

[rivescript.com](https://www.rivescript.com)

Tutorial - RiveScript.com

50-64 minutes

RiveScript::Tutorial - Learn to write RiveScript code.

INTRODUCTION

This tutorial will help you learn how to write your own chatbot personalities using the RiveScript language.

What is RiveScript?

RiveScript is a text-based scripting language meant to aid in the development of interactive chatbots. A chatbot is a software application that can communicate with humans using natural languages such as English in order to provide entertainment, services or just have a conversation.

Getting Started

To **write** your own RiveScript code, you will only need a simple text editing program. You can use Notepad for Windows, or gedit for Linux, or any other text editors you have available.

A RiveScript document is a text file containing RiveScript

code. These files will have a `.rive` extension. An example file name would be `greetings.rive`. If you're writing RiveScript documents on Windows, you may need to put the file name in quotes when you save it, for example `"greetings.rive"` to be sure it gets saved with the file extension (and not `"greetings.rive.txt"`).

To **execute** and test your RiveScript code, you will need a **RiveScript Interpreter**, or, a program that uses a RiveScript library to read and execute RiveScript code.

The Perl [RiveScript](#) module comes with a ready-to-use interpreter that can be used for this tutorial! If you've installed the Perl RiveScript module on a Linux, Unix or Mac OS system, the RiveScript interpreter will probably be installed in a place like `/usr/bin/rivescript` or `/usr/local/bin/rivescript`. You can open a terminal window and run the command `"rivescript"` to use the RiveScript interpreter.

If you're using Windows, you can open a Command Prompt window and run the command `"rivescript"` to use the RiveScript interpreter.

If you are able to run the RiveScript interpreter that is shipped with the Perl distribution, you will be able to continue with this tutorial. If you're using a different RiveScript library, such as the Python version, please refer to the documentation of the library to see if it comes with a RiveScript interpreter that you can use.

If you need to write your own interpreter program, see

["Writing an Interpreter"](#) for an example of how to do this in Perl.

This tutorial will assume you are using the `rivescript` command shipped with the Perl RiveScript library.

Project Directory

For this tutorial, you should create a folder to save your RiveScript documents to. The following are some recommended locations, but you can place them wherever you like.

For Linux, Unix and Mac OS users, I recommend making a folder in your home directory, like so:

Unix: `/home/USER/rstut`

Mac: `/Users/USER/rstut`

Substitute `USER` with your username of course.

For Windows users, make a directory in the `C:\` drive, like so:

After you begin writing RiveScript documents, you can test your code at any time by running the RiveScript interpreter and pointing it at your reply directory.

For Linux, Unix and Mac OS users, open a terminal window and run a command such as the following:

For Windows, open a command prompt window (push the Start button, type `cmd` and hit Enter. For Windows XP or older, push the Windows key + R on your keyboard, and type `cmd` and hit enter in the Run dialog). Navigate to the

"bin" folder in the Perl RiveScript distribution by using the `cd` command, and then run the following command:

If you're having trouble getting this to work on Windows, see ["Windows Troubleshooting"](#) for help.

FIRST STEPS

Hello, Human!

Let's write our first few lines of RiveScript code!

In your text editor, create a new file and write the following in it:

```
! version = 2.0
```

```
+ hello bot
```

```
- Hello, human!
```

Save this in your project directory as `hello.rive`, and then run the RiveScript interpreter on that directory (see ["Project Directory"](#) for reference). You should see something along these lines in your terminal window:

```
RiveScript Interpreter - Interactive Mode
```

```
-----
```

```
RiveScript Version: 1.30
```

```
    Reply Root: rstut
```

You are now chatting with the RiveScript bot. Type a message and press Return to send it.

When finished, type `'/quit'` to exit the program. Type `'/help'` for other options.

You>

At the prompt, type "Hello bot" and press Enter. The bot should respond with "Hello, human!"


Try saying something else to the bot, such as "how are you?" and see what it says. The bot should respond with, "ERR: No Reply Matched". This is because you said something that there was no RiveScript code written to handle. There is only a handler for "hello bot", and nothing else. Later you will learn how to write a "catch-all" response that will be used when you say something the bot wasn't programmed to handle. But first, let's go over the code you wrote for `hello.rive`.

The Code, Explained


RiveScript code is really simple. Each line of the text file is a separate entity (RiveScript is a line-based scripting language). Lines of RiveScript code always begin with a command symbol (in this example, the symbols we see are `!`, `+`, and `-`) and they always have some kind of data that follows them. The data depends on the command used.

The line, `"! version = 2.0"` tells the RiveScript interpreter that your code follows version 2.0 of the

RiveScript specification. This way, future versions of the language can be backwards compatible with existing code by looking at the version number and responding accordingly. It is a good idea to always include the version line in your code (but it isn't the end of the world if you leave it out -- you'll just leave the interpreter to make its best guess about which version your code is using).

The  command is how you define a **trigger**. A trigger is a line of text that is used to match the user's message. In this case, "hello world" is exactly the message that we're matching. You will learn about some more complex features of triggers later in this tutorial.

Important Note: a trigger is ALWAYS lower cased, and it doesn't contain punctuation symbols. Even if you write a trigger that contains the proper noun "I", that "I" should be lowercased too. You may have noticed when testing your code that the interpreter doesn't care about the capitalizations used in your messages: you can say "Hello Bot", "HELLO BOT", or "hello bot" and it will match the trigger all the same.

The  command is how you define a response to a trigger. In this case, when the user matches the "hello bot" trigger, the bot should respond to the user by saying "Hello, human!"

Random Replies

Making your bot *a/ways* respond exactly the same way to

something the user says will get boring really quickly. For this reason, RiveScript makes it easy to add random responses to a trigger!

Getting random responses is as easy as entering multiple Response commands for the same trigger. To see this in action, open your `hello.rive` file from ["Hello, Human!"](#) and add the following lines to it:

```
+ how are you
- I'm great, how are you?
- I'm good, you?
- Good :) you?
- Great! You?
- I'm fine, thanks for asking!
```

Save this and then test it with the RiveScript interpreter. Ask your bot, "How are you?" a few times and see how it responds. It will say one of these five things at random each time you ask!

You can also use random strings *inside* a reply by using the `{random}` tag (see ["TAGS"](#)). Example:

```
+ say something random
- This {random}message|sentence{/random}
  has a random word.
```

Between the `{random}` and `{/random}` tags, you separate the strings with a pipe symbol and one will be chosen randomly.

A Note About Style

To keep your RiveScript documents nice and tidy and easy to read (and maintain!) you should follow these style guidelines:

- Use blank lines to separate logical groups of code.

For example, a trigger line and its responses should be grouped together, and a blank line should separate them from a different trigger and its responses.

- Indent code inside a topic or begin block.

You'll learn about these later in this tutorial.

So, our `hello.rive` should look like this now:

```
! version = 2.0

+ hello bot
- Hello, human!

+ how are you
- I'm great, how are you?
- I'm good, you?
- Good :) you?
- Great! You?
- I'm fine, thanks for asking!
```

Let's Talk About Weight

While random responses are certainly useful, there will be times when you would prefer that *some* replies would be chosen more frequently than others. For example, you might be writing a bot whose personality is that he's

secretly an alien pretending to be a human that's pretending to be a bot, and you want the bot to respond in some unintelligible gibberish every once in a while.

You can use the `{weight}` tag in a reply to override how frequently that reply will be chosen compared to the others. For our alien gibberish example, you could write a reply like this:

```
+ greetings
- Hi there!{weight=20}
- Hello!{weight=25}
- Yos kyoco duckeb!
```

Here, we've assigned a weight to each of the English responses, and left the gibberish one alone. The effect that this has is that "Hi there!" will be picked 20 times out of 46, "Hello!" will be picked 25 times out of 46, and "Yos kyoco duckeb!" will be chosen only 1 time out of 46.

You can test this by saying "greetings" to your bot over and over again. It should *very rarely* choose the "Yos kyoco duckeb!" response compared to the other two.

The weight value controls the probability that the reply is chosen. Replies that don't explicitly include a weight tag automatically have a weight of 1. The probability of each reply being chosen is the reply's weight divided by the sum of all the weights combined (in this example, $20 + 25 + 1 = 46$, so each reply has its weight out of 46 chance of being chosen).

Weight values can't be zero and they can't be negative.

You **can not** use weights inside a `{random}` tag.

Line Breaking

There will be times when you're writing a really long line of RiveScript code and you'd like to break it to span multiple lines. For these cases, you can use the `^` command (**Continuation**). The `^` command automatically extends the data from the previous line. Here is an example:

```
+ tell me a poem
- Little Miss Muffit sat on her tuffet,\n
^ In a nonchalant sort of way.\n
^ With her forcefield around her,\n
^ The Spider, the boulder,\n
^ Is not in the picture today.
```

Note that the Continuation command doesn't automatically insert a space between the previous line and the continuation line. Consider the following example:

```
// There will be no space between
"programmed" and "using"!
+ what are you
- I am an artificial intelligence
programmed
^ using RiveScript.
```

If you asked "what are you" with this reply, the bot would say, "I am an artificial intelligence programmedusing RiveScript.", with no space between "programmed" and "using".

To make sure there's a space between continuations, use the escape sequence `\s` where you want the space to appear.

```
// This one will have a space.  
+ what are you  
- I am an artificial intelligence  
programmed\s  
^ using RiveScript.
```

From the "tell me a poem" example, the escape sequence `\n` inserts a line break instead of a space.

If you find that you frequently want to use continuations and you almost always want to have spaces (or line breaks) between each one, you can tell the parser to always insert these symbols automatically when it sees a `^` command.

You do this with a "local parser option" which only applies to the current file and affects the lines that come after the parser option. For example:

```
// Tell the parser to join continuation  
lines with line breaks  
! local concat = newline  
  
// Now we don't need to explicitly write  
the \n characters every time!  
+ tell me a poem  
- Little Miss Muffit sat on her tuffet,  
^ In a nonchalant sort of way.
```

```
^ With her forcefield around her,  
^ The Spider, the boulder,  
^ Is not in the picture today.  
  
// Now change the concat mode to spaces  
! local concat = space  
  
// Here we don't have to use \s like in the  
earlier example.  
+ what are you  
- I am an artificial intelligence  
programmed  
^ using RiveScript.  
  
// Go back to the default concatenation  
mode (which doesn't insert ANY  
// character when joining lines)  
! local concat = none
```

Because the `! local concat` setting changes the way the parser handles continuation lines, this setting is "file scoped" and only affects the parser while in the current file. When the parser is done with one file and begins processing the next one, the concatenation setting is set back to the default (`none`) and no spaces or line breaks will be automatically added in continuation commands.

The supported options for this are:

- `none` -- the default, nothing is added when continuation lines are joined together.

- `space` -- continuation lines are joined by a space character (`\s`)
- `newline` -- continuation lines are joined by a line break character (`\n`)

ANATOMY OF A RIVESCRIPT BRAIN

The Begin File

You now know some of the basics about how triggers and replies relate to each other. Before continuing, you should know how RiveScript brains are typically organized.

RiveScript brains (a "brain" is a set of RiveScript documents) should, by convention, include a document named `begin.rive` that contains some configuration settings for your bot's brain. The most useful settings that would be set here include **substitutions**, which are able to make changes to the user's message *before* a reply is looked for.

You may have noticed that the RiveScript interpreter doesn't care about punctuation in your messages (you can say "Hello bot!!!" and it ignores the exclamation marks), so what does that mean for things such as the word "what's"? By default, the word "what's" would be converted into "whats" before the interpreter looks for a reply for it. With substitutions, you can see to it that "what's" is expanded into "what is" instead -- allowing you better control over how you reply to the user's message!

Let's start with our `begin.rive` file. In your text editor, create a new document and write the following code into it (this code will be explained below):

```
! version = 2.0

// Bot variables
! var name = Tutorial
! var age  = 5

// Substitutions
! sub i'm      = i am
! sub i'd      = i would
! sub i've     = i have
! sub i'll     = i will
! sub don't    = do not
! sub isn't    = is not
! sub you'd    = you would
! sub you're   = you are
! sub you've   = you have
! sub you'll   = you will
! sub what's   = what is
! sub whats    = what is
! sub what're  = what are
! sub what've  = what have
! sub what'll  = what will
! sub who's    = who is
```

Save this as `begin.rive` in your project directory. Now, for an explanation on what this code is doing.

Definitions

You've already seen the `!` command used for the version line, but what are all these other lines? More generally, the `!` command is used for **Definitions** (just like the `+` is for **Triggers** and the `-` is for **Replies**). In the version line, we are *defining* that the version is 2.0.

Also, you may be wondering what the `//` characters are for. Like in most programming languages, RiveScript allows you to include **Comments** in your source code. The `//` characters denote the start of a comment. The interpreter will ignore these comments when it reads your code; they're only there for the humans (you!) who have to read and maintain the code.

Now let's go over these new definition types.

First, we defined a couple of **Bot Variables**. These are pieces of information that describe your bot, such as its name and age in this example. These will come in handy later. With bot variables, we can write replies in which the bot can tell the user a little something about itself.

Then, we defined a handful of **Substitutions**.

Substitutions are *always lowercased*. On the left side of the `=` sign, you write the "original text" that may appear in the user's message, and on the right you place the substituted text. The text on the right *should not* contain any special symbols or punctuation, and it should also be lowercased.

With these substitutions, if a user says to the bot, "what's up?" or "I've been at work all day", the RiveScript interpreter will expand these messages out to "what is up" and "i have been at work all day", respectively, before it starts looking for a reply.

Here is some more code that you can add to your `hello.rive` file from earlier that demonstrates how substitutions work:

```
+ what is up
- Not much, you?
- nm, you?
- Not a lot, you?

+ you are a bot
- How did you know I'm a machine?
```

You can then ask your bot, "What's up?" or "You're a bot" and see that it matches these replies accordingly. Notice that the substitutions didn't apply to the bot's response to "you're a bot" -- it says "How did you know I'm a machine?" -- substitutions only apply to the user's incoming message. They're also used with the `% Previous` command, but we'll get to that later.

There are other types of definition commands available too: `array`, `global`, and `person`. These are useful for more advanced replies and they'll be covered later in this tutorial.

Note: there will be many more examples of RiveScript

code in this tutorial. You can put these in any RiveScript document you wish; you can create a new `.rive` file for them if you like. Now and then I'll mention a recommended file name, though, but the names don't really matter that much.

TRIGGERS

Open-Ended Triggers

So far, the triggers you've seen have been what I call "atomic" -- they describe a user's message *perfectly*. For example, the user must say exactly "hello bot"; they can't say "hello there" or "hello robot" and still match one of your triggers, unless you've written individual triggers for each possible thing they could say!

This is where **Wildcards** come into play. With wildcards, you can mark a part of the trigger as being open-ended. The best way to demonstrate this is with an example:

```
+ my name is *  
- Nice to meet you, <star1>!  
  
+ * told me to say *  
- Why would <star1> tell you to say  
"<star2>"?  
- Did you say "<star2>" after <star1> told  
you to?  
  
+ i am * years old
```

- A lot of people are <star1> years old.

With these triggers, a user can say "My name is Noah", or "I am 24 years old", and the bot will be able to match these messages all the same. Wildcards are very useful to match messages that may contain "variable" data, such as names or numbers. They're also useful for your bot to be able to fake knowledge about a subject:

```
+ where is *
```

- Where it belongs.
- Where you left it.
- Where the heart is.

You can write triggers for common questions like "who is", "where is", and "what is" by using wildcards; so, if the user asks your bot about something that your bot doesn't have a special trigger to handle, it can give a sort of "generic" response that will be at least somewhat relevant to the question.

You may have noticed the <star1> and <star2> tags that appeared in some of those replies up there. These tags can be used in a reply in order to repeat the words matched by the wildcards. When the user says "my name is Noah", the first wildcard in that trigger would catch the name, and <star1> would be "noah" in this case.

If you only have a single wildcard, you may just use the <star> tag without the number "1" as a shortcut:

```
+ who is *
```

- I don't know who <star> is.

While we're on the topic of wildcards...

Catch-All Trigger

Remember back in ["Hello, Human!"](#) where the bot would say **"ERR: No Reply Matched"** whenever we said something it couldn't reply to? Well, we can remedy this by writing a catch-all trigger.

A catch-all trigger is one that simply consists of a single wildcard:

```
+ *  
- I'm not sure how to reply to that.  
- Try asking your question a different way.  
- Let's change the subject.
```

Anything the user says now that doesn't get matched by a more relevant trigger will fall back to the `*` trigger. This way, you can avoid letting the bot say "ERR: No Reply Matched", and use it to try to steer the conversation back on track.

Conventionally, your catch-all trigger should go into a file named `star.rive`, so that when you're looking for it later to make changes you'll know exactly where you put it.

Specialized Wildcards

Wildcards are great, but what if you want to restrict what a wildcard is allowed to match? For example, the trigger `"i am * years old"` would match a message like `"I am twenty four years old"` just as well as `"I am 24 years old"`.

There are two other wildcard symbols you may use. The `#` symbol is a wildcard that will *only* match a number. The `_` symbol is one that will *only* match a word with no numbers or spaces in it.

You can have multiple triggers that look the same but use different wildcards and they will work how you'd expect:

```
+ i am # years old
```

```
- A lot of people are <star> years old.
```

```
+ i am _ years old
```

```
- Tell me that again but with a number this time.
```

```
+ i am * years old
```

```
- Can you use a number instead?
```

Regardless of the type of wildcard you use, you can use the `<star>` tags to pull them into the reply.

Alternatives and Optionals

What if you want to use something like a wildcard, but you want to limit the possible words to a select few? This is where optionals come into play.

The syntax for these is a little tricky. Let's start with some examples:

```
+ what is your (home|office|cell) number
```

```
- You can reach me at: 1 (800) 555-1234.
```

```
+ i am (really|very|super) tired
- I'm sorry to hear that you are <star>
  tired.
```

```
+ i (like|love) the color *
- What a coincidence! I <star1> that color
  too!
- I also have a soft spot for the color
  <star2>!
- Really? I <star1> the color <star2> too!
- Oh I <star1> <star2> too!
```

In these examples, a user can say "what is your home number", or "what is your office number", or "what is your cell number" and match the first trigger. Or they can say "I am really tired", "I am very tired", or "I am super tired" and match the second one. And so on. But, if the user says "I am extremely tired", it won't match because "extremely" wasn't listed in the alternatives!

The alternative that the user used in their message can be captured with a `<star>` tag too, just like wildcards.

Alternatives don't have to be single words, either.

```
+ i (will|will not) *
- It doesn't matter to me whether you
  <star2> or not.
```

Optionals are like alternatives, but they don't *need* to be present in the user's message *at all*! But, if the user does say them, it will help match the reply anyway.

```
+ how [are] you
- I'm great, you?

+ what is your (home|office|cell) [phone]
number
- You can reach me at: 1 (800) 555-1234.

+ i have a [red|green|blue] car
- I bet you like your car a lot.
```

Since optionals don't have to be present in the user's message, they *can not* be captured with `<star>` tags. If you had a wildcard or alternative before and after an optional, `<star1>` would be the first wildcard or alternative, and `<star2>` would be the second; the optional would be skipped.

A clever thing you can do with optionals is write "keyword" triggers: if the user says a magic word ANYWHERE in their message, your trigger will match!

```
+ [*] the machine [*]
- How do you know about the machine!?
```

You can also use these `[*]` optionals to ignore parts of a message by putting it before or after your trigger instead of on both sides.

Arrays in Triggers

Consider something a human might say to a bot: "what color is my blue shirt?" You might be able to program a

reply to this using wildcards, but alternatives would be even better, since you can limit the color to a small set.

```
+ what color is my (red|blue|green|yellow)
*
- Your <star2> is <star1>, silly!
```

Wouldn't it be useful to re-use this list of colors for other triggers without having to copy and paste it all over the place? Well, that's exactly the reason why **Arrays** exist! In RiveScript, you can make a list of words or phrases, give that list a name, and then use it in a trigger (or multiple triggers!)

You define an array using the `! array` command, which was first mentioned in the ["Definitions"](#). By convention, all definitions belong in `begin.rive`, so write the following code in `begin.rive`:

```
! array colors = red blue green yellow
```

Now, you can refer to this array by name in your triggers. Here are a couple examples you can use (you can use arrays in as many triggers as you want):

```
+ what color is my (@colors) *
- Your <star2> is <star1>, silly!
- Do I look dumb to you? It's <star1>!
```

```
+ i am wearing a (@colors) shirt
```

```
- Do you really like <star>?
```

Just like wildcards and alternatives, the word the user

used out of the array can be captured in a `<star>` tag. If you don't want this to happen, you can use the array without the parenthesis around it:

```
// Without parenthesis, the array doesn't
go into a <star> tag.
+ what color is my @colors *
- I don't know what color your <star> is.
```

Arrays can be used in optionals too. They don't go into `<star>` tags though, because optionals *never* do!

```
// Arrays in an optional
- i just bought a [@colors] *
- Is that your first <star>?
```

When defining arrays, you can either separate the array items with spaces (useful for single words) or pipe symbols (for phrases). Examples:

```
// Single word array items
! array colors = red blue green yellow

// Multiple word items
! array blues = light blue|dark blue|medium
blue
```

If you use Continuations when defining an array, you can swap between spaces and pipes on each line. Here is a very thorough array of colors:

```
// A lot of colors!
! array colors = red blue green yellow
orange cyan fuchsia magenta
```



```
^ light red|dark red|light blue|dark
blue|light yellow|dark yellow
^ light orange|dark orange|light cyan|dark
cyan|light fuchsia
^ dark fuchsia|light magenta|dark magenta
^ black gray white silver
^ light gray|dark gray
```

Priority Triggers

You're almost done learning about all the things that can be done to a trigger!

The last thing is weighted, or priority triggers. You've seen the `{weight}` tag applied to responses before. Well, the same tag can also be used in a trigger!

A weighted trigger has a higher matching priority than others. This is useful to "hand tune" how well a trigger matches the user's message. An example of this would be, suppose you have the following two triggers:

```
+ google *
- Google search: <a href="http://google.com
/search?q=<star>">Click Here</a>

+ * perl script
- You need Perl to run a Perl script.
```

What if somebody asked the bot, "google write perl script"? They might expect the bot to provide them with a Google search link, but instead the bot replies talking

about needing Perl. This is because `"* perl script"` has more words than `"google *"`, and therefore would usually be a better match.

We can add a `{weight}` tag to the Google trigger to make that trigger "more important" than anything with a lower weight.

```
+ google *{weight=10}
- Google search: <a href="http://google.com
/search?q=<star>">Click Here</a>

+ * perl script
- You need Perl to run a Perl script.
```

Now, if the user starts their message with "google", that trigger will have a higher priority for matching than anything else. The weights on triggers are arbitrary, and higher numbers just mean it has a higher priority than ones with lower numbers. Triggers that don't have a `{weight}` tag automatically have a weight of 1.

You can't have a zero or negative weight value.

If you have multiple triggers with the same weight value, these triggers are considered equals, and their matching order will be the same as usual (triggers with more words are tested first). If no triggers with a given weight can match the user's message, then triggers with a lower weight are tried.

See the "Sorting +Triggers" section of the RiveScript Working Draft for a detailed explanation of how triggers

are sorted.

MORE COMMANDS

Redirections

If a user matches a trigger, you can have that trigger simply redirect them somewhere else, as though they had asked a different question. Example:

```
+ hello
- Hi there!
- Hey!
- Howdy!
```

```
+ hey
@ hello
```

```
+ hi
@ hello
```

In this example, if the user says "hey" or "hi", the bot redirects them to the "hello" trigger, as though they had said hello to begin with.

Of course, with alternatives and the other advanced trigger features, redirects like this aren't always useful. But you can also use redirects *inside* of replies. One of my favorite examples of this:

```
+ * or something{weight=100}
- Or something. {@ <star>}
```

If the user says, "Are you a bot or something?", the bot might reply, "Or something. How did you know I'm a machine?"

If you just want to use `{@ <star>}`, you can use a shortcut tag instead: `<@>`. Use the `{@ . . . }` format for everything else:

```
+ hello *  
- {@ hello} <@>
```

```
+ hello  
- Hi there!
```

```
+ are you a bot  
- How did you know I'm a machine?
```

This trigger would reply to "Hello, are you a bot?" with a reply like "Hi there! How did you know I'm a machine?"

Short Discussions

Suppose you want to program your bot to be able to play along with a user who is telling a Knock-Knock joke? You can pull this off with the `%` command (**Previous**):

```
+ knock knock  
- Who's there?
```

```
+ *  
% who is there  
- <star> who?
```

```
+ *  
% * who  
- LOL! <star>! That's funny!
```

This example introduces the `%` command. The `%` command is similar to the `+` used for triggers, except it looks at the bot's previous response to the user instead. In the second trigger here, if the bot's previous response was "who is there", anything the user says (`*`) will match, and the bot will continue playing along with the joke.

In the `%` command, the bot's previous response is sent through the same substitutions as the user's messages are. Notice that the bot's reply was "Who's there?", but the `%` line on the next trigger says "who is there". This is because the "Who's" was substituted for "who is" due to the substitution defined in your `begin.rive` file!

`%` Previous lines need to be lowercased just like triggers do.

Here is another example:

```
+ i have a dog  
- What color is it?  
  
+ (@colors)  
% what color is it  
- That's a silly color for a dog!
```

Now you can say "I have a dog" and the bot will ask what color it is. If you tell it the color, it will say "That's a silly

color for a dog!" -- but, if you ignore the bot's question and say something else, the bot will just reply to your new message as usual.

Conditionals

Learning Things

You now know most of the RiveScript commands and how to use them. But what good is a chatbot if it can't even remember your name?

RiveScript has the capability to store and repeat variables about users. To set a user variable, we use the `<set>` tag, and to retrieve the variable we use `<get>`. Here are some examples of how we can learn and repeat information about the user.

```
+ my name is *  
- <set name=<star>>It's nice to meet you,  
<get name>.
```

```
+ what is my name  
- Your name is <get name>, silly!
```

```
+ i am # years old  
- <set age=<star>>I will remember that you  
are <get age> years old.
```

```
+ how old am i  
- You are <get age> years old.
```

While we're talking about variables, what about those bot variables we defined in `begin.rive`? You can retrieve them in a similar fashion:

```
// The user can ask the bot its name too!  
+ what is your name  
- You can call me <bot name>.  
- My name is <bot name>.  
  
+ how old are you  
- I am <bot age> years old.
```

Now, you may notice that if you tell the bot, "My name is Noah", it will store your name as "noah" -- lowercased. To store the name as a proper noun instead, you can use the formal tag. See ["TAGS"](#).

```
// Store the name with the correct casing  
+ my name is *  
- <set name=<formal>>Nice to meet you, <get  
name>!
```

The `<formal>` tag is a shortcut for `{formal}<star>{/formal}`, so you will need to use the `{formal}...{/formal}` syntax to formalize other things. See ["TAGS"](#).

Writing Conditionals

And with learning information about the user, conditionals let us pick replies based on the values of those variables!

You may notice that if you asked the bot what your name was *before* you told the bot your name, it would say "Your

name is undefined, silly!" This doesn't look very professional and would give away that the bot is just a program.

With conditionals, you can make sure the bot knows a user's name before it opens its big mouth, and say something else if it doesn't know.

Here is an example:

```
+ what is my name
* <get name> == undefined => You never told
me your name.
- Your name is <get name>, silly!
- Aren't you <get name>?
```

Now, if you ask the bot your name, it will see if your name is "undefined", and if so, it will say "You never told me your name." Otherwise, it will give one of the other replies.

Conditions are used to compare variables to values. You can also compare variables to other variables. Here is a more advanced way of telling the bot what your name is:

```
+ my name is *
* <formal> == <bot name> => Wow, we have
the same name!<set name=<formal>>
* <get name> == undefined => <set
name=<formal>>Nice to meet you!
- <set oldname=<get name>><set
name=<formal>>
^ I thought your name was <get oldname>?
```

If you're feeling a little bit cramped on the condition lines,

using the `^ Continuation` command is useful to get some more room.

Conditions are checked in order from top to bottom. If no condition turns out true, the normal – replies are used. If you want to have a random response when a condition is true, you need to use the `{random}` tag.

With conditionals, you can use the following equality tests:

```
== equal to
eq equal to (alias)
!= not equal to
ne not equal to (alias)
<> not equal to (alias)
```

The following equality tests can be used on variables that contain numbers only:

```
< less than
<= less than or equal to
> greater than
>= greater than or equal to
```

Here is an example using different equality tests:

```
+ what am i old enough to do
* <get age> == undefined => I don't know
  how old you are.
* <get age> > 25 => You can do anything
  you want.
* <get age> == 25 => You're old enough to
  rent a car with no extra fees.
* <get age> > 21 => You're old enough to
```

drink, but not rent a car.

* <get age> == 21 => You're exactly old enough to drink.

* <get age> > 18 => You're old enough to gamble, but not drink.

* <get age> == 18 => You're exactly old enough to gamble.

- You're not old enough to do much of anything yet.

LABELED SECTIONS

There are three types of labeled sections. Labels are defined using the `>` command symbol, and they're ended with `<`. All labeled sections should be properly closed when done (even if it's at the end of the file). For style purposes, you should indent the contents of a labeled section too. Labeled sections can not be embedded inside each other.

Topics

Topics are logical groupings of triggers. When a user is in a topic, they can only match triggers that belong to that topic. Here's an example:

+ i hate you

- You're really mean! I'm not talking again until you apologize.{topic=sorry}

> topic sorry

```
// This will match if the word "sorry"
exists ANYWHERE in their message
+ [*] sorry [*]
- It's OK, I'll forgive
you!{topic=random}

+ *
- Nope, not until you apologize.
- Say you're sorry!
- Apologize!
```

< topic

In this example, if the user tells the bot that they don't think very much of it, the bot will force the user to apologize before continuing conversation. Once the user has been put into the "sorry" topic, the **ONLY** triggers available for matching are the two in that topic.

The default topic is "random", and you change the user's topic using the `{topic}` tag, for example `{topic=sorry}` and `{topic=random}`.

You might be wondering why you can't use `<set topic=random>` instead. Well, you *can*, but there is a small difference in how the two tags will behave:

The `<set>` tag can appear multiple times in a reply and each one is processed in order. The `{topic}` tag can only appear once (if there are multiple ones, the first one

wins). So, they'll both do the same job, but `{topic}` is a little shorter to type.

Topics are capable of inheriting and including triggers from other topics, too. But, this is for more advanced users and is outside the scope of this tutorial (see the file `rpg.rive` that comes with standard RiveScript distributions for a practical example of this).

You may be wondering what happens if you accidentally set the topic to one that doesn't exist? Would the user be unable to chat with the bot anymore since no replies can be matched? Fortunately, the RiveScript libraries are smart enough to detect this and will place the user back in the "random" topic automatically.

The Begin Block

The Begin Block is an optional feature of a RiveScript brain. The Begin Block will typically be found in `begin.rive`. They work in a similar fashion as topics. Here is an example:

```
> begin

+ request
- {ok}

< begin
```

The Begin block serves as a pre-processor *and* post-processor for fetching a response. If the Begin Block is

present, the `request` trigger will be tried for each message the user says. If the response to this contains the `{ok}` tag, then a reply is fetched for the user's message and the `{ok}` tag is substituted out.

Here is a longer example that use the pre-processing capabilities of the `Begin` Block to introduce itself to new users and interview them:

```
> begin
```

```
    // If we don't know their name, set the
    new_user topic and continue.
```

```
    + request
```

```
    * <get met> == undefined => <set
    met=true>{topic=new_user}{ok}
    - {ok}
```

```
< begin
```

```
> topic new_user
```

```
    + *
```

```
    - Hi! I'm <bot name>! I'm a chatbot
    written in RiveScript.\s
```

```
    ^ What is your name?{topic=asked_name}
```

```
< topic
```

```
> topic asked_name
```

```
+ #
- Your name is a number?

+ *
- I only want your first name.

+ _
- <set name=<formal>>Nice to meet you,
<get name>!{topic=random}

< topic
```

And here is another example that would use the post-processing capability for the Begin Block:

```
> begin

    // Change the reply formatting based on
the bot's mood
    + request
    * <bot mood> == happy => {sentence}
{ok}/{/sentence}
    * <bot mood> == angry => {uppercase}
{ok}/{/uppercase}
    * <bot mood> == sad    => {lowercase}
{ok}/{/lowercase}...
    - {ok}

< begin
```

With this example, the bot would change the formatting of its response based on a "mood" variable. It will be left as an exercise for you to decide how the mood variable would be set (you can use e.g. `<bot mood=happy>` to change the value of a bot variable from a reply).

Object Macros

In RiveScript, an Object Macro is a programming function written in another programming language. Usually, a RiveScript library written in a dynamic programming language such as Perl or Python will support Object Macros of the same language. The RiveScript libraries will also allow you to add custom language handlers to support other languages, but this will be up to the programmer and is outside the scope of this tutorial.

Object macros allow you to do more powerful things with your bot's responses. For example, you can allow the user to ask the bot questions about the current weather or about movie ticket prices, and your bot can call an object macro that goes out to the Internet to fetch that information.

All object macros should define the programming language they're written in, in lowercase (e.g. "perl", "python", "javascript").

Here is an example of an object macro written in Perl (you will be able to run this code using the Perl RiveScript interpreter).

```
// The object name is "hash", written in
Perl
> object hash perl
    my ($rs, $args) = @_;
    my $method = shift @{$args};
    my $string = join " ", @{$args};

    # Here, $method is a hashing method (MD5
or SHA1), and $string
    # is the text to hash.

    if ($method eq "MD5") {
        require Digest::MD5;
        return Digest::MD5::md5_hex($string);
    }
    elsif ($method eq "SHA1") {
        require Digest::SHA1;
        return Digest::SHA1::sha1_hex($string);
    }
< object

// You call an object using the <call> tag.
+ what is the md5 hash of *
- The hash of "<star>" is: <call>hash MD5
<star></call>

+ what is the sha1 hash of *
- The hash of "<star>" is: <call>hash SHA1
<star></call>
```


Here is another example of an object that would tell you the bot's public IP address, using icanhazip.com. You'd probably want to secure this question and only allow the bot's owner to get the IP address for obvious privacy reasons. So, you can have an "authentication" system before the bot will allow you to see its IP address.

```
// To gain botmaster power, say "I am your
master"...
+ i am your master
- Then you must know the secret password:

// And then enter the botmaster password...
+ *
% then you must know the secret password
* <star> == rivescript is awesome =>
Correct password!<set master=true>
- That's not the right password. :-P

// And after authenticated, let them get
the bot's IP address!
+ what is your ip address
* <get master> == true => My IP address is:
<call>myip</call>
- You're not my master so you don't need to
know! :-P

// The object macro that fetches an IP
address.
```

```
> object myip perl
    my ($rs, $args) = @_;

    # Fetch the IP.
    use LWP::Simple;
    my $ip = get "http://icanhazip.com";
    return $ip;
< object
```

As you can see from this example, you don't need to declare your object macro before calling it with a `<call>` tag. The RiveScript interpreter parses *all* of your code before you can start getting replies, so it will find your object macro definition no matter where you put it.

You can test this code with the Perl RiveScript interpreter like so:

```
You> What is your IP address?
Bot> You're not my master so you don't need
to know! :-P
You> I am your master.
Bot> Then you must know the secret
password:
You> RiveScript is awesome
Bot> Correct password!
You> What is your IP address?
Bot> My IP address is: 67.205.20.243
```

See the documentation of your RiveScript library to see how to write object macros (in particular, to see the format

in which arguments are passed to your object).

MORE DEFINITIONS

To go back to `begin.rive` definition commands, there are two more types that may come in handy.

Global Variables

Global variables are similar to bot variables, but they're not particularly related to your bot as an entity. Global variables may be defined by your RiveScript interpreter (a common example would be to make all the environment variables of your program available as globals in RiveScript, for example `<env REMOTE_ADDR>` could be used to show the user's IP address in a CGI environment). However, there are two special globals that affect the RiveScript interpreter directly:

- Debug Mode

```
! global debug = true
! global debug = false
```

This global variable will turn debug mode on or off in the RiveScript interpreter. Debug mode typically will print lines of text to the terminal window that will thoroughly document *what* the interpreter is doing (for example, if debug mode is on while parsing files from disk, it will print every line of RiveScript code it sees and a short summary of what it's doing with it. When it's on while fetching a reply, it will print every trigger that it's trying to compare

the user's message to, etc.)

The value should be `true` or `false`.

- **Recursion Depth**

Because RiveScript replies can redirect to each other, there is some protection in place to avoid infinite recursion (for example, a trigger that redirects to another one, and that trigger redirects back to the first one, forever). By default the recursion depth limit is set to 50, meaning if RiveScript can't find a reply after 50 redirects it will give up and say "ERR: Deep Recursion Detected".

This value should be a positive number higher than zero.

Person Substitutions

These are a special kind of substitution that is intended to swap first- and second-person pronouns. Here is how you define the person substitutions (usually in your `begin.rive` file):

```
! person i am      = you are
! person you are   = i am
! person i'm       = you're
! person you're    = I'm
! person my        = your
! person your      = my
! person you       = I
! person i         = you
```

Notice in this example that each pair of substitutions

works in both directions.

To invoke person substitutions, you can use the `<person>` tag (to substitute on the `<star>`), or `{person}...{/person}` to substitute on something else. Here is a practical example to show why person substitutions can come in handy:

```
+ say *  
- Umm... "<person>"
```

Now, if the user says, "say I am the greatest", the bot will reply with "Umm... "you are the greatest"". Without person substitutions, the bot would repeat "i am the greatest" instead.

TAGS

By now, you've seen examples of all the different RiveScript commands. Sprinkled here and there throughout the examples were tags. Examples of tags you've seen are `<star>`, `<get>`, and `{topic}`. There are many more tags available! This section will teach you about all the tags and how to use them.

In general, a tag that has brackets are tags that insert text in their place, or tags that set a variable silently. Tags that have `{curly}` brackets modify the text around them.

Tags can be used with all the RiveScript commands except where explicitly noted.

- `<star>`, `<star1>` - `<starN>`

The star tag is used for capturing values used in wildcards, alternatives and arrays present in the matched trigger. You've seen many examples of this in this tutorial. If you have multiple stars, you can use `<star1>`, `<star2>`, `<star3>`, etc. to use the stars in order from left to right. The `<star>` tag is an alias for `<star1>`, so if you only have one wildcard this tag can be useful.

These tags can not be used with `+ Trigger`.

- `<botstar>`, `<botstar1>` - `<botstarN>`

This tag is similar to `<star>`, but it captures wildcards present in a `% Previous` line. Here is an example:

```
+ i bought a new *  
- Oh? What color is your new <star>?  
  
+ (@colors)  
% oh what color is your new *  
- <star> is a pretty color for a <botstar>.
```

Like the `<star>` tag, `<botstar>` is an alias for `<botstar1>`.

These tags can not be used with `+ Trigger`.

- `<input>`, `<reply>`

The input and reply tags are used for showing previous messages sent by the user and the bot, respectively. The previous 9 messages and responses are stored, so you can use the tags `<input1>` through `<input9>`, or `<reply1>` through `<reply9>` to get a particular message

or reply. `<input>` is an alias for `<input1>`, and `<reply>` is an alias for `<reply1>`.

Here are a couple examples:

```
// If the user repeats the bot's previous message
```

```
+ <reply>
```

```
- Don't repeat what I say.
```

```
// If the user keeps repeating themselves over and over.
```

```
+ <input1>
```

```
* <input1> == <input2> => That's the second time you've repeated yourself.
```

```
* <input1> == <input3> => If you repeat yourself again I'll stop talking.
```

```
* <input1> == <input4> => That's it. I'm not talking.{topic=sorry}
```

```
- Please don't repeat yourself.
```

```
// An example that uses both tags
```

```
+ why did you say that
```

```
- I said, "<reply>", because you said, "<input>".
```

- `<id>`

This tag inserts the user's ID, which was passed in to the RiveScript interpreter when fetching a reply. With the interpreter shipped with the Perl RiveScript library, the

`<id>` is, by default, `localuser`.

RiveScript uses user IDs to keep multiple users separate. You can use the same RiveScript interpreter to serve responses to multiple users, and it will keep their user variables separate based on their ID.

Here is an example of how you might distinguish the botmaster from other users based on a screen name (for example, for an instant messenger bot, where the user ID is set to the user's screen name).

```
! var master = kirsle

+ am i your master
* <id> == <bot master> => Yes, you are. Hi Kirsle!
- No, <bot master> is my master, and you are <id>.
```

- `<bot>`

The `<bot>` tag is used for retrieving a bot variable. It can also be used to set a bot variable.

Bot variables can be considered "global" to the RiveScript interpreter instance. That is, if you set the bot's name to Aiden, its name will be Aiden for everybody who asks, regardless of the user's ID. This is in contrast to user variables which are tied to a specific user ID.

```
+ what is your name
- You can call me <bot name>.
```



```
+ tell me about yourself
- I am <bot name>, a chatterbot written by
<bot master>.
```

```
// Setting a bot variable dynamically
+ i hate you
- Aww! You've just ruined my day.<bot
mood=depressed>
```

- `<env>`

The `<env>` tag is used for retrieving global variables. It can also be used to set a global variable.

For example, if the RiveScript interpreter copies all its environment variables into RiveScript globals, a CGI-based RiveScript bot could tell a user their IP address.

```
+ what is my ip
- Your IP address is: <env REMOTE_ADDR>
```

And here is an example of how you can set a global using this tag. In this example, the bot's master is able to turn debug mode on or off dynamically.

```
+ set debug mode (true|false)
* <id> == <bot master> => <env debug=
<star>>Debug mode set to <star>.
- You're not my master.
```

- `<get>`, `<set>`

These tags are used to get or set a user variable, respectively. User variables are arbitrary name/value

pairs. You can make up any variable name you want.

Here are some examples:

```
+ my name is *  
- <set name=<formal>>Nice to meet you, <get  
name>.
```

```
+ i am # years old  
- <set age=<star>>I will remember that you  
are <get age> years old.
```

```
+ what do you know about me  
- I know your name is <get name> and you  
are <get age> years old.
```

If you attempt to `<get>` a variable that had never been defined for the user before, it will insert the word "undefined" in its place instead. You can use this feature to test whether a variable has been defined:

```
+ do you know my name  
* <get name> != undefined => Yes, your name  
is <get name>.  
- No, you've never told me your name  
before.
```

`<get>` can be used in a trigger, but `<set>` can not.

- `<add>`, `<sub>`, `<mult>`, `<div>`

These tags can add, subtract, multiply or divide a numeric user variable, respectively.

```
+ give me 5 points
- <add points=5>You have been given 5
points. Your balance is: <get points>.
```

If you operate on a variable that isn't defined, it will be initialized to zero first. If you operate on a variable that doesn't currently contain a number, an error message will appear in the place of the tag.

These tags can not be used with `+ Trigger`.

- `{topic}`

This tag changes the client's topic.

```
+ play hangman
- {topic=hangman}Now playing hangman. Type
"quit" to quit.
```

```
> topic hangman
```

```
+ quit
- Quitting the game.{topic=random}
```

```
+ *
- <call>hangman <star></call>
```

```
< topic
```

This tag can not be used with `+ Trigger`.

- `{weight}`

This tag applies a weight to a trigger or response. When

used with a trigger, it controls the matching priority of the trigger (a higher weight means higher priority). When used with a reply, it controls how frequently that reply will be randomly chosen.

```
+ * or something{weight=10}
- Or something. <@>
```

```
+ hello
- Hi there!{weight=20}
- Hey!{weight=10}
- Howdy!
```

This tag can only be used with `+ Trigger` and `- Reply`.

- `{@}`, `<@>`

This tag performs an inline redirection to a different trigger.

`<@>` is an alias for `{@ <star>}`.

```
+ your *
- I think you mean to say "you are" or
  "you're", not "your". {@you are <star>}
```

You don't need to include a space between the `@` and the trigger text.

This tag can not be used with `+ Trigger`.

- `{random}`

Insert a sub-set of random text. This can come in handy in conditional lines, if you want a random reply for a condition.

Between the `{random}` and `{/random}` tags, separate your possible texts with a pipe symbol. Here is an example:

```
+ hello
* <get name> != undefined => {random}
^ Hello there, <get name>!!
^ Nice to see you again, <get name>!!
^ Hey, <get name>!{/random}
- Hello there!
- Hi there!
- Hello!
```

In this example, you give a random response that includes the user's name if the bot knows it, or a normal random response otherwise.

This tag can not be used with `+ Trigger`.

- `{person}`, `<person>`

Processes person substitutions on some text. See ["Person Substitutions"](#).

`<person>` is an alias for `{person}<star>{/person}`.

This tag can not be used with `+ Trigger`.

- `<formal>`, `<sentence>`, `<uppercase>`, `<lowercase>`

Change the case formatting on some text. `<formal>` is an alias for `{formal}<star>{/formal}`. The same applies for the other tags.

Formal text makes the first letter of each word uppercase.

This is useful for names and other proper nouns.

Sentence text makes the first word of each sentence uppercase.

Uppercase and **lowercase** make the entire string upper or lower case.

- `<call>`

This tag is used to call an object macro. Example:

```
// Call a macro named "reverse" and give it
an argument
+ say * to me in reverse
- <call>reverse <star></call>
```

The first word in the `<call>` tag is the name of an object macro to call (see ["Object Macros"](#)). The macros may optionally take some arguments, and you can pass them in by placing them in the `<call>` tag as shown in this example.

- `{ok}`

This is only used in the Begin Block in response to the `request` trigger. It indicates that it's OK to fetch a reply for the user's message. The reply will replace the `{ok}` tag.

- `\s`

This inserts a space character. It's useful when using the `^` `Continue` command to extend a line of RiveScript code.

- `\n`

This inserts a line break.

CONCLUSION

This concludes the RiveScript tutorial. By now you should have a thorough understanding of how RiveScript code is written and you should be able to start creating replies for your own chatterbot.

For more information about RiveScript, see <http://www.rivescript.com/>

VERSION AND AUTHOR

This tutorial was last updated on June 8, 2012.

This tutorial was written by Noah Petherbridge, <http://www.kirsle.net/>

APPENDIX

Windows Troubleshooting

Command Prompt Help

If you're not familiar with DOS commands and how to change directories in a command prompt window, you should look up a quick tutorial on how to use the command line. One such tutorial can be found at <http://www.computerhope.com/issues/chusedos.htm>

For the lazy, you can create a simple batch file and place it in the same folder as the RiveScript interpreter that

would open a command prompt window there, without needing to `cd` to that folder yourself.

Open Notepad and type the following code into a new file:

Save the file as "`cmd.bat`" (*with* the quotation marks!) in the same folder as the `rivescript` file from the Perl RiveScript distribution. Now, navigate to that folder in Windows Explorer and double-click on the `cmd.bat` file (the file might appear as simply "`cmd`" if file extensions are hidden).

Writing an Interpreter

Here is an example of how to write your own RiveScript interpreter application in Perl 5.

```
#!/usr/bin/perl

use strict;
use warnings;
use RiveScript;

# Create a new RiveScript interpreter
object.
my $rs = RiveScript->new();

# Load a directory full of RiveScript
documents.
$rs->loadDirectory("./replies");
```



```
# You must sort the replies before trying
to fetch any!
$rs->sortReplies();

# Enter a loop to let the user chat with
the bot using standard I/O.
while (1) {
    print "You> ";
    chomp(my $message = <STDIN>);

    # Let the user type "/quit" to quit.
    if ($message eq "/quit") {
        exit(0);
    }

    # Fetch a reply from the bot.
    my $reply = $rs->reply("user", $message);
    print "Bot> $reply\n";
}
```

SEE ALSO

[RiveScript](#)

[RiveScript::WD](#) - The RiveScript Working Draft -

<http://www.rivescript.com/wd/RiveScript.html>

Web design and content copyright © 2021 Noah
Petherbridge.