# Stay Out Of Trouble

Now that our scripts are getting a little more complicated, I want to point out some common mistakes that you might run into. To do this, create the following script called **`trouble.bash`**. Be sure to enter it exactly as written.

```
#!/bin/bash

number=1

if [ $number = "1" ]; then
    echo "Number equals 1"
else
    echo "Number does not equal 1"
fi
```

When you run this script, it should output the line "Number equals 1" because, well, **`number`** equals 1. If you don't get the expected output, check your typing; you made a mistake.

## Empty Variables

Edit the script to change line 3 from:

```
number=1
```

to:

```
number=
```

and run the script again. This time you should get the following:

```
[me@linuxbox me]$ ./trouble.bash
/trouble.bash: [: =: unary operator expected.
Number does not equal 1
```

As you can see, **bash** displayed an error message when we ran the script. You probably think that by removing the "1" on line 3 it created a syntax error on line 3, but it didn't. Let's look at the error message again:

```
./trouble.bash: [: =: unary operator expected
```

We can see that **./trouble.bash** is reporting the error and the error has to do with "**[**". Remember that "**[**" is an abbreviation for the **test** shell builtin. From this we can determine that the error is occurring on line 5 not line 3.

First, let me say there is nothing wrong with line 3. **number=** is perfectly good syntax. You will sometimes want to set a variable's value to nothing. You can confirm the validity of this by trying it on the command line:

```
[me@linuxbox me]$ number=
[me@linuxbox me]$
```

See, no error message. So what's wrong with line 5? It worked before.

To understand this error, we have to see what the shell sees. Remember that the shell spends a lot of its life expanding text. In line 5, the shell expands the value of **`number`** where it sees **`$number`**. In our first try (when **`number=1`**), the shell substituted 1 for **`$number`** like so:

```
if [ 1 = "1" ]; then
```

However, when we set number to nothing (**`number=`**), the shell saw this after the expansion:

```
if [ = "1" ]; then
```

which is an error. It also explains the rest of the error message we received. The "**=**" is a binary operator; that is, it expects two items to operate upon - one on each side. What the shell is trying to tell us is that there is only one item and there should be a unary operator (like "!") that only operates on a single item.

To fix this problem, change line 5 to read:

```
if [ "$number" = "1" ]; then
```

Now when the shell performs the expansion it will see:

```
if [ "" = "1" ]; then
```

which correctly expresses our intent.

This brings up an important thing to remember when you are writing your scripts. Consider what happens if a variable is set to equal nothing.

## Missing Quotes

Edit line 6 to remove the trailing quote from the end of the line:

```
    echo "Number equals 1
```

and run the script again. You should get this:

```
[me@linuxbox me]$ ./trouble.bash
./trouble.bash: line 8: unexpected EOF while looking for
matching "
./trouble.bash: line 10 syntax error: unexpected end of
file
```

Here we have another case of a mistake in one line causing a problem later in the script. What happens is the shell keeps looking for the closing quotation mark to tell it where the end of the string is, but runs into the end of the file before it finds it.

These errors can be a real pain to find in a long script. This is one reason you should test your scripts frequently when you are writing them so there is less new code to test. I also find that text editors with syntax highlighting make these kinds of bugs easier to find.

## Isolating Problems

## Isolating Problems

Finding bugs in your programs can sometimes be very difficult and frustrating. Here are a couple of techniques that you will find useful:

**Isolate blocks of code by "commenting them out."** This trick involves putting comment characters at the beginning of lines of code to stop the shell from reading them. Frequently, you will do this to a block of code to see if a particular problem goes away. By doing this, you can isolate which part of a program is causing (or not causing) a problem.

For example, when we were looking for our missing quotation we could have done this:

```
#!/bin/bash

number=1

if [ $number = "1" ]; then
    echo "Number equals 1
#else
#   echo "Number does not equal 1"
fi
```

By commenting out the **`else`** clause and running the script, we could show that the problem was not in the **`else`** clause even though the error message suggested that it was.

**Use echo commands to verify your assumptions.** As you gain experience tracking down bugs, you will discover that bugs are often not where you first expect to find them. A common problem will be that you will make a false assumption about the performance of your program. You will see a problem develop at a certain point in your program and assume that the problem is there. This is often incorrect, as we have seen. To combat this, you should place **`echo`** commands in your code while you are debugging, to produce messages that confirm the program is doing what is expected. There are two kinds of messages that you should insert.

The first type simply announces that you have reached a certain point in the program. We saw this in our earlier discussion on stubbing. It is useful to know that program flow is happening the way we expect.

The second type displays the value of a variable (or variables) used in a calculation or test. You will often find that a portion of your program will fail because something that you assumed was correct earlier in your program is, in fact, incorrect and is causing your program to fail later on.

## Watching Your Script Run

It is possible to have **`bash`** show you what it is doing when you run your script. To do this, add a "**`-x`**" to the first line of your script, like this:

```
#!/bin/bash -x
```

Now, when you run your script, bash will display each line (with expansions performed) as it executes it. This technique is called *tracing*. Here is what it looks like:

```
[me@linuxbox me]$ ./trouble.bash
+ number=1
+ '[' 1 = 1 ']'
+ echo 'Number equals 1'
Number equals 1
```

Alternately, you can use the **set** command within your script to turn tracing on and off. Use **set -x** to turn tracing on and **set +x** to turn tracing off. For example.:

```
#!/bin/bash

number=1

set -x
```

```
if [ $number = "1" ]; then
    echo "Number equals 1"
else
    echo "Number does not equal 1"
fi
set +x
```