test_iterator2, has to start at the beginning of the list, test, and thus currently stores the value 1.

next(test_iterator)

This will return 5 and move the iterator to the next element in the test list.

next(test_iterator)

Now, since there are no elements left to iterate through, this call results in a StopIterationError which isn't necessarily bad, it just indicates that we have gotten through all of the elements in the list. An important thing to note is that we cannot simply "reset" an iterator once it has already gone through all of the elements in its iterable. If you wanted to go through all the elements again, you would need to create a new iterator.

Warm-Up Questions:

Iterator/Iterable/Neither -- Identify the following as one of these options.

1. list = [1, 2, 3]
2. 1
3. dict = {1: 'iterators' , 2: 'are' , 3: 'fun' }
4. iter(1)
5. iter(dict)

Create an Iterator over the list, practice = [1,2,3].

Write code using an iterator that will give the third element of the list, work = ['apple', 'banana', 'pear', 'grape']. (You may use as many lines as you would like)

Look at the code below and next to each line write error if the line would cause an error or for the lines that would return a value, write what python would return.

```
test = ['a', 'b', 'c', 'd']
iter(test)
test[0] = 'f'
next()
test_iter = iter(test)
next(test_iter)
next(test_iter)
```

Deeper Thinking! Here's the cool part. Can an iterator be processed sequentially? In other words, can we loop through all the elements in an iterator? The answer is yes--an iterator is indeed also an iterable!

Deeper Thinking/Fun Fact (pt.2)! The ‘for’ loops that we have long been using have inherent iterators! Just because we don’t see it, doesn’t mean it’s not there!