Your handy guide to the language of creating artificial conversational programs

ANIRUDH KHANNA

Coding With QuickScript

Your handy guide to the language of creating artificial conversational programs

anirudhkhanna.cse@gmail.com

IN THIS DOCUMENT

1. The QuickScript Project

- **What is QuickScript?**
- **Behind the Idea**
- **The QuickScript Engine**

2. First Steps

- **Making a QS File**
- **Including a QS File in the QuickScript Engine**

3. QuickScript Syntax

- **The Basics: Queries, Patterns & Responses**
- **SRAI**
- **Wildcards**

1. The QuickScript Project

What is QuickScript?

QuickScript is a simple and easy to learn special-purpose language created by Anirudh Khanna, which can be used to design artificial conversational agents and other programs that involve textual dialogue between humans and computers. QuickScript can be used to create something as plain as a dictionary as well as something as clever as a chatbot!

QuickScript is special because of its simplicity and is intended to generate interest in beginners in the field of virtual conversational entities, more popularly known as chatbots.

The features of QuickScript are inspired by AIML, but both the features and the way of writing code has been even more simplified. QuickScript not only provides the basic gear that one would appreciate when starting his/her own chatbot, but it also simplifies the task to such an extent that anyone can playfully learn the fundamentals of this art.

The project's repository on GitHub: https://github.com/anirudhkhanna/QuickScript
(Released under the GNU GPL v3.0: http://www.gnu.org/licenses/gpl-3.0.html)

Behind the Idea

In 2015, I started working on a chatbot in AIML and was really impressed by the features it offers for the very purpose. I wanted to spread the field of virtual conversational agents to even more people and so I decided to take some wonderful features of AIML (wild cards, SRAI etc.), simplify them, and make the easiest script for designing such entities.

The QuickScript Engine

One can run QuickScript code in the QS Engine, which is written in C and compiled on Code::Blocks 13.12 (MinGW GCC Compiler 4.8).

2. First Steps

Making a QS File

Users can work with QuickScript code in any text-editor of their choice and save it with the extension ".qs". Let's make our first QuickScript file.

Go to the QuickScript folder, make a text file and open it in a text-editor like Notepad or Notepad++ and write the following code:

```
>> HELLO
## Hello, user!
```

Now save the file with a *[dot]qs* extension, for example "mynewfile.qs", and congratulations, you just created your first QuickScript code! The two lines written above make a simple but complete QuickScript program, which contains only one pattern and its one response.

Note: Remember to give a line feed between the first and the second line. Newlines are the backbone of QuickScript code. They work in the same way as the semicolon works in C language.

Including a QS File in the QuickScript Engine

Now we need to include this file into the list of files which the QS Engine will interpret. **Look for a text file in the QuickScript project's folder, named "files.txt". This file contains the names (and paths, if necessary) of the QS files to be run.** (By chance, if there is no such file then you need to create a text file there and save it as "files.txt".)

Open this text file and write the name (and relative path, if required) of your newly created QuickScript file in a pair of angular brackets '<' and '>' in such a way:

```
<path of file.../filename.qs>
```

For example, if your QuickScript file is named "mynewfile.qs" and it is in the folder where the QS Engine is placed itself, then writing the file name in "files.txt" will be sufficient:

```
<mynewfile.qs>
```

Now suppose your file is in a folder named "My QuickScript Files", which is in turn placed in the main QuickScript folder itself, then we need to include the relative path also:

```
<My QuickScript Files/mynewfile.qs>
```

Or, you may give the complete path to the file, somewhat like this:

```
<C:/QuickScript/My QuickScript Files/mynewfile.qs>
```

After making an entry of your QS file in "files.txt", save the changes and run "QuickScript.exe" to start the QS Engine. It is the program that will understand all the QuickScript code written in the file(s) you have included.
The QS Engine is written in C and compiled on Code::Blocks 13.12 (MinGW GCC Compiler 4.8). The source code is available under the GNU General Public License v3.0.

After starting the QuickScript interface, press ENTER to load all the included QS files and get ready to chat with the program. If the file is successfully included, the code should be interpreted without any errors and the program will prompt the user to enter a query.

Write "Hello" and press ENTER. The program, as per the above code, should print "Hello, user!" in response.

Note: See that a number of files can be included at a time. The response of the program depends on the order of writing the file names in "files.txt" – the topmost file will be searched for any input first, then the file below it and so on. So, if the same pattern like >> HELLO is in two QuickScript files, the response given in the file which is included first in "files.txt" (going from top to bottom) will take precedence.

3. QuickScript Syntax

The reason for QuickScript to exist is its simplicity. QuickScript can teach you to walk easily on the path of programming conversational agents. Before delving more into QuickScript, a few terms must be introduced here:

- **Entry:** A QuickScript code consists of lines of text called "entries". Each entry must be in a separate line. An entry has two parts – a "prefix" and some "content".
  E.g. `>> HELLO`
  `## Hello, user!`
  `++ How are you?`

  - **Prefix:** An entry starts with a "prefix". It is a fixed set of symbols which specifies the type of an entry.
    E.g. `>>`
    `##`
    `++`

  - **Content:** The "content" is whatever the botmaster decides for the bot to recognize/learn.
    E.g. `HELLO`
    `Hello, user!`
    `How are you?`

- **Pattern:** A "pattern" is a string of characters which the bot is intended to recognize. Means, if the bot should recognize the user saying "hello" and respond with a suitable reply, then the pattern "hello" must be stored in the code along with a reply.

  A pattern starts with a ">>" prefix.
  E.g. `>> HELLO` or `>> hello` (QS is case-insensitive.)

- **Response/Reply:** A "reply" is the chatbot's response to a matched pattern. For instance, "Hello! How are you?" can be a suitable reply to the above pattern "Hello".

  A reply starts with a "##" prefix.
  E.g. `## Hello! How are you?`

- **Query:** A "query" is anything that the user asks the chatbot. The query is matched with patterns already stored in the database and when a suitable match is found, the corresponding reply is shown.

- **Comments:** "Comments" can be given in QuickScript code with a "//" prefix. Again, a comment should be an independent line of code. Comments do not affect the working of code; they are there just for the botmaster's understanding and note-making.
  E.g.
  ```
  // I am a comment.
  // I do not affect the working of your code.
  ```

  Note that it's a **WRONG EXAMPLE** to do this:

  ```
  >> HELLO        //This is a pattern
  ## Hey there!   //This is a reply
  ```

  Mixing up of other entries and comments in one line is wrong. They need to be written in **separate lines**.

  This is the **CORRECT WAY**:

  ```
  // This is a pattern:
  >> HELLO
  // The following is its reply:
  ## Hey there!
  ```

- **SRAI:** The term "SRAI" is directly taken from AIML, where AI stands for *Artificial Intelligence* but SR can stand for various contexts like *Syntactic Rewrite*, *Synonym Resolution*, *Symbolic Reduction* etc. This reflects the variety of ways in which SRAI can be implemented.

  Simply stating, SRAI can be used to redirect a number of queries to a single answer and hence, you don't need to write the same answer again and again for each possible pattern having the same meaning.

  In QuickScript, SRAI can be simply implemented with a "==" sign.
  E.g.
  ```
  >> HELLO
  ## Hello! How are you?

  //Now redirecting similar patterns with SRAI:
  ```

```
>> HI
== HELLO
>> HOLA
== HELLO
>> HOWDY
== HELLO
```

The Basics: Queries, Patterns & Responses

What QuickScript code mostly consists of is patterns and their corresponding responses, which are matched with user's queries. Remember, **QuickScript code as well as the matching process is case-insensitive**.

QUERIES

User can enter any query consisting of letters, numbers and special characters, but before the matching process begins, the inputted query goes though the following changes:

- Extra spaces are trimmed.
- Punctuation/special characters are removed, except the apostrophe.

Hence, a pattern like this will match with both *"Who created QuickScript"* and *"…who created quickscript ??? "*:

```
>> WHO CREATED QUICKSCRIPT
## Anirudh Khanna started the QuickScript project in 2016.
```

When the user makes a query, the matching process is started. The query is matched in the included files one by one, **starting from the topmost included file to the bottom**. Also, in any particular file, the user's query is matched with **patterns starting from top to bottom**. As soon as a pattern matches with the query, the reply is displayed and the program is ready to take the next input.

The following will give the first definition as response to the query *"What do you mean by apple"* every time:

```
>> WHAT DO YOU MEAN BY APPLE
## It is a well-known fruit.

>> WHAT DO YOU MEAN BY APPLE
## It is a famous technology company.
```

PATTERNS AND RESPONSES

Patterns are strings of text which have a ">>" prefix. It has already been explained above how patterns are matched with input queries.

"EXIT" as a Pattern: To exit the chatting interface and go back to home screen, user has to enter *"exit"* as the input query. This is the correct way of going back. (Other queries like *"bye"* or *"abort"* can also be given the same functionality as *"exit"*; we will later see how this can be done with the help of SRAI.)

IT IS RECOMMENDED TO EXIT THE CHATTING INTERFACE CORRECTLY.
Various temporary files are created during the program execution and they are deleted when you properly exit the program. If you see any new stray files with an ending like "_temp.qs", do not worry about them. They are just the temporary files created while your bot was running but could not be deleted successfully. You may delete them manually too, if you feel necessary. Just be careful not to delete an original QS file accidently!

Multiple lines in a reply: Two prefixes are generally associated with responses: "##" and "++". As we already know that a response line starts with a "##" prefix. But a response may consist of multiple lines too. If the very next line is also to be shown as a line of the reply, then "++" is used before it. This means that the first line of any reply has the prefix "##", and all the subsequent lines (if they are to be displayed as the lines in the reply) start with the "++" prefix. If the reply consists of only one line, then there is no need to use "++" prefix.

E.g.
```
>> WHAT DO YOU MEAN BY APPLE
## It is a well-known fruit.
++ I like to eat apples very much!
++ It is also the name of a famous technology company.
```

The above code will display all the three lines as the reply, like this:

```
USER: What do you mean by apple?
BOT: It is a well-known fruit.
     I like to eat apples very much!
     It is also the name of a famous technology company.
```

Random Replies: The botmaster can give a number of replies for a given pattern and any of the replies will be shown randomly when a matching query is asked by the user. Each random reply is started with a new "##" prefix.

E.g.
```
>> WHAT DOES A BIRD HAVE
## Lungs.
## Two eyes.
## A beak.
## A tail.
## Wings.
## Feathers.
```

```
>> HEY THERE
// A response:
## Hello, dear.
++ How are you doing?

// Another random response:
## Hi.
++ What a beautiful day!
```

Random responses make an artificial conversation much more human. You would not like your chatbot to give exactly the same reply every time someone says hello!

SRAI

As already stated above, SRAI (taken from AIML) can stand for many contexts. AI stands for Artificial Intelligence but SR can mean various things: *Syntactic Rewrite*, *Synonym Resolution*, *Symbolic Reduction* etc.

SRAI is used to direct a number of patterns (generally having the same meaning) to a single answer. For example, the patterns *"How are you"*, *"How do you do"* and even *"How R U"* can have the same answer.

SRAI is implemented with the "==" prefix.

E.g.
```
>> HOW ARE YOU
## I am fine. Thank you.
## I'm doing fine! Thanks for asking.
++ How about you?
## I am doing very well!

// Now using SRAI for similar patterns:
>> HOW DO YOU DO
== HOW ARE YOU

>> HOW R U
== HOW ARE YOU
```

SRAI in QuickScript: As shown in the above example, SRAI can be used to redirect one pattern to another. When SRAI is used with a pattern, then that pattern cannot have its own replies.
As compared to the SRAI of AIML, the functionality of SRAI in QuickScript is relatively limited and simple.

HOW IS SRAI SEARCHED?

When a SRAI prefix is encountered after a pattern, the redirected pattern is searched **IN THE SAME FILE FIRST**. If the pattern is not found in that very file, then it is searched **IN ALL THE INCLUDED FILES** starting from the first included file as any normal pattern is searched in QuickScript.

Consider two QS files which are included in the Engine at a time: "first.qs" and "second.qs".

Let the contents of "first.qs" be the following:

```
// FIRST.QS

>> BIRD
## Birds are winged, warm-blooded, egg-laying vertebrates.
```

And the contents of "second.qs" are the following:

```
// SECOND.QS

>> WHAT IS A BIRD
== BIRD

>> BIRD
## Birds are cute, feathery creatures which mostly can fly.
```

For an input *"What is a bird"*, the output will always be *"Birds are cute, feathery creatures which mostly can fly."* as given in "second.qs", because the SRAI is satisfied in the same file. Now if the pattern is removed and "second.qs" is changed to this:

```
// SECOND.QS

>> WHAT IS A BIRD
== BIRD
```

In this case, the SRAI pattern "BIRD" would not match in the same file and then, every file will be searched for it from the start. The result would be *"Birds are winged, warm-blooded, egg-laying vertebrates."* as found in the file "first.qs".

USING SRAI WITH "EXIT":

We know that the query *"exit"* is used to go back to the starting screen of the program. Other patterns can also be made to carry out the function of *"exit"* with help of SRAI.

E.g.
```
>> BYE
== EXIT
>> GO BACK
== EXIT
>> ABORT
== EXIT
```

Many times while designing your chatbot, you will see SRAI is a very useful and handy concept and apart from making your program much cleverer, it can save you from writing a lot of redundant code.

Note: The danger of SRAI is that it permits the botmaster to create infinite loops. For instance, entries like `>>x`, `==y` and `>>y`, `==x` will pose the risk of program entering an infinite loop.

Wildcards

The asterisk (*) and underscore (_) symbols are two wildcard characters used in QuickScript. The meaning of a QuickScript wildcard, wherever it occurs, is "**anything may or may not occur here**".

Let's say you want to provide the same answer to various patterns like *"What is Google"*, *"What is Google Inc"*, *"What is Google Dot Com", "What is Google Search"* and so on.

A wildcard can easily handle this situation with just one pattern, which is like this:

```
>> What is Google*
## Google is a technology company specializing in Internet-
related services and products like search, advertising and
cloud computing.
```

The above pattern actually means: *"What is Google[anything may or may not occur here]"*. It will match to all the queries that start as *"What is Google…",* **including the query *"What is Google"* itself**.

Note that there is a difference between the following two similar-yet-different uses of the wildcard symbol:

```
>> What is Google*

>> What is Google *
```

The pattern with a space between *Google* and * **will not match the query *"What is Google"***. This is because the query does not have a space after Google. However, it will match all the queries starting with *"What is Google<space>"* **and having some word(s) after that**.

Let's see another similar situation. Consider:

```
>> What is Git*
```

This pattern will match with *"What is Git"* as well as *"What is GitHub"* as well as *"What is Git used for".* Had the pattern been **`>> What is Git *`**, it would not match with the first two of the queries, only with *"What is Git used for"*.

Note: The wildcard character '_' can also be used in the above examples. The reason of having two different wildcard characters will be cleared in subsequent text.

USING `[STAR]` AND `[UNDERSCORE]`:

Sometimes, it may be needed to use/display whatever text is there in place of the wildcard character. Here is when **`[star]`** and **`[underscore]`** come to our help.

Consider the following:

```
>> I LIKE *
## I also like [star].
```

If user enters *"I like apples"*, then the bot will respond *"I also like apples"*, because there is *"apples"* in place of the wildcard character '*'. Similarly, if user enters *"I like talking to people"*, then the bot will say *"I also like talking to people"*.

Suppose we have a QuickScript file that defines a lot of words with simple pattern-response pairs. We can efficiently redirect different patterns to suitable words, with help of SRAI and wildcards:

```
>> WHAT IS AN _
== [underscore]
```

```
>> WHAT IS A_
== [underscore]
```

```
>> WHAT IS _
== [underscore]
```

Multiple Wildcards: More than one wildcards can be used in a pattern, but QuickScript will remember the text corresponding to **only the last occurrence** of a '*' and a '_' for replacing **[star]** and **[underscore]** respectively.

See the following snippet:

```
>> A * IS A *
## A [star].


>> A _ CAN BE USED FOR _ AND _
## For [underscore].
```

Let's see the outcome of the above code:

```
USER: A mango is a juicy fruit.
BOT: A juicy fruit.
```
(Last occurrence was *"juicy fruit"*.)

```
USER: A pencil can be used for writing and drawing.
BOT: For drawing.
```
(Last occurrence was *"drawing"*.)

The two wildcards can be used in a pattern simultaneously when two wildcard text replacements are required.

E.g.
```
>> A * CAN BE USED FOR * AND _
## For [star] and [underscore]
```

Which results in this:

```
USER: A pencil can be used for writing and drawing.
BOT: For writing and drawing.
```

Note: As of now, not more than two wildcard text replacements (as shown above) can be done, although any number of wildcard characters can be used in a pattern.

* * *