Handson - MapReduce

Question 1:
The first parameter is the number of map tasks.
The second parameter is the number of reduce tasks.
The number of map tasks splits up the inputs into that many temporary files, and the number of reduce tasks splits up the work into that many different processes.

Question 2:
run() is called by WordCount's superclass — MapReduce class. run() calls doMap() method for the map tasks and then calls doReduce() method for the reduce tasks. The doMap() method works by taking in the split input and calling Map() method defined in WordCount class, and then dumping the results i nto a file. After that, doReduce() method is called, it takes in the files created by doMap() method, and calls Reduce() method defined in WordCount class on the files.

Question 3:
The keyvalue is the byte offset from the beginning of the file. It's the key of the file portion the Map() method is run-ing on.
The value is the portion of the file where the Map() method is actually running on.

Question 4:
The key is the word that is found in the file that we want to count.
The keyvalue is a list that contains many tuples indicate how many times the word occurred
in the file, for example ('key', 1).

Question 5:
The current set up has us run doMap 4 times and doReduce 2 times because when initialization we set maptask as 4 and reducetask as 2.

Question 6:
All of the doMap() jobs run in parallel and of course all the doReduce() jobs run in parallel too. This is the case because the code that calls these jobs spawns enough cores or processes to processes all these iterations in parallel.

Question 7:
A single doMap() processes about 1208690 bytes of input, but this number changes by a couple bytes because the inputs are split at the first whitespace character that is greater than the chunk size of 1208690.

Question 8:
A single doReduce() processes about 2250 keys, but this number is not exact and depends on the number of title words in the associated map jobs for each doReduce() job.

Question 9:
I found that having about a medium or reasonable amount of jobs for both provided the best overall speed. I think the reason behind this is that once the jobs get too big, it runs much slower because we are limited on the number of cores that the process runs on.

Question 10:

```
# Produce a (key, value) pair for each title word in value
def Map(self, keyvalue, value):
        results = []
        i = 0
        n = len(value)
        while i < n:
            # skip non-ascii letters in C/C++ style a la MapReduce paper:
            while i < n and value[i] not in string.ascii_letters:
                i += 1
            start = i
            while i < n and value[i] in string.ascii_letters:
                i += 1
            w = value[start:i]
            if start < i and w.istitle():
                results.append ((w.lower(), 1))
        return results

# Reduce [(key,value), ...])
def Reduce(self, key, keyvalues):
      return (key, sum(pair[1] for pair in keyvalues))
```