Unit 6

Learn something new

1. Since Unit 2, we've been opening file objects and reading lines of text from them. Here's the code pattern we've normally used:

```
with open('pap.txt') as book:
    for line in book:
        to process one line of the book
```

   Another way of achieving, exactly, the same effect is this:

```
filename = 'pap.txt'
book = open(filename)
lines = book.readlines()
book.close()

for line in lines:
    to process one line of the book
```

   This approach is a bit more awkward, but it is useful to understand for a reason that will soon be clear.

   The new approach doesn't take advantage of the `with` statement. Using with, we open a file object, give it a name and, in addition, arrange for it to be *closed* when we're done. That is, the connection between the file object—`book` in this case—and the actual file—`pap.txt` —is broken when the `with` statement is completed.

   Here, instead, we explicitly break the connection by calling the file method `close`.
   The new approach also differs in how it gets information from the file. Rather than reading lines one by one, just in time for each to be processed, we use the file method `readlines` to get a complete list of all lines once and for all. Then we close the file and work with the list.

   The new approach doesn't have much to recommend it when we're working with ordinary files. But it does very nicely lay the foundation for reading files that—instead of residing on our computer—are available on the Internet. As an example, here's code to read an online file that contains the full text of Mark Twain's *Huckleberry Finn*:

```
import urllib.request

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

for line in lines:
```

In place of `open`, we've used the `urlopen` function from the `urllib.request` module. And in place of the file name `pap.txt`, we've used the **URL**, or web address, of the file we want.[1] Otherwise, everything stays the same. This means we can use files on the web almost as easily as local ones.

There are a few technical details to take care of though. Let's look at a sample line from the list returned by `readlines`. Here's code to print the line with index 461.

```
import urllib.request

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

print(lines[461])
```

And here's the output.

```
b'course that was all right, because she done it herself.\r\n'
```

The first thing to notice is the letter `b` just before the first quote. This indicates that the line is not a string object, but rather a **bytes** object. When we download information from the web, it may well be an image or sound rather than text. A bytes object is designed to hold raw binary data—the `1`'s and `0`'s used to represent all kinds of digital information—so it's appropriate in this context. In our case, though, we know the binary information is, in fact, text. To convert to a string object, we call the bytes method `decode`. Here's the revised code.

```
import urllib.request

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

print(lines[461].decode())
```

Before running this, though, check the last part of the previous output: '...`\r\n`'. Inside quotes, the backslash character is used in designating characters that would otherwise be hard to represent. For example, `\n` is used for the 'newline' character, the one you get by pressing the Enter key, and `\t` is used for the character you get by pressing Tab.

The statement

```
print('hello\n\thello')
```

[1] URL stands for uniform resource locator. It's a system for specifying where information can be found on the Internet and what method to use to retrieve it.

produces the output

```
hello
   hello
```

Note also that `len('\n\t')` is 2, not 4, since `\n` and `\t` represent one character each.

Apparently, each line of the *Huckleberry Finn* file ends with two of these special characters. Before we use a line, we might like to slice these off. We'll do that in just a moment.

If you look at the original *Huckleberry Finn* file using a web browser, you'll see that the first several lines contain bibliographical information and that the end is a long statement about legal uses of the file. The actual beginning and end of the novel is marked with lines that start with `'***'`. Let's find those lines.

```
import urllib.request

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

index = 0
for line in lines:
    line = line.decode()[:-2]
    if '***' in line:
        print(index, line)
    index += 1
```

The first line in the body of the `for` statement decodes the bytes object into a string and removes the last two characters. As we go through the lines, we count through the indices so that we can report them when we find lines flagged with stars.
The output of this program is:

```
21 *** START OF THE PROJECT GUTENBERG EBOOK ADVENTURES OF HUCKLEBERRY FINN ***
11990 *** END OF THE PROJECT GUTENBERG EBOOK ADVENTURES OF HUCKLEBERRY FINN ***
```

Apparently, the actual novel runs from index 21 up to—but not including—index 11990. If we want to process only lines in the novel, we can slice out the rest as follows:

```
import urllib.request

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

lines = lines[21:11990]

index = 0
for line in lines:
    line = line.decode()[:-2]
```

2. If you've been running the example programs, you've probably noticed that they're painfully slow. That's because each new run starts by reading an entire book from the Internet. This is silly. Once we've read the book once, we shouldn't need to do it again. In fact, we also shouldn't need to decode bytes and strip off extraneous lines and characters each time we want to do something with this text.

   What we'd like is to create a list with just the lines we want, in just the form we want, and then to save it permanently.

   The `shelve` module provides a general facility for saving and reusing Python objects. We create a shelve object—or connect to an existing one—using the `shelve.open` function. Then we use it exactly as if it were a dictionary. To save any object, we just make it the value of some key. The big difference is that ordinary dictionaries disappear when our program finishes executing. Shelve objects **persist** between runs—they remain available for use.

   As an example, let's make a persistent list of the lines in *Huckleberry Finn* so that we can quickly access the list in any new program.

```
import urllib.request, shelve

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

lines = lines[21:11990]

finalLines = []

for line in lines:
    line = line.decode()[:-2]
    finalLines.append(line)

shelf = shelve.open('books')
shelf['Huckleberry Finn'] = finalLines
shelf.close()
```

   In the first line, we import the `shelve` module along with `urllib.request`. Note that you can import as many modules as you like in one line—just separate module names with commas.

   The middle of the program builds up the list we want, starting with an empty list and appending new lines to it one by one. The lines come only from the actual text of the novel and we decode them and slice off the last two characters before appending them to our final list. The last three lines of the program create a shelve object called `books`, save the list we've created as the value of the key `'Huckleberry Finn'` and then close the shelve object.[2]

[2] The last step is not strictly necessary since open connections will be closed automatically when the program is done. But it is good policy. If we don't think to include the code for closing now, we're likely to forget it later if we extend our program so that it doesn't stop at this point.

This program takes some time to run. The payoff is that, after we've run it once, we'll never again need to read and process the webbased file. Here's a program to list lines in the novel that contain the string `'cat'`. Note how fast it runs.

```
import shelve

shelf = shelve.open('books')
book = shelf['Huckleberry Finn']
shelf.close()

for line in book:
    if 'cat' in line:
        print(line)
```

3. In compiling the list of lines for *Huckleberry Finn*, we used a common pattern. We started with an empty list and added new objects to it one by one within a `for` statement. In outline, the pattern looks like this:

```
result = []

for item in target:
    result.append(                    )
```

Python provides a simpler way to accomplish the same task. We just specify the end result we want using a **list comprehension**. Here's the rewritten code:

```
result = [                    for item in target]
```

Applying this streamlined pattern, we can rewrite the program that compiles lines from *Huckleberry Finn*:

```
import urllib.request, shelve

url = 'https://www.gutenberg.org/files/76/76-0.txt'
book = urllib.request.urlopen(url)
lines = book.readlines()
book.close()

finalLines = [line.decode()[:-2] for line in lines[21:11990]]

shelf = shelve.open('books')
shelf['Huckleberry Finn'] = finalLines
shelf.close()
```

Check back and you'll see that the single highlighted line does the work of five lines in the original.

4. We've been working a lot with text. Now let's analyze some interesting numbers. Each month the U.S. government publishes the Consumer Price Index (CPI), the price of a carefully selected "market basket" of consumer goods. When you hear news stories reporting the inflation rate, they're just telling you how this price has risen. Assuming government economists have chosen a group of items for the basket that's representative of what typical consumers actually buy, this rise reflects a general increase in the cost of living.

   A file containing CPI data from 1913 to 2013 is located at the following URL:

   `https://futureboy.us/frinkdata/cpiai.txt`

   If you look at the file, you'll see that it contains a number of header lines followed by the actual data in the following format:

Year	Jan.	Feb.	Mar.	Apr.	May	June	July	Aug.	Sep.	Oct.	Nov.	Dec.	Annual Avg.	Percent change Dec-Dec	Percent change Avg-Avg
1913	9.8	9.8	9.8	9.8	9.7	9.8	9.9	9.9	10.0	10.0	10.1	10.0	9.9		
1914	10.0	9.9	9.9	9.8	9.9	9.9	10.0	10.2	10.2	10.1	10.2	10.1	10.0	1.0	1.0
1915	10.1	10.0	9.9	10.0	10.1	10.1	10.1	10.1	10.1	10.2	10.3	10.3	10.1	2.0	1.0
1916	10.4	10.4	10.5	10.6	10.7	10.8	10.8	10.9	11.1	11.3	11.5	11.6	10.9	12.6	7.9
1917	11.7	12.0	12.0	12.6	12.8	13.0	12.8	13.0	13.3	13.5	13.5	13.7	12.8	18.1	17.4
1918	14.0	14.1	14.0	14.2	14.5	14.7	15.1	15.4	15.7	16.0	16.3	16.5	15.1	20.4	18.0
1919	16.5	16.2	16.4	16.7	16.9	16.9	17.4	17.7	17.8	18.1	18.5	18.9	17.3	14.5	14.6
1920	19.3	19.5	19.7	20.3	20.6	20.9	20.8	20.3	20.0	19.9	19.8	19.4	20.0	2.6	15.6
1921	19.0	18.4	18.3	18.1	17.7	17.6	17.7	17.7	17.5	17.5	17.4	17.3	17.9	-10.8	-10.5

   The first column gives the year and the next 12 columns give the value of the CPI for each of the 12 months of the year. There are some additional columns that we won't be using. Also, note that the last row (not pictured above) gives data for the current year and therefore will list the CPI only for months which have already past.

   If we'd like to work with this data, our first task is simply to read it and store it in a reusable permanent object. Our plan is to create a dictionary called `cpi` with years as keys.
   The value associated with any year will be a list starting with the year again—we'll explain why in a moment—and then the monthly CPI values for the year. For example,

   ```
   cpi[1913]
   ```

   will be

   ```
   [1913, 9.8, 9.8, 9.8, 9.8, 9.7, 9.8, 9.9, 9.9, 10.0, 10.0,
   10.1, 10.0]
   ```

   Note that, in this list, the year is in position `0` and the actual CPI values are in positions `1` to `12`.
   Because of this, `1` corresponds to January, `2` to February and so on, making it easy to access values in a natural way.
   The CPI for March 1942 will be stored in `cpi[1942][3]`: `cpi` is the whole dictionary,

   - `cpi[1942]` is the list associated with the key, the integer `1942`, and
   - `cpi[1942][3]` is the item in position `3` of that list.

Here's a program to compile the dictionary and store it permanently in a shelve object:

```
import urllib.request, shelve

url = 'https://futureboy.us/frinkdata/cpiai.txt'
file = urllib.request.urlopen(url)
lines = file.readlines()
file.close()

cpi = {}

for line in lines:
    items = line.decode().split()
    if len(items) > 0 and items[0].isdigit():
        cpi[int(items[0])] = [float(item) for item in items[:13]]

shelf = shelve.open('cpi')
shelf['cpi'] = cpi
shelf.close()
```

Most of this code should look familiar. The first five lines read the data file from the Internet and store it as a list of lines. The last three lines store the dictionary called `cpi` in a shelve object.

The middle group of lines compiles the dictionary. This code is a bit tricky. We start the dictionary empty and go one by one through the lines in the file, adding an entry for each line, if appropriate. The first step for each line is to convert it from a bytes object to a string (using `decode`) and then to split the string into a list of individual items. The result for a typical line would look like this:

```
['1914', '10.0', '9.9', '9.9', '9.8', ...]
```

Before we use this items list to create a dictionary entry, we need to be sure the line we're processing isn't part of the header that precedes the data or one of the blank lines the separates the data into groups of five years. Our code uses an `if` statement to create an entry only if there is at least one item (ruling out the blank lines) and if the first item is a string consisting entirely of digit characters (ruling out all of the header lines). The condition part of the `if` statement uses the string method `isdigit`, which returns `True` when called on a string of digits.[3] It also connects two separate conditions into one, using and in a natural way.

Once we've determined that we're working with actual data, we need to use it to produce a key and the associated value. The main difficulty is that items holds a series of *strings*; we want to get the associated *numbers*. Converting the first element of the list to an integer gives us our key: `int(items[0])`. This is the integer `1914` in the sample line above.
Note that using an unconverted string would be clumsy. We would have to write `cpi['1914']` rather than just `cpi[1914]` and we wouldn't be able to find get CPI data for ten years later by writing `cpi[1914+10]`.

[3] There are a host of useful string methods, including two we've already seen, `lower` and `split`. If you're wondering where you can find a complete list, check Section 5.6.1 of the Python documentation at: http://docs.python.org/3.0/library/stdtypes.html. You might also like to look at the list methods (including, e.g., `sort`) given in Section 5.6.4, as well as the file methods (e.g. `readlines`) given in Section 5.9.

Once we have the key, say `1914`, we need to produce the value to associate with it. This is a list of numbers, e.g. `[1914, 10.0, 9.9, ...]`. But note that most of these are not integers—they're **floating point numbers**, numbers that may include a decimal part.
The Python function `float` works just like `int`, except that it converts strings to floating point numbers rather than integers. What we want to do, then, is apply float one by one to the elements of `items` and collect the results in a new list. We could do this by creating an empty list and appending new elements to it one at a time, but, as before, a list comprehension is easier and clearer.

The following code specifies a list of floating point numbers derived from the strings in `items`:

```
[float(item) for item in items]
```

A slight variation uses at most the first 13 elements in `items`, so that extra columns that give percentages and averages are ignored.[4]

```
[float(item) for item in items[:13]]
```

And, finally, the actual code in the program uses the result of this list comprehension as the value of the key that specifies the year:

```
cpi[int(items[0])] = [float(item) for item in items[:13]]
```

5. Once the data has been processed into a convenient form and stored in a persistent object, using it is quick and easy. Here's a program that lists years from 1920 to 1990 in which prices fell between January and February.

```
import shelve

shelf = shelve.open('cpi')
cpi = shelf['cpi']
shelf.close()

for year in range(1920, 1991):
    if cpi[year][1] > cpi[year][2]:
        print(year)
```

This program uses the `range` function in a new way. If we pass in two integer values instead of one, `range` generates values from the first up to, but not including, the last. Otherwise, the program is simple and straightforward.

[4] If there are *less* than 13 elements, as there likely will be in the current year, the code still works fine. The second index in a slice just specifies a limit. If there are only seven elements in items, the limit of 13 in `items[:13]` has no effect. All seven items are used.