CS 121, Spring 2014

wmr — Additional problems: WebMapReduce

1. Type in the files `wcmapper.py` and `wcreducer.py` that are shown in the handout, and use WebMapReduce (WMR) to apply the mapper an reducer from those files to count word frequencies, as described in that handout.

2. Run the program `wmrtest` with `wcmapper.py` and `wcreducer.py`. Does it produce the desired output for input you provide?

3. Type in or copy the files `idmapper.py` and `idreducer.py` from the `∼cs121` directory. To copy them, you can enter the following Linux commands.

   ```
   % cp ∼cs121/idmapper.py .
   % cp ∼cs121/idreducer.py .
   ```

   Be sure to include the final isolated "dot" in each of these commands, since that dot represents where to copy the files (namely, to your directory).

   Once you have those two files, run WMR using `wcmapper.py` with `idreducer.py`, in order to see the output from the word-count mapper, as described in the handout. Also run `idmapper.py` with `wcreducer.py` as described in that same handout.

4. (a) Create some different example data than the fish example discussed in the handout. (b) Then *predict* the output from `wcmapper.py` for that data. (c) Finally, test your prediction using WMR, by running `wcmapper.py` with `idreducer.py`.

5. (a) Write down some key-value pairs consisting of words for the keys and integers for the values. Include some duplicate words among your key-value pairs; mix up the words, so they are out of order. (b) Then *predict* the output if your list of key-value pairs is shuffled into the word-count reducer `wcreducer.py` . (c) Finally, test your prediction using WMR, by entering your list of key-value pairs as data and running `idmapper.py` with `wcreducer.py` .

6. The word-count example in the handout counts the words "`fish`" and "`Fish`" as different words, because they are different strings. Modify the mapper and/or reducer to lower the case of all the words, yielding a combined count for the word "`fish`".
   Example input:

   ```
   The cat in the hat
   In the Hat!
   ```

   Desired map-reduce output (`TAB` separated):

   ```
   cat     1
   hat     1
   hat!    1
   in      2
   the     3
   ```

   Notes:

   - Start by making a "map-reduce plan" as described in the handout, deciding what the mapper will produce and what the reducer will produce. Be sure to decide which of these stages should change the case of the word.

7. The Python3 string class has methods `lstrip()` and `rstrip()` for removing all appearances of specified characters from the left or right end of a given string. Examples:

```
’www.example.com’.lstrip(’czmow.’)  --> ’example.com’
’mississippi’.rstrip(’ipz’) --> ’mississ’
```

Here, all characters `c z m o w` or `.` that appear at the *left* end of the string `’www.example.com’` were removed, one at a time, until none of them was left; note that the `m`’s were *not* removed, because they never appeared at the left end.

Modify `wcmapper.py` and/or `wcreducer.py` to produce a list of words after lowering the case *and* removing all punctuation from both ends of those words.
Example input:

```
The ‘‘cat,’’ in the hat--
In "the" Hat!
```

Desired map-reduce output (`TAB` separated):

```
cat     1
hat     2
in      2
the     3
```

Notes:

- As before, start by making a “map-reduce plan” that indicates what the mapper will produce and what the reducer will produce. Be sure to decide which of these stages should change the case of the word, which should remove the punctuation, etc.

8. Some industry programming tasks call for using a *programming framework* system. With a framework, most of a program can be reused, and a programmer only supplies one or more segments of code in order to affect the computation.

For example, a *web-app framework* makes it convenient to produce a web-based software program. A programmer provides three kinds of code segments: code for the *model* (the data needed by the webapp); code for the *view* (i.e., the web pages seen on the user’s browser); and code for the *control* (which governs how the program reacts to user interactions such as button clicks and text entry). After providing code segments for the model, view, and control, a programmer can reuse well-tested code for implementing webapps, thus saving time and avoiding mistakes.

As a simple example of a framework, consider the (locally written) program `netflix`, which supports computations with data about movie ratings. By default, this program acts on lines of data having the following format:

movieID,reviewerID,rating,date

Example:

```
6,110641,5,2003-12-15
```

Our `netflix` framework requires two code segments.

- A function named `extract()`, for selecting data from a line in `netflix.dat`.

`extract`

1 Argument: A string from `netflix.dat`, containing data about a single movie rating.

State change: The function `extract()` calls a predefined function ¡code¿emit()¡/code¿ to produce a string value accoerding to the data *arg1*.

Return: None.

- A function named `reduce()`, for combining values `emit`ted by `extract()`.

`reduce`

1 Argument: An iterator (suitable for controlling a `for` loop) that provides a series of string values.

State change: The function `reduce()` calls a predefined function ¡code¿emit()¡/code¿ to produce string values, based on the values in the iterator *arg1*.

Return: None.

Both of these functions `extract()` and `reduce()` deliver their results using a function `emit()` that is predefined within the `netflix` framework.

-

`emit`

1 Argument: A string.

State change: The string value *arg1* is forwarded to the framework for further processing.

Return: None.

The function `emit()` represents a new way to deliver results from a function `extract()` or `reduce()`, which is neither a return value nor printed output.

As an example of using the `netflix` framework, consider the following problem: *Determine the highest rating among all movie ratings in* `netflix.dat`. (We expect the answer 5, since the 100 movies in `netflix.dat` are rated between 1 and 5.)

Here are `extract()` and `reduce()` definitions that solve the problem using the framework.

```
def extract(data):
    lis = data.split(',')
    emit(lis[2])

def reduce(iter):
    max = -1
    for v in iter:
        val = int(v)
        if (max < val):
            max = val
    emit(str(max))
```

Comments

- The argument `data` of the `extract()` function holds *one line* of data from `netflix.dat`, representing a single movie rating as a string. After `split`ting that string into a list according to the delimiter character `','`, the `extract()` function delivers its results to the framework using `emit()`. For this problem, the value to deliver is the rating number for the movie, which is the third element of the list produced by `split()` (index 2).

- The function `reduce()` has an iterator argument that yields all of the movie ratings produced by calls of `extract()`. A `for` loop visits each of those ratings in turn. These ratings arrive as strings `v`, not numbers. Since we want the maximum number (highest movie rating), we convert `v` into an `int` value `val`, then update the variable `max` if this rating `val` exceeds the value of `max`. After the `for` loop, the variable `max` must hold the highest numerical value among all movie ratings in `netflix.dat`, which is the value we sought in this problem. Finally, we use the framework's `emit()` and the type-change function `str()` to deliver a string version of that desired value.
- The `netflix` framework is responsible for (a) retrieving lines of data from `netflix.dat`, (b) calling `extract()` with each of those lines of data, (c) collecting the values produced by those calls via `emit()` into an iterator, (d) passing that iterator as the argument for a single call to `reduce()`, (e) receiving the value produced by `reduce()` via `emit()`, and (f) printing that result on output. By using the framework, we avoid having to program these steps ourselves, which takes advantage of code reuse.

DO THIS: Verify that the solution above solves the maximum-rating problem by using the `netflix` framework on a Link machine, as follows.

a) Create a file `code.py` (or use another name if you prefer) that contains the definitions of the two function `extract()` and `reduce()`. (No `import` or other context is required, just the two function definitions as above.)

b) Run the program `netflix` with that file `code.py` as a command-line argument *on a link computer.*

```
% netflix code.py
```

(The framework takes care of finding the data file `netflix.dat`, calling the functions `extract()` and `reduce()` at the right times with the right argument values, etc.) Note that you may need to `source  rab/css-setup` in order to access the program `netflix` .

c) The output from that program `netflix` should be a single number 5, representing the highest movie rating among the 100 ratings in `netflix.py`.

9. Use the `netflix` framework to determine the highest (numerical) reviewer ID among the movies in `netflix.dat`.

10. Use the `netflix` framework to determine the *average* (mean) rating among the movies in `netflix.dat`.

    Hint: Instead of using the `for` loop within the `reducer()` for determining a maximum value, use that `for` loop to determine both the count of values seen and a sum of the values seen so far. Then, use `emit()` to deliver a string representation of the quotient of that count and that sum.

11. Use the `netflix` framework to determine the average (mean) rating *for each movie* in `netflix.dat`.

    Hints:

    - The `extract()` function will have to emit not only each movie's rating but also its movieID, in order for the `reduce()` function to have enough information to compute averages *per movie*. These must be emitted together, so pack them into a single string, separated by a comma or space character
    - In `reduce()`, you must maintain a separate count and sum for *each movie*. One way to store these values uses two lists, `count` and `sum`, with nine elements each (This is enough, because the movieIDs that appear in `netflix.dat` happen to be between 1 and 8 inclusive. By allowing for 9 elements in the lists, you can conveniently use a movie id as an index for those lists). Initialize all nine elements to zero in each of the two lists.
    - Use two `for` loops in the `reduce()` function, one for tabulating the `count` and `sum` lists according to the values from the iterator, and one for `emit()`ting the resulting averages.

- In the first loop of `reduce()`, each string `v` from the iterator has a comma or space separator (which was inserted during an *extract()* call). Use `split()` to remove that separator and produce a list of two strings, then convert both of those strings to integers using `int()`. One of these integers will be the movie id, and the other one will be the rating. Use the movie id as an index for `count` and `sum`, and update both arrays appropriately.

- In the second loop of `reduce()`, you'll be ready to `emit()` all the movie-rating averages. You may end up emitting as many as 8 averages in this loop. *However*, use `if` to check that the count for each movie is non-zero before attempting to emit that movie's average, in order to avoid division by zero. If a movie didn't receive any ratings, just skip it.

12. Use the `netflix` framework to determine the lowest (numerical) reviewer ID among the movies in `netflix.dat`.

13. Use the `netflix` framework to determine the latest date among ratings in the `netflix.dat` data set.

    Note: Unlike comparisons in the other fields of `netflix.dat`, *do not* convert the date field into an integer for comparison! The date field does not convert directly to an integer, because of hyphens used to separate the year, month, and day.

    Instead, compare two dates from `netflix.dat` *as strings* using a comparison operator such as `<`. The ordering of dates in the format used in `netflix.dat` agrees with their ordering as strings using `<` or other comparison operators, so `<` can be used in a loop to determine the latest date.

14. Use the `netflix` framework to determine the earliest *and* latest dates among ratings in the `netflix.dat` data set.

    Hint: Try modifying your code for determining the latest date to keep track of *both* the earliest and latest dates, instead of keeping track of only the latest date.

15. Use the `netflix` framework to count the number of lines in the file `netflix.dat`.

    Hint: One approach is to simply `emit()` the string `"1"` once for each call of `extract()` (no matter what the contents of that line may be), then in `reduce()` to convert each `"1"`s into an integer and add those integers.