# Solutions to a Few Problems in Core Python Applications by Wesley J. Chun

John Weatherwax

To my family.

# Introduction

As a final comment, I've worked hard to make these notes as good as I can, but I have no illusions that they are perfect. If you feel that that there is a better way to accomplish or explain an exercise or derivation presented in these notes; or that one or more of the explanations is unclear, incomplete, or misleading, please tell me. If you find an error of any kind – technical, grammatical, typographical, whatever – please tell me that, too. I'll gladly add to the acknowledgments in later printings the name of the first person to bring each problem to my attention.

# Chapter 1 (Regular Expressions)

## Exercises

The exercises for this chapter are implemented in the `python` code `exercises.py` found on web site holding my solutions.

# Chapter 2 (Network Programming)

**Exercise 2-1**

Connection-oriented sockets mean that the two sockets initiate some sort of handshaking or agreement that they will communicate between each other. With connectionless socks at least one of the parties in the communication pair just sends out its information without acknowledgment that the other party received it or understood it. If you have some information that everyone needs to know and that perhaps changes frequently you could broadcast this information on a connectionless socket. If a receiver didn't obtain understand/receive the message another would be sent very quickly and this mode of communication might be good enough for your application.

# Chapter 4 (Multithreaded Programming)

**Exercise 4-1**

Process are what the operating system uses to divide up computational resources. Thus each application program is allocated a process that is then managed by the operating system. Threads are a programming resources available for an application program to/divide up its processing into parts that the program itself can/manage.

**Exercise 4-2**

Python will do better using threads with I/O-bound tasks. This is because during an I/O operation the program will context switch and during that time other threads can run.

**Exercise 4-3**

For general threading, with multiple CPU's each CPU will execute the thread that it views as the most important at the time. Thus I think that these multiple threads could run concurrently. In python due to the limitation of the GIL I believe that one thread will have to halt (by explicitly halting or executing an I/O call) before another thread can be processing. Thus in python there might not be a difference between running a threaded program on a multiple CPU system.

**Exercise 4-4**

Based on how reading a single file from disk is optimized I decided to change this exercise to count the number of occurrences of a character in a given large list of files (say all files that end `*.py`). Then we can use threads to read each file, count the number of occurrences of that character in that file, and add this number to a running total.

**Part (a):** See the python code `character_count_serial.py` where we implement a serial version of the above exercise. Running that code I'm getting

```
python3 ./character_count_serial.py
Starting at Fri Dec 14 22:06:25 2018
Need to read 25 files...
Done! Total number of "c" counted= 332
Ended at Fri Dec 14 22:06:50 2018... took 25 seconds
```

Notice that it took 25 seconds (basically taking a second per file to process each file).

**Part (b):** See the python code `character_count_parallel.py`. In that code we read each file in its own thread and have each thread count the number of time a given character is found inside it. Running that code I'm getting

```
python3 ./character_count_parallel.py
Starting at Fri Dec 14 22:07:31 2018
Need to read 25 files...
Done! Total number of "c" counted= 332
Ended at Fri Dec 14 22:07:32 2018... took 1 seconds
```

The entire process took only one second which shows the power of speeding up an I/O bound application using threads.

## Exercise 4-5

My research seems to indicate that reading the full contents of a file from disk has been optimized extensively since it is the most common use case. Because of that, breaking the processing of a single file (even if very large) into threads will most likely not result in sufficient speed up over just processing that one file in serial. What will provide significant speed up is the threaded processing of multiple files as to read each one during the I/O operation another thread can run. Under that modification this exercise is much like that in Exercise 4-4 (above) but involves changing the/operation we perform on each file. Converting that code to do this task should be simple to do.

## Exercise 4-7

In this exercise we will use a list to hold URLs that the worker threads will have to download and parse looking for more URLs. Any new URLs found are added to this global list. Once a certain amount of time has passed during which no new URLs are found we assume that the total job is "over" and we can exit the application. This exercise is implemented in the `python` code `mtcrawl.py`. There are a large number of improvements that could be made to this code to make it more useful.

## Exercise 4-8

See the `python` code `prodcons.py` where we now have one producer that adds items to the queue and several (three) consumers that remove items. I've made sure that the producer produces a large number of items so that the larger number of consumers will most likely have something to do and not stall the program waiting for the queue to get an item (that might never come). In a real application the consumer might actually find an empty queue and the application designer will need to decide what to do in those situations.

**Exercise 4-9 and 4-10**

These are basically the same as Exercise 4.4 (see also Exercise 4.5) above but changing the operation we perform on each file. Converting that code to do this task would be simple to do.

**Exercise 4-11**

It is a very good exercise to write simple "tester" programs demonstrating the functionality of the `threading` module. Rather than implement these "from scratch" I'll borrow and modify the program code found in [**?**]. In that book, many components from the `threading` module are tested by implementing small "demo" programs designed to exercise their behavior on a small scale. Using these code snippets makes it very easy to experiment and "play" with the given functionality to make sure one understands how they work. In fact I would recommend running (or at least reading) the programs there on any module you are interested in learning more about. The code for that book can be downloaded

```
https://doughellmann.com/blog/the-python-standard-library-by-example/
```

For this exercise I decided to implement/study the following

- TEST

**Exercise 4-12**

I will run `2to3` as

```
2to3 -w candy.py
```

and then

```
mv candy.py candy3.py
mv candy.py.bak candy.py
```

To mirror how the book deals with the different python versions. When I did this it produced a file that ran without any issues.

**Exercise 4-13**

Working with the `candy3.py` file produced in the previous exercise (I didn't rename it) we can access the `_value` attribute of the `BoundedSemaphore` object `candytray`. When this is done the output from this code (for a single individual run) looked like

```
python3 ./candy3.py
starting at: Fri Dec 14 22:14:43 2018
THE CANDY MACHINE (full with 5 bars)!
Buying candy...inventory: 4
Refilling candy...inventory: 5
Buying candy...inventory: 4
Buying candy...inventory: 3
Refilling candy...inventory: 4
Refilling candy...inventory: 5
Buying candy...inventory: 4
Buying candy...inventory: 3
Refilling candy...inventory: 4
Buying candy...inventory: 3
Buying candy...inventory: 2
Buying candy...inventory: 1
Buying candy...inventory: 0
Refilling candy...inventory: 1
all DONE at: Fri Dec 14 22:14:50 2018
```

Due to randomness your results maybe different. We now print the inventory after each `buy` and `refill`.

# Chapter 13 (Web Services)

**Exercise 13-1**

Web services allow the application developer the ability to issue command to a web site and have the responses sent back for further processing/display. They enable the server to provide "services" to connecting computers. Authorization (the root of the word is "authorize") is the process of verifying what you have access to, namely what on the host computer you can access. Authentication is the process of confirming who you are to the remote machine. In other words we provide some sort of credentials that indicate your identity.

**Exercise 13-2**

REST is name of one type of API used that allows clients to connect to web services and issue commands. XML and JSON are formats used to provide data back to the requesting caller. The full state needed for the request is stored by the client and is transferred to the server when the request is made. The RESTful URL's look different (in how they are typed/displayed) than the "old school" URL calls but can functionally perform the same tasks.

# Chapter 14 (Text Processing)

## CSV

### Exercise 14-1

The CSV format is a loosely defined format for storing tabular data. For many years there was no “official” specification of it but is loosely based on items of text separated by commas. It is suited for applications where the amount of data is not huge and can be thought of as consisting of “files”.

### Exercise 14-2

One cannot use `str.split()` when the “items” in the CSV text have commas themselves in them. In that case the entries are normally quoted as

```
v1,v2,comment
2,3,"some text"
4,5,"some text, and more text"
```

In the above example splitting on a comma would not give the correct tokens.

### Exercise 14-3

I found a `csv` file to practice loading using the `csv` module and then to print all columns with uniform widths.

### Exercise 14-4

One can use the `csv` reader with a different delimiter by specifying the `delimiter` argument to the `csv.reader` or `csv.writer` calls. See the associated code for this exercise for examples of reading `csv` files with delimiters of colons (like an `/etc/passwd` file).

## JSON

### Exercise 14-5

JSON is a string based format for exchanging data between applications. A python dictionary is a data structure that can be used in the language to perform fast key value look-ups. When printed the string representation of a python dictionary looks like a JSON string.

### Exercise 14-6

See the code in `lort2json.py` where we convert lists or tuples to `json` object strings. We use the `json` module's `dump` function to perform this function.