ChatScript Analytics Manual

Copyright Bruce Wilcox, gowilcox@gmail.com Revision 1/31/2018 cs8.0 Your bot has been written and debugged and released. You are getting log files from users. What can you learn from them? That's the job of the analytics tools.

Components of a log file

A typical log file will have one or more "Start" lines and some number of "Respond" lines.

```
Start: user:test bot:rose ip: rand:3089 (~introductions) 0 ==> Hello, I'm Rose. When:Jan15'1
Build1:Jan15'16-16:23:39 0:Jan15'16-17:03:44 F:0 P:0 Why:~introductions.1.0.~control.7.0
Respond: user:test bot:rose ip: (~introductions) 1 where do you live ==> I'm not from around
Why:~no_task.0.0.~control.21.0=MAIN
```

A `start` line shows conversation initiation. After Start, the line contains the user name, the bot name, the IP address (using a default configuration). Then the current value of the random seed and what topic it ended in in parens and the volley count. That completes the input side.

The `==>` indicates the upcoming outputs side. You see the startup message it issued (eg *Hello, I'm Rose.*). Then a datestamp of when this was issued, followed by the CS engine version, the date stamps when build0, build1, and build2 were created.

`F:` and `P:` are specially volley markers for reconstructing conversations. And Why tells you the rule tags of the rule and possibly the reuse prior rule that immediately generated the output. The respond log entry shows you similar information, the only difference being that immediately before the `==>` is the actual input from the user.

The Abstract

Looking at your entire source script is tedious. It's extensive and hard to read. If you want to see what you've got in a reasonable overview, you need `:abstract`. It can do a variety of tasks.

`:abstract`

This prints out a view of the entire topic system showing its structure (gambits, rejoinders, responders) as well as conditions on gambits and normal text content of everything, but omits actual code complexity. It is useful for seeing what will be said in response to sample input (if you use the #! and #! x commands on rules. E.g

```
Topic: ~introductions[]

t: ( $old %input=0 %hour<12 $name ) Good morning, .
t: ( $old %input=0 %hour>11 %hour<18 $name ) Good afternoon, .
t: ( $old %input=0 %hour>18 $name ) Good evening, .

t: Where do you live?
     a: "Fukushima" => I've heard of _0. Were you born there?
     a: "I live in Japan" => I've visited Japan.
     a: "I live in California" => That's where I live!
     a: "Libya" => I would have thought you lived in Japan, not _0 .
     a: "Earth" => Yes, we all live on Earth.
     a: "Mars" => I don't believe you.

t: What do you do for a living?

u: "Am I welcome here?" => Of course you are welcome.

s: "I'm back" =>
 [ Where did you go? ]
 [ Where have you been? ]
 [ I'm glad. ]

s: "Knock" => Who's there?
```

While `:abstract` primary does topics, you can get it to do fact data as well. It will attempt to call a function `^abstract_facts()`, so if you define that, you can do whatever you want for abstracting facts.

```
:abstract ~topicname
```

Calling `:abstract` with a topic will limit it to doing just that topic.

```
:abstract 100 ~topicname
```

If you want to adjust output of yours that would be too long for something like a phone screen, you can ask `:abstract` to show you all rules whose output would likely exceed some limit (here 100). Again with a topic name restricts it to that topic and without the name it does the entire system.

```
:abstract censor ~mywords
```

will note all output which contains any words in mywords. Of course regular uses may also appear. The censor command looks for any words referred to by the concept given.

```
:abstract pretty
```

will prettyprint topics.

```
:abstract canon
```

will prettyprint and rewrite patterns using the canonical form of words.

`:abstract nocode`

will not display rules that only have code output You can do all topics in a file by naming the file name instead of the topicname. Don't use the full path, just the actual name of the file.

Time View over Log file - `:timelog`

Given a log file (usually a server log), it will compute the average, min, and max NLU engine response times of the volleys. And for LINUX, how long it took from first detection of incoming connection to get to the engine (including reading incoming data).

Views over User Logs - `:trim`

If you get a lot of user logs (say thousands), reading through them becomes a chore. The logs have a bunch of excess information and are in a bunch of different files. This is where `:trim` comes in, making it easier to see things. `:trim` assumes all files in the LOGS directory are user logs and will process them in some manner. It will normally put its output in `TMP/tmp.txt`.

If you give a quoted string as first argument and it has "keepname" in it, then instead of dumping everything into tmp.txt, it will create files named after the user logs it discovers (username.txt)

Your first argument to trim can also be just the name of a user (whose log file will be in `USERS/log-xxx.txt`) or if that doesn't exist then it is the directory to use, or you can use the name of a log file within the USERs directory (the name should begin `log-` and not include the directory and need not include the `.txt` suffix). E.g.,

`:trim c:\FULLLOGS 6`

a directory to read all files within

`:trim log-bob 6`

named log file with `.txt` defaulted

`:trim bob 6`

user with logfile log-bob.txt in USERS

`:trim n`

Trim will read every file and generate output depending on the integer code given it. The codes are:

n	description
`0`	puts the what the user said, followed by what the chatbot said, on single lines, removing all the excess junk.
`1`	similar to 0, but puts what the chatbot said first, and what the user said after. This is useful for seeing all the responses users have made and can be aggregated to figure out what clever rejoinders you might want.

n	description
2	similar to 0 (user first), but puts the name of the topic the chatbot ended in before either. You can see the flow of topics better with this view.
3	similar to 2 (topic shown), but puts what the chatbot said before the user.
4	puts the user and chatbot on separate lines, indenting the chatbots line. Easier to read.
5	similar to 3, but indents the user instead of the chatbot.

n	description
6	only lists the users inputs. This is good for creating a file that can recreate a user's experience, if you want to recreate it for debugging or regression.

n	description
7	display rule responsible for output. Analogous to :why, it shows the rule tag, the sample input comment if there is one, the rule type and pattern, the input from the user and the output from the chatbot. If the rule doing the output was the target of a local `^reuse (same topic)`, then the data about the rule comes from the calling rule, not the output rule.

n	description
8	puts the user and chatbot on separate lines, indenting the chatbots line and prefixes it with the topic generating the response. Easier to read and debug.
11	puts the timestamp and user on first line and and chatbot on second line,indenting the chatbots line.
12	output per line, 2nd rule label generating output, input, => output.
13	output per line, 1st rule label generating output, input, => output.

'12' and '13' are useful for generating where input went (what rule output came from) and then you can sort the tmp.txt file to cluster all inputs that went to the same place. '12' assumes you always output OOB data from postprocessing,

so it skips that rule label and uses the second label (the actual user output). `13` assumes the only data written is data to the user.

Normally trim displays everything. But with an optional 3rd argument `nooob`, you can omit out-of-bands data from output. E.g.,

```
:trim bob 6 nooob
```

You can separately choose to trim input and output oob using a numeric bits, where `1` is input and `2` is output (the same as `noob`) and `3` is both

`:trim 11` looks like this:

```
Jan25'17-13:49:21  when is the help desk open?
               The Helpdesk is available Monday through Friday from 7:00AM to 7:00PM Pacific
```

Deduping a file - `:dedupe`

`:dedupe  filepath` outputs into tmp/filename just unique lines from the input (ignoring data after a hash comment marker).