CITS4407 Open Source Tools and Scripting Conditionals

Unit coordinator: Arran Stewart

Overview

This week:

- Assignment questions
- Conditionals

Conditionals in Bash

We've seen that Bash has "`if`" statements and "`while`" loops – both of these use what are often called *conditional expressions* – expressions which evaluate in some sense to being "true" or "false".

```
if conditional ; then
  statement 1;
  statement 2;
fi
```

```
while conditional ; do
  statement 1;
  statement 2;
done
```

Conditionals in Bash

What is actually happening when Bash encounters a conditional in an "`if`" statement? What does it expect to see in that spot?

The answer is, it expects to see a *command*, which succeeds or fails.

Commands in conditionals

So the following will print "no such directory" (assuming the directory /xxxx doesn't exist):

```
if ls /xxx; then
  echo it exists;
else
  echo no such directory;
fi
```

Commands in conditionals

```
if ls /xxx; then
  echo it exists;
else
  echo no such directory;
fi
```

The command "`ls`" succeeds when it is able to list something, and fails when it can't.

In Bash, *every* command can "succeed" or "fail".

Exit codes

The way an external command tells the operating system whether it succeeded or failed is by returning an *exit code*.

In Unix-like systems, an exit code of 0 means "success", and anything else means "failure".

For instance, `grep` returns "success" when it finds matching lines, and "failure" when it doesn't.

(However, it *also* returns "failure" when something else went wrong – for instance you tried to `grep` for a pattern in a file that doesn't exist.)

Exit codes

Linux has two programs that do nothing but return an exit code:

- The "`true`" program always returns an exit code of 0
- The "`false`" program always returns an exit code of 1

```
$ if true; then echo hi there; fi
hi there
```

Examining exit codes

Bash stores the exit code of the most recently executed command in a special variable called "$?" (it allows you to *query* the most recent exit code, hence the question mark).

```
$ true
$ echo $?
0
$ false
$ echo $?
1
```

Exit codes and non-external commands

In Bash, *every* command has an exit code – not just external programs, but also built-in commands and user-defined functions.

When defining a function, you can use the "return" statement to specify the exit value of your function.

```
always_fails () {
  return 1;
}
```

Exit codes and non-external commands

```
always_fails () {
  return 1;
}
```

If you don't specify a return value, the exit value of the function will be that of the last command it executes. So the following function is equivalent to the one above.

```
always_fails () {
  false;
}
```

Exit codes

Even built-in commands like “declare” (which can be used to explicitly declare variables) have an exit code.

```
$ declare myvar=0
$ echo $?
0
$ declare 000=0
bash: declare: `000=0': not a valid identifier
$ echo $?
1
```

Arithmetic expansion

In Bash, an arithmetic expression inside double brackets ("((" and "))") also has an "exit code".

```
$ (( 1 == 1 && 2 == 2))
$ echo $?
0
$ (( 1 > 10 ))
$ echo $?
1
```

It exits with 1 (failure) if the expression inside evaluates to "false" or "0", and 0 (success) otherwise.

Arithmetic expansion

If you have several conditionals you want to check, joined with “and” or “or” – you might want to see if you can write them using arithmetic expansion, which is often more convenient than using square brackets (“[” and “]”).

```
if [ "$var1" -eq 1 ] && [ "$var2" -eq 1 ] && [ "$var3" -eq 1 ] ; then
  echo "correct";
fi
```

versus,

```
if (( var1 == 1  &&  var2 == 1 && var3 == 1 )) ; then
  echo "correct";
fi
```

Square brackets (test)

The left-square-bracket ("[") is just another command, as far as Bash is concerned. In fact, it's usually available as an external program:

```
$ which [
/usr/bin/[
```

Square brackets (test)

The square-bracket command takes multiple arguments, and expects the last argument to be a right-square-bracket ("]"). And it then exits with success or failure depending on its interpretation of those arguments.

```
$ [ -d /tmp -a -d / ]
$ echo $?
0
```

```
$ [ -d /tmp -a -d /xxx ]
$ echo $?
1
```

Square brackets (test)

However, because it's so frequently used, Bash also defines a built-in command called "[", and this will normally get called instead of the external program.

If you absolutely wanted to use the external program, you would have to write:

```
$ /usr/bin/[ -d /tmp -a -d /xxx ]
$ echo $?
1
```

This explains why Bash (or rather, the "[" command) is so picky about spacing – it needs spaces to tell it where different arguments start and end, and looks for "]" as its last argument.

Square brackets (test)

The `/usr/bin/[` program also comes in a variant called `/usr/bin/test`, which doesn't take any square brackets.

```
$ test -d /tmp -a -d /xxx
$ echo $?
1
```