Files

Like any modern programming language, Python provides good support for reading and writing files. Here's a short overview of the API.

- `open(filename, mode)` returns a *file object*
  - `filename` is a path to a file
  - `mode` is a string where
    - 'r' - open for reading (default)
    - 'w' - open for writing, truncating the file first
    - 'x' - open for exclusive creation, failing if the file already exists
    - 'a' - open for writing, appending to the end of the file if it exists
    - 'b' - binary mode
    - 't' - text mode (default)
    - '+' - open a disk file for updating (reading and writing)
- file objects are *iterable* and support methods to *read*, *write*, and *flush* data as well as *close* the file

Here's a simple example of reading the contents of a file, line-by-line, and printing them back out to the terminal. Notice that because `fin` is iterable, we can use the `for` syntax to iterate over the file one line at a time. The `close` method frees up system resources associated with the open file. This code is compact, clear, and has a small memory footprint because only one line of the file is stored in memory at a time. This style of procedure is known as a *streaming algorithm*.

```
1 import sys
2
3 fin = open(sys.argv[1],'r')
4 for line in fin:
5     print(line, end='')
6 fin.close()
```

If a file is small, you can read the entire contents as a string using `read()`. This is usually a bad idea unless you really want to store the entire contents of the file in memory.

```
1 import sys
2
3 fin = open(sys.argv[1], 'r').read()
4 print(fin.read())
5 fin.close()
```

The preferred way of opening files is to use the `with` keyword in conjunction with `open`.

```
1 import sys
2
3 with open(sys.argv[1], 'r') as fin:
4     for line in fin:
5         print(line, end='')
```

This has the advantage of automatically closing the file for you once exiting the `with` block. This happens even if there is an exception thrown.

Practice

Here is a Python program that prints every odd line of a file. This program uses some really nice Python features. First, there is the `from module import name` syntax. This just allows you to import `name` into the current namespace so that you don't have to use the dot notation `module.name`. Second, there is `itertools`. This is a module that has a bunch of functions for creating *iterators*. Iterators are python objects that sequentially iterate over some structure using a `next` method. Iterators are iterable. This means you can use a `for` loop to scan the sequence. More on this later. The `count(i)` iterator takes a value i and represents the range from $[i, \infty)$. The `zip` function takes k iterable objects and produces a new iterator that returns k-tuples. Each tuple is comprised of the first object in each iterable, then the second, and so on. It stops when one of the iterables runs out of objects.

```
1 import sys
2 from itertools import count
3
4 with open(sys.argv[1], 'r') as fin:
5     for line, lineno in zip(fin, count(1)):
6         if (lineno % 2 == 1):
7             print(line, end='')
```

Tangent: `zip` examples

Here's an example of how `zip` truncates based on the size of its input.

```
>>> a = range(3)
>>> b = range(3,10)
>>> c = list(zip(a,b))
>>> c
[(0, 3), (1, 4), (2, 5)]
```

Interleaving two files

Write a program called `merge.py` that takes two files as input and outputs to the terminal the contents of those files interleaved. For example if `file1` contains

```
1
3
5
7
```

and `file2` contains

```
2
4
6
8
```

then

```
$ python3 interleave.py file1 file2
1
2
3
4
5
6
7
8
```

```
1 import sys
2
3 with open(sys.argv[1],'r') as fin1, open(sys.argv[2],'r') as fin2:
4     for line1,line2 in zip(fin1,fin2):
5         print(line1,end='')
6         print(line2,end='')
```