Sequences, Iterables, For Loops

A sequence is just a list of things in a particular order. Lots of things are sequences:

- a word - a sequence of letters
- a sentence - a sequence of words or a sequence of letters
- a musical performance - a sequence of notes or combinations of notes
- a grocery list - a sequence of things your kitchen is missing

Here are some examples of sequences in Python by type:

```
# a string
'Mr. Jones'

# a list
[4,3,2,1]

# a tuple
(4,3,2,1)

# a list of lists
[[4,3,2,1],[4,3,2,1],[4,3,2,1]]

# a tuple of tuples
((4,1), (3,0), (2,0), (0,1))
```

The Iterable

An **iterable** is a Python object with a special interface designed to let you step through its underlying data in an item-by-item fashion. The mechanics that make this behavior possible are somewhat hidden from plain sight, but they are important to be mindful of and we will cover them in greater detail in a later chapter.

Say you create an iterable called `letters`:

```
letters = ['a','b','c']
```

`letters` is both a list and an iterable and this is because **lists are iterables**.

Iterating through `letters` will give you the values, `'a'`, `'b'`, `'c'`, one at a time, but how can we do this? We need to pick a control structure, which is to say we need a Python looping mechanism. What we want is something that will repeat some bit of code while some condition is true. Let's try the `for` loop.

The `for` Loop

The best way to get acquainted with the `for` loop is to pick a name to refer to each new value in the `letters` sequence. Lets call it `letter`, with no `s`. Then you decide what you want to do with that letter by putting your code in an indented block on the next line. When you put these together in the correct Python syntax, you can do something like this:

```
for letter in letters:
    print letter
#a
#b
#c
```

Or even this:

```
letters = [
    'Dear John, much time has passed since we last spoke ...',
    'Dear Wendy, please find the enclosed photograph you requested ...',
    'Hi Wayne, School is going good. Miss you ...'
]

for letter in letters:
    if letter == 'Dear john, much time has passed since we last spoke ...':
        print '42'

# [ prints ]
# Dear john, much time has passed since we last spoke ...':
```

Each execution of the `for` has access to a new value in `letters` and so each new letter is accessible, whether you use it or not. When there is no next item left, the `for` loop will stop.

The Interface

I said that an iterable has an interface that provides access to data in a certain way. I'll expand on that a little further now. Instead of a list, let's say you have a string called `letters`:

```
letters = 'abc'
```

The code we wrote above will result in the exact same output if we run it on the string version of `letters`:

```
for letter in letters:
    print letter
#a
#b
#c
```

And that is because an iterator is an abstraction of a data type. In other words, an iterator has a contract that defines how the underlying data is treated. So if an object has the requisite properties, it meets the contract, so its data can be accessed through iteration. Things get cooler, as you can make your own custom objects that can have properties like iteration which the language supports directly.

Below, we are trying to iterate through an integer. We get a `TypeError` because we've tried to use a control structure that expects an iterable but we've given it the integer `number`. You can think of a `TypeError` as an error that results from the violation of a contract.

```
number = 5
for n in number:
    print n
# TypeError: 'int' object is not iterable
```

Indexing: getting a value in a sequence

You might want to get a single value out of an iterable as opposed to the entire sequence. One way of doing this is to target this value by its index. Python follows the common convention of using 0-based indexes, so the first element of any sequence is 0.

```
name = 'Mr. Jones'
print a[0]
print a[1]
print a[2]
print a[3]
print a[4]
print a[5]
print a[6]
print a[7]
print a[8]
```

This will print each letter on a separate line. It works fine, but a more reasonable way to do this would be to use a `for` loop:

```
# prints 'name' as a single entity
print name

# prints name, letter by letter
name = 'Mr. Jones'
for letter in name:
    print letter

# and a single character is also an iterable/sequence
letter = name[0]
print letter[0]
```

Strings have a funny property ...

```
# these print the same thing
```

```
print name[0]
print letter

# so do these:

print name[0]
print name[0][0]
print name[0][0][0]
print name[0][0][0][0]
print name[0][0][0][0][0]
print name[0][0][0][0][0][0]
print name[0][0][0][0][0][0][0]
print name[0][0][0][0][0][0][0][0]
print name[0][0][0][0][0][0][0][0][0]
print name[0][0][0][0][0][0][0][0][0][0]

...
```