Rebol scripting basics

Presented by developerWorks, your source for great tutorials

ibm.com/developerWorks

Table of Contents

If you're viewing this document online, you can click any of the topics below to link directly to that section.

Section 1. Tutorial tips

Should I take this tutorial?

This tutorial will introduce you to a powerful Internet-enabled scripting language called Rebol. You should take this tutorial if you'd like to add Rebol to your programming arsenal and have no prior experience with it, or if you've tried to learn Rebol in the past but found it confusing.

This tutorial provides clear demonstrations of Rebol fundamentals, including detailed explanations of the parts of Rebol that differ from more conventional programming languages. It's designed to make learning Rebol really easy.

Once you've mastered Rebol basics, you'll be directed to appropriate online resources (like the *Rebol User's Guide*) where you can continue your study of this revolutionary language.

Navigation

Navigating through the tutorial is easy:

* Use the Next and Previous buttons to move forward and backward through the tutorial.
* When you're finished with a section, select Next section for the next section. Within a section, use the Section menu button to see the contents of that section. You can return to the main menu at any time by clicking the Main menu button.
* If you'd like to tell us what you think, or if you have a question for the author about the content of the tutorial, use the Feedback button.

Getting help

For technical questions about the content of this tutorial, contact the author, Daniel Robbins, at *drobbins@gentoo.org* .

Residing in Albuquerque, New Mexico, Daniel Robbins is the President/CEO of *Gentoo Technologies, Inc.* , the creator of **Gentoo Linux**, an advanced Linux for the PC, and the **Portage** system, a next-generation ports system for Linux. He has also served as a contributing author for the Macmillan books *Caldera OpenLinux Unleashed*, *SuSE Linux Unleashed* and *Samba Unleashed*. Daniel has been involved with computers in some fashion since the second grade, when he was first exposed to the Logo programming language as well as a potentially dangerous dose of Pac Man. This probably explains why he has since served as a Lead Graphic Artist at **SONY Electronic Publishing/Psygnosis**. Daniel enjoys spending time with his wife, Mary, and his new baby daughter, Hadassah.

Section 2. Introducing Rebol

A description of Rebol

How does one describe the Rebol language? Revolutionary. Powerful. Flexible -- and different. In fact, Rebol is probably unlike any other language that you've used before. Designed to tackle real-world programming tasks in a straightforward yet "unconventional" way, Rebol offers a refreshing new programming toolkit for those who are looking for something more in a scripting language.

For us developers, learning Rebol can be both refreshing and intimidating. While promising something completely different, Rebol also uses new programming paradigms that, while somewhat similar to those found in more traditional languages, are "different enough" to breed confusion for the seasoned developer.

Rebol features

A lot of things make Rebol different. It uses a natural English-like syntax, and supports tons of datatypes, including special datatypes designed to store Internet-related data, like URLs and email addresses. Rebol is an Internet-enabled language; using Rebol, you can download a Web page or send an email message with a single line of code. Not only that, but Rebol provides advanced functionality called "dialecting", which allows you to create your own special-purpose sub-languages that are specifically tailored to the tasks *you* use Rebol for.

However, before you can access these killer features, it's important to master the basics of Rebol. That's what we're going to do in this tutorial. After you're well versed in the basics, I'll show you some online resources that you can use to learn Rebol's more advanced features.

Installation

Using Rebol begins with a visit to *http://www.rebol.com* . In this tutorial, we'll be using the basic version of Rebol, called Rebol/Core, available for about 40 different platforms. The Rebol-related parts of this tutorial will apply to every platform, although we will only be covering Linux installation.

Download Rebol/Core

After you arrive at *http://www.rebol.com* , head over to the developer section, and then click on "Downloads". Select the most recent version of Rebol/Core that's available for your platform. Currently, there are versions available for libc5 and libc6 x86 Linux, as well as Linux/Alpha, Linux/PPC, Linux/Sparc and UltraSparc, and even Linux/StrongARM, MIPS, and 68K! Obviously, the Rebol developers take their porting effort seriously.

Downloading Rebol/Core, continued

For libc6 x86 Linux, the Rebol/Core archive will be called "core042.tar.gz". After downloading the appropriate archive, make a directory called "rebol" in your home directory, and place the archive inside this directory. After entering the directory, extract the contents of the archive by typing:

```
$ tar xzvf core042.tar.gz
```

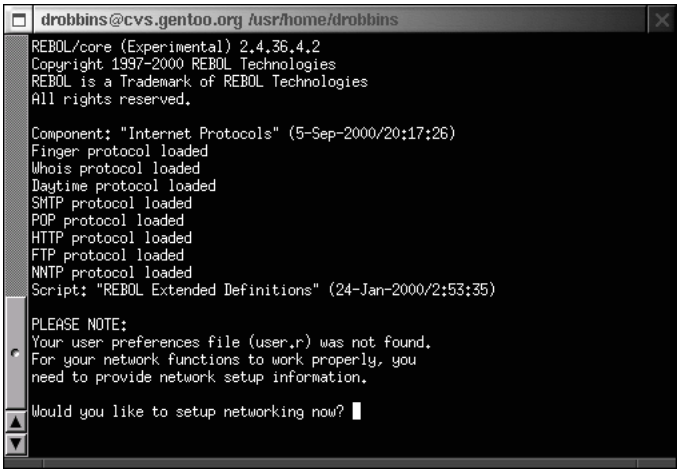
About five files, including the rebol executable, will be extracted to your current directory.

Starting Rebol

To start Rebol, type:

```
$ ./rebol
```

You'll see something that looks like this:



Configuring Rebol

At this point, you will be prompted to configure networking options for your system, which will include entering your email address, an SMTP and POP server, and network proxy settings. Since Rebol is an Internet-enabled language, it needs these settings to access the Internet.

Configuring Rebol, continued

After you've completed the configuration steps, Rebol will ask for permission to create a "user.r" file in the current directory. Your settings will be stored in this file. Go ahead and say "Y". This is an example of Rebol's built-in security support, protecting you from any unauthorized modifications to your filesystem.

The next time you start Rebol, remember to start it from inside your "rebol" directory. That way, Rebol will find the user.r file and your network settings will be loaded automatically.

The first prompt

After configuration finishes, you'll be inside the interactive Rebol interpreter. The Rebol prompt looks like this:

```
>>
```

Go ahead and type "help", and hit Enter to view Rebol's built-in help page. The "help" command can also accept an argument. Try typing "help print". You'll see a helpful summary of the print function appear on your screen.

Rebol -- what?

Exactly what Rebol functions are available? To see a complete list, type "what" at the prompt:

```
>> what
```

The "what" command displays a list of all globally defined functions.

Quitting Rebol

That brings us to one of the most essential Rebol commands -- quit. If you ever need to leave the interpreter, simply type:

```
>> quit
```

....and you'll be dropped back at the shell prompt. Now that you know how to get in and out of the interpreter, it's time to get familiar with the language itself.

Section 3. Rebol -- the language

Introducing datatypes

One of the wonderful things about Rebol is that it has a huge number of built-in datatypes, designed to represent values that you find in the "real world", like email addresses, URLs, dates, times, and monetary values. Let's start getting familiar with some of them.

Introducing datatypes, continued

At the Rebol prompt, go ahead and type:

```
>> drobbins@gentoo.org
== drobbins@gentoo.org
```

After typing this email address into the interpreter, Rebol just repeated it back to us. This is standard fare for Rebol -- when a literal value is typed into the interpreter, it's repeated back. Now, you may wonder what *kind* of value drobbins@gentoo.org is. Most languages would store an email address as a string, so you might guess that Rebol does too. However, you'll be surprised to find that Rebol does something much more useful.

Exploring datatypes

To see what kind of datatype drobbins@gentoo.org is, type:

```
>> type? drobbins@gentoo.org
== email!
```

The type? word is a function that accepts one argument, returning the datatype of the argument. As you can see, Rebol actually recognizes drobbins@gentoo.org as an email address -- quite impressive! We didn't need to use special syntax to specify an email address; Rebol recognized this value as an email address because it noticed the embedded "@" character.

Exploring datatypes, continued

Let's continue. Type this in:

```
>> type? $5.00
== money!
```

Again, Rebol understands that $5.00 represents a monetary value.

You may be wondering why there are exclamation points at the end of email! and money!. The answer's simple -- in Rebol, all datatype names end with an exclamation point. In addition, many functions that query attributes (like the type? function) end with a question mark. Rebol tries to be more user-friendly by structuring its syntax similarly to a traditional written language, like English.

Many datatypes

Try typing in the following Rebol expressions and see what datatype Rebol recognizes each value as:

```
>> type? 1
>> type? 1.0
>> type? "foo"
>> type? {foo}
>> type? http://www.gentoo.org
>> type? true
>> type? yes
>> type? :print
>> type? money!
```

Section 4. Rebol words

Introducing words

Just as languages like C have variables that hold data, Rebol has its own kind of variable, called a word. Here's an example of how to associate a word with a value:

```
>> myval: $5.00
```

Introducing words, continued

In this example, myval has been associated with the value $5.00. Using myval in an expression will cause it to be immediately evaluated:

```
>> myval
== $5.00
>> myval + 1
== $6.00
```

In Rebol, it's very common to associate words with values, creating a word/value pair. Words are also case *insensitive*, so typing myval is the same as typing MyVal.

Words with no value

Now, try typing any old word into the interpreter. You'll likely receive an error:

```
>> foo
** Script Error: foo has no value.
** Where: foo
```

By default, Rebol will automatically evaluate all words. By "evaluate", I mean that Rebol will automatically replace the word with its associated value. If a word isn't associated with a value, then Rebol returns an error.

Defining new words

We can also define new words based on the values of existing words. Try this example:

```
>> myval: $5.00
== $5.00
>> myval2: myval + 1
== $6.00
>> myval2
== $6.00
>> myval
== $5.00
>> myval: myval + 1
== $6.00
>> myval2
== $6.00
```

Here, we defined myval2 based on myval. Then, we incremented myval, and printed the contents of myval2, finding that myval2's value didn't change.

A description of Rebol

As you can see, Rebol words appear to act identically to variables in other languages like C and C++. You may even be wondering why the Rebol guys decided to call their versions of variables "words" rather than sticking with the standard terminology. Good question! As we continue with the tutorial, I'll show you some of the unique qualities of Rebol words that set them apart from standard C variables.

Word basics -- summary

When the Rebol interpreter encounters a word, its normal response is to *evaluate* it. When a word is evaluated, it is replaced with its associated value. So, for example, if the word **myemail** were associated with the value **drobbins@gentoo.org**, typing this at the Rebol interpreter prompt:

```
>> myemail
```

....would cause the interpreter to return:

```
== drobbins@gentoo.org
```

This is myemail's value. This shows us that according to Rebol, the word myemail has a meaning, and represents the drobbins@gentoo.org email address. While this may seem obvious, Rebol also allows you to do some very unusual things with words, as we'll see in a bit.

Math operators

When we were playing with dollar amounts a few panels back, we used the standard math operator +. Of course, Rebol sports a whole bunch of math operators, including the standard +, -, *, and /. In addition, it also features a // remainder operator.

When you're writing mathematical expressions in Rebol, there are two important things that you need to know. First, you need to put spaces between values and operators; 1+7 can't be parsed correctly by the interpreter, but 1 + 7 can. Second, Rebol evaluates mathematical expressions from left to right, period. There's no special operator precedence like in C. If you want to be explicit about the order of mathematical evaluation, you can surround parts of your expression with parentheses.

Prefix operators

In addition to the standard "infix" (in between) math operators, Rebol also has built-in "prefix" math operators. Here's how they work:

```
>> abs -23
== 23
>> add 23509 230
== 23739
>> divide 4 2
== 2
>> multiply 30 30
== 900
>> remainder 9 4
== 1
>> subtract 4 9
== -5
```

Dates

Rebol also has excellent built-in support for dates. To get the current date, use the word "now":

```
>> print now
30-Sep-2000/22:41:19-6:00
```

By adding or subtracting an integer from a date, you can offset the date by a certain amount of days. For example, you can express two weeks from now as:

```
>> print now + 14
14-Oct-2000/22:43:13-6:00
```

Many, many datatypes

Rebol has many more datatypes; we simply don't have room to cover all of them here. Here's a partial list: integer, decimal, time, date, money, logic, char, none, string, binary, email, file, url, issue, tuple, tag, block, hash, list, paren, path, and word.

Many datatypes have their own special operators and functions, and datatypes can interact in a myriad of ways. For example, you can extract a time from a date; append to strings; and open, close, and write to files. All these datatypes and their associated operators and functions are described in detail in the *Rebol User's Guide* (see Resources on page 24for more information).

Section 5. Words and expressions

A description of Rebol

Previously, when we evaluated myemail, the Rebol interpreter wrote "== drobbins@gentoo.org" to the screen. When you use the interpreter interactively, Rebol prints the values of any evaluated expressions. However, when Rebol code is executed from a file (not using the interpreter, but as a shell script), we won't see this output. In these situations, you should use the **print** command to generate output. We can also use print interactively:

```
>> print myemail
drobbins@gentoo.org
```

Rebol displayed the value of myemail, but did not print "== drobbins@gentoo.org". This is because print outputs the value of email to the screen, but does not return any value itself.

Probing

Sometimes you'll want to print the value of a word to the screen, but you'll also want to perform additional processing on the word's value. For example, you may type:

```
>> print myemail
drobbins@gentoo.org
>> type? myemail
== email!
```

This does the trick, but Rebol provides a much cleaner way to perform these two steps at once...

Probing, continued

Simply type:

```
>> type? probe myemail
drobbins@gentoo.org
== email!
```

The probe word expects a single word or block (covered later) as an argument. It will print the value of this word, and *also return the value* of the word. In this example, this value can then be evaluated by the type? word, displaying the type of the drobbins@gentoo.org value. You can think of probe as Rebol's version of the standard UNIX "tee" command.

Literal words

Most of the time, when we refer to a word in Rebol, Rebol does what we want -- it evaluates the word, passing its value to a function or displaying the value in the interactive interpreter. However, sometimes this isn't exactly what we want to do. There are times when we may want to refer to the word itself, rather than its associated value. For these instances, Rebol provides a handy literal word syntax. Take a look at this example:

```
>> type? myemail
== email!
>> type? 'myemail
== word!
```

Speakin' it!

To help you grasp this concept of literal words, you're going to actually speak out loud (with your mouth, in English or your favorite language!) what the code is doing. Yes, this means that you will be talking to yourself while staring at the monitor. Don't fear -- if you're embarrassed by this kind of thing, just pretend that you're Jean Luc Picard (Captain of the USS Enterprise) and that you're barking orders to the ship computer.

Speakin' it -- the key

Here's the key to this exercise. When you see a word written like this:

```
foo
```

....you're supposed to say "foo".

When you see a word written like this:

```
'foo
```

....you're supposed to say "the word foo".

First order

Here's a command that you've already typed into the Rebol interpreter:

```
>> myemail: drobbins@gentoo.org
```

If you were Jean Luc Picard, you wouldn't use something as horribly antiquated as a computer keyboard to enter this command. Instead, you'd say this out loud (hint):

"Computer, associate the word myemail with drobbins@gentoo.org."

Second order

And, being Jean Luc Picard, instead of typing this monstrosity into the keyboard, risking a bad case of carpal tunnel syndrome:

```
>> myword: 'myemail
```

....you'd get up from your orthopedically-designed Captain's chair and authoritatively project:

"Associate the word myword with myemail"

Notice the difference? It's subtle but very important. The ' prefix means you're talking about **the word**. Without the ', you're talking about the word's **value**.

Evaluating words by hand

Now, you have a word called myword that has a value which is itself also a word. OK, pretty cool, but how can you tell Rebol to evaluate your word? Easy:

```
>> do myword
== drobbins@gentoo.org
```

Sha-blamm! Now *we* can control exactly when Rebol will evaluate a word.

Much ado about nothing

Thanks to the do function, we can also type:

```
>> do 'myemail
== drobbins@gentoo.org
```

However, this is a bit pointless since typing the word by itself will accomplish the same thing:

```
>> myemail
== drobbins@gentoo.org
```

Word associations

Here's another way to associate a word with a value. Instead of typing:

```
>> myvar: $5.00
```

You can also type:

```
>> set 'myvar $5.00
```

Both expressions do the same thing. Read them both out loud as "set **the word** myvar to the value $5.00". The **set** word opens up a lot of powerful possibilities.

More word associations

Consider:

```
>> myword: 'cost
== cost
>> set myword $5.00
== $5.00
```

The last expression doesn't actually affect the value of the word myword; instead, it associates cost with the value $5.00. (Type "print cost" if you don't believe me.) You can read the last line out loud as "set the value of myword to $5.00" or "set the word cost to $5.00". Since the value of myword *is* the word cost, both of these sentences mean the same thing.

Unset

Words can be set, and they can also be unset:

```
>> unset 'cost
```

This will remove any value associated with the word cost ('cost). If you type "print cost" immediately after this command, you'll get an error, because cost has no value. Try typing "print cost" just for fun.

Unset, continued

Even though the word cost no longer has any value, the following code will not raise an error:

```
>> print 'cost
cost
```

This is because we executed the command "print the word cost". The print function goes ahead and prints the name of the word, rather than its value. The ' character prevents cost from being evaluated.

Helping Rebol out

While Rebol is very good at automatically determining the type of a literal value, there are times where Rebol simply doesn't have enough information to automatically figure out what type a particular value should be. For example, imagine a situation where I want to set the word myemail so that it's associated with the email address "drobbins". You may be tempted to type:

```
>> myemail: drobbins
```

Helping Rebol out, continued

Unfortunately, this doesn't work. There's no way for the interpreter to know what "drobbins" is. Rebol doesn't see an "@" embedded in the string, so it assumes that drobbins is a regular word, and tries to evaluate it. Unfortunately, the word drobbins doesn't have an associated value, so we get this error:

```
** Script Error: drobbins has no value.
** Where: myemail: drobbins
```

How to help

Fortunately for us, there's a way around this. Instead of typing:

```
>> myemail: drobbins
```

....we can type:

```
>> myemail: make email! "drobbins"
```

We don't get an error, and if we query the type of myemail, we see that we got the desired result:

```
>> type? myemail
== email!
```

The make function allows us to create a value of a specific type. We specify the type, rather than relying on Rebol to try to detect the type of the data for us.

Make it work

The first argument to the make function should be a datatype, like money!, email! or url!. The second argument should be a string containing the value that you would like to create. As long as you specify a valid string, make will create exactly the value that you want. If you specify an invalid string, you'll get an error. You can see that Rebol isn't able to satisfy this request, and with good reason:

```
>> mymoney: make money! "drobbins"
** Script Error: Invalid argument: drobbins.
** Where: mymoney: make money! "drobbins"
```

Section 6. Blocks

Building blocks

Now we're going to look at a Rebol construct called a block. In Rebol, you can use blocks to organize collections of data. To create a block, surround a bunch of values with a [and a]:

```
>> myblock: [ this and that ]
== [this and that]
```

One of the neat things about blocks is that they can contain data or code. They can contain words that don't have any associated value, and the contents of the block are *not* immediately evaluated. For example, Rebol accepts the definition of myblock even though the words this and that have no value.

Code blocks

You can insert code into a block, as follows:

```
>> mycode: [ print "foo" ]
== [print "foo"]
```

To actually execute the code, use the handy do function:

```
>> do mycode
foo
```

Block features

Here's a block of code that doesn't actually do anything when executed, but it does demonstrate a Rebol feature:

```
>> mydata: [ 1 2 3 ]
== [1 2 3]
```

This looks like a data block, but we can also execute it as code. As code, this block contains three separate expressions, 1, 2, and 3. Each expression returns a value of 1, 2, and 3 respectively. Now, let's execute the block.

Block features, continued

```
>> do mydata
== 3
```

As you can see, Rebol will return the last value encountered in a code block. However, the entire code block *is* executed in order, not just the last element.

Nesting blocks

Blocks can be nested:

```
>> instructions: [
[    [ remove hard drive from chassis ]
[    [ throw hard drive out window ]
[    [ scream "foo foo foo!" ]
[    ]
```

The [characters that begin the second through fifth lines are actually the Rebol prompt. It's Rebol's way of reminding us that we are entering a multi-line block. Try typing this block into the interpreter and you'll see what I mean.

Accessing blocks

We can access the elements of a block:

```
>> myblock: [ jimmy cracked corn ]
== [jimmy cracked corn]
>> first myblock
== jimmy
>> myblock/1
== jimmy
>> last myblock
== corn
>> myblock/3
== corn
>> myblock/2
== cracked
```

Block paths

We can also "pair up" words (used as keys) with values, and use the words as indexes into the block:

```
>> order: [
[    amount 10
[    cost $3.50
[    email drobbins@gentoo.org
[    ]
== [
    amount 10
    cost $3.50
    email drobbins@gentoo.org
]
>> order/email
== drobbins@gentoo.org
>> order/cost
== $3.50
```

Notice that the order block contains six separate words, and that the word amount is not associated with any value (we didn't type "amount: 10", we typed "amount 10", which are two separate unrelated words). "order/cost" is called a *path*.

Section 7. Resources

Getting the User's Guide

We've covered a lot in this tutorial, but we've just scratched the surface of the Rebol programming language. To dig deeper, I strongly recommend that you download the *Rebol User's Guide*. It's well written, quite thorough, and now that you know the basics of Rebol, quite understandable. To have Rebol automatically download the *User's Guide*, type:

```
>> do do http://www.rebol.com/users-guide.r
```

Rebol will ask you if it can create files on disk. Just hit A for "All", and the Rebol User's Guide will be automatically downloaded and created in your current working directory! Pretty neat, eh?

Additional resources

For additional developer resources, make sure you check the developer section of *http://www.rebol.com* on a regular basis. There's a good (very high-volume) mailing list available, as well as several free Rebol introductory and intermediate articles of various kinds.

I hope that you've enjoyed this tutorial, and that you'll continue to grow in your Rebol knowledge!

Section 8. Feedback

Feedback

Please send us your feedback on this tutorial. Please be advised that we cannot be held responsible for any malpractice claims filed against you, nor can we offer you legal assistance or political asylum to protect you from the surviving relatives of your former patients. With those caveats in mind, we look forward to hearing from you!

Colophon

This tutorial was written entirely in XML, using the developerWorks Toot-O-Matic tutorial generator. The Toot-O-Matic tool is a short Java program that uses XSLT stylesheets to convert the XML source into a number of HTML pages, a zip file, JPEG heading graphics, and two PDF files. Our ability to generate multiple text and binary formats from a single source file illustrates the power and flexibility of XML.