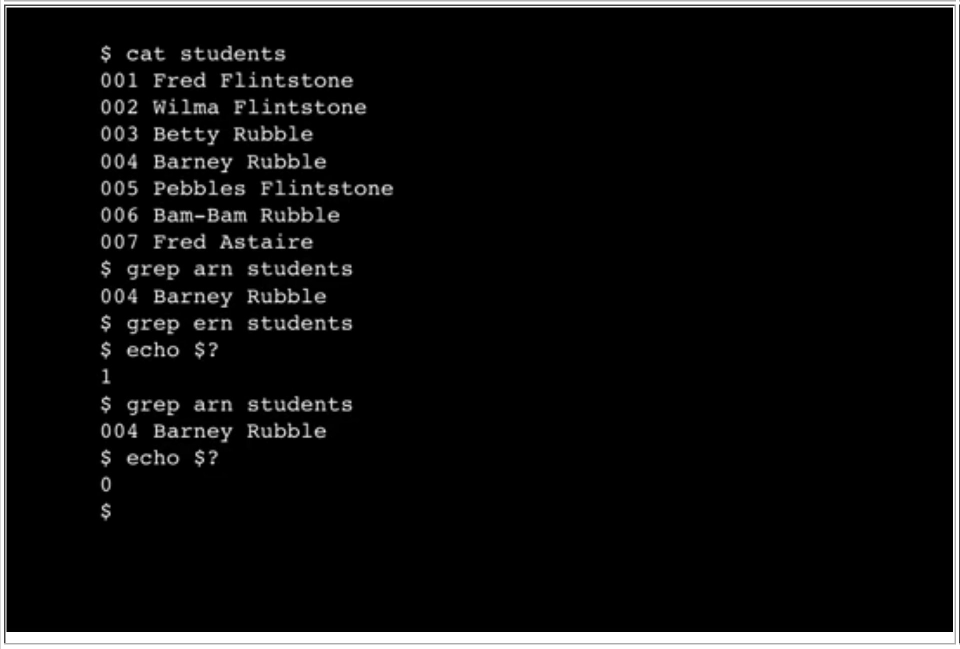
Shell programming 2 of 6

Hover over the image to see the time range in the original video. Click to play that excerpt. Full video at https://www.youtube.com/watch?v=iZuXiEYv_D4

```
$
```

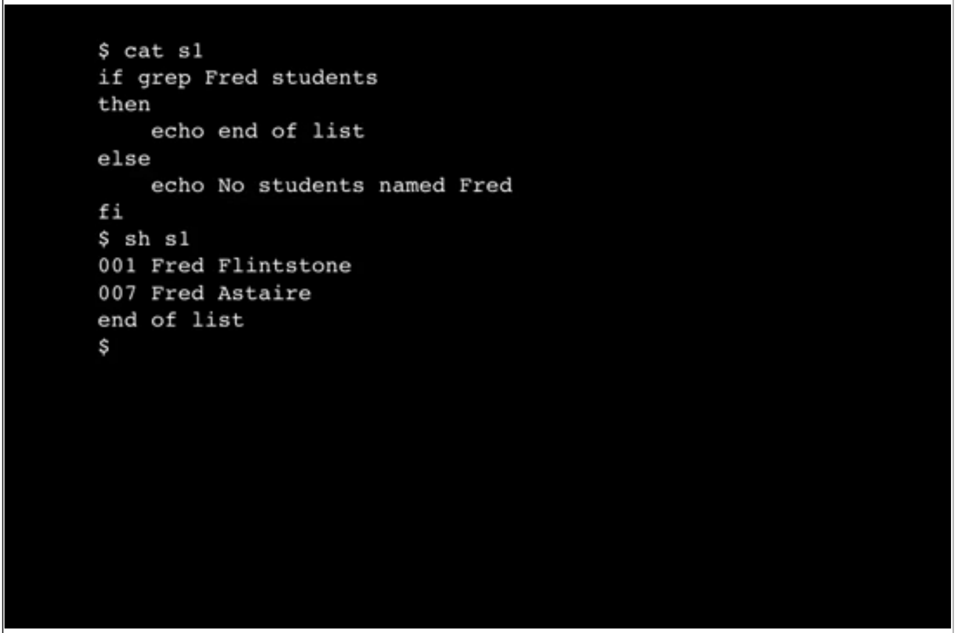
This is my second video about shell programming. In the first video, I talked about a number of things you need to know to do shell programming, but I covered fairly little of what you are used to having in a programming language. Based on your experience programming in more conventional programming languages, you will be expecting control constructs.

Conditional constructs in *sh* are based on commands' exit statuses. It's a very solid and clean design.



```
$ cat students
001 Fred Flintstone
002 Wilma Flintstone
003 Betty Rubble
004 Barney Rubble
005 Pebbles Flintstone
006 Bam-Bam Rubble
007 Fred Astaire
$ grep arn students
004 Barney Rubble
$ grep ern students
$ echo $?
1
$ grep arn students
004 Barney Rubble
$ echo $?
0
$
```

A command in unix succeeds or fails. For example, given this file ("students"): If we do grep arn students it succeeds, but if we do grep ern students it fails. How do we tell?
Well, for interactive exploration we can use a special shell variable "$?", which tells us the exit status of the last command.
And we can use the echo command to see the value of this variable. ("echo $?" yields 1)
Whereas grep arn students succeeds. ("echo $?" yields 0)
In unix or linux, zero is the success exit status, and anything nonzero indicates failure.

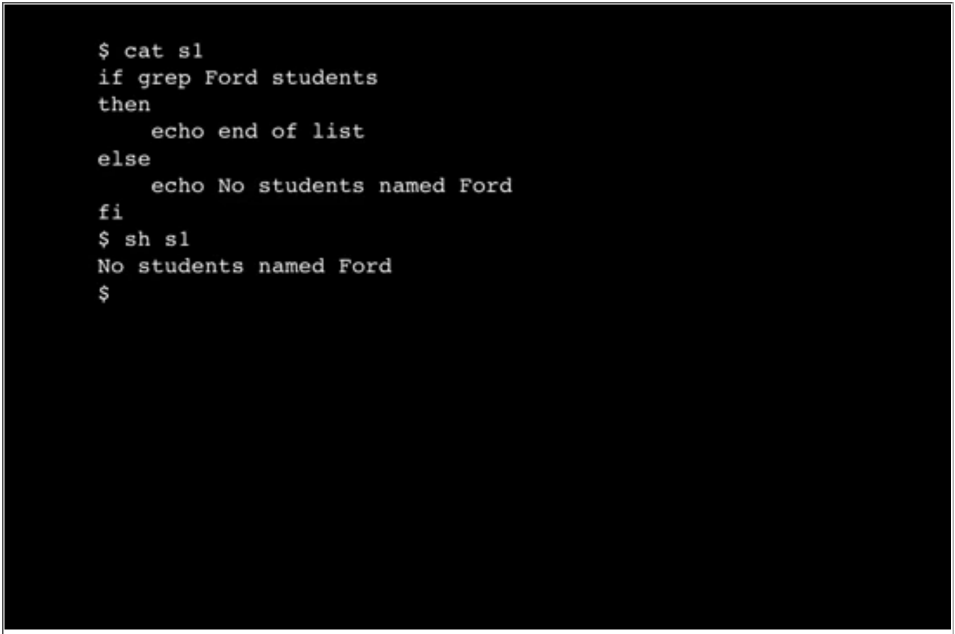


```
$ cat s1
if grep Fred students
then
    echo end of list
else
    echo No students named Fred
fi
$ sh s1
001 Fred Flintstone
007 Fred Astaire
end of list
$
```

Look at this shell script:

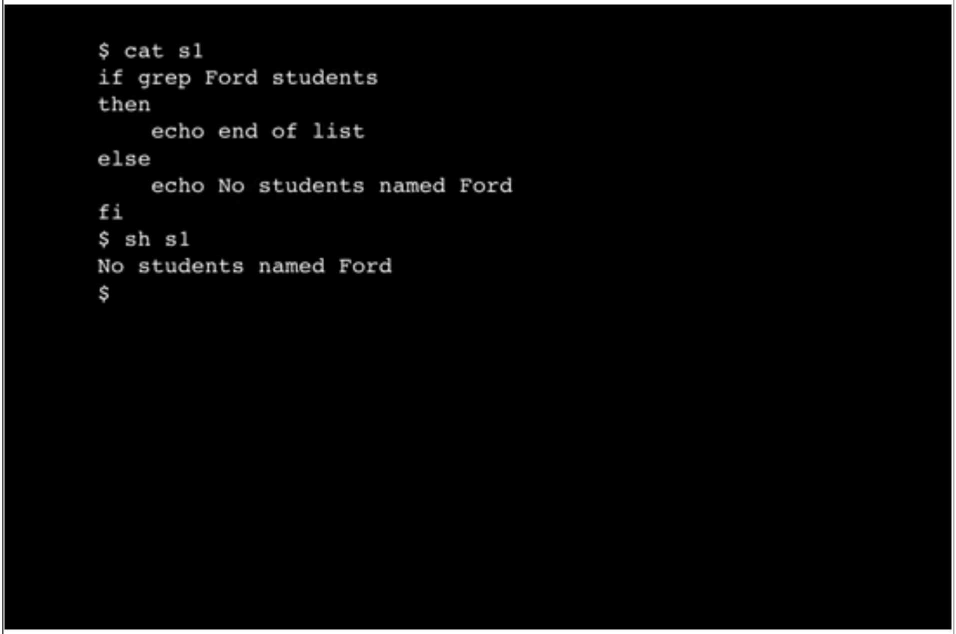
```
if grep Fred students
then
    echo end of list
else
    echo No students named Fred
fi
```

Here we have an 'if'. The 'if' keyword in *sh* is followed by a command — an arbitrarily-complex command, using any of the features of sh.
We run this shell script...
we can run it by passing it to the shell...
For one, we'll see why I said "echo end of list" above. That's because the command passed to 'if' *is* executed. If the command succeeds, we do the 'then' clause; if the command fails, we do the 'else' clause, if any.



```
$ cat s1
if grep Ford students
then
    echo end of list
else
    echo No students named Ford
fi
$ sh s1
No students named Ford
$
```

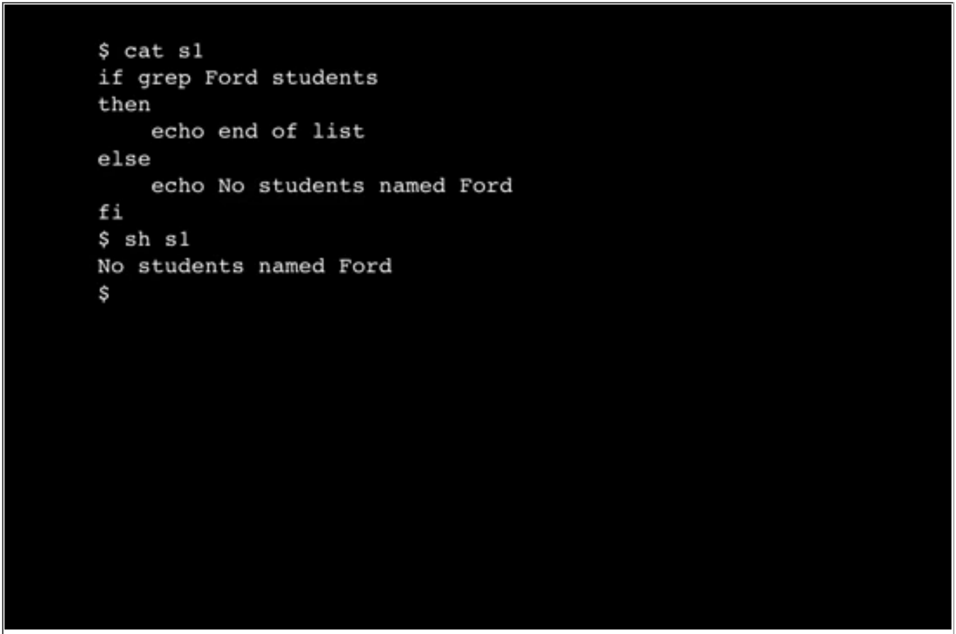
Let's watch it fail...
change "Fred" to "Ford", there, and there...
I run it... the *grep* will fail... and so we will do the 'else' clause.



```
$ cat s1
if grep Ford students
then
    echo end of list
else
    echo No students named Ford
fi
$ sh s1
No students named Ford
$
```

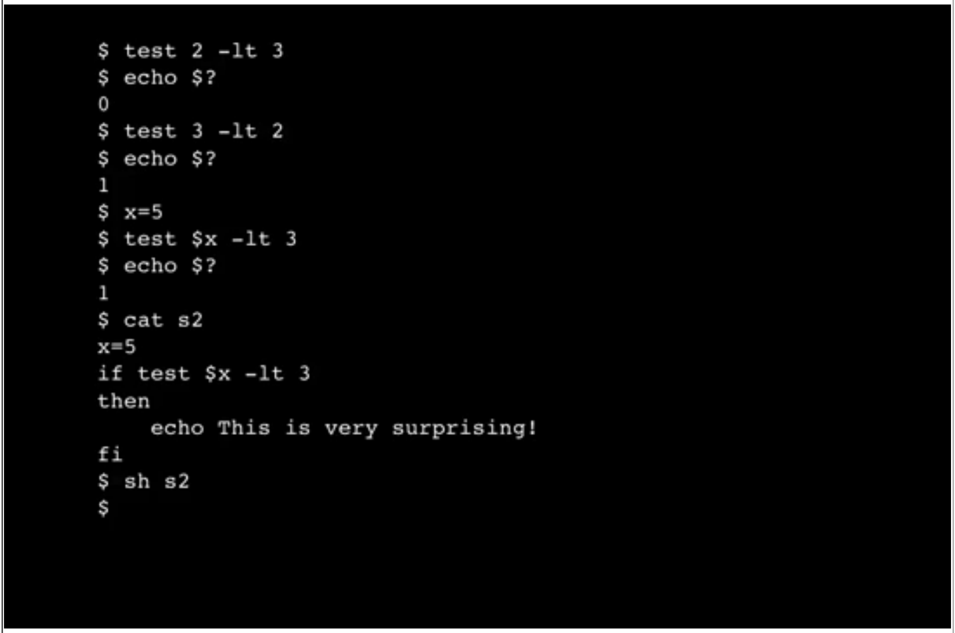
So the syntax of the 'if' statement is: we have the 'if' keyword; then we have an arbitrary *sh* statement, as I said; the next command is 'then'; and then we have one or more statements for the 'then' clause; then optionally the 'else' keyword and one or more statements for the 'else' clause. Then we have the keyword to end the 'if' construct, which is 'fi' — 'if' spelled backwards. This is from a European programming languages tradition, reversing the keyword for the 'end' keyword... it kinda grows on you.

Remember that the 'if' command executes an arbitrary other command. This is why the 'then' has to be on the next line. If we were to put the 'then' up here... there's no way for the shell to tell that that's not supposed to be an argument to *grep*. It is entirely possible to run "grep Ford students then" as a command. So to make it uniquely parseable, we have to put the 'then' on the next line, and that's the syntax of 'if' in *sh*.
You might argue that we don't need a 'then' at all; that would be sensible, but that's just not the syntax in this particular programming language.



```
$ cat s1
if grep Ford students
then
    echo end of list
else
    echo No students named Ford
fi
$ sh s1
No students named Ford
$
```

Most normal modern programming languages are "free-format", meaning that any white space counts as the same. Spaces, tabs, newlines. Python is not like that though; in Python, newlines and indentation levels are significant syntactically.
sh comes closer to being free-format than Python does, but of course we want "return" to separate commands interactively, so it also does in shell scripts. There's also a command separator which is the semi-colon.



```
$ test 2 -lt 3
$ echo $?
0
$ test 3 -lt 2
$ echo $?
1
$ x=5
$ test $x -lt 3
$ echo $?
1
$ cat s2
x=5
if test $x -lt 3
then
    echo This is very surprising!
fi
$ sh s2
$
```

Again, though, by looking at what's simplest in *sh*, we're still wondering how to do the things which are the simplest in normal programming languages. How do you write "if x is less than 3"?

There's a general testing command which exists for just this purpose. It's called "test".

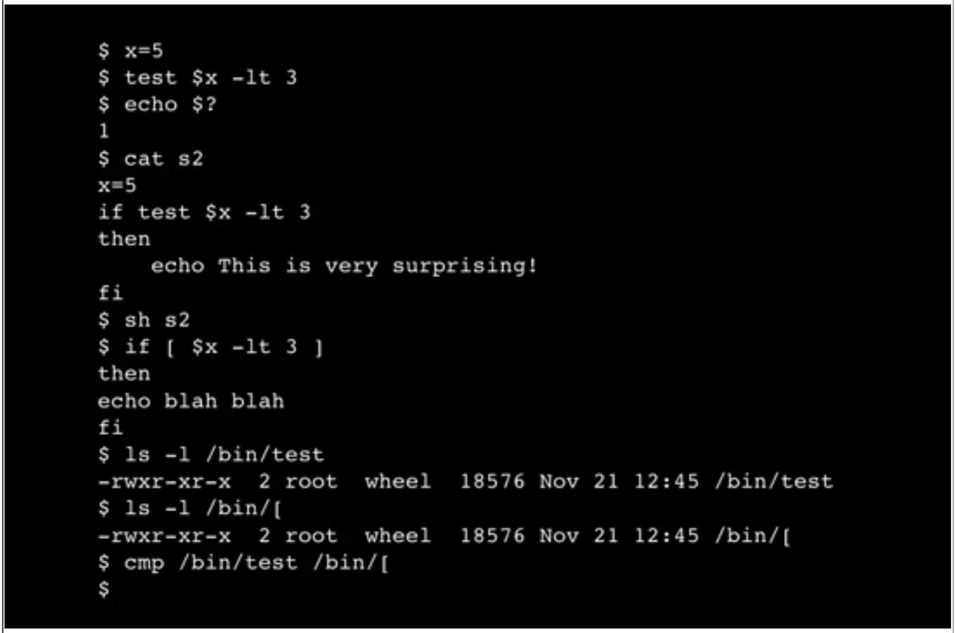
```
test 2 –lt 3
```

This is a command, running *test*; it produces no output, but it succeeds, because 2 is less than 3 — that's what that means, that's what that's testing. [types "echo $?"] We can see that it succeeded.
If we do "test 3 –lt 2", it fails. So this is useful in an 'if' statement.
But in practice, of course, one of those arguments might be a variable: if we say "x=5", then we could run "test $x –lt 3". The shell will substitute that variable x, to 5, and run "test 5 –lt 3". So that fails, because it's not the case that 5 is less than 3.

Here's how we would use it in a shell script:

```
x=5
if test $x –lt 3
then
    echo This is very surprising!
fi
```

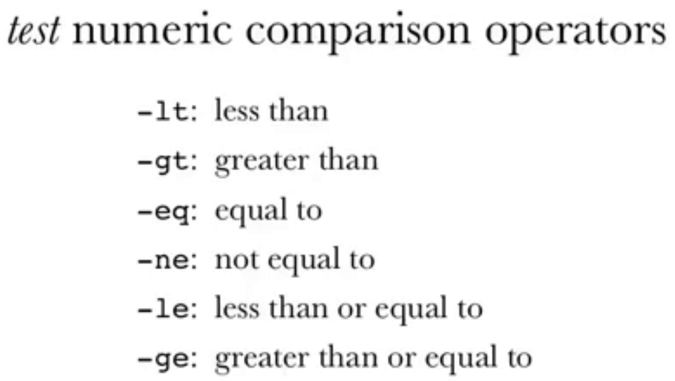
Then we can run that shell script, and the output is not surprising.
Remember that our use of '$?' farther up above is just for exploration. In real life we would put the test command in an 'if'; we wouldn't use $? for this.



```
$ x=5
$ test $x -lt 3
$ echo $?
1
$ cat s2
x=5
if test $x -lt 3
then
    echo This is very surprising!
fi
$ sh s2
$ if [ $x -lt 3 ]
then
echo blah blah
fi
$ ls -l /bin/test
-rwxr-xr-x  2 root  wheel  18576 Nov 21 12:45 /bin/test
$ ls -l /bin/[
-rwxr-xr-x  2 root  wheel  18576 Nov 21 12:45 /bin/[
$ cmp /bin/test /bin/[
$
```

Something else I should mention at this point: You may see people writing things like "if [$x -lt 3]". (And then you can write the 'then', and whatever.) This looks cute, but the way that it's implemented is a hack.
Let's look at the test command; it's in /bin in this version of unix; so this is an "ls –l" of the test command;
and there's also "/bin/[" (slash bin slash left-square-bracket). That's a file, in /bin, and that's what we're actually running, above, where we said "if left-square-bracket". 'if' is *always* followed by a command.
These ('test' and '[') are the same program, as we can tell by using the unix tool *cmp* to compare the files. No output means that they're identical.

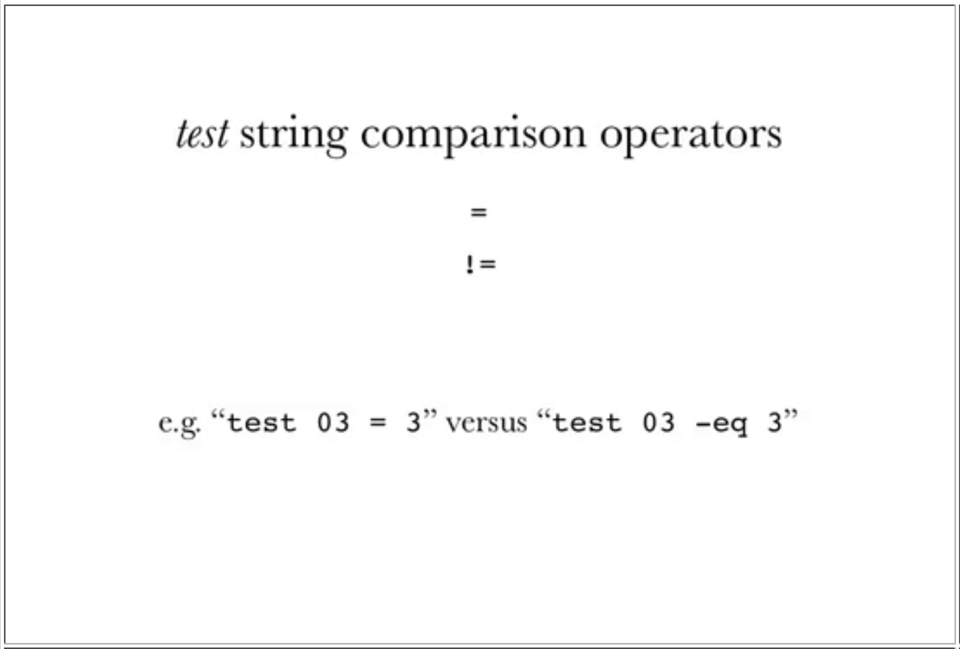
So, you can write test or you can write left-square-bracket; but it's important to understand that what 'if' really does is, what follows 'if' is a command; the command is executed and has all of the effects it has from executing; its success or failure exit status determines whether we take the 'then' or the 'else' clause in the 'if'.



test numeric comparison operators

`-lt`:	less than
`-gt`:	greater than
`-eq`:	equal to
`-ne`:	not equal to
`-le`:	less than or equal to
`-ge`:	greater than or equal to

test has a number of numeric comparison operators: –lt for less than, –gt for greater than, –eq for equal to, –ne for not equal to; possibly less-defensively, –le for less than or equal to, and –ge for greater than or equal to.
When the *test* program was devised, pretty much all programmers knew the Fortran programming language, and these are the two-letter codes from the numeric comparison operators in the Fortran programming language. These days people mostly learn these two-letter codes when they learn about the *test* command. But that's ok.



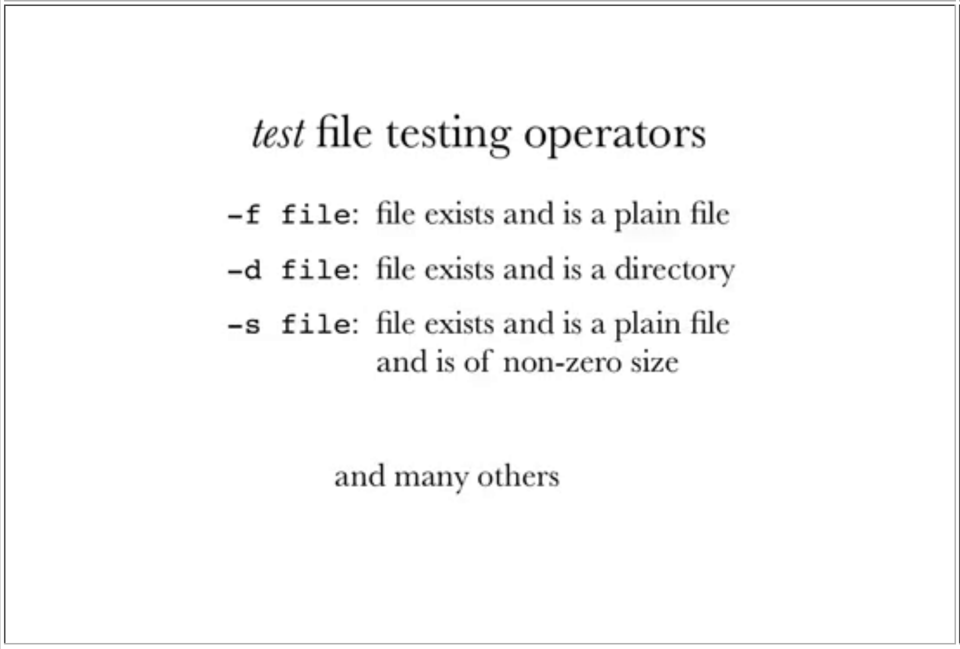
test string comparison operators

`=`

`!=`

e.g. “`test 03 = 3`” versus “`test 03 -eq 3`”

test also does string comparisons. The basic operators are "=" (equals) and "!=" (not equals). Now, everything in sh *is* a string, but this still makes a difference. If you compare something like, for example, "03" and "3", if you use the string comparison they're unequal, if you use the numeric comparison they're equal.



test file testing operators

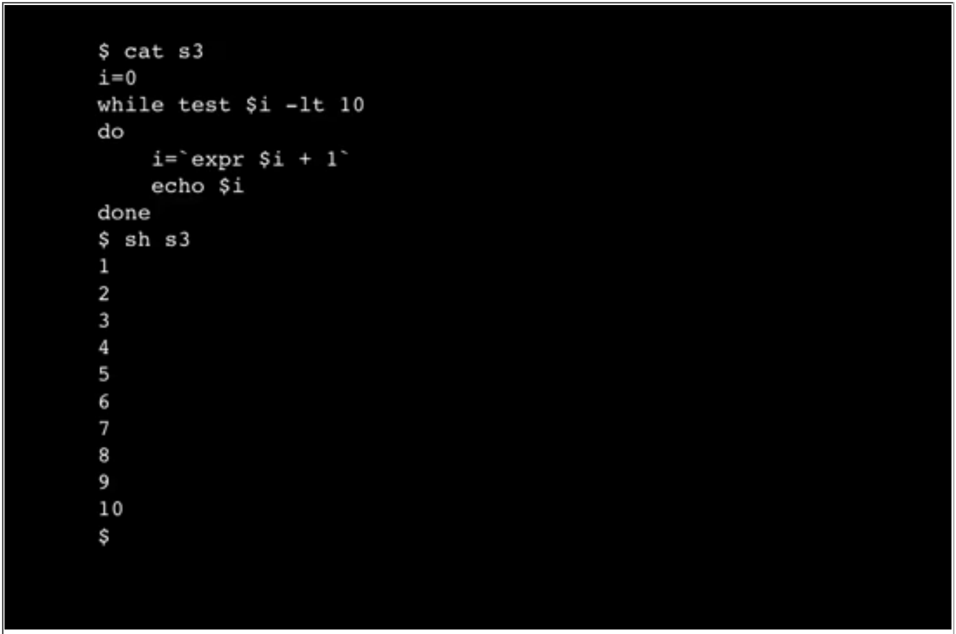
`-f file`: file exists and is a plain file

`-d file`: file exists and is a directory

`-s file`: file exists and is a plain file and is of non-zero size

and many others

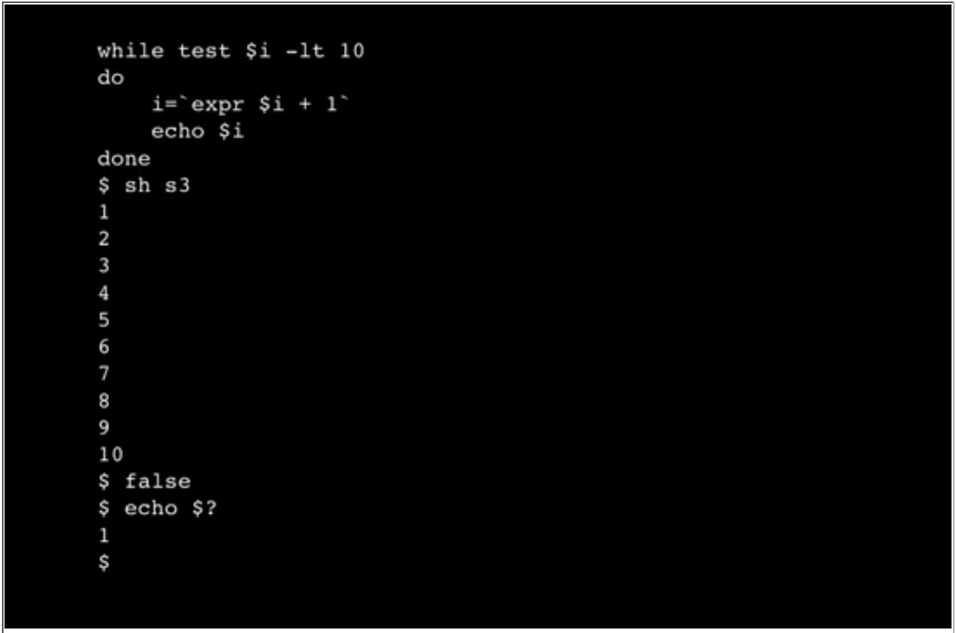
test also has file-testing operators. –f and a file name says that the file exists and is a plain file. So if the file does not exist by that name, or if it exists but is not a plain file, then this would fail.
–d space file: file exists and is a directory;
and here's one that turns out to be useful from time to time:
–s: file exists **and** is a plain file **and** is of non-zero size.
And there are many other file-testing operators; and other things such as boolean operators; all this is in "`man test`".



```
$ cat s3
i=0
while test $i -lt 10
do
    i=`expr $i + 1`
    echo $i
done
$ sh s3
1
2
3
4
5
6
7
8
9
10
$
```

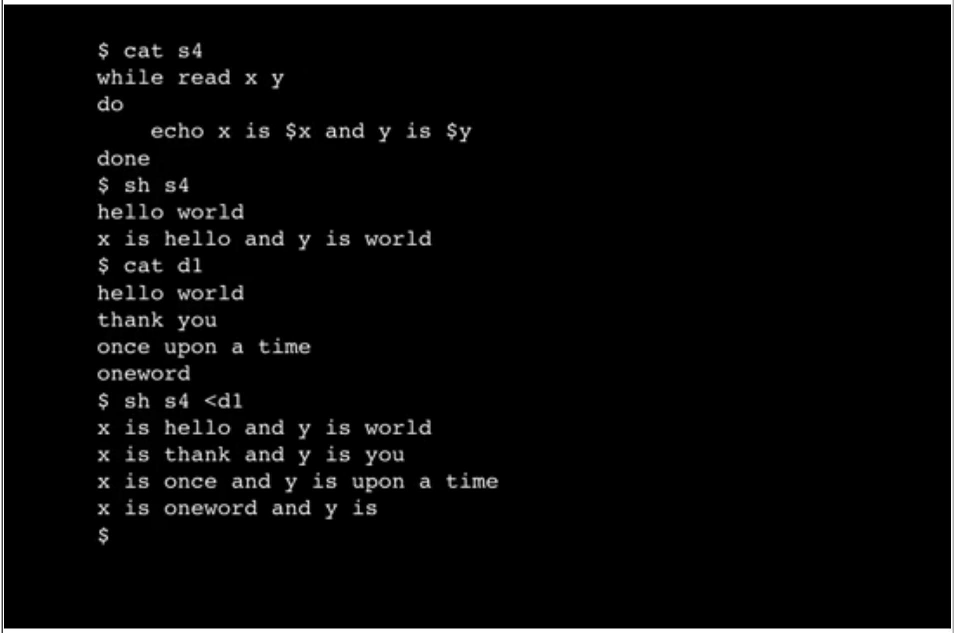
Ok, so that's 'if'. And, the general idea of using a command's exit status as a boolean value.
We also use this command exit status concept for 'while'. The 'while' keyword is also followed by an arbitrary command.
i=0
while test something
That can be any command; it doesn't have to be *test*. While i is less than 10, we increment i and output it.

```
i=0
while test $i -lt 10
do
    i=`expr $i + 1`
    echo $i
done
```



```
while test $i -lt 10
do
    i=`expr $i + 1`
    echo $i
done
$ sh s3
1
2
3
4
5
6
7
8
9
10
$ false
$ echo $?
1
$
```

Once we have these commands as booleans we might need the equivalent of boolean constants: There is a command "true" which is just exit 0; and "false" which is just exit 1.



```
$ cat s4
while read x y
do
    echo x is $x and y is $y
done
$ sh s4
hello world
x is hello and y is world
$ cat d1
hello world
thank you
once upon a time
oneword
$ sh s4 <d1
x is hello and y is world
x is thank and y is you
x is once and y is upon a time
x is oneword and y is
$
```

In the previous video we saw the command "read". read can be used in a 'while' because it fails on end-of-file.

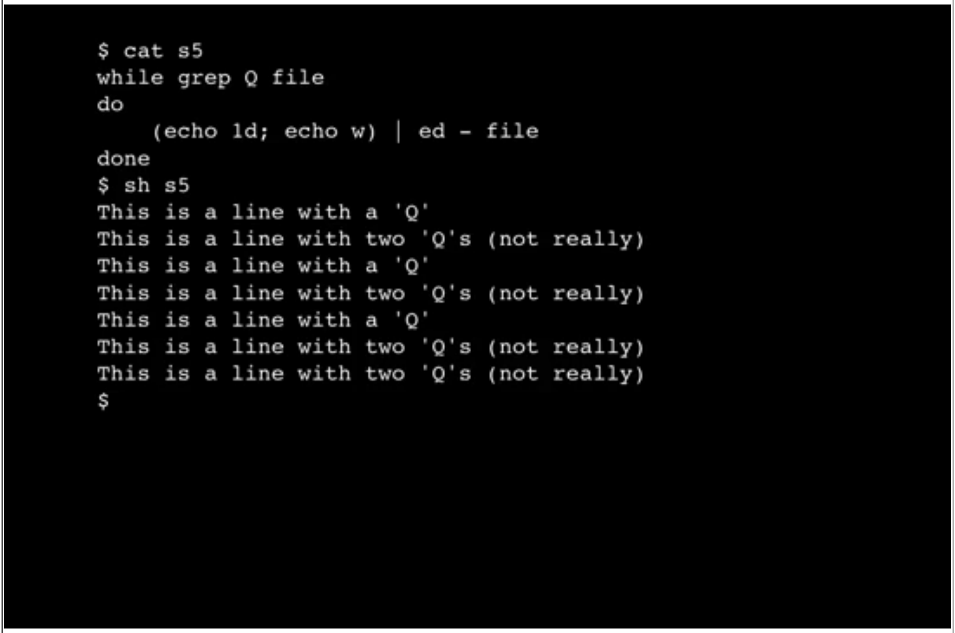
```
while read x y
do
    echo x is $x and y is $y
done
```

"while read x y" will do that read command, and if it succeeds, we do the body of the loop and continue; if we hit end-of-file, it fails, and the loop is done.

So I can type "hello world", return; x is hello and y is world.

I'm going to signal end-of-file from the terminal — there is a special control character for that, control-D by default:

Like all processing until end-of-file, this makes more sense non-interactively: Here's a data file; I can run s4 with that data file as input...



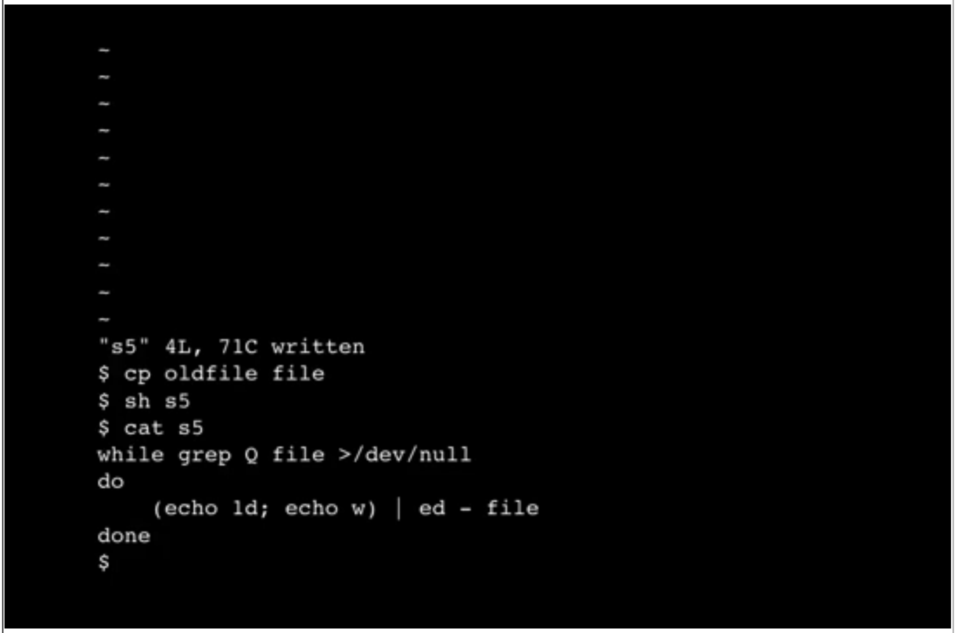
```
$ cat s5
while grep Q file
do
    (echo 1d; echo w) | ed - file
done
$ sh s5
This is a line with a 'Q'
This is a line with two 'Q's (not really)
This is a line with a 'Q'
This is a line with two 'Q's (not really)
This is a line with a 'Q'
This is a line with two 'Q's (not really)
This is a line with two 'Q's (not really)
$
```

Here's a loop which keeps using the editor "ed" to remove the first line of a file so long as the file has a capital Q in it.

```
while grep Q file
do
    (echo 1d; echo w) | ed - file
done
```

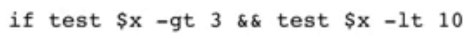
"while grep Q file", so as long as there is a Q in file, then we do this compound statement, which formulates input to *ed* using two shell commands, echo 1d, which means delete the first line, echo w, which means write the file. We don't need a quit command becaue when *ed* gets end-of-file on the standard input, it will exit.
The extra argument to *ed*, the minus sign, tells it to suppress some of the interactive stuff that it would normally output, like prompts.

This script will work, but will produce some messy output. All of that is the output from the 'grep' command, being executed multiple times around the loop.



```
~
~
~
~
~
~
~
~
~
~
~
"s5" 4L, 71C written
$ cp oldfile file
$ sh s5
$ cat s5
while grep Q file >/dev/null
do
    (echo 1d; echo w) | ed - file
done
$
```

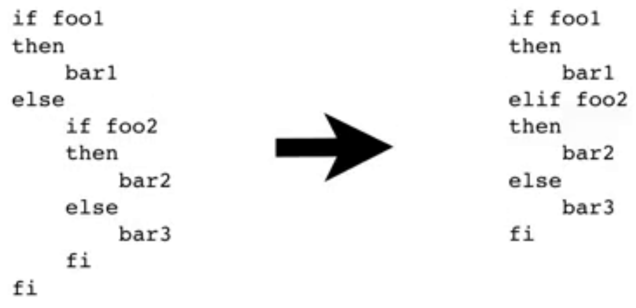
But we really wanted to use grep only for its exit status, not to *see* the lines with the Qs, but just to check whether there is a Q in the file.
Let's modify this.
We could throw away the 'grep' output by redirecting it to some file name we don't care about.
But better yet, we will use the special file "/dev/null".
This is a device file with the simplest possible driver: the driver just says "return 0". It does nothing. Therefore, the data is lost. Which is what we want here.
To run this, I'll restore the file from before we edited it and removed all the 'Q's...
Run this...
The unwanted output is discarded.
That file again...



```
if test $x -gt 3 && test $x -lt 10
```

With all these conditions, we might want one more feature: the double vertical bar and the double ampersand operators, which combine boolean statuses just like in C or Java (or like the "or" and "and" keywords in Python). They also have the "short-circuit" behaviour where the right operand is only evaluated if necessary. Which in *sh* means a command which is only *executed* if necessary.

For example, we can write something like "`if test $x -gt 3 && test $x -lt 10`".
So, this is an 'if'... as I said the command to an 'if' can be an arbitrary *sh* statement, as complex as we like... in this case it's a compound statement of two statements.
"`test $x -gt 3`" is executed. If this fails, then the right command is not executed, and the exit status of the compound command is that exit status of the left command.
If, on the other hand, the left command succeeds, then the right command is also executed, and the exit status of the compound command is the exit status of the right command.



```
if foo1
then
    bar1
else
    if foo2
    then
        bar2
    else
        bar3
    fi
fi
```

```
if foo1
then
    bar1
elif foo2
then
    bar2
else
    bar3
fi
```

Something else about the 'if' statement. Suppose we have an if-then-else statement like this.

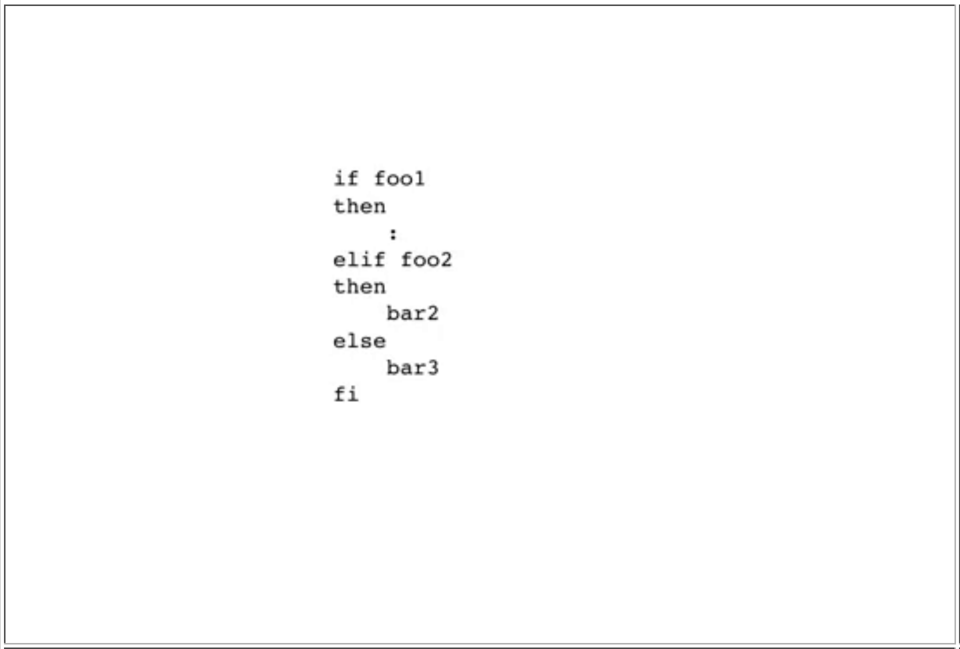
```
if foo1
then
    bar1
else
    if foo2
    then
        bar2
    else
        bar3
    fi
fi
```

This kind of cascading if-then-else is pretty common, but if we write it like this it's pretty messy. And if we had more conditions, it would be worse. Now, since *sh* is free-format we *could* write it like this:

```
if foo1
then
    bar1
else
if foo2
then
    bar2
else
    bar3
fi
fi
```

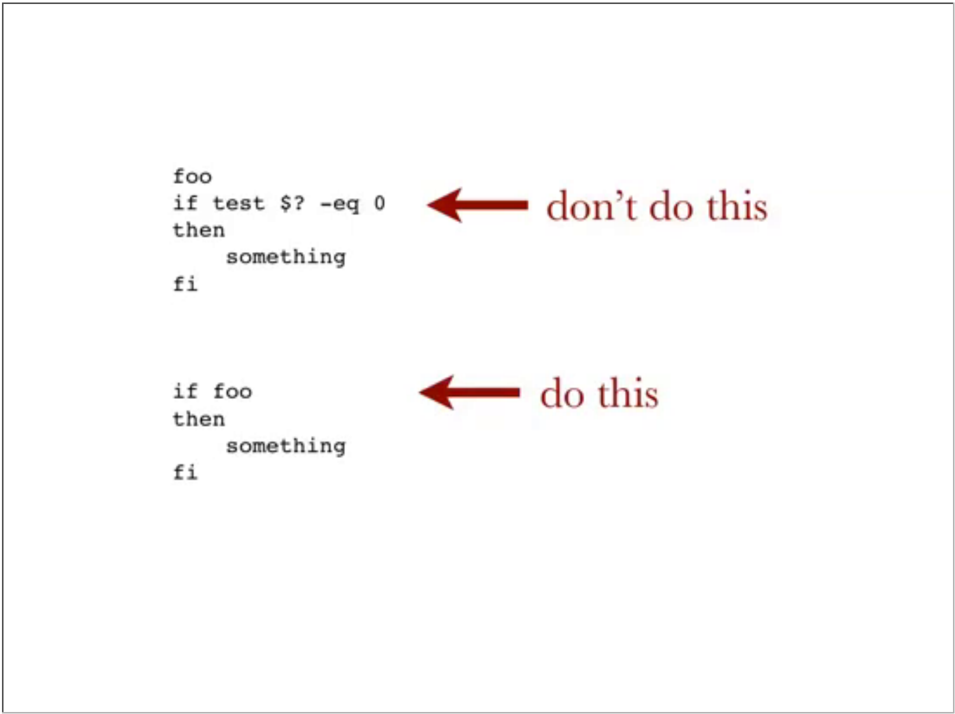
That helps a little to show the structure... The weirdest thing about this is that we will have an increasing number of 'fi's at the end all in a row as it gets more complicated. This also gets error-prone.

To deal with this there is a special combined else-if keyword, that looks like this. The difference is that this combined else-if keyword does not introduce an additional nesting level, so we have only one 'fi' on the end, and the one on the right is correctly indented.



```
if foo1
then
    :
elif foo2
then
    bar2
else
    bar3
fi
```

Suppose we didn't have anything to do in the 'then' case. We wouldn't usually write things like this, but every once in a while it's easier or clearer to write it this way than to negate the condition, so we end up with an empty 'then' clause. Written like this [with an empty 'then' clause] this is not syntactically valid in *sh* because the 'then' clause is *one or more* statements, not the usual "zero or more" statements.
So for a situation like this we need the *sh* null statement, the statement which doesn't do anything. Which in sh is just a colon. This is similar to the "pass" statement in Python, or just a semicolon in C or Java.



```
foo
if test $? -eq 0
then
    something
fi
```

```
if foo
then
    something
fi
```

One final reminder to leave you with: Don't do this:

```
foo
if test $? -eq 0
then
    ...
fi
```

Don't run a command, then use its exit status as '$?' in *test* in an 'if'. This is what 'if' *does*! 'if' checks whether a command succeeds or fails.

So do this:

```
if foo
then
    ...
fi
```

Just supply the command directly to an 'if', if what you're trying to do is to check whether it succeeds or fails.

The third video in this series is about some other control constructs in *sh*. But it begins with a detailed discussion of quoting in *sh*.