ChatScript Document Reader

© Bruce Wilcox, gowilcox@gmail.com brilligunderstanding.com

Revision 7/3/2014 cs5.5

Reading Documents

Normally the system reads a line of input from the user and responds to it, completing a volley. But there is another style of input available, called document mode. You cannot intermix the two (you can't ask for something, have the system dynamically read a document, and then reply).

The `:document` command is given the name of a document or directory of documents.

```
:document myfile.txt
:document mydirectory/
```

The document will automatically be broken into sentences and fed to the system one at a time. The system will execute the prepass once at start (the prepass of the bot), and the postpass once at end (the postpass of the bot). All sentences of the document are treated as a single volley and when the document is completed, the system reloads the user's prior state. So you can't store stuff except external to ChatScript (e.g., a file using `^export` or `^log`).

Optional arguments after the file/directory name are:

`echo` – the lines after preparation (spell check etc) will be dumped to TMP/out.txt so you can see what it read.

`single` – read a single line at a time, even if it is not obviously a complete sentence. Multiple sentences on a line are still separated.

`stats` – at each document completion show memory and time used

Prior to reading a document, the system will execute a preprocess topic `~document_pre`, if you define it. When it is done reading a document the system will execute a postprocess topic `~document_post`, if you define it. Such topics are always invoked in gambit mode, so should consist of gambits and not responders.

`%document` is a system variable you can use to tell if you are in document mode or not, to make your main pass topic act conditionally on whether you are in

document mode or not. And the current document path is stored on the variable `$$document`.

You can analyze a document using standard ChatScript patterns, including those based on pos-tagging and parsing. The pos-tagger/parser is very much a work in progress and will continue to improve with future releases, but at least you can tell if it completely accounted for everything in its parse because %parsed will be true if it did.

If you are not careful, it is easy to have document reading not work properly. The intent of reading a document is to gather information, not to generate output on each sentence.

So in general it is inappropriate to be generating output while reading the lines. The Harry control script is set to call for a gambit if no response is generated or to say I don't know what to say if all else fails. That's not good, so you'd need to remove all the control script code for generating gambits. You only get one such message no matter how many lines you read because the Harry script also only generates one response before shutting down.

If you did want to generate responses on the fly while reading (which won't be transmitted by a server but would end up in logs), you'd need to revise the control script so that it could react more than once. And you must remember that generating output stops further rules from firing unless you explicitly use `^fail(rule)` after the output.

If you want stuff displayed to the user at the moment something is happening, use `^log(OUTPUT_ECHO xxxx)`. Normal output will be displayed all at once when the document is finished being read. Should you generate more than 20 replies during a document, all later replies will be discarded.

But typically you'd do something after the document was finished, which means doing it from a post-process topic and perhaps using ^postprintafter to issue your message. An example of a filter script for simple wikipedia data mining (using the output of `:wikitext`) is this:

```
#### establish the default bot
table: defaultbot (^name)
    ^createfact(^name defaultbot defaultbot)
DATA:
reader

#### define the bot
outputmacro: reader()
    $control_main = ~main_control
    $token = #DO_ESSENTIALS | #DO_CONTRACTIONS | #STRICT_CASING |
             #NO_HYPHEN_END | #NO_COLON_END | #NO_SEMICOLON_END | #DO_PARSE

#### define the bot control scripts
```

```
topic: ~document_pre system repeat()

t: $$linecount = 0

Log(OUTPUT_ECHO \n Begin $$document ) # instant display

topic: ~main_control system repeat () # executed for each sentence of input

# on startup, do introduction
u: (%input<%userfirstline) Reader ready.

u: (%document)
    # give confirmation echo that all is progressing periodically
    $$linecount += 1
    $$tmp = $$linecount - 1
    $$tmp1 = $$tmp % 1000 # every 1000 lines
    if ($$tmp1 == 0) {Log(OUTPUT_ECHO .)}
    respond(~filter)

topic: ~document_post system repeat() # tell him we are done with this document

t: ^postprintafter(done.)

# and the filter script:
topic: ~filter system repeat ()

# input is just a blank line
u: (%length=0) FAIL(TOPIC)

#! [ out of band info ]
u: ( < \[ _* \] %more ) $$kind = _0 ^nextinput() ^refine()

    # input is a wiki title designator
    a: ( $$kind=title: _* ) $$title = '_0 LOG(FILE tmp/is.txt \n ***title: '_0 \n )
        FAIL(topic)

    # input is a wiki heading designator
    a: ($$kind=heading: ) $$heading = '_0 LOG(FILE tmp/is.txt \n heading: '_0 \n )
        FAIL(topic)

    # input is a wiki category designator
    a: ( $$kind=category: ) FAIL(TOPIC)

    # input is all others like [*]
    a: () FAIL(TOPIC)
```

```
# dump inputs based on parse status
u: (_*) refine()
    # successfully parsed items
    a: (%parsed ) LOG(FILE tmp/parsed.txt '_0 \n )

    # partially parsed items
    a: (<< ~mainsubject ~mainverb >>) LOG( FILE tmp/semiparsed.txt '_0 \n )

    # not parsed items
    a: () LOG( FILE tmp/unparsed.txt '_0 \n )

# find all X is a Y statement for definitions
#! a dog is a cat
# u: ( _* be ['a 'an the] _*) LOG(FILE tmp/is.txt '_0 ^" |" be ^"| " '_1 \n )
```

Memory management with documents

Normally you don't have to pay attention to memory management in ChatScript. It starts out with a pool of available facts, dictionary nodes, and text string space (which you can control) and uses and releases that as appropriate after each volley. But when you can read all of Wikipedia as one volley (since everything read by `:document` is a single volley), all bets are off. When you run out of preallocated memory, you die.

As scary as that sounds, it will likely not be a problem. CS automatically cleans up, releases used memory, and writes out the current user's state into his topic files at the end of every document read (eg when reading a directory of documents). CS can easily handle a 10MB document file in Windows. With Linux I understand memory allocation much less and have to reserve a smaller memory pool. Maybe there is some way to raise the default process memory limit that I don't know about. If you know how to use more space in Linux, please let me know.

However, if for each sentence in a document you create facts, then those facts will get written out into the user topic file and read back in again. This means at some point the topic file will get bigger than is allowed, or you will be accumulating too many facts. If you want to generate a lot of facts, you need to write out your facts into a separate file. Those facts should become part of the base system that gets loaded in on startup, not part of the user's own fact space. I will try to remember to document how to do that at some point in the future. Otherwise, you can write your data using ^log into files for some other future use.

If you want to monitor how big a document you can manage, use the stats parameter to tell you after each document how much was used up and how much was still available. See also `^memorymark()`/`^memoryfree()` for controlling

memory use. And be aware that setting user variables uses memory.

If you set a variable like `$$linecount` for every sentence you read, you are using up text memory on every assignment. You can reduce this burden by using the match variables like _19 whenever you can. Match variables above _10 are unlikely to ever be used in your patterns, so they can hold temporary data in a way that requires no additional memory per assignment.

Match variable are initialized at startup of ChatScript and hold their values until explicitly overwritten by script. Even if your bot does nothing while reading, you may find it useful to use `:document` with echo, and get every sentence of document, one-per-line, written out for some other tool.

Debugging Documents

You can place any `:xxx` command at the start of lines in a document file and the system will treat that line as a command and execute it.

`:wikitext` inputfile outputdirectory sizelimit

The ultimate document to read, the largest fact document in existence, is Wikipedia.

Data-mining it for information is something various people are interested in. ChatScript makes it easy to use all of ChatScript's abilities to do that. Of course, Wikipedia starts out in some arcane syntax notation, which is unsuitable for natural language analysis.

Therefore, ChatScript comes with a transcoder that reads that in and spouts out only relevant text and with special out-of-band annotations useful to ChatScript. `:wikitext` reads files written in MediaWiki fomat (e.g., a dump file of Wikipedia or Simplepedia). You can get a wikipedia dump via:

`http://en.wikipedia.org/wiki/Wikipedia:Database_download`

Currently it's about 10GB compressed and 50GB uncompressed. ChatScript only reads an uncompressed version. You name the source file path (relative to ChatScript). This can be a file or a directory ending with /. If the latter, the files will be named file0.txt and upwards, as generated by a prior `:wikitext` command using SPLIT.

Simplepedia has an earlier dump and I don't know where later ones are:

`https://archive.org/details/simplewiki_20100610`

Be advised that ChatScript supports UTF-8, so if you read in ANSI or UTF-16 or whatever, all foreign characters may well be wrong. I don't find this a problem for reading english wikipedia because I'm not looking much for foreign data,

but trying to read any other language wiki without insuring it is first in UTF-8 would be bad. The output directory is where you want output files to be placed.

Various OS's and hardware have limits on how large a file size can exist. And I find I like to be able to look at the converted files at times, and my editor doesn't do well with big files, so I tend to generate multiple files. That's where the size limit comes into play.

Based on the size limit, the system will generate just the useful text portion of the input. Starting with file0.txt, whenever a wiki page overflows the size limit it will complete and CS will increment the file number for the next batch. The sizelimit is in kilobytes, so 1 = 1000 bytes, 1000 = 1 megabyte. I usually set my size limits for around 10MB.

Processing all of wikipedia in a single round will take a while. And I like to consult the original from time to time, to see if CS made a transcription error or the original author made a formatting error. So if you have a machine and hardware which can handle the full file, you can add SPLIT at the end of the :wikitext command. It will not convert the input to text, it will merely split the original xml into multiple files based on the size limit. So this series:

```
:wikitext wikipedia.xml X 10000 split
:wikitext X/ Y 10000
```

would read the full wikipedia and break it into smaller 10Mb files. Then read all the smaller files and put the output text into Y. Then, one would likely do :document on the Y directory.

Similarly I set size limits on the output files of translation, because using :document on all of wikipedia will take a lonnnnng time, and it's handy to be able to break off and not have to restart at the beginning.

The text converter outputs some special tags that you can make use of while using `:document`. And, in fact, if you are going to write a document bot to process wiki data in some way, you owe it to yourself to go read an outputted file0.txt file to see what it looks like.

Each wikipedia page as a title, output as out-of-band header plus data plus a closing period to stop the document reader from continuing onto more text to try to make a sentence.

```
[ title ] Paul Newman .
```

The actual text begins with

```
[text ]
```

and may have section headers also terminating in periods, e.g.,

```
[ header ] Early life .
...
[header ] Later years .
```

Whenever bullet points exist in the wiki, those will be reflected as: `[ * ]` this was bulleted text The categorization information is output as:

```
[category ] Movie Actors .
```

^memorymark()

Reading a document consists of performing a single volley of the entire document. This can tie up a lot of memory in keeping facts, dictionary entries, user variables, etc. If you are careful in what you do, you can make the memory burden go away. `^memoryMark()` notes where memory is currently at, and is best done within the document_pre topic. Then you can release memory after every sentence of the document, so it doesn't accumulate.

^memoryfree()

This releases memory back to the last ^memorymark(). It is best done after your main control of the document bot has finished processing a sentence. E.g.,

```
topic: ~document_pre system repeat()
t: ^memorymark() # note start
  Log(OUTPUT_ECHO \n Begin $$document ) # instant display

topic: ~main_control system repeat () # executed each sentence of document
u: (%document)
  respond(~filter)
  ^memoryfree()
```

The caveats and warnings about how this works. Whenenver you free memory, the system will clear all fact sets. It will clear all user variables set after the memory mark (leaving the ones before alone). It will then release facts, text, and dictionary nodes created after the mark.