ChatScript Tutorial

This tutorial will show you how to use ChatScript for building a simple but useful chatbot - one that can help a human plan a travel.

First download ChatScript, read the manual, go through the overview, and make sure you can chat with the default bot about its childhood.

Now, in order to build a new chat-script from scratch: Create a folder called TEST inside the RAWDATA folder. Create an empty file named `tutorial.top`, and put it inside this TEST folder.

Copy the file `RAWDATA/filesHarry.txt` that comes with the zip file into a new file `RAWDATA/filesmine.txt`.

In `filesmine.txt`, insert a reference to your newly created file and revise the reference to HARRY. `filesmine.txt` looks like this:

```
RAWDATA/HARRY/simplecontrol.top # default bot control
RAWDATA/TEST/tutorial.top # tutorial bot data
```

From now on, we will work with the file `tutorial.top`.

NOTE: This tutorial will cover only a small part of ChatScript's features - only those features needed for building our traveling-agent bot. You can find many more features in the documentation.

Contents

Speaking (`topic: , t:`)

The simplest possible chat-script should include a topic and a sentence to say, for example:

```
topic: ~introductions []

t: Hello, talk to me!
```

Where:

- The keyword topic: defines a topic for conversation. We called our topic `~introductions` (topic names always start with `~`).
- The keyword `t:` defines a sentence to say within a topic (it's called a "gambit", as the bot says it without waiting for our input).

Put this text in `tutorial.top`. Then, at the client's prompt, type `:build mine`. You should see something like this:

```
>:build mine

----Reading file simplecontrol.top
Reading outputmacro: ^harry
Reading outputmacro: ^georgia
Reading table tbl:defaultbot
Reading topic ~control
Reading topic ~alternate_control

----Reading file tutorial.top
Reading topic ~introductions
No errors or warnings in build
Read 302,955 Dictionary entries
Read 304,588 Dictionary facts
Read 110,851 Build0 facts
Read 1 Build1 facts
Concept sets: 1421
Free space: 123,252,280 bytes FreeFacts: 4,584,560

Hello, talk to me!
>
```

Chatting with this bot results in a dialog such as this:

```
Hello, talk to me!

>hi
I don't know what to say.

>why?
I don't know what to say.
```

The bot just says what we told it to say, and then stuck, as it does not have anything else to say.

If you are tired of speaking with the bot, you can exit the client using the `:quit` command.

Saying more (^keep, random selection [], ^repeat)

Let's make our bot a bit more interesting:

```
topic: ~introductions []

t: ^keep() [Hello] [Hi] [Hey], [talk] [speak] [say something] to me!
```

Here:

- We called the ^keep function, which tells the bot not to erase the rule after it is used (normally, rules are used only once and then discarded).
- We also added some options in brackets, from which the bot may select at random. In any continuous sequence of bracketed phrases, the bot may select one at random.

Building this chatbot will result in a dialog such as this:

```
Hello, say something to me!

>hi
Hey, say something to me!

>what
Hey, talk to me!

>why?
 Hello, speak to me!

>who
Hi, talk to me!

>abc
I don't know what to say.

>def
I don't know what to say.
```

The bot tries some random combinations from our output pattern, and then gets stuck again as it starts to repeat itself. A bot normally doesn't like to repeat itself, except when we tell it explicitly using the ^repeat() macro:

```
topic: ~introductions []

t: ^keep() ^repeat() [Hello] [Hi] [Hey], [talk] [speak] [say something] to me!
```

With this bot we can chat forever.

Listening (`u:`, `^reuse`)

Our previous bots could only talk - they did not listen to us. Now let's build a bot that listens and reacts to what we say:

```
topic: ~introductions []

t: [Hello] [Hi] [Hey], [talk] [speak] [say something] to me!

u: (what are you) ^keep() ^repeat() I am a bot.

u: (where do you live) ^keep() ^repeat() I live on your computer.

u: ([what where]) ^keep() ^repeat() Good question.
```

Here:

- The `u:` keyword tells the bot to match the user input against the pattern given in parenthesis.
- Brackets `[ ]` inside the pattern mean disjunction, so the third line means "If there is 'what' or 'where' anywhere in the user's input, say"Good question".

Here is a possible dialog with this bot;

```
Hello, say something to me!

>hi
I don't know what to say.

>What are you?
I am a bot.

>Tell me where do you live.
I live on your computer.

>What is the time?
Good question.

>why?
I don't know what to say.

>Remind me where did you live please.
I live on your computer.
```

The bot selects the first line that generates output, prints the output, and stops matching (conclusion: order your patterns from the more specific to the more general). If it doesn't find any match, it just replies with the default *I don't know what to say.*

Note that the match is case-insensitive ('what' matches 'What'), and uses canonical forms of verbs and nouns ('do' matches 'did'). The engine uses a POS tagger to detect nouns, verbs and other types of words, and convert them to canonical form.

If we want an exact match, we can use an apostrophe, for example, if we replace the "where do you live" line with this line:

```
u: (Where 'do you live) ^keep() ^repeat() I live on your computer.
```

we will get this dialog:

```
Hey, speak to me!

>where do you live
I live on your computer.

>where did you live
Good question.
```

If we use a non-canonical form, it will match only that exact form, even if we don't use an apostrophe, so the patterns "(Where 'did you live)" and "(Where did you live)" will both match only that exact phrase, and not "where do you live".

What if we want to use a totally different input pattern, but still get the same output? We can, of course, repeat the same line with a different output, but we can also ^reuse:

```
topic: ~introductions keep repeat []

t: HI () [Hello] [Hi] [Hey], [talk] [speak] [say something] to me!

u: WHAT (what are you) I am a bot.

u: (tell me about yourself) ^reuse(WHAT) ^reuse(HI)

u: (where do you live) I live on your computer.

u: ([what where]) good question.
```

Here:

- we moved `keep` and `repeat` to the topic definition, so we don't have to repeat them for each statement.
- We gave the first two rules a label.
- We added a new third rule, which reuses the first two rules.

Here is a sample dialog:

```
Hello, speak to me!

>tell me about yourself
I am a bot. Hi, talk to me!

>what are you
I am a bot.
```

Short-term memory (* _)

Our previous bots could listen and talk. The third thing a bot should do is remember. In ChatScript, the memory of a bot is contained in variables.

Using variables we can start building our travel-agent bot. Our bot will try to understand where the user is and where he wants to be, and will try to supply useful information.

```
topic: ~introductions keep repeat []

t: [Hello] [Hi] [Hey], I will help you plan your travel. Just tell me where you want to go.

u: (I want to go to _*) OK, you want to go to _0.
```

Here:

- We use * as a wildcard that means "0 or more words", because we want to match any sentence where the user says his destination.
- In the input pattern, we use _ before the wildcard to capture what the user actually said.
- In the output, we use _0 to refer to the first thing we captured (if we have more patterns to capture, we can use _1, _2, ... _9)

Here is a sample dialog:

```
Hello, I will help you plan your travel. Just tell me where you want to go.

>I want to go to the moon.
OK, you want to go to a moon.

>I want to go to earth.
OK, you want to go to earth.

>I want to go back to the moon.
I don't know what to say.
```

Note that the last sentence did not match because of the word "back". We could use another wildcard to make our sentence more general, such as:

```
u: (I want to go * to _*) OK, you want to go to _0.
```

In this case we don't need to capture the first wildcard, so we don't put a _ before it.

Long-term memory ($)

The _0, _1... variables are "short-term memory" because they exist only in the output line next to the pattern where they were created. We can create "long-term memory" by using $-variables. Let's use these variables to extend our travel-agent bot:

```
topic: ~introductions keep repeat []

t: [Hello] [Hi] [Hey], I will help you plan your travel. Just tell me where you are and
where you want to go to.

u: (be * at _*)
    OK, you want to go from _0.
    $source = '_0

u: ([go fly travel] to _*)
   $target = '_0
   OK, you want to go to $target .

u: (what do I want)
   You want to go from $source to $target .
```

Notes: We can put the different parts of a rule (input pattern, output pattern, update commands) in different lines, to make it more readable. Chat Script doesn't care.

Here is a sample dialog:

```
Hey, I will help you plan your travel.
Just tell me where you are and where you want to go to.

>I am at Earth.
OK, you want to go from Earth.

>I would like to fly to the moon.
OK, you want to go to the moon.

>What do I want?
You want to go from Earth to the moon.

>Actually, I want to go to the sky.
```

```
OK, you want to go to the sky.

>What do I want?
You want to go from Earth to the sky.
```

The above dialog with some explanations:

```
Hey, I will help you plan your travel.
Just tell me where you are and where you want to go to.

>I am at Earth.
```

'be' in the pattern is the cannonical form of 'am'.

```
OK, you want to go from Earth.
```

the bot prints the temporary variable `_0`, which contains 'Earth'.

```
>I would like to fly to the moon.
OK, you want to go to the moon.
```

the bot prints the permanent variable '$target', where we put 'the moon'.

```
>What do I want?
You want to go from Earth to the moon.
```

the bot remembers the permanent variables.

```
>Actually, I want to go to the sky.
OK, you want to go to the sky.

>What do I want?
You want to go from Earth to the sky.
```

the bot updated the variable '$target'.

Dialog Management (conditions)

The bots we built so far decided what to do based solely on the user's input. But suppose we need to make a more sophisticated decision, for example, check if the user already said where he wants to go, and if not, ask him specifically. For this kind of tasks, we can use conditions.

The most basic condition is testing whether a variable is set. This is done simply by putting the variable inside the parens of the input pattern.

```
topic: ~introductions repeat keep []

t: [Hello] [Hi] [Hey], I will help you plan your travel. Just tell me where you are and
where you want to go to.

u: (be * at _*)
```

```
   $source = '_0

u: (go to _*)
   $target = '_0

u: (!$source)
   Where are you?

u: (!$target)
   Where do you want to go?

u: ($source $target)
   You want to go from $source to $target .
```

The last rule will be selected only if both `$source` and `$target` are defined. The previous rule will be selected only if `$target` is NOT defined, and similarly for the rule before it.

This script allows the user to specify what he wants in any order. It will fill the appropriate variables, and ask appropriate questions if they are not set yet:

Hello, I will help you plan your travel. Just tell me where you are and where you want to go to.

```
Hello, I will help you plan your travel.
Just tell me where you are and where you want to go to.

>great!
Where are you?

>Hod Hasharon
Where are you?

>I am at Hod Hasharon
Where do you want to go?

>I don't know
Where do you want to go?

>I want to go to Jerusalem
You want to go from Hod Hasharon to Jerusalem.

>really?
You want to go from Hod Hasharon to Jerusalem.

>Actually I am at Haifa.
You want to go from Haifa to Jerusalem.
```

Those familiar with Dialog System terminology will probably notice that we implement an information-state-update dialog manager. The information state is defined by the variables. In each volley, the system first tries to understand what the user said, then updates the current state, and then generates output to the user.

Our previous bot didn't pay any attention to context - it just looked for an input pattern, so it didn't understand that the words *Hod Hasharon* are actually a reply to the previous question *Where are you?*.

One solution to this is to add another variable that will remember the subject of the last question asked. Here is the modified script, where we added some comments after the `#` symbols:

```
topic: ~introductions repeat keep []

t: [Hello] [Hi] [Hey], I will help you plan your travel.
   $issue = null # initialize the current issue

u: SOURCE (be * at _*)
   $source = '_0 # remember the source
   $issue = null # clear the current issue

u: TARGET (go to _*)
   $target = '_0
   $issue = null

u: ($issue=source _*) ^reuse(SOURCE)
   # If current issue is source, then assume the
   # user answered the question about his source

u: ($issue=target _*) ^reuse(TARGET)

u: (!$source)
   $issue = source # Remember that we asked the user about his source
   Where are you?

u: (!$target)
   $issue = target
   Where do you want to go?

u: ($source $target)
   You want to go from $source to $target .
```

Note: When testing variable values inside input patterns, there must be no spaces between the variable name and the `=` operator (e.g. `$issue=source`), otherwise the engine will parse it as two different input conditions `$issue` and `source`.

```
Hello, I will help you plan your travel.

>great!
Where are you?

>Jerusalem
Where do you want to go?

>Actually I am at Haifa
Where do you want to go?

>Hebron
You want to go from Haifa to Hebron.
```

Implicit Confirmations (`^respond`)

Our previous bot understood the user perfectly, but usually it is not the case, we would like our bot to confirm that it understood the user correctly. There are two types of confirmation:

- Implicit confirmation - just tell the user what we understood.
- Explicit confirmation - tell the user what we understood, and ask him/her if it's correct.

Let's start with implicit confirmation. For simplicity, we will start this section with the bot at the beginning of the "Dialog Management" chapter (without context), and add the confirmation statements from the bot at the "Long-term memory" chapter:

```
topic: ~introductions repeat keep []

t: [Hello] [Hi] [Hey], I will help you plan your travel. Just tell me where you are and
where you want to go to.

u: (be * at _*)
   $source = '_0
   OK, you want to go from $source .

u: (go to _*)
   $target = '_0
   OK, you want to go to $target .

u: (!$source)
   Where are you?

u: (!$target)
```

```
   Where do you want to go?

u: ($source $target)
   You want to go from $source to $target .
```

This works, but produces an awkward dialog:

```
  Hello, I will help you plan your travel.
  Just tell me where you are and where you want to go to.

  >I am at Haifa
  OK, you want to go from Haifa.

  >well?
  Where do you want to go?

  >I want to go to Jerusalem.
  OK, you want to go to Jerusalem.

  >what now?
  You want to go from Haifa to Jerusalem.
```

By default, the bot stops processing the rules when it finds a rule that produces output, so after the confirmation *you want to go from Haifa* the bot doesn't go on and ask the next question *Where do you want to go*, until the user says another thing.

We can change this by creating a new topic, and using the `^respond` command;

```
topic: ~introductions []

t: Hi, I will help you plan your travel.
 ^respond(~question)

u: SOURCE (be * at _*)
   OK, you want to go from _0.
   $source = '_0
   ^respond(~question)

u: TARGET (go to _*)
   $target = '_0
   OK, you want to go to $target .
   ^respond(~question)

u: DEFAULT ()
   ^respond(~question)

topic: ~question repeat keep nostay []
```

```
u: (!$source)
   Where are you?

u: (!$target)
   Where do you want to go?

u: ($source $target)
   You want to go from $source to $target .
```

The `^respond` tells the bot to use the `~question` subtopic, and add whatever text it gets from it to the current statement.

Notes: * We use `^respond` after all 4 rules - after the initial gambit, after the user replies with a source or a target, and also in the default case where we don't understand the user. * We added the `nostay` directive to the `~question` topic. By default, if the bot gets a response from a certain topics, it 'stays' in this topic and gives it priority over other topics. In this case we don't want the `~question` topic to get priority, because our first priority is to handle the user response, so we override this behavior.

This produces a much shorter dialog:

```
Hi, I will help you plan your travel. Where are you?

>I am at Haifa
OK, you want to go from Haifa. Where do you want to go?

>I don't know
Where do you want to go?

>go to Jerusalem
OK, you want to go to Jerusalem. You want to go from Haifa to Jerusalem.
```

Explicit Confirmation (rejoinders: `a: b: c: ...`)

Sometimes we want to explicitly ask the user what he meant. In this case we expect a yes/no answer, but the meaning of the answer obviously depends on the context - the question that we just asked. One way to handle this is to use rejoinders.

ChatScript allows us to understand the user input based on the current context of the dialog, by labeling statements with `a:`, `b:`, `c:`, ... `q:`, instead of the u and t we used so far. For example:

```
topic: ~introductions repeat keep []

t: [Hello][Hi][Hey], I will help you plan your travel.
```

```
   ^respond(~question)

u: SOURCE (at _*)
   Is your current location \" '_0 \" ?
   a: (~yes)
       $source = '_0
       ^respond(~question)
   a: (~no)
       OK, so where are you?

u: TARGET (to _*)
   Is your destination \" '_0 \" ?
   a: (~yes)
       $target = '_0
       ^respond(~question)
   a: (~no)
       OK, so where do you want to go?

u: DEFAULT ()
   ^respond(~question)

topic: ~question repeat keep nostay []

u: (!$source)
   Where are you?

u: (!$target)
   Where do you want to go?

u: ($source $target)
   You want to go from $source to $target .
```

Here, every time the user answers our question, we ask him if that's what he meant.

Note that we use `~yes` and `~no` as the patterns for yes and no; this will also match "sure", "yeah" and other forms.

Here is a sample dialog:

```
Hello, I will help you plan your travel. Where are you?

> at Haifa
Is your current location "Haifa "?

> no
OK, so where are you?
```

```
> at Ramat Gan
Is your current location "Ramat Gan "?

> yes
Where do you want to go?

> actually I am at Givat Shmuel
Is your current location "Givat Shmuel "?

> sure
Where do you want to go?

> to Jerusalem
Is your destination "Jerusalem "?

> y
Where do you want to go?

> to Jerusalem
Is your destination "Jerusalem "?

> yeah
You want to go from Givat Shmuel to Jerusalem.

> ...
```

Knowledge (`^createfact`, `table:`)

Our previous bot could understand what the user wants, but it couldn't really help him reach his goal because it had no knowledge of transportation means. We can give our bot some knowledge by creating facts.

— THE TUTORIAL AS ORIGINALLY AUTHORED ENDS HERE —

TO DO