q for smarties 2

Files and Interprocess Communication Accessing files is very easy though the syntax may look unfamiliar. Here for example is the code to number each line in a file.

```
/  Numbers the lines in a file
/ q number  filename


/ APPLICATION

/ Does the numbering
myparse:{[line]
  linenum+: 1;
  :(string linenum), (": "),(line)}

/ EXECUTION

if[0 = count .z.x; til -3] / should have a file argument
file: .z.x[0]

linenum: 0 / global variable
a: read0 ` $ (":"),file / read the file

(` $ (":"),file,("out")) 0: myparse each a / parse and output the ↩
   file
```

q number.q file

Points to understand:

1. How to read in a file.
2. How globals work.
3. How to construct an output file name.
4. How to do the output.

Exercise 1. (Easy) Read in a file and count the number of characters per line then output the numbers in the form 1: 23 2: 0 ...

```
    q countchar.q <filename>
```

Now we might want to parse all the words in the file (where words are delimited by blanks) and count the number of instances of words.
Let's work up to this.
We will have the same basic structure in which we read in one line at a time and we parse it. Only this time, parsing it will consist of separating the words based on spaces.

Consider how to parse a line so we can get just the words:

```
/ deletes all blanks
delblanks:{[x]
 ii: where not x = " ";
 x[ii] }

/ separate into words
sepwords:{[x]
  y: (" ")vs x;
  c: count each y;
  ii: where not c = 0;
  y[ii]}
```

Points to notice:

1. Use of *vs*
2. Use of *count*
3. Need for 0 to eliminate blanks

Exercise 2: (Medium) Parse all the words in the file (where words are delimited by blanks) and find all distinct words.

Exercise 2a: (Medium) Parse all the words in the file (where words are delimited by blanks) and count the number of instances of words. Display the pair (word, number) for all words having at least 3 instances in descending order by count.

```
    q wordcount.q filename
```

In the following exercise, we read a q program and try to find elementary syntax errors.
A q program needs semi-colons within its procedures except on the last line of a procedure or if we're inside parentheses. Note that we don't want to put the semi-colon after the comment in a line or on blank lines.

Hints:

```
line: "5 + 2" / this is an addition
(line til 6),("; "),(6 _ line)
jj: line ss " / "
```

Exercise 3 (Middle). Read in a file and check that it has semicolons where needed and add them.

```
q findmissing.q test.q -- Adds in semi-colons where needed
```

Client-server communication Doing client-server communication is easy.
Let's think about the information that is required.
The server has to establish an address.
The client sends to that address.
The client has to make a few decisions:

1. which function to call on the server
2. whether the call is
   (a) synchronous (client waits for the response before proceeding)
   (b) or asynchronous (client goes on).

For example, here is a simple server:

```
/ q simpleserver.q -p 1111

/ Returns twice the number sent in
simple:{[num]
  2*num}
```

And here is a simple synchronous client:

```
h: hopen `::1111; / opens the port on the current machine
h"simple[9876.5]" / calls the function with an argument
```

Here is an asynchronous client:

```
h: hopen `::1111;
neg[h]"simple[9876.5]"
```

The trouble with the asynchronous client is that it doesn't return anything. So we will need to keep state in the server and call for it using a synchronous call. Let's see if you can figure that out.

Exercise 4. (Easy) Communicate from a client to a server where the server computes the present value of an income stream given as a list given a certain interest rate value. Use synchronous calls.

```
    q presvalserver.q -p 1234
    q presvalclient.q
```

Exercise 4 async. (Easy) Communicate from a client to a server where the server computes the present value given a certain interest rate value for four different asynchronous calls to presvalue e.g.

```
neg[h]"presval[6.5; 100 200 100 300]"
neg[h]"presval[6.0; 100 200 100 300]"
neg[h]"presval[5.5; 100 200 100 300]"
neg[h]"presval[5.0; 100 200 100 300]"
```

The client then makes a synchronous call to collect all these results.

```
    q presvalserverasync.q -p 1234
    q presvalclientasync.q
```

Communicating to many servers is just a matter of declaring those servers and then using the appropriate handles. Note you can do e.g.

```
\sleep 3
```

if you think the server will take time.

```
    q presvalclientasyncwithdelay.q
```

Exercise 5. (Middle) Communicate from a client to three servers to compute the present value assuming a variety of interest rates.

```
    q presvalserver.q -p 1000
    q presvalserver.q -p 1001
    q presvalserver.q -p 1002
    q presvalclientmulti.q
```

One way in which we might wish to parallelize jobs is to have a reliable client which has a queue of jobs (jobqueue) and acts as client to a bunch of server.
Each time the client dispatches a job to a server it puts the job on the back of the jobqueue.

Periodically, the client queries each server to see which jobs the server has done.
The client reports these in its answer queue.
The client eliminates all jobs that are in its answer queue from its jobqueue.

pseudo-code of client:

```
do several asynchronous calls to the servers
 for jobs that haven't completed
do synchronous calls to servers to get results
 and cause the server to reset its answer buffer
```

pseudo-code of server:

```
put the result of each call on its answer buffer.

    q presvalserverasyncslow.q -p 10001
    q presvalserverasyncslow.q -p 10002
    q presvalserverasyncslow.q -p 10003
    q presvalclientasyncslow.q
```

Catching Errors Suppose we have a server function that performs a certain function presval but a client calls another function foobar. How do we prevent the client from crashing.

```
  q presvalserver.q -p 1234
  q carefulclient.q
```