UNIX tips and tricks for a new user, Part 4: Some nifty shell tricks

Learn the basics of scripting with these powerful techniques

Skill Level: Intermediate

Michael Stutz (stutz@dsl.org)
Author
Consultant

20 Feb 2007

When writing a shell program, you often come across some special situation that you'd like to handle automatically. This tutorial includes examples of such situations from small Bourne shell scripts. These situations include base conversion from one string to another (decimal to hex, hex to decimal, decimal to octal, and so on), reading the keyboard while in a piped loop, subshell execution, inline input, executing a command once for each file in a directory, and multiple ways to construct a continuous loop. Part 4 of this series wraps up with a collection of shell one-liners that perform useful functions.

Section 1. Before you start

Learn what to expect from this tutorial and how to get the most out of it.

About this series

This series of tutorials provides a concise introduction to basic UNIX® concepts, written for the perspective of a new user. The three previous tutorials in the series provided a brush-up on UNIX systems geared toward new users coming from a Microsoft® Windows® background, describing the file system and common commands, an introduction to `vi` (the most ubiquitous of UNIX editors), and a quick primer on filters and regular expressions using the `grep`, `sed`, and `awk` tools.

About this tutorial

This tutorial provides a collection of shell tricks and tips that are handy for new users. It shows how to automate special situations using small scripts written in the Bourne shell, including automatic base conversion, reading keyboard input, executing commands in a subshell, executing commands on all the files in a directory, and various forms of looping. The tutorial concludes with a collection of useful shell one-liners.

Objectives

The objective of this tutorial is to show new users how to use and implement many of the shell's methods for providing automation at various levels. It demonstrates these methods by giving tricks and tips for special situations, and it also presents a rundown of useful shell one-liners for common tasks.

Prerequisites

This tutorial is written for users who are relatively new to UNIX. The only prerequisites are basic knowledge of the UNIX file system and the commands to manipulate it, the command line itself, and editing text files with an editor, such as `vi`. All of these concepts are fully described in the previous tutorials of this series.

System requirements

You need user-level access to a UNIX system with a Bourne-compatible shell environment, such as the popular `bash` shell. This is the only system requirement for this tutorial.

Section 2. Shell command execution

The best way to learn shell scripting is by example. Any command that you would execute in a script you can try right at the command line, and that's how many of the hands-on examples are given throughout this tutorial. For example, the `echo` command writes a line of text to the standard output. (Many shells offer their own version of `echo` as a built-in command, including the IBM AIX® implementation of the Bourne shell. If that's your case, then when you run `echo`, it's actually your shell's version of the command that's being run.)

Quotation

Try outputting a short message with `echo` by enclosing it in quotes:

```
$ echo "Hello, world"
Hello, world
```

Shell *quotation*, either right at the command line or in scripts, is a means of passing a string to the shell so that there is no ambiguity about any of the special metacharacters that the string might contain. You use quotation when strings contain more than a single word and to enclose a phrase containing space characters. You also quote an individual character when it happens to be a shell metacharacter and you want to take away its special meaning, such as when you want to pass a dollar sign ($) as a literal dollar sign character and not the special metacharacter that precedes a variable name.

Various *expansion* occurs inside quoted text. Inside of double-quoted text, for instance, variables are expanded to their values, while literal variable names are not expanded when referenced inside of single-quoted text.

There are three important types of quotation to know:

1. Quote a single character by preceding it with a backslash (\). This passes the literal character and not any special meaning it might have, such as a space character or shell metacharacter. For instance, to quote an asterisk (*), which is a shell metacharacter, use `\*`. To quote an actual backslash character, use `\\`.

2. Pass an *expanded quotation* by enclosing a text string in double quotation marks (`"`). The dollar sign ($) and single quotation mark (`'`) characters keep their meanings. So, among other things, any variable names referenced in the quotation are replaced by their values. Backslashes that precede a new line or certain characters (`$` `` ` `` `"\`) are removed, but the characters they quote are passed.

3. Pass a *literal quotation* of a text string -- passing all variable names, metacharacters, and so on as the characters themselves and not their meanings or values -- by enclosing the text in single quotation marks (`'`).

Note that the exact rules of quotation differ across shells. Consult the `man` pages of your particular shell to see its precise rules.

Try assigning a variable and then outputting it with various quotation styles, as shown in Listing 1.

Listing 1. Demonstrating shell variable quotation styles with echo

```
$ myvar = "Hello, world"
```

```
$ echo $myvar
Hello, world
$ echo "$myvar"
Hello, world
$ echo '$myvar'
$myvar
$ echo \$myvar
$myvar
$ echo \'$myvar\'
'Hello, world'
$ echo "'$myvar'"
'Hello, world'
$ echo '"$myvar"'
"$myvar"
$ echo \"$myvar\"
"Hello, world"
```

Notice how the variable is interpreted in different ways depending on the quoting style used.

Comments

In the shell, the hash mark (#) begins a comment line. The hash and everything that follows it on the line is ignored. Try typing lines interspersed with comments, as shown in Listing 2:

Listing 2. Using comments in the shell

```
$ # a comment does nothing
$ echo "Hello, world" # This text is ignored
Hello, world
$ echo # This will not output

$ echo 'But a hash (#) can be quoted'
But a hash (#) can be quoted
$ echo "# Even in double quotes"
# Even in double quotes
$
```

Make a shell script

As you can see, you can work right at the command line to test these shell programming constructs. But, when you graduate beyond one-line commands and actually start composing longer programs, you need to write them to files called scripts. A *script* is a text file that has its executable bit set and contains a program consisting of commands in the shell language. The UNIX shell is an *interpreted language*, meaning that its programs are not compiled but are read by an interpreter, which is the shell executable itself, such as `/bin/sh`, `/bin/bsh`, or `/bin/bash`.

The first line of a shell script is always the same:

```
#!/bin/sh
```

This is a special comment line used by the shell itself to determine the language or

contents of the file. The exclamation point, often called *bang* in the UNIX and typesetting idioms, followed by the path name indicates the interpreter the shell should use to execute the file. In this case, it's `/bin/sh`, which on many systems is the Bourne shell executable itself. A script written specifically for the Korn shell, for example, might begin with the line `#!/usr/bin/ksh` instead, just as a Ruby script would begin with `#!/usr/bin/ruby`. When `bash` is installed, `/bin/sh` is usually a symbolic link to the `bash` binary. And, for compatibility, using `/bin/sh` is preferable to using `/bin/bash`. On some systems, such as IBM AIX 5L™, the name for the Bourne shell executable is `bsh`, and it's located at /usr/bin/bsh.

Listing 3 gives a short example shell script.

Listing 3. Sample shell script

```
#!/bin/sh
# This is a shell script
message = "Hello, world!"
echo "The message is '"$message"'"
```

Use `vi` to type it in and save it to a file named myscript, as described in a previous tutorial in this series (see Resources). Then, use `chmod` to make it executable by setting execute permission on the file:

```
$ chmod u+x myscript
```

This command makes it executable only by you. If you want to let all users on the system run it, you can always set the execute permission for all:

```
$ chmod a+x myscript
```

Now you can run it. Give the file name as it exists in relation to the current working directory, which is specified in the path as a dot character (`.`):

```
$ ./myscript
The message is 'Hello, world!'
$
```

The shell variable `PATH` contains a list of directories delimited by colons. This is said to be your *path*, and the shell always "sees" any files in those directories. The purpose of the UNIX path is to make it convenient to run binary files. That's why you just type the base file names of commands, such as `ls` and `echo`, instead of giving their full or relative path names. If you move the script to a directory on your path, you can just type its name to run it. The exact path depends on your UNIX implementation and local setup, but normally the directories on the path include /bin, /usr/bin, and /usr/local/bin.

Some users configure their shell so that the `PATH` variable includes the current

working directory, which is specified in the path as a dot character, ("."). That way, to run a script in the current directory, you can just type its name without specifying the relative directory. The shell searches along the path in the order given; so to protect against trojans or accidental mishaps, it's never wise to put the current working directory anywhere but at the end of the path.

To see your path, use `echo` to show the contents of the `PATH` variable, as shown in Listing 4.

Listing 4. Changing the PATH

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11
$ myscript
myscript: command not found
$ PATH = $PATH":."
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/bin/X11:.
$ myscript
The message is 'Hello, world!'
$
```

Special options or flags can follow the name of the interpreter, such as `/usr/bin/bsh -n`, which is used for debugging purposes. A hyphen turns the option off, while a plus turns the option on. The special built-in environment variable, `-` (a hyphen), contains the full list of options for the current shell.

Try showing which options are set in your current interactive shell. Do this by using `echo` to display the contents of the `-` variable:

```
$ echo $-
himBH
$
```

Consult the `man` page of your shell to get a current list of flags and options. Table 1 provides a list of common flags for the Bourne shell on AIX®, along with a short description of what each one does.

Table 1. Common options for the AIX Bourne shell

Flag	Description
`-a`	Export all variables that are assigned values.
`-c` *Variable*	Execute the commands read from *Variable*.
`-e`	Exit immediately if a command meets one of the following criteria: the command exits with a return value greater than 0; the command isn't part of a `while`, `until`, or `if` construction; the command isn't being tested by `AND` or `OR`; or the command is not a pipeline preceded by bang.
`-f`	Disable all filename substitution.
`-h`	Locate and remember all commands called within the functions as the functions are defined.

`-i`	Specify an *interactive* shell.
`-k`	Place all *keywords* in the environment for the command.
`-n`	Read commands, but don't execute them.
`-r`	Call a *restricted* shell.
`-s`	Read commands from standard input, and write output to standard error (excluding the output of shell built-ins).
`-t`	Read and execute one single command, and then exit.
`-u`	In scripts, treat all *unset* variables as errors. Exit upon attempting variable substitution.
`-v`	Show the input line as it's being read.
`-x`	Show the full commands (all arguments and options) before executing them.

Section 3. Shell arithmetic and base conversion

The shell provides for a number of basic arithmetic operations, which are useful in scripts. The shell evaluates arithmetic expressions you give it, performing *arithmetic expansion*, where it replaces the expression with its result. Give arithmetic expressions in this form:

```
$(( expression )
```

You can see arithmetic expansion at work by using `echo` to display the results at the command line. Try what's shown in Listing 5 now.

Listing 5. Arithmetic expansion in the Bourne shell

```
$ echo $((10+40))
50
$ echo $((5*(3+3)))
30
```

You can also assign expansions to variables. Try what's shown in Listing 6.

Listing 6. Assigning arithmetic expansions to shell variables

```
$ myvar = 10
$ echo $myvar
10
$ echo $(($myvar-2))
```

```
8
$ myvar = $(($myvar+5))
$ echo $myvar
15
$ result = $(($myvar-10))
$ echo $result
5
$
```

Table 2 lists some of the valid operators that might be used between expressions in most Bourne and Bourne-compatible shells. As in the second example above, statements grouped in their own set of parentheses take precedence. In fact, shell arithmetic precedence is generally determined according to the rules of the C language.

Table 2. Shell conditionals

Operator	Description
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Remainder
<	Less than (1 if true, 0 if false)
<=	Less than or equal to (1 if true, 0 if false)
>	Greater than (1 if true, 0 if false)
>=	Greater than or equal to (1 if true, 0 if false)
<<	Left bitwise shift: shifts the given integer of the first expression to the left by the number of bits of the second expression
>>	Right bitwise shift: shifts the given integer of the first expression to the right by the number of bits of the second expression

Base conversion with shell arithmetic

Say you have some number but, in your script, you need to work on it in another base. Automating this conversion is done easily with shell arithmetic. One way is to use shell arithmetic to convert a number from a given base to decimal. If a number is given in an arithmetic expansion, it's assumed to be in decimal notation *unless* it's prefaced by a either a zero -- in which case it's assumed to be in octal -- or 0x -- in which case it's assumed to be in hexadecimal. Type the following to get decimal output for some octal and hex values:

```
$ echo $((013))
$ echo $((0xA4))
```

You can also specify any arbitrary base between 2 and 64 with the following format:

```
$((BASE#NUMBER))
```

Try converting numbers in binary, octal, hex, and other bases to decimal by typing the lines shown in Listing 7 at the shell prompt.

Listing 7. Outputting numbers in arbitrary bases in the shell

```
echo $((2#1101010))
echo $((8#377))
echo $((16#D8))
echo $((12#10))
echo $((36#ZZYY))
```

Base conversion using bc

Another trick for doing base conversion in the shell is to use `bc`, the arbitrary precision calculator language, which is available on most UNIX installations. Because it lets you specify an output base, this is a good technique for when you need output in something other than decimal.

The special `bc` variables, `ibase` and `obase`, contain the value of the base used for input and output, respectively. By default, both are set to 10. To perform base conversion, convert one or both of them, and then give it a number. Try it now, as shown in Listing 8.

Listing 8. Performing base conversion with bc

```
$ bc -ql
                    10
10
obase=16
                    10
A
ibase=2
                    10
2

                        Control-D

$
```

To do a quick base conversion, use `bc` in conjunction with `echo` to make a quick one-liner, piping the given values to `bc`. Type what's shown in Listing 9.

Listing 9. Shell one-liners for bc

```
$ echo 'obase=16; 47' | bc
2F
$ echo 'obase=10; ibase=16; A03' | bc
2563
$
```

A note of caution: After you set the input base in `bc`, all numbers you input to `bc` after that are taken in that base, including any number you give to set the output base. So it's better to set the output base first or you might get unexpected results, as shown in Listing 10.

Listing 10. Precedence matters when setting the input and output base

```
$ echo 'ibase=16; obase=10; A' | bc
A
$ echo 'ibase=16; obase=A; A' | bc
10
$
```

Section 4. Inline input

While the `echo` trick to pipe to an interactive command, such as `bc`, makes for a quick one-liner at the command line, it isn't practical for multiple-line input, such as when the contents of an actual file might be used. But there's another useful way to do this. The shell has a facility called *here documents*, or inline input, which is a great way to construct a file on the fly, such as inside of a script, and redirect the contents of this file to a command.

Specify a here document using the shell << operator and follow it on the same line with a *limit string*, which is the string that marks the termination of input and can be any text you choose, so long as it's a single word without space characters. Follow this with the lines of input that constitute your input file, and terminate the input with the limit string on a line of its own -- it can't have any text before or after it, or that line is considered to be part of the input. Try it with `cat`, as shown in Listing 11.

Listing 11. Making a here document

```
$ cat << END
>  END of input text
> ENDspace

> This is still not the END
> ENDING SOON
> THE END
> END
 END of input text
END
This is still not the END
ENDING SOON
THE END
$
```

The limit string, in this case `END`, can appear anywhere in the input -- only when it appears on a line of its own with no spaces or other characters does it function as

the termination of input.

Inline input in scripts

Inline input is often used in scripts to output usage information to standard output. This is normally done by sending a here document to `cat`, as in the script in Listing 12. Use `vi` to type it in and save it to a file named baseconv, and make the file executable (see the Make a shell script section).

Listing 12. Using a here document to give shell script usage information

```
#!/bin/sh
cat << EOF
baseconv is a program to convert a number from one base to another.

Usage: baseconv [options]

Options:

-i BASE input base
-o BASE output base
-h display this message

For more information, consult the baseconv man page.
EOF
```

When the script is executed, the contents of the here document is sent (using `cat`) to the standard output. Try it now, as shown in Listing 13.

Listing 13. Output of shell script usage information from a here document

```
$ baseconv
baseconv is a program to convert a number from one base to another.

Usage: baseconv [options]

Options:

-i BASE input base
-o BASE output base
-h display this message

For more information, consult the baseconv man page.
$
```

Additionally, most implementations of the Bourne shell recognize inline input redirected with the optional hyphen character. The optional hyphen character removes all leading tab characters from the beginning of all input lines, as well as the line containing the limit string itself. This is helpful when writing scripts where you want to keep the current indentation. Because inline input is normally taken literally and the limit string has to be given at the beginning of the line, the input would break your current indentation and make the script look unsightly. Thus, you can rewrite the script in Listing 12 to match Listing 14, and the output will be identical.

Listing 14. A shell script here document with leading indentation

```
#!/bin/sh

        cat <<- EOF
                baseconv is a program to convert a number from one base to another.

                Usage: baseconv [options]

                Options:

                -i BASE input base
                -o BASE output base
                -h display this message

                For more information, consult the baseconv man page.
                EOF
```

Inline input at the command line

At the command line, inline input is used with one-liners that call an interactive program, such as the bc calculator discussed in the Base conversion using bc section. You can use a here document to substitute a real file, or any number of lines of real input, for any interactive command.

Try sending multi-line input to bc with a here document. Type what's shown in Listing 15.

Listing 15. Sending inline input to an interactive program

```
$ bc << EOF
> ibase=16
> A
> EOF
10
$
```

Variables are normally expanded with inline input. Try what's shown in Listing 16.

Listing 16. How variable expansion occurs in inline input

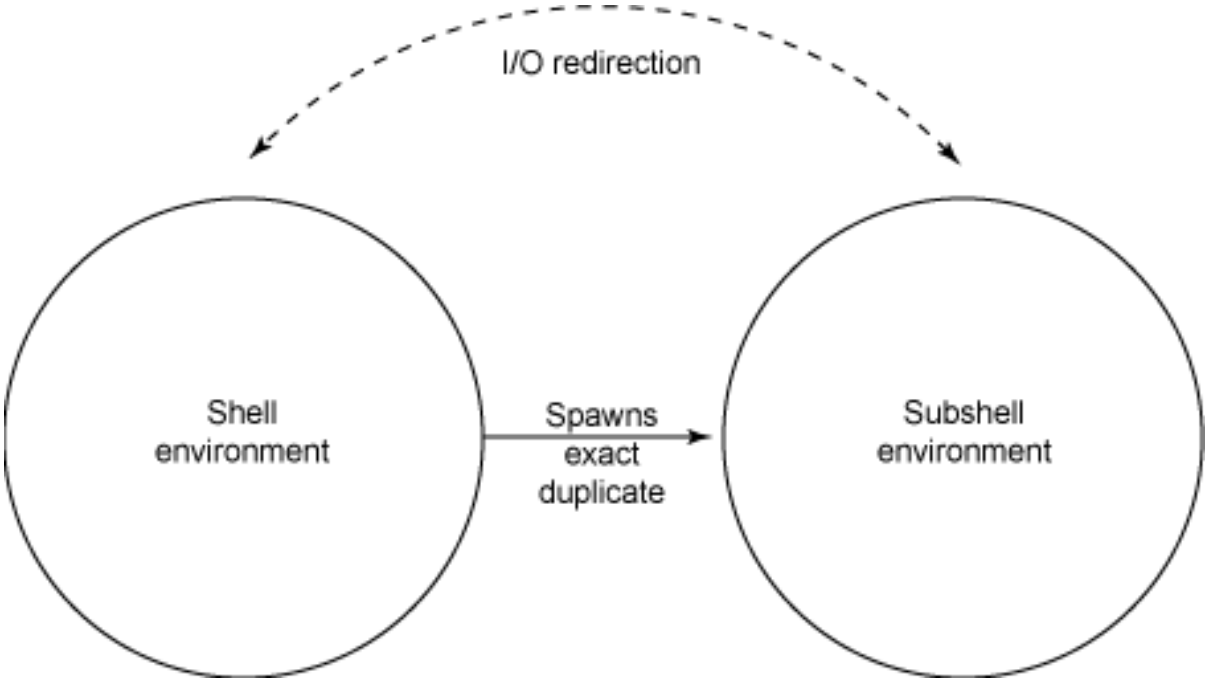
```
$ BASECON=16
$ bc << EOF
> ibase=16
> $BASECON
> EOF
22
$
```

Section 5. Subshell execution

A command or list of commands can be executed in a new shell called a *subshell*,

whose parent is the current shell. Subshells take the environment of their parent. I/O redirection can occur between the child and parent shells, but the subshell can never modify the parent environment. This is desirable when you want to change the shell environment for the execution of those commands, such as by setting variables, but you do not want to change the environment that the script itself is running in. It's also desirable to run subshells when you want to have multiple, long-running processes started and running in the background at the same time. A shell can spawn multiple subshells, and subshells, in turn, can recursively spawn any number of their own subshells. Figure 1 illustrates the process.

Figure 1. How a subshell interacts with its parent shell



The shell sometimes automatically spawns subshells of its own, such as when built-ins are used on a pipeline. When in a subshell, the shell $ parameter expands to the process ID (PID) of the parent shell, never of the subshell.

Running commands in a subshell

To run a group of commands in a subshell, enclose them in parentheses. You can use redirection to send input to the subshell's standard input or to send its collective output to a file or a pipeline.

Try typing what's shown in Listing 17 in your home directory. It creates an example directory and some test files, provided that you don't already have a directory named example.

Listing 17. Making a group of files in a subshell

```
$ pwd
/home/user
$ (mkdir example; cd example; touch A B C)
```

```
$ pwd
/home/user
$ cd example; ls
A B C
$ pwd
/home/user/example
$
```

In this example, the shell spawns a subshell that runs in the background, making the example directory and using `touch` to create three dummy files in that directory. Meanwhile, the shell returns to the prompt in your home directory.

Subshells are convenient both at the command line and in scripts when you have a group of commands that take a long time to execute. To keep your shell free, you can run it in the background, or run many of them in the background:

```
( group-of-long-running-commands ) &
( another-group-of-long-running-commands ) &
( yet-another-group-of-long-running-commands ) &
```

Subshells and variables

It's important to understand how variables work with subshells. Since the subshell environment is a duplicate of its parent, it inherits all of the variables of the parent. But the parent shell never sees any changes that are made in the subshell environment, and the subshell, in turn, never sees any changes that are made in the parent after the subshell is spawned.

As an example, use the `vi` editor to save the script in Listing 18 to a file in your home directory named vartest, and make it executable (see the Make a shell script section).

Listing 18. Shell script demonstrating variable behavior in subshells

```
#!/bin/sh
# Demonstrates variable behavior in a subshell environment

VAR=10

echo "VAR is" $VAR

(

echo "In the subshell, VAR is still" $VAR

VAR=$(($VAR+5))

echo "The new value of VAR in the subshell is" $VAR

)

echo "Outside of the subshell, VAR is" $VAR
```

Now try executing the script by typing its name, as shown in Listing 19.

Listing 19. Output of the vartest script

```
$ vartest
VAR is 10
In the subshell, VAR is still 10
The new value of VAR in the subshell is 15
Outside of the subshell, VAR is 10
$
```

Section 6. Continuous loops

Now it's time to look at loops, which let you perform iterative tasks, such as performing some action or command on a group of files. The shell has several methods for constructing loops.

Make a for loop

The most common loop construction is the `for` loop. It begins by defining a variable that names the loop, gives a list of members, which can be any word, including integers and file names, and then gives the commands to be executed in each iteration. Each of the commands is terminated by a semicolon character (;), and the entire grouping is enclosed by the words `do` and `done`. Its structure is described in Listing 20.

Listing 20. Structure of a shell for loop

```
for loopname in members

do

        command;
        command;
        ...
        command;

done
```

In the first iteration of the loop, the *loopname* variable takes the value of the first member. Then *loopname*'s value is replaced by the value of the next member in the list, and it continues this iteration until there are no more members.

In most shells, the `do` and `done` can be replaced by curly brackets, as shown in Listing 21.

Listing 21. Alternate structure of a shell for loop

```
for loopname in members

{
```

```
        command;
        command;
        ...
        command;

}
```

Type the text from Listing 22 to run a simple loop with three members:

Listing 22. Changing the value of a variable with a loop

```
$ for i in 1 2 3
> {
> VAR = $(($VAR+$i))
> echo $i:$VAR
> }
1:1
2:3
3:6
$
```

Execute commands on each file in a directory

You can use loops to execute a command or group of commands on a given set of files. If you give the names of the files as the members of a `for` loop, the loop operates on each of those files in the order that you give them. You can give the same file twice and the loop operates on it in turn. Try it in your example directory using the text from Listing 23.

Listing 23. Constructing a loop with a list of files

```
$ cd ~/example
$ ls
A B C
$ for file in C B B C
> {
> echo $file
> }
C
B
B
C
$
```

To perform an operation on every file in a directory, use an asterisk (*) as the sole member of the loop, as shown in Listing 24. The shell expands it to all the files in the directory. Then, for the commands inside the loop that you wish to operate on all of the files, use the *loopname* variable as the proper argument or option.

Listing 24. Executing a command on all files in a directory

```
$ ls
A B C
$ for file in *
> {
> mv $file $((0x$file))
```

```
> }
$
```

If you've been running all of the examples in this tutorial, the contents of your example directory should be changed:

```
$ ls
10 11 12
$
```

What happened was the `mv` command in the loop changed the name of the file from a hexadecimal value (constructed by inserting `0x` to the beginning of the name) to its decimal equivalent.

Make a while loop

You can make a loop that runs for as long as some condition is true. Do this with the `while` conditional, whose format is described in Listing 25.

Listing 25. Structure of a shell while loop

```
while [ condition ]; do

        command;
        command;
        ...
        command;
done
```

In the loop, a *condition* can be a statement built with an operator (see Table 3), or it can be as simple as a variable name. As long as the value is non-zero, it's true.

Table 3. Common shell operators

Operator	Description
`-eq`	Equal to
`-ne`	Not equal to
`-lt`	Less than
`-le`	Less than or equal to
`-gt`	Greater than
`-ge`	Greater than or equal to

There are a few notes of caution to keep in mind when constructing `while` loops. First, there *must always be whitespace between the condition and its surrounding brackets*. Second, if a variable is used for a numeric comparison in the condition, *the variable must be defined first*, before the `while` statement.

Type the text from Listing 26 to execute a short `while` loop:

Listing 26. Changing a variable with a while loop

```
$ VAR=0
$ while [ $VAR -lt 10 ]; do
>   echo $VAR;
>   VAR=$(($VAR+1));
> done
0
1
2
3
4
5
6
7
8
9
$
```

Make an until loop

The `until` conditional is similar to `while` and uses the same operators, but it does the opposite. It only executes the loop if the condition is false, and the loop continues to iterate *until* the given condition becomes true. Its format is described in Listing 27.

Listing 27. Structure of a shell until loop

```
until [ condition ] ; do

        command;
        command;
        ...
        command;
done
```

Try running a short `until` loop by typing the text from Listing 28:

Listing 28. Changing a variable with an until loop

```
$ VAR=10
$ until [ $VAR -eq 0 ]; do
>   echo $VAR;
>   VAR=$(($VAR-1));
> done
10
9
8
7
6
5
4
3
2
1
$
```

Nest multiple loops

You can nest loops and combine the various types of loops to perform assorted complex operations. Since the elements of a `for` loop need not be numeric nor in any kind of chronological order, you might use command names to be later executed as commands in some inner loop, such as `printf`, `echo`, `stop`, `resume`, and so on.

Try running the sample in Listing 29. It's an `until` loop that performs arithmetic substitution while nested in a `for` loop whose loop words are not in numeric order.

Listing 29. Arithmetic substitution with nested loops

```
$ for i in 250 100 2136 875
>  {
>    VAR=10;
>    until [ $VAR -eq 0 ]; do
>      echo "$i / $VAR = $(($i/$VAR))  $i * $VAR = $(($i*$VAR))\
            $i + $VAR = $(($i+$VAR))  $i - $VAR = $(($i-$VAR))";
>      VAR=$(($VAR-1));
>    done;
>  }
250 / 10 = 25  250 * 10 = 2500  250 + 10 = 260  250 - 10 = 240
250 / 9 = 27  250 * 9 = 2250  250 + 9 = 259  250 - 9 = 241
250 / 8 = 31  250 * 8 = 2000  250 + 8 = 258  250 - 8 = 242
250 / 7 = 35  250 * 7 = 1750  250 + 7 = 257  250 - 7 = 243
250 / 6 = 41  250 * 6 = 1500  250 + 6 = 256  250 - 6 = 244
250 / 5 = 50  250 * 5 = 1250  250 + 5 = 255  250 - 5 = 245
250 / 4 = 62  250 * 4 = 1000  250 + 4 = 254  250 - 4 = 246
250 / 3 = 83  250 * 3 = 750  250 + 3 = 253  250 - 3 = 247
250 / 2 = 125  250 * 2 = 500  250 + 2 = 252  250 - 2 = 248
250 / 1 = 250  250 * 1 = 250  250 + 1 = 251  250 - 1 = 249
100 / 10 = 10  100 * 10 = 1000  100 + 10 = 110  100 - 10 = 90
100 / 9 = 11  100 * 9 = 900  100 + 9 = 109  100 - 9 = 91
100 / 8 = 12  100 * 8 = 800  100 + 8 = 108  100 - 8 = 92
100 / 7 = 14  100 * 7 = 700  100 + 7 = 107  100 - 7 = 93
100 / 6 = 16  100 * 6 = 600  100 + 6 = 106  100 - 6 = 94
100 / 5 = 20  100 * 5 = 500  100 + 5 = 105  100 - 5 = 95
100 / 4 = 25  100 * 4 = 400  100 + 4 = 104  100 - 4 = 96
100 / 3 = 33  100 * 3 = 300  100 + 3 = 103  100 - 3 = 97
100 / 2 = 50  100 * 2 = 200  100 + 2 = 102  100 - 2 = 98
100 / 1 = 100  100 * 1 = 100  100 + 1 = 101  100 - 1 = 99
2136 / 10 = 213  2136 * 10 = 21360  2136 + 10 = 2146  2136 - 10 = 2126
2136 / 9 = 237  2136 * 9 = 19224  2136 + 9 = 2145  2136 - 9 = 2127
2136 / 8 = 267  2136 * 8 = 17088  2136 + 8 = 2144  2136 - 8 = 2128
2136 / 7 = 305  2136 * 7 = 14952  2136 + 7 = 2143  2136 - 7 = 2129
2136 / 6 = 356  2136 * 6 = 12816  2136 + 6 = 2142  2136 - 6 = 2130
2136 / 5 = 427  2136 * 5 = 10680  2136 + 5 = 2141  2136 - 5 = 2131
2136 / 4 = 534  2136 * 4 = 8544  2136 + 4 = 2140  2136 - 4 = 2132
2136 / 3 = 712  2136 * 3 = 6408  2136 + 3 = 2139  2136 - 3 = 2133
2136 / 2 = 1068  2136 * 2 = 4272  2136 + 2 = 2138  2136 - 2 = 2134
2136 / 1 = 2136  2136 * 1 = 2136  2136 + 1 = 2137  2136 - 1 = 2135
875 / 10 = 87  875 * 10 = 8750  875 + 10 = 885  875 - 10 = 865
875 / 9 = 97  875 * 9 = 7875  875 + 9 = 884  875 - 9 = 866
875 / 8 = 109  875 * 8 = 7000  875 + 8 = 883  875 - 8 = 867
875 / 7 = 125  875 * 7 = 6125  875 + 7 = 882  875 - 7 = 868
875 / 6 = 145  875 * 6 = 5250  875 + 6 = 881  875 - 6 = 869
875 / 5 = 175  875 * 5 = 4375  875 + 5 = 880  875 - 5 = 870
875 / 4 = 218  875 * 4 = 3500  875 + 4 = 879  875 - 4 = 871
875 / 3 = 291  875 * 3 = 2625  875 + 3 = 878  875 - 3 = 872
875 / 2 = 437  875 * 2 = 1750  875 + 2 = 877  875 - 2 = 873
875 / 1 = 875  875 * 1 = 875  875 + 1 = 876  875 - 1 = 874
$
```

Section 7. Reading keyboard input

You can also read keyboard input in your scripts or from the command line itself. This is accomplished with the `read` command, which is a built-in function that takes any number of variable names as arguments. It reads values into the variables from the standard input, reading in a single line of input and assigning each input word to each variable.

Try reading a variable, as shown in Listing 30:

Listing 30. Reading a variable with read

```
$ read VAR
                    23
$ echo $VAR
23
$
```

Use the `-p` option to give a prompt for each read. Give the prompt as a string enclosed in quotes, as shown in Listing 31. Variable expansion occurs.

Listing 31. Using a prompt with a variable read

```
$ read -p "Instead of $VAR, what number would you like? " VAR
Instead of 23, what number would you like? 17
$ echo $VAR
17
$
```

If there are more words in the keyboard input than there are variables, the variables are assigned the input words in turn until reaching the last variable, which is then assigned the remainder of the input line. (If *fewer* words are given in the input than variables, then variables are assigned values until all of the input is assigned, after which all remaining variables are given null values.)

Reading in loops

You can use `read` as a conditional in a loop. Try it now using Listing 32:

Listing 32. Reading a list of file names in a loop

```
$ while read -p "File? " file; do ls $file; done
File? 10
10
File? 12
12
File? 42
```

```
42: no such file or directory
File?
                        Carriage return

10 11 12
File?
                        Control-C

$
```

This technique is often used when the input to the loop is piped. Try typing the text from Listing 33, which replaces the output of the `ls` command with a loop:

Listing 33. Reading from a pipeline

```
$ ls | while read file; do ls $file; done
10
11
12
$
```

You can also operate on the variable over multiple lines, such as sending a message to standard output and also performing shell arithmetic on the loopname variable (see the Shell arithmetic and base conversion section). Try the example provided in Listing 34:

Listing 34. A longer loop with a piped read

```
$ ls | while read file; do echo "The file is " `ls -i $file`; \
echo "If the number were in hex, the value would be $((16#$file))"; done
The file is  100267120 10
If the number were in hex, the value would be 16
The file is  100267121 11
If the number were in hex, the value would be 17
The file is  100267122 12
If the number were in hex, the value would be 18
$
```

You can read multiple values in one piped `read`, as shown in Listing 35.

Listing 35. Reading multiple variables from a pipeline

```
$ ls -i | while read inode file; do \
                    echo "File $file has inode $inode"; done
File 10 has inode 100267120
File 11 has inode 100267121
File 12 has inode 100267122
$
```

Section 8. Putting it all together

This concluding section combines the various tricks and techniques you've just

learned for some useful, real-life one-liners. It also includes a simple shell script that performs arbitrary base conversion.

Useful one-liners

The following examples are sample shell one-liners that perform useful functions. All of them consist of various constructions described in this tutorial.

- Take a group of files in the current directory that are exactly two characters long and rename them with a .ppm extension:

```
for i in ??; { mv $i $i.ppm; }
```

- Copy an entire directory tree using `tar` and subshells while keeping all of the same file permissions:

```
( cd source ; tar pcf - * ) | ( cd target ; tar pxvf - )
```

- Read a binary number and output the value in decimal:

```
read BINLOC;echo $((2#$BINLOC))
```

- Find all of the files in the /usr/local directory tree that have an .mp3 extension -- these files might have space characters in their name -- and compress them with the `bzip2` utility:

```
find /usr/local -name "*.mp3" | while read name ; do bzip2 $name; done
```

- Output all the decimal numbers in a given file to their hexadecimal value:

```
cat file | while read number ; do echo $((0x$number)); done
```

- Convert all the decimal numbers in a given file to their hexadecimal value and output them to a new file with a .hex extension:

```
cat file | while read number ; do echo $((0x$number)) >> file.hex; done
```

- Make ten iterations of a loop, running *command* with the number (from 0 to 90 in increments of ten) passed as an argument:

```
i=0; while [ $i -ne 100 ]; do command $i; i=$(($i+10)); done
```

A sample script: convert numbers to another base

Some of the various tricks discussed in this tutorial are combined together in Listing 36. It's a simple script, `baseconv`, that converts numbers from a given input base to an output base. Give it the values for the input and output bases as arguments, and it reads numbers from the keyboard input until it gets a number of 0.

Listing 36. A simple script to convert bases

```
#!/bin/sh
# baseconv, convert numbers from one base to another.
#

NUMBER=1
while [ $NUMBER ]; do
    read -p "Input base: " IN
    read -p "Output base: " OUT
    read -p "Number: " NUMBER
    bc -ql <<- EOF
                obase=$OUT
                ibase=$IN
                $NUMBER
                EOF
done
```

When you have it in an executable file (see the Make a shell script section), try running it, as shown in Listing 37:

Listing 37. Output of the baseconv script

```
$ ./baseconv
Input base: 10
Output base: 16
Number: 33
21
Input base: 2
Output base: 1100
Number: 101
5
Input base: 16
Output base: A
Number: ACA
2762
Input base: 10
Output base: 10
Number:
                        Carriage return

$
```

Section 9. Conclusion

Wrap-up

Whew! This tutorial sure has covered a lot of ground, sweeping you through a whirlwind tour of many basic shell programming concepts. While taking this tutorial, you learned many core concepts of shell programming: continuous loops, inline input, reading keyboard input, base conversion, and subshell execution. You also learned how shell code snippets can be run as one-liners right from a shell prompt and how to put them together in a file as an executable script. These are some of the most important shell scripting concepts you can learn. If you use what you learned in this tutorial along with what you learned in the previous tutorials in this series, you're well on your way to becoming a UNIX pro.

Resources

Learn

- UNIX tips and tricks for a new user: Check out other parts in this series.
- "Working in the bash shell" (developerWorks, May 2006): A step-by-step tutorial that describes the features of `bash`.
- "Learn 10 good UNIX usage habits" (developerWorks, December 2006): This article gives examples of good usage habits in the shell, including hints on quoting variables and running subshells.
- AIX documentation: Read this documentation on inline input.
- "Produce device-independent documentation with Groff" (developerWorks, June 2006): This tutorial uses inline input to show how to process input documents with the Groff typesetting system.
- Check out other articles and tutorials written by Michael Stutz:
  - AIX and UNIX zone
  - Across developerWorks
- Search the AIX and UNIX library by topic:
  - System administration
  - Application development
  - Performance
  - Porting
  - Security
  - Tips
  - Tools and utilities
  - Java™ technology
  - Linux®
  - Open source
- AIX and UNIX: The AIX and UNIX developerWorks zone provides a wealth of information relating to all aspects of AIX systems administration and expanding your UNIX skills.
- New to AIX and UNIX: Visit the New to AIX and UNIX page to learn more about AIX and UNIX.
- AIX 5L Wiki: A collaborative environment for technical information related to AIX.

- Safari bookstore: Visit this e-reference library to find specific technical resources.
- developerWorks technical events and webcasts: Stay current with developerWorks technical events and webcasts.
- Podcasts: Tune in and catch up with IBM technical experts.

Get products and technologies

- IBM trial software: Build your next development project with software for download directly from developerWorks.

Discuss

- Participate in the developerWorks blogs and get involved in the developerWorks community.
- Participate in the AIX and UNIX forums:
  - AIX 5L -- technical forum
  - AIX for Developers Forum
  - Cluster Systems Management
  - IBM Support Assistant
  - Performance Tools -- technical
  - Virtualization -- technical
  - More AIX and UNIX forums

About the author

Michael Stutz
Michael Stutz is author of *The Linux Cookbook* , which he also designed and typeset using only open source software. His research interests include digital publishing and the future of the book. He has used various UNIX operating systems for 20 years. You can reach Michael at stutz@dsl.org.

Trademarks

countries, or both.
UNIX is a registered trademark of The Open Group in the United States and other countries.