```
        echo $word is not a dictionary word;
fi
```

The `aspell list` command returns output text when the given input is not a dictionary word, and does not output anything when the input is a dictionary word. A `-z` command checks whether `$output` is an empty string or not.

The `look` command will display lines that begin with a given string. You might use it to find the lines in a log file that start with a given date, or to find words in the dictionary that start with a given string. By default, `look` searches `/usr/share/dict/words`, or you can provide a file to search.

```
$ look word
```

Alternatively, this can be used:

```
$ grep "^word" filepath
```

Consider this example:

```
$ look android
android
android's
androids
```

Use this to find lines with a given date in `/var/log/syslog`:

```
$look 'Aug 30' /var/log/syslog
```

Automating interactive input

We looked at commands that accept arguments on the command line. Linux also supports many interactive applications ranging from `passwd` to `ssh`.

We can create our own interactive shell scripts. It's easier for casual users to interact with a set of prompts rather than remember command line flags and the proper order. For instance, a script to back up a user's work, but not to back up and lock files, might look like this:

```
$ backupWork.sh
```

- What folder should be backed up? `notes`
- What type of files should be backed up? `.docx`

Automating interactive applications can save you time when you need to rerun the same application and frustration while you're developing one.

Getting ready

The first step to automating a task is to run it and note what you do. The script command discussed earlier may be of use.

How to do it...

Examine the sequence of interactive inputs. From the previous code, we can formulate the steps of the sequence like this:

```
notes[Return]docx[Return]
```

In addition to the preceding steps, type `notes`, press `Return`, type `docx`, and finally press `Return` to convert into a single string like this:

```
        "notes\ndocx\n"
```

The `\n` character is sent when we press Return. By appending the return (`\n`) characters, we get the string that is passed to `stdin` (standard input).

By sending the equivalent string for the characters typed by the user, we can automate passing input to the interactive processes.

How it works...

Let's write a script that reads input interactively for an automation example:

```
#!/bin/bash
# backup.sh
# Backup files with suffix. Do not backup temp files that start with ~
read -p " What folder should be backed up: " folder
read -p " What type of files should be backed up: " suffix
find $folder -name "*.$suffix" -a ! -name '~*' -exec cp {} \
    $BACKUP/$LOGNAME/$folder
echo "Backed up files from $folder to $BACKUP/$LOGNAME/$folder"
```

Let's automate the sending of input to the command:

```
$ echo -e "notes\ndocx\n" | ./backup.sh
Backed up files from notes to /BackupDrive/MyName/notes
```

This style of automating an interactive script can save you a lot of typing during developing and debugging. It also insures that you perform the same test each time and don't end up chasing a phantom bug because you mis-typed.

We used `echo -e` to produce the input sequence. The `-e` option signals to `echo` to interpret escape sequences. If the input is large we can use an input file and the redirection operator to supply input:

```
$ echo -e "notes\ndocx\n"  > input.data
$ cat input.data
notes
docx
```

You can manually craft the input file without the `echo` commands by hand–typing. Consider this example:

```
$ ./interactive.sh < input.data
```

This redirects interactive input data from a file.

If you are a reverse engineer, you may have played with buffer overflow exploits. To exploit them we need to redirect a shell code such as `\xeb\x1a\x5e\x31\xc0\x88\x46`, which is written in hex. These characters cannot be typed directly on the keyboard as keys for these characters are not present. Therefore, we use:

```
echo -e \xeb\x1a\x5e\x31\xc0\x88\x46"
```

This will redirect the byte sequence to a vulnerable executable.

These echo and redirection techniques automate interactive input programs. However, these techniques are fragile, in that there is no validity checking and it's assumed that the target application will always accept data in the same order. If the program asks for input in a changing order, or some inputs are not always required, these methods fail.

The expect program can perform complex interactions and adapt to changes in the target application. This program is in worldwide use to control hardware tests, validate software builds, query router statistics, and much more.

There's more...

The expect application is an interpreter similar to the shell. It's based on the TCL language. We'll discuss the spawn, expect, and send commands for simple automation. With the power of the TCL language behind it, expect can do much more complex tasks. You can learn more about the TCL language at the `www.tcl.tk` website.

Automating with expect

`expect` does not come by default on all Linux distributions. You may have to install the expect package with your package manager (`apt-get` or `yum`).

Expect has three main commands:

Commands	**Description**
`spawn`	Runs the new target application.
`expect`	Watches for a pattern to be sent by the target application.
`send`	Sends a string to the target application.

The following example spawns the backup script and then looks for the patterns `*folder*` and `*file*` to determine if the backup script is asking for a folder name or a filename. It will then send the appropriate reply. If the backup script is rewritten to request files first and then folders, this automation script still works.

```
#!/usr/bin/expect
#Filename: automate_expect.tcl
spawn ./backup .sh
expect {
  "*folder*" {
     send "notes\n"
     exp_continue
   }
  "*type*" {
     send "docx\n"
     exp_continue
  }
}
```

Run it as:

```
$ ./automate_expect.tcl
```

The `spawn` command's parameters are the target application and arguments to be automated.

The `expect` command accepts a set of patterns to look for and an action to perform when that pattern is matched. The action is enclosed in curly braces.

The `send` command is the message to be sent. This is similar to echo `-n -e` in that it does not automatically include the newline and does understand backslash symbols.

Making commands quicker by running parallel processes

Computing power constantly increases not only because processors have higher clock cycles but also because they have multiple cores. This means that in a single hardware processor there are multiple logical processors. It's like having several computers, instead of just one.

However, multiple cores are useless unless the software makes use of them. For example, a program that does huge calculations may only run on one core while the others will sit idle. The software has to be aware and take advantage of the multiple cores if we want it to be faster.

In this recipe, we will see how we can make our commands run faster.

How to do it...

Let's take an example of the `md5sum` command we discussed in the previous recipes. This command performs complex computations, making it CPU-intensive. If we have more than one file that we want to generate a checksum for, we can run multiple instances of `md5sum` using a script like this:

```
#/bin/bash
#filename: generate_checksums.sh
PIDARRAY=()
for file in File1.iso File2.iso
do
  md5sum $file &
  PIDARRAY+=("$!")
done
wait ${PIDARRAY[@]}
```

When we run this, we get the following output:

```
$ ./generate_checksums.sh
330dcb53f253acdf76431cecca0fefe7  File1.iso
bd1694a6fe6df12c3b8141dcffaf06e6  File2.iso
```

The output will be the same as running the following command:

```
md5sum File1.iso File2.iso
```

However, if the `md5sum` commands run simultaneously, you'll get the results quicker if you have a multi–core processor (you can verify this using the `time` command).

How it works...

We exploit the Bash operand `&`, which instructs the shell to send the command to the background and continue with the script. However, this means our script will exit as soon as the loop completes, while the `md5sum` processes are still running in the background. To prevent this, we get the PIDs of the processes using `$!`, which in Bash holds the PID of the last background process. We append these PIDs to an array and then use the `wait` command to wait for these processes to finish.

There's more...

The Bash `&` operand works well for a small number of tasks. If you had a hundred files to checksum, the script would try to start a hundred processes and might force your system into swapping, which would make the tasks run slower.

The GNU parallel command is not part of all installations, but again it can be loaded with your package manager. The parallel command optimizes the use of your resources without overloading any of them.

The parallel command reads a list of files on `stdin` and uses options similar to the find command's `-exec` argument to process these files. The `{}` symbol represents the file to be processed, and the `{.}` symbol represents the filename without a suffix.

The following command uses **Imagemagick's** `convert` command to make new, resized images of all the images in a folder:

```
ls *jpg | parallel convert {} -geometry 50x50 {.}Small.jpg
```