Converting the preceding steps `1`, `Return`, `hello`, and `Return` by observing the characters that are actually typed in the keyboard, we can formulate the following string:

```
"1\nhello\n"
```

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

Hence, by sending the equivalent string for the characters typed by the user, we can automate the passing of input in the interactive processes.

How it works...

Let's write a script that reads input interactively and uses this script for automation examples:

```
#!/bin/bash
#Filename: interactive.sh
read -p "Enter number:" no ;
read -p "Enter name:" name
echo You have entered $no, $name;
```

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

```
$ echo -e "1\nhello\n" | ./interactive.sh
You have entered 1, hello
```

Thus crafting input with `\n` works.

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

```
$ echo -e "1\nhello\n"  > input.data
$ cat input.data
1
hello
```

You can also manually craft the input file without the `echo` commands by hand typing. For 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 shellcode such as `"\xeb\x1a\x5e\x31\xc0\x88\x46"`, which is written in hex. These characters cannot be typed directly through the keyboard as keys for these characters are not present in the keyboard. Therefore, we should use:

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

This will redirect the shellcode to a vulnerable executable.

We have described a method to automate interactive input programs by redirecting expected input text through `stdin` (standard input). We are sending the input without checking the input the program asks for. We are also expecting the program to ask for input in a specific (static) order. If the program asks for input randomly or in a changing order, or sometimes certain inputs are never asked for, the aforementioned method fails. It will send the wrong inputs to different input prompts by the program. In order to handle a dynamic input supply and provide input by checking the input requirements by the program on runtime, we have a great utility called `expect`. The `expect` command supplies the correct input for the correct input prompt by the program.

There's more...

Trailing from the previous section, let's see how to use `expect`. Automation of interactive input can also be done using other methods. Expect scripting is another method for automation. Let's go through it.

Automating with expect

`expect` does not come by default with most of the common Linux distributions. You have to install the expect package manually using your package manager.

`expect` expects for a particular input prompt and sends data by checking messages in the input prompt:

```
#!/usr/bin/expect
#Filename: automate_expect.sh
spawn ./interactive .sh
expect "Enter number:"
send "1\n"
expect "Enter name:"
send "hello\n"
expect eof
```

Run it as follows:

```
$ ./automate_expect.sh
```