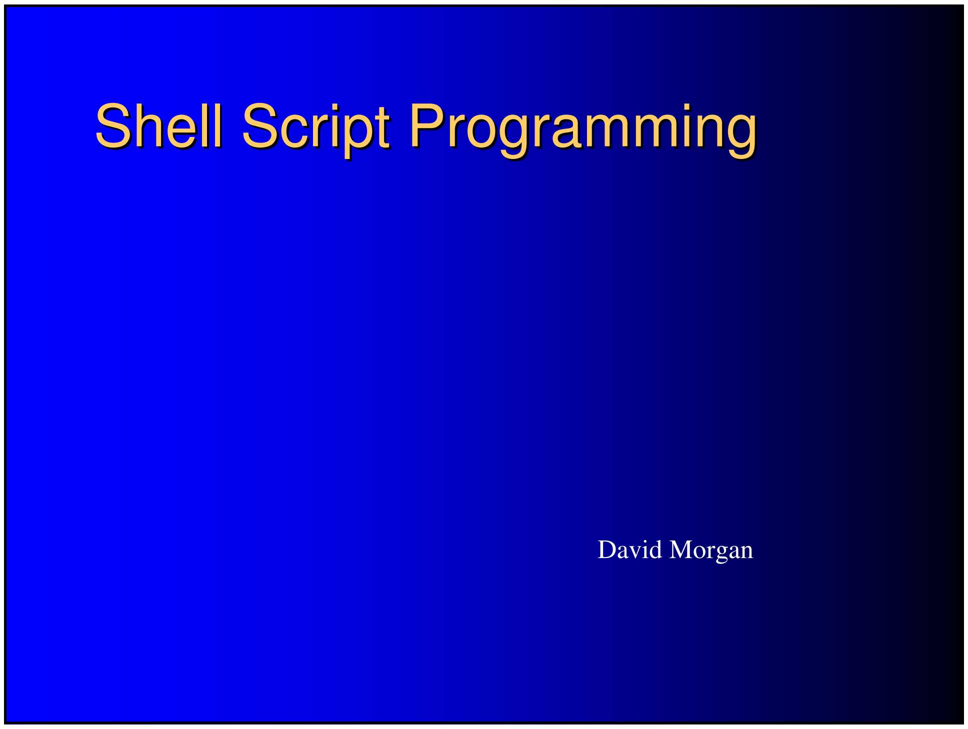
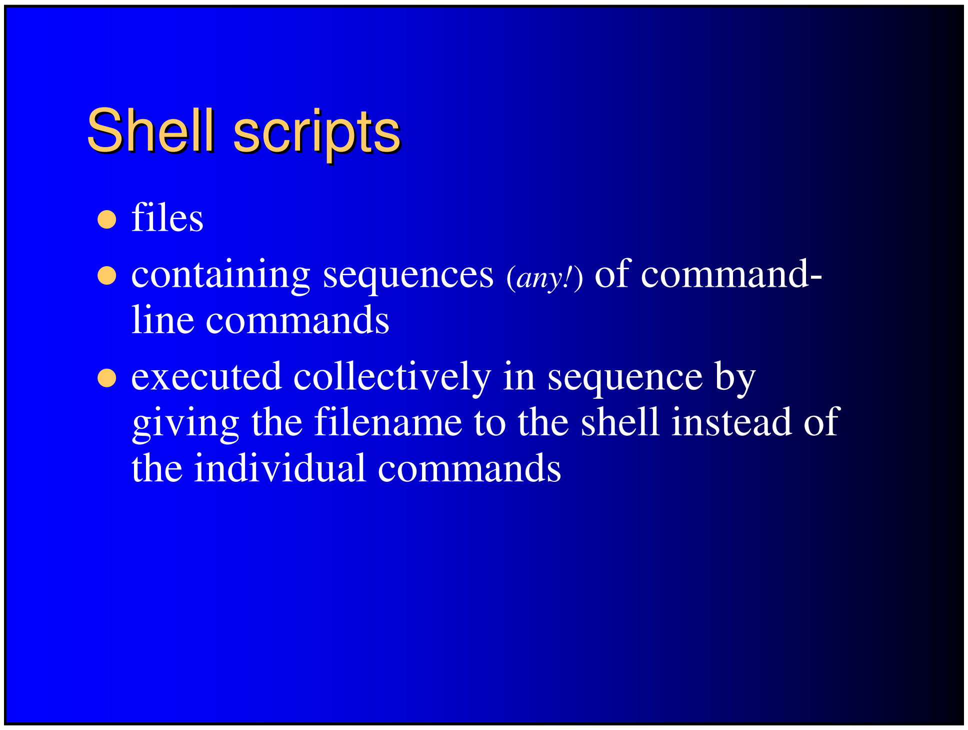


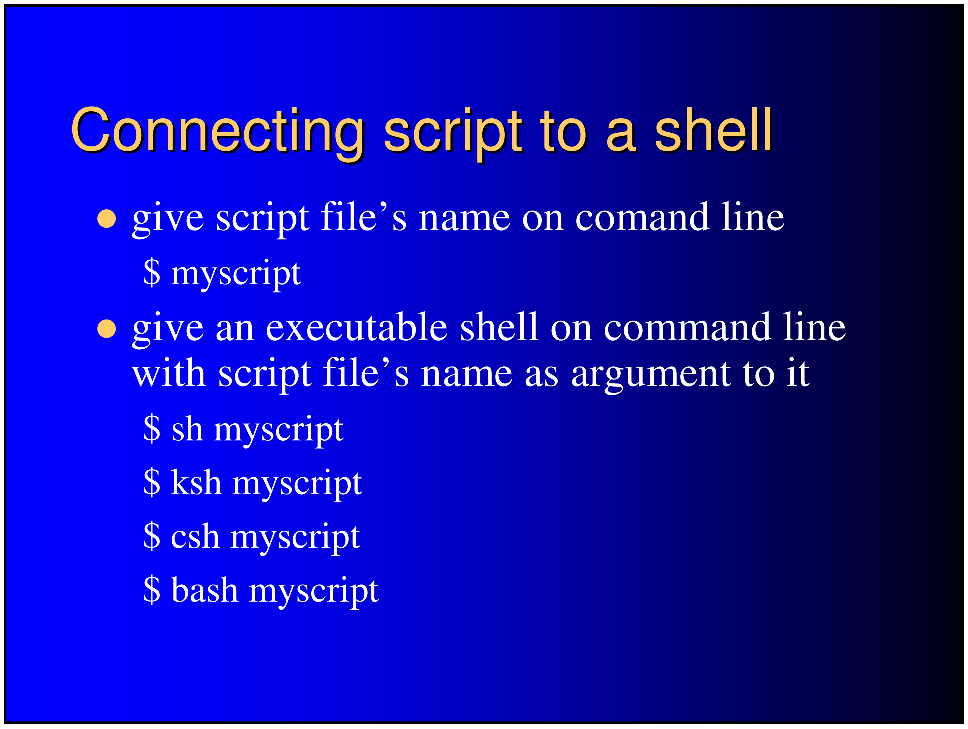
Shell Script Programming

David Morgan



Shell scripts

- files
- containing sequences *(any!)* of command-line commands
- executed collectively in sequence by giving the filename to the shell instead of the individual commands



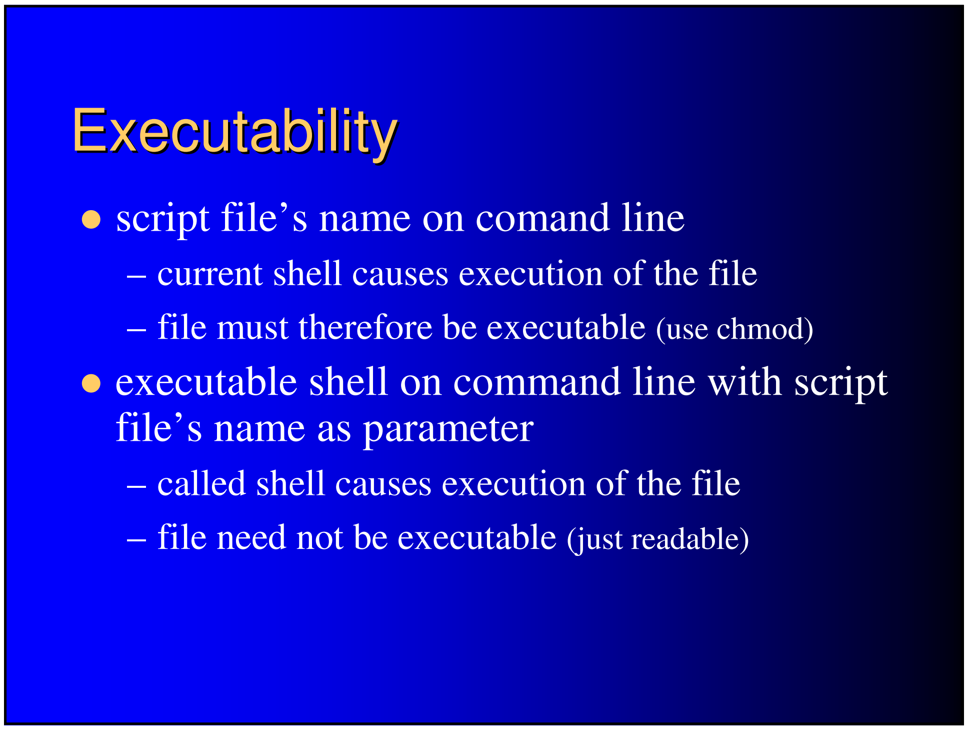
Connecting script to a shell

- give script file’s name on comand line

  ```
  $ myscript
  ```

- give an executable shell on command line with script file’s name as argument to it

  ```
  $ sh myscript
  $ ksh myscript
  $ csh myscript
  $ bash myscript
  ```



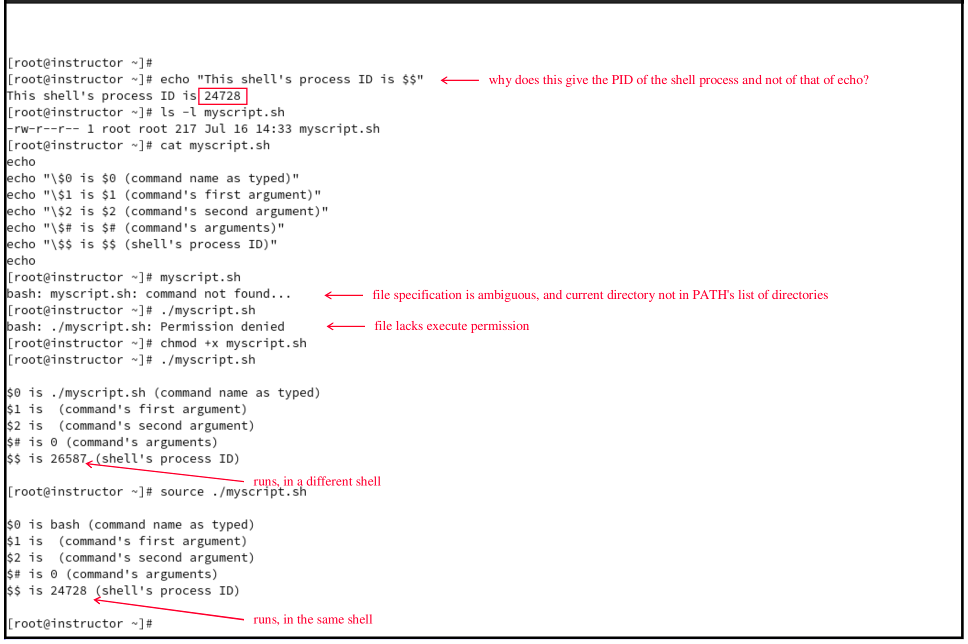
Executability

- script file’s name on comand line
  - current shell causes execution of the file
  - file must therefore be executable (use chmod)
- executable shell on command line with script file’s name as parameter
  - called shell causes execution of the file
  - file need not be executable (just readable)

Executing in the current shell
using the "source" builtin

- "."or "source" on command line followed by script's name
  - current shell executes the file (*not* a child shell)
- commands are executed in current shell
  - e.g., variables created in current shell's space (so, persist)
  - e.g., exit collapses current shell (instead of returning to it from another one-- there is no other one)

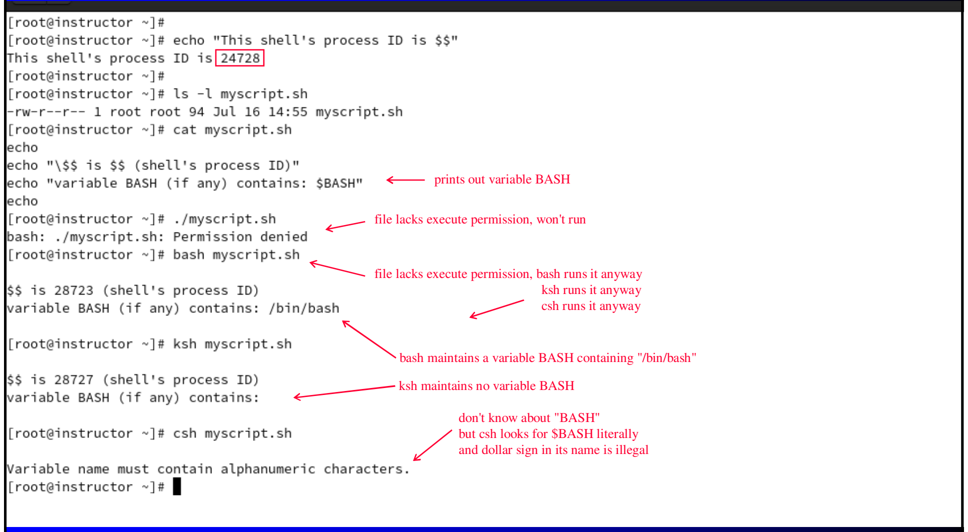
Getting a script to run



```
[root@instructor ~]#
[root@instructor ~]# echo "This shell's process ID is $$"
This shell's process ID is 24728
[root@instructor ~]# ls -l myscript.sh
-rw-r--r-- 1 root root 217 Jul 16 14:33 myscript.sh
[root@instructor ~]# cat myscript.sh
echo
echo "\$0 is $0 (command name as typed)"
echo "\$1 is $1 (command's first argument)"
echo "\$2 is $2 (command's second argument)"
echo "\$# is $# (command's arguments)"
echo "\$$ is $$ (shell's process ID)"
echo
[root@instructor ~]# myscript.sh
bash: myscript.sh: command not found...
[root@instructor ~]# ./myscript.sh
bash: ./myscript.sh: Permission denied
[root@instructor ~]# chmod +x myscript.sh
[root@instructor ~]# ./myscript.sh

$0 is ./myscript.sh (command name as typed)
$1 is  (command's first argument)
$2 is  (command's second argument)
$# is 0 (command's arguments)
$$ is 26587 (shell's process ID)

[root@instructor ~]# source ./myscript.sh

$0 is bash (command name as typed)
$1 is  (command's first argument)
$2 is  (command's second argument)
$# is 0 (command's arguments)
$$ is 24728 (shell's process ID)

[root@instructor ~]#
```

why does this give the PID of the shell process and not of that of echo?
file specification is ambiguous, and current directory not in PATH's list of directories
file lacks execute permission
runs, in a different shell
runs, in the same shell

Getting another shell to run your script

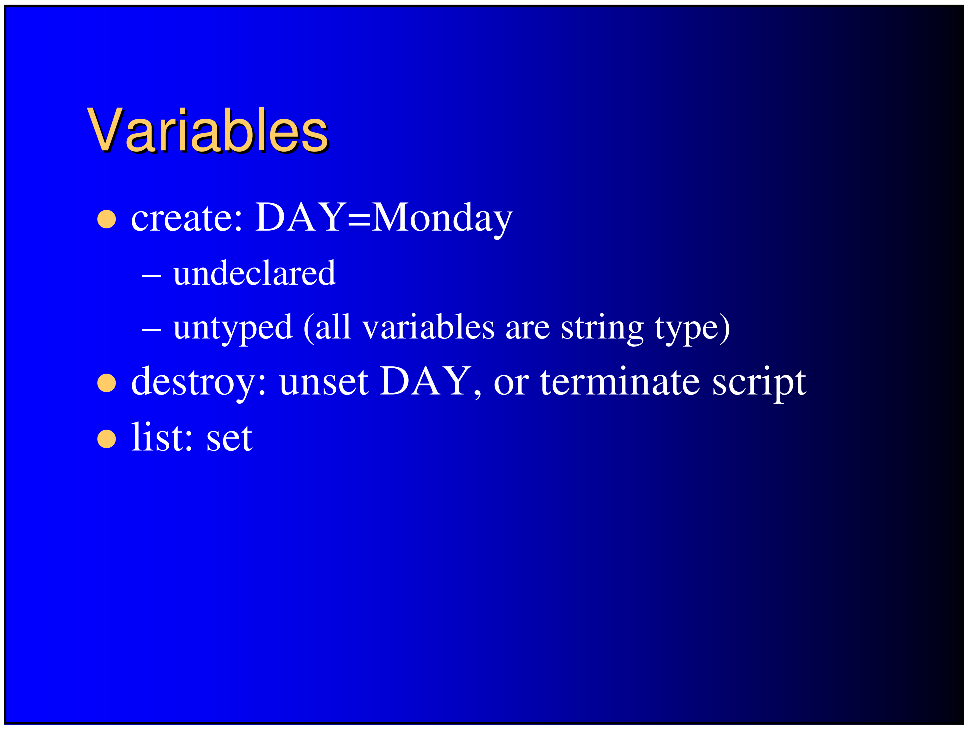


```
[root@instructor ~]#
[root@instructor ~]# echo "This shell's process ID is $$"
This shell's process ID is 24728
[root@instructor ~]#
[root@instructor ~]# ls -l myscript.sh
-rw-r--r-- 1 root root 94 Jul 16 14:55 myscript.sh
[root@instructor ~]# cat myscript.sh
echo
echo "\$$ is $$ (shell's process ID)"
echo "variable BASH (if any) contains: $BASH"
echo
[root@instructor ~]# ./myscript.sh
bash: ./myscript.sh: Permission denied
[root@instructor ~]# bash myscript.sh

$$ is 28723 (shell's process ID)
variable BASH (if any) contains: /bin/bash

[root@instructor ~]# ksh myscript.sh

$$ is 28727 (shell's process ID)
variable BASH (if any) contains:

[root@instructor ~]# csh myscript.sh

Variable name must contain alphanumeric characters.
[root@instructor ~]# █
```

prints out variable BASH
file lacks execute permission, won't run
file lacks execute permission, bash runs it anyway
ksh runs it anyway
csh runs it anyway
bash maintains a variable BASH containing "/bin/bash"
ksh maintains no variable BASH
don't know about "BASH"
but csh looks for $BASH literally
and dollar sign in its name is illegal

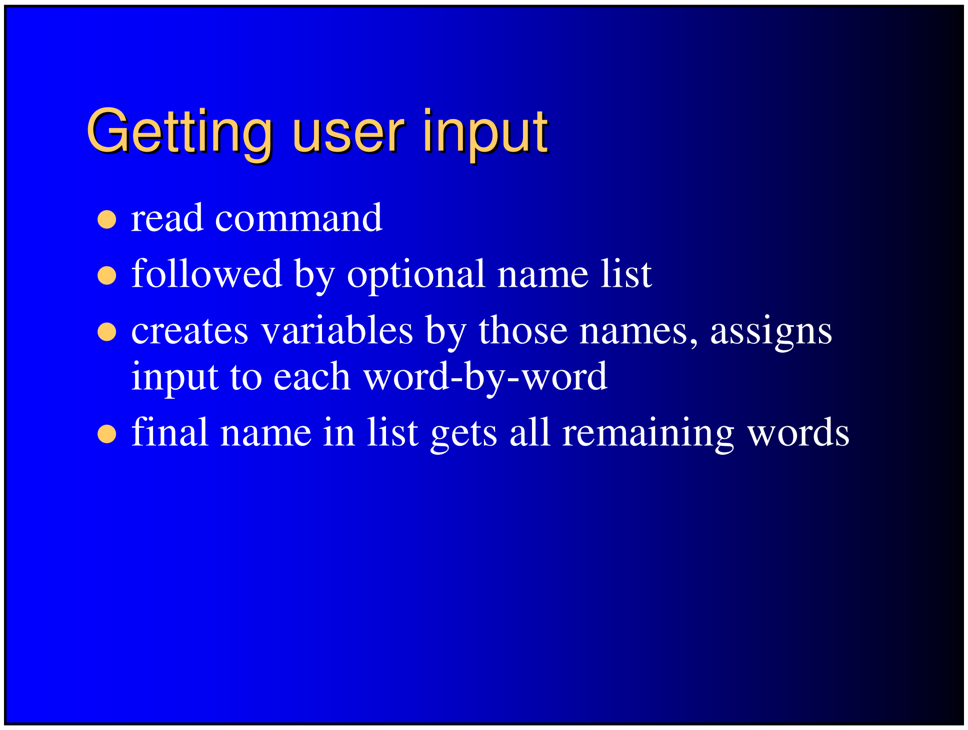
Parameters

- variables (named parameters)
- positional parameters
  - $1, $2, etc – command line arguments
- special parameters
  - $0 - command line script name
  - $# - number of positional parameters
  - $* - positional parameters collectively
  - $$ - process ID (PID) of the shell (from which executed)
  - $? – exit status of most recent command



Variables

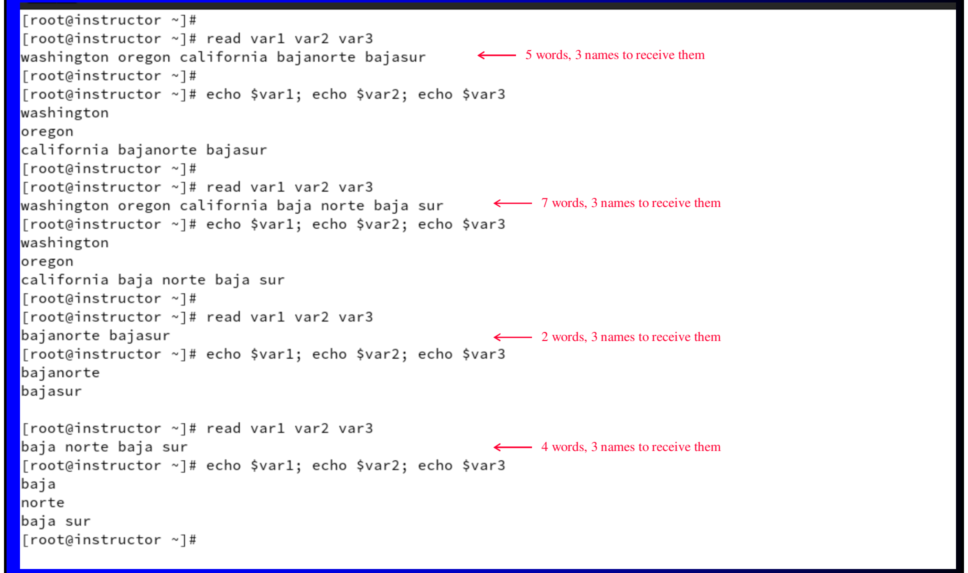
- create: DAY=Monday
  - undeclared
  - untyped (all variables are string type)
- destroy: unset DAY, or terminate script
- list: set



Getting user input

- read command
- followed by optional name list
- creates variables by those names, assigns input to each word-by-word
- final name in list gets all remaining words

How read distributes words to names



```
[root@instructor ~]#
[root@instructor ~]# read var1 var2 var3
washington oregon california bajanorte bajasur      ← 5 words, 3 names to receive them
[root@instructor ~]#
[root@instructor ~]# echo $var1; echo $var2; echo $var3
washington
oregon
california bajanorte bajasur
[root@instructor ~]#
[root@instructor ~]# read var1 var2 var3
washington oregon california baja norte baja sur      ← 7 words, 3 names to receive them
[root@instructor ~]# echo $var1; echo $var2; echo $var3
washington
oregon
california baja norte baja sur
[root@instructor ~]#
[root@instructor ~]# read var1 var2 var3
bajanorte bajasur                                     ← 2 words, 3 names to receive them
[root@instructor ~]# echo $var1; echo $var2; echo $var3
bajanorte
bajasur

[root@instructor ~]# read var1 var2 var3
baja norte baja sur                                   ← 4 words, 3 names to receive them
[root@instructor ~]# echo $var1; echo $var2; echo $var3
baja
norte
baja sur
[root@instructor ~]#
```

Conditions: what are they syntactically?

- conditions are lists

- a list - one or more pipelines

  pipeline1; pipeline2; pipeline3

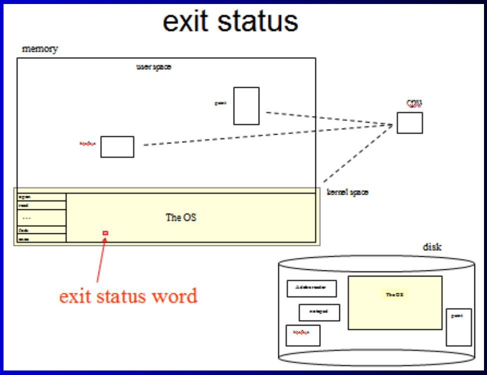
- a pipeline - one or more commands

  command1 | command 2 | command3

Observations:
a pipeline can be a single command
a list can be a single pipeline
a list can therefore also be a single command, and a single command is a list
technique: use semi-colons to string multiple commands on a single line

Conditions: what are they physically?

- values of an in-kernel storage word
- available values 0-255



Conditions: what uses/reads them?

- some commands that branch
  - if
- some commands that loop
  - while
  - until

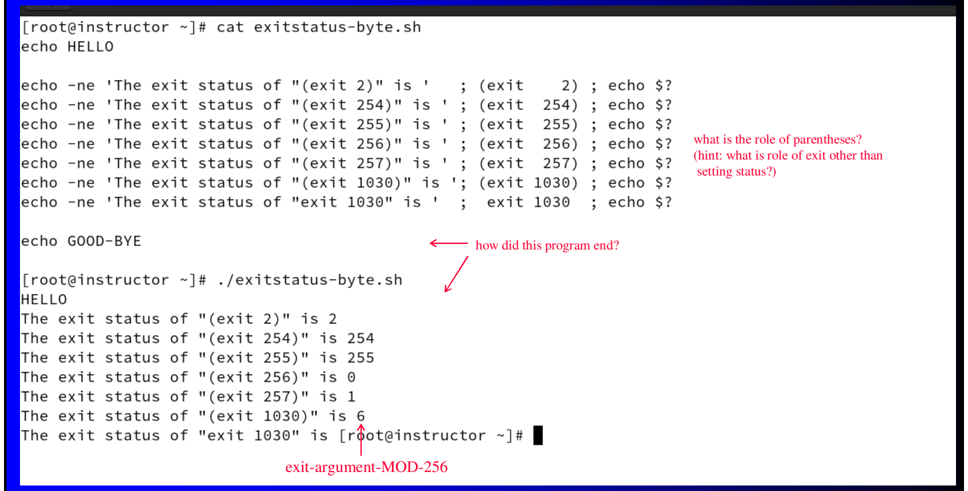
```
general:        less general:

if list1        if command
then            then
  list2           commands
fi              fi


while list1     while command
do              do
  list2           commands
done            done
```

- you, with $? special parameter

Conditions: what sets/writes them?

- the exit( ) system function
- used by
  - every command
  - the shell "exit" builtin (which in turn calls the system function)

exit status: what is its value range?

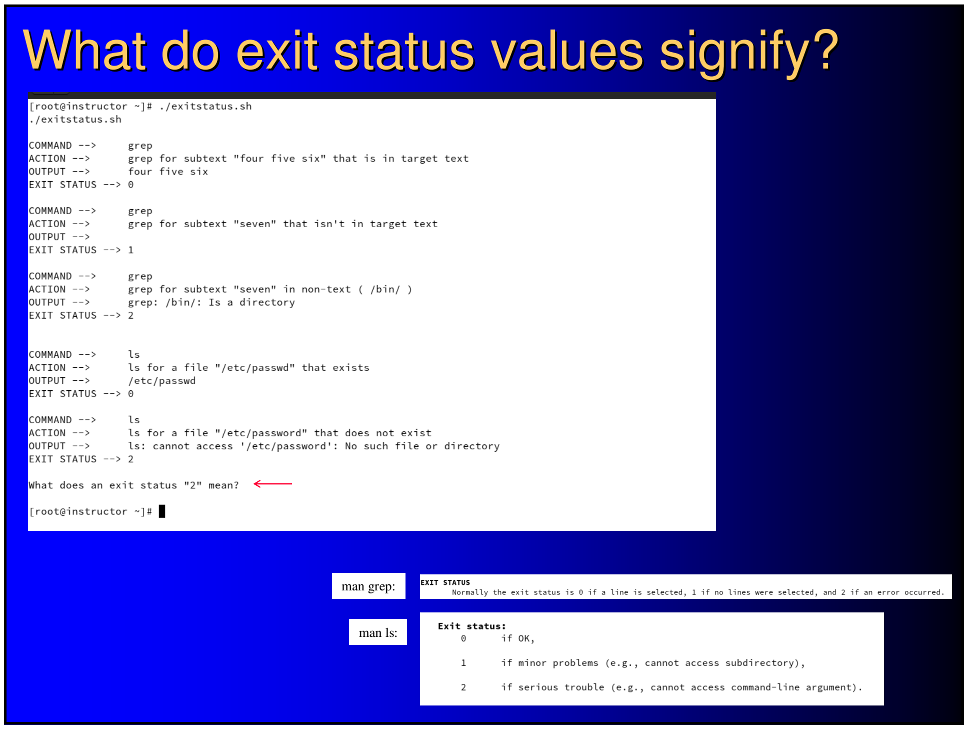


```
[root@instructor ~]# cat exitstatus-byte.sh
echo HELLO

echo -ne 'The exit status of "(exit 2)" is '   ; (exit    2) ; echo $?
echo -ne 'The exit status of "(exit 254)" is ' ; (exit  254) ; echo $?
echo -ne 'The exit status of "(exit 255)" is ' ; (exit  255) ; echo $?
echo -ne 'The exit status of "(exit 256)" is ' ; (exit  256) ; echo $?
echo -ne 'The exit status of "(exit 257)" is ' ; (exit  257) ; echo $?
echo -ne 'The exit status of "(exit 1030)" is '; (exit 1030) ; echo $?
echo -ne 'The exit status of "exit 1030" is '  ;  exit 1030  ; echo $?

echo GOOD-BYE

[root@instructor ~]# ./exitstatus-byte.sh
HELLO
The exit status of "(exit 2)" is 2
The exit status of "(exit 254)" is 254
The exit status of "(exit 255)" is 255
The exit status of "(exit 256)" is 0
The exit status of "(exit 257)" is 1
The exit status of "(exit 1030)" is 6
The exit status of "exit 1030" is [root@instructor ~]# █
```

what is the role of parentheses?
(hint: what is role of exit other than setting status?)

how did this program end?

exit-argument-MOD-256



What do exit status values signify?

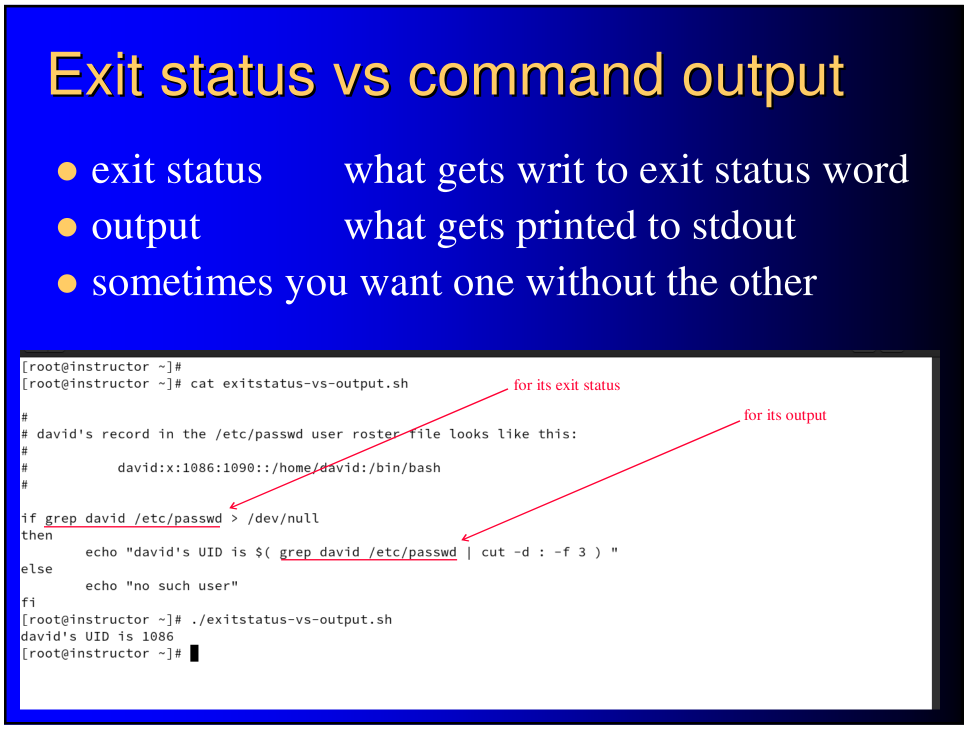
```
[root@instructor ~]# ./exitstatus.sh
./exitstatus.sh

COMMAND -->     grep
ACTION -->      grep for subtext "four five six" that is in target text
OUTPUT -->      four five six
EXIT STATUS --> 0

COMMAND -->     grep
ACTION -->      grep for subtext "seven" that isn't in target text
OUTPUT -->
EXIT STATUS --> 1

COMMAND -->     grep
ACTION -->      grep for subtext "seven" in non-text ( /bin/ )
OUTPUT -->      grep: /bin/: Is a directory
EXIT STATUS --> 2


COMMAND -->     ls
ACTION -->      ls for a file "/etc/passwd" that exists
OUTPUT -->      /etc/passwd
EXIT STATUS --> 0

COMMAND -->     ls
ACTION -->      ls for a file "/etc/password" that does not exist
OUTPUT -->      ls: cannot access '/etc/password': No such file or directory
EXIT STATUS --> 2

What does an exit status "2" mean?  ←

[root@instructor ~]# █
```

man grep:

```
EXIT STATUS
       Normally the exit status is 0 if a line is selected, 1 if no lines were selected, and 2 if an error occurred.
```

man ls:

```
   Exit status:
       0      if OK,

       1      if minor problems (e.g., cannot access subdirectory),

       2      if serious trouble (e.g., cannot access command-line argument).
```



Exit status vs command output

- exit status   what gets writ to exit status word
- output   what gets printed to stdout
- sometimes you want one without the other

```
[root@instructor ~]#
[root@instructor ~]# cat exitstatus-vs-output.sh

#
# david's record in the /etc/passwd user roster file looks like this:
#
#            david:x:1086:1090::/home/david:/bin/bash
#

if grep david /etc/passwd > /dev/null
then
        echo "david's UID is $( grep david /etc/passwd | cut -d : -f 3 ) "
else
        echo "no such user"
fi
[root@instructor ~]# ./exitstatus-vs-output.sh
david's UID is 1086
[root@instructor ~]# █
```

for its exit status

for its output

if – conditional execution

```
if condition *
then
        commands
fi
```

*remember, conditions are lists and lists are made of commands

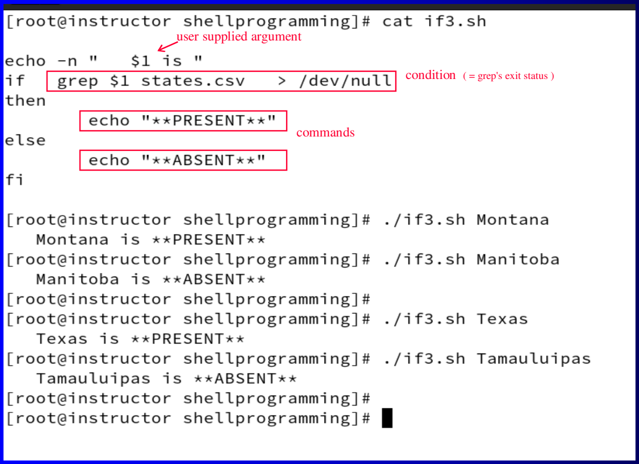
man bash:

```
if list; then list; [ elif list; then list; ] ... [ else list; ] fi
       The  if  list  is  executed.  If its exit status is zero, the then list is exe-
       cuted.  Otherwise, ...
```

if WHAT??

the exit status is the condition
the conditions of if's are commands' exit statuses

if – conditional execution



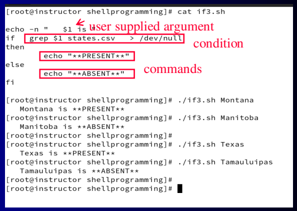
```
[root@instructor shellprogramming]# cat if3.sh
                                      user supplied argument
echo -n "   $1 is "
if   grep $1 states.csv    > /dev/null    condition  ( = grep's exit status )
then
       echo "**PRESENT**"
                                           commands
else
       echo "**ABSENT**"
fi

[root@instructor shellprogramming]# ./if3.sh Montana
   Montana is **PRESENT**
[root@instructor shellprogramming]# ./if3.sh Manitoba
   Manitoba is **ABSENT**
[root@instructor shellprogramming]#
[root@instructor shellprogramming]# ./if3.sh Texas
   Texas is **PRESENT**
[root@instructor shellprogramming]# ./if3.sh Tamauluipas
   Tamauluipas is **ABSENT**
[root@instructor shellprogramming]#
[root@instructor shellprogramming]#
```

man grep:

```
EXIT STATUS
       Normally the exit status is 0 if a line is selected, 1 if no lines were selected, and 2 if an error occurred.
```

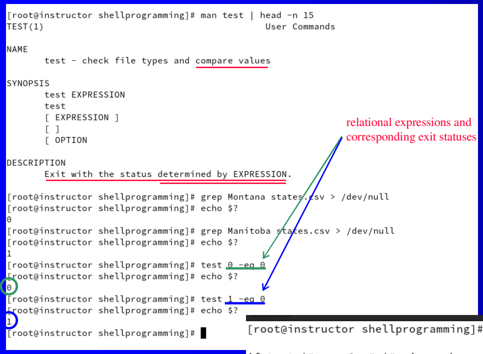
if – conditional execution

- what if user supplies no argument?
- protect with another, initial if
  - count the arguments
  - if none, exit
  - desired condition: the relational "count exceeds zero"
    - conditions are exit statuses
    - relationals are not exit statuses
    - how to turn a relational into an exit status in order to use it as a condition?

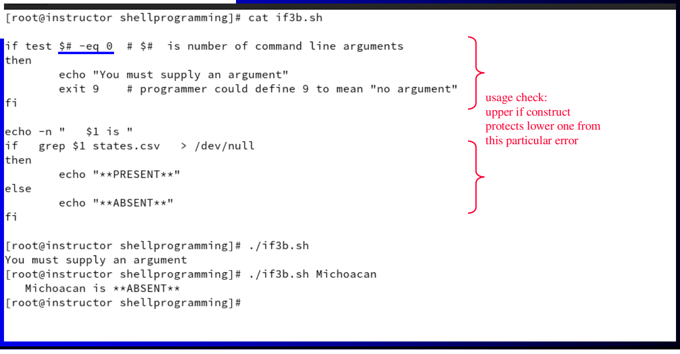


```
[root@instructor shellprogramming]# cat if3.sh

echo -n "   $1 is "
if  grep $1 states.csv   > /dev/null
then
        echo "**PRESENT**"
else
        echo "**ABSENT**"
fi

[root@instructor shellprogramming]# ./if3.sh Montana
   Montana is **PRESENT**
[root@instructor shellprogramming]# ./if3.sh Manitoba
   Manitoba is **ABSENT**
[root@instructor shellprogramming]#
[root@instructor shellprogramming]# ./if3.sh Texas
   Texas is **PRESENT**
[root@instructor shellprogramming]# ./if3.sh Tamauluipas
   Tamauluipas is **ABSENT**
[root@instructor shellprogramming]#
[root@instructor shellprogramming]# 
```

"test" command – converts relationals to exit statuses

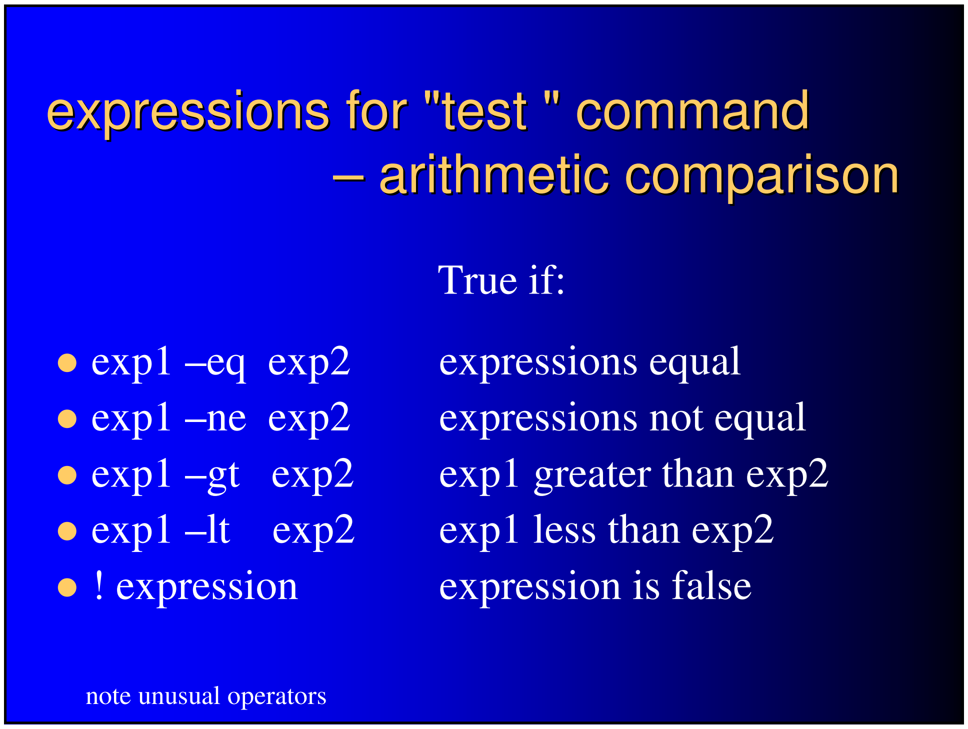


```
[root@instructor shellprogramming]# man test | head -n 15
TEST(1)                                               User Commands

NAME
       test - check file types and compare values

SYNOPSIS
       test EXPRESSION
       test
       [ EXPRESSION ]
       [ ]
       [ OPTION

DESCRIPTION
       Exit with the status determined by EXPRESSION.

[root@instructor shellprogramming]# grep Montana states.csv > /dev/null
[root@instructor shellprogramming]# echo $?
0
[root@instructor shellprogramming]# grep Manitoba states.csv > /dev/null
[root@instructor shellprogramming]# echo $?
1
[root@instructor shellprogramming]# test 0 -eq 0
[root@instructor shellprogramming]# echo $?
0
[root@instructor shellprogramming]# test 1 -eq 0
[root@instructor shellprogramming]# echo $?
1
[root@instructor shellprogramming]# 
```



```
[root@instructor shellprogramming]# cat if3b.sh

if test $# -eq 0  # $#  is number of command line arguments
then
        echo "You must supply an argument"
        exit 9    # programmer could define 9 to mean "no argument"
fi

echo -n "   $1 is "
if   grep $1 states.csv   > /dev/null
then
        echo "**PRESENT**"
else
        echo "**ABSENT**"
fi

[root@instructor shellprogramming]# ./if3b.sh
You must supply an argument
[root@instructor shellprogramming]# ./if3b.sh Michoacan
   Michoacan is **ABSENT**
[root@instructor shellprogramming]#
```

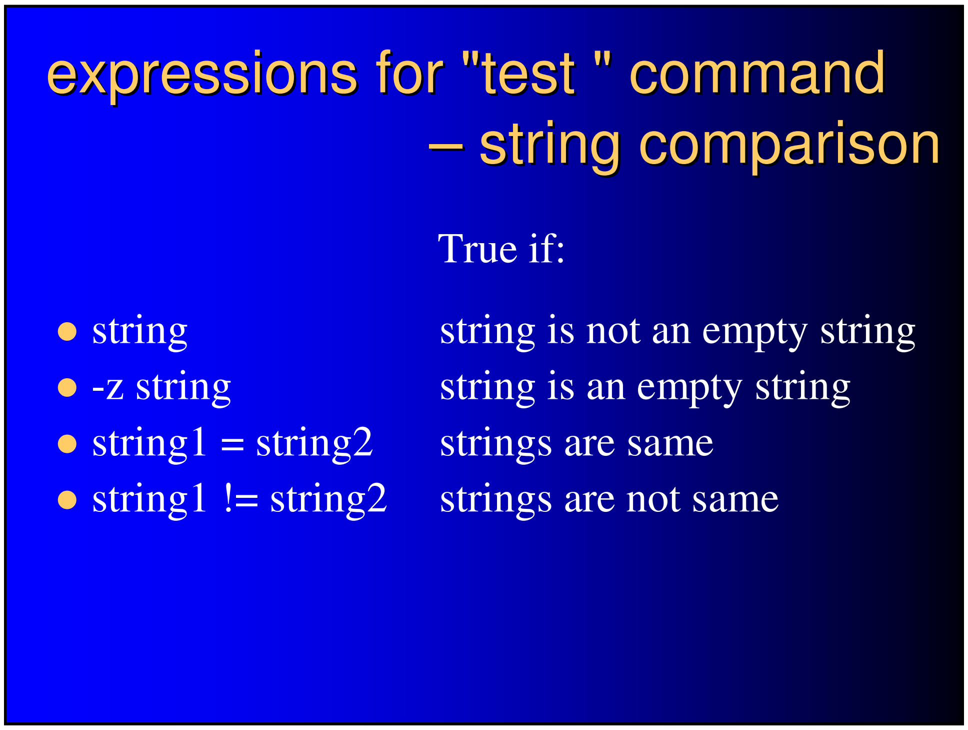
usage check: upper if construct protects lower one from this particular error



expressions for "test " command – arithmetic comparison

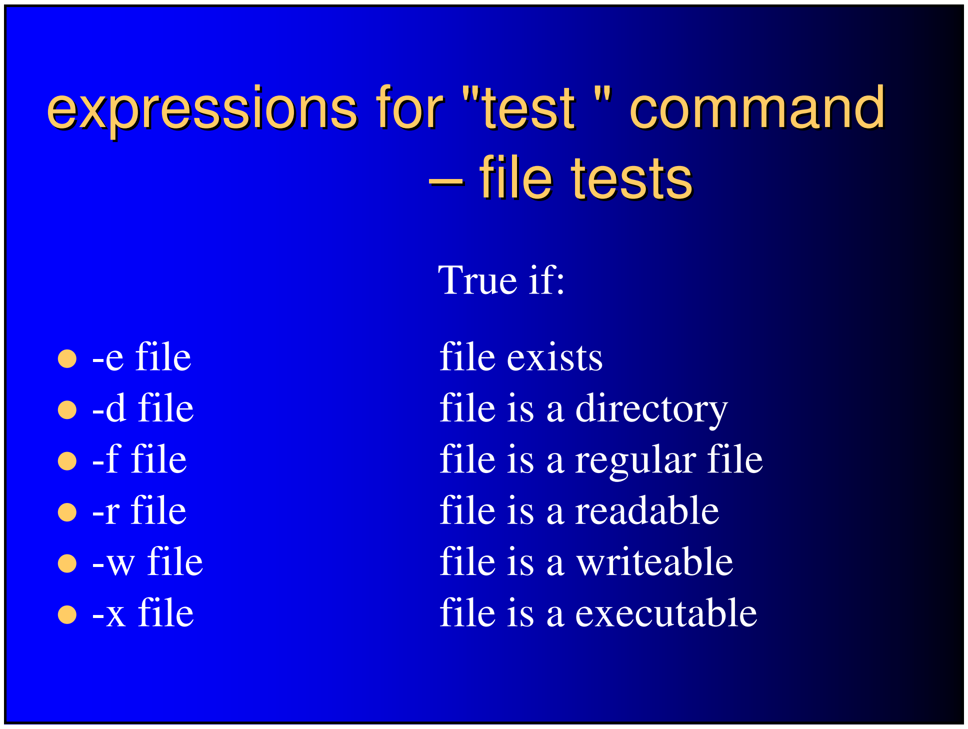
	True if:
● exp1 –eq exp2	expressions equal
● exp1 –ne exp2	expressions not equal
● exp1 –gt exp2	exp1 greater than exp2
● exp1 –lt exp2	exp1 less than exp2
● ! expression	expression is false

note unusual operators



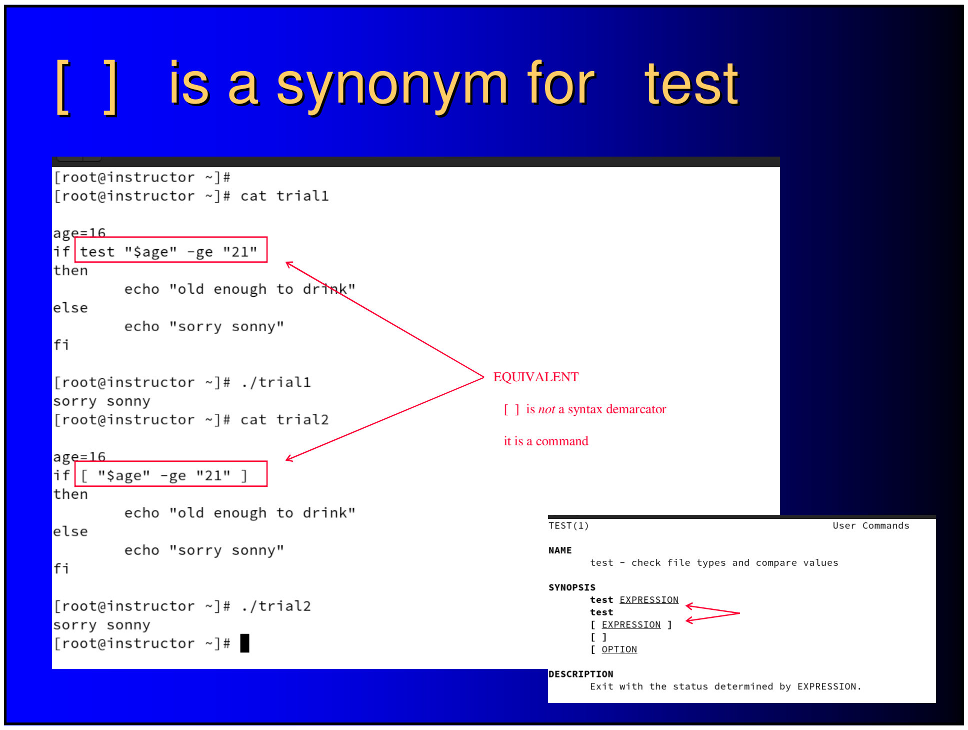
expressions for "test " command – string comparison

	True if:
● string	string is not an empty string
● -z string	string is an empty string
● string1 = string2	strings are same
● string1 != string2	strings are not same



expressions for "test " command – file tests

	True if:
● -e file	file exists
● -d file	file is a directory
● -f file	file is a regular file
● -r file	file is a readable
● -w file	file is a writeable
● -x file	file is a executable



[ ] is a synonym for test

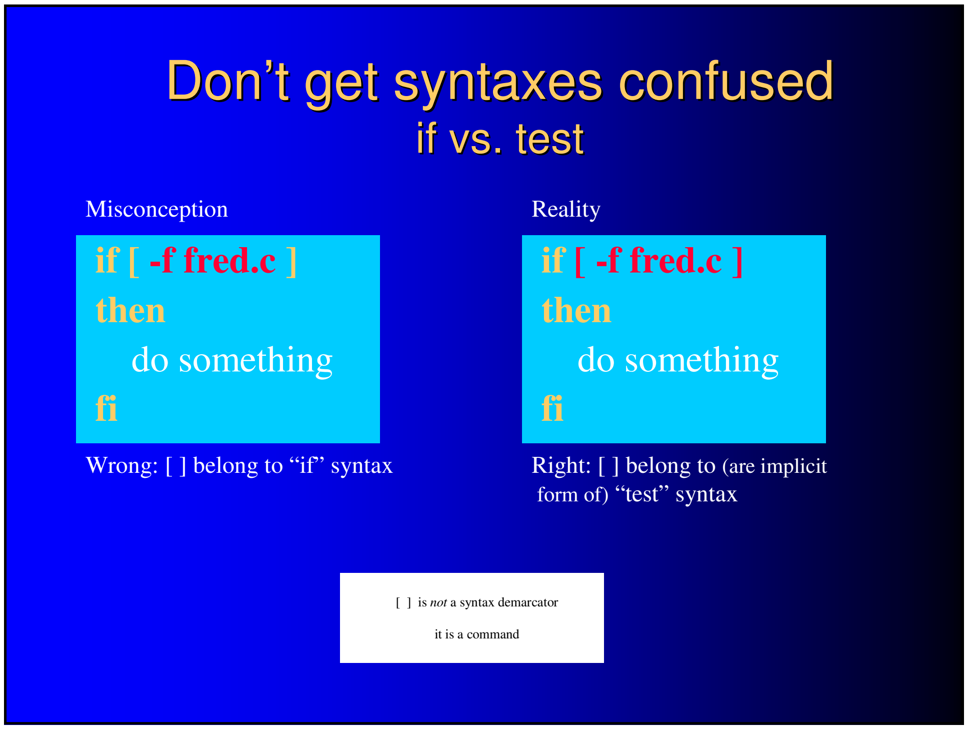
```
[root@instructor ~]#
[root@instructor ~]# cat trial1

age=16
if test "$age" -ge "21"
then
        echo "old enough to drink"
else
        echo "sorry sonny"
fi

[root@instructor ~]# ./trial1
sorry sonny
[root@instructor ~]# cat trial2

age=16
if [ "$age" -ge "21" ]
then
        echo "old enough to drink"
else
        echo "sorry sonny"
fi

[root@instructor ~]# ./trial2
sorry sonny
[root@instructor ~]#
```

EQUIVALENT

[ ] is *not* a syntax demarcator

it is a command

```
TEST(1)                                   User Commands

NAME
       test - check file types and compare values

SYNOPSIS
       test EXPRESSION
       test
       [ EXPRESSION ]
       [ ]
       [ OPTION

DESCRIPTION
       Exit with the status determined by EXPRESSION.
```



Don’t get syntaxes confused
if vs. test

Misconception

```
if [ -f fred.c ]
then
    do something
fi
```

Wrong: [ ] belong to “if” syntax

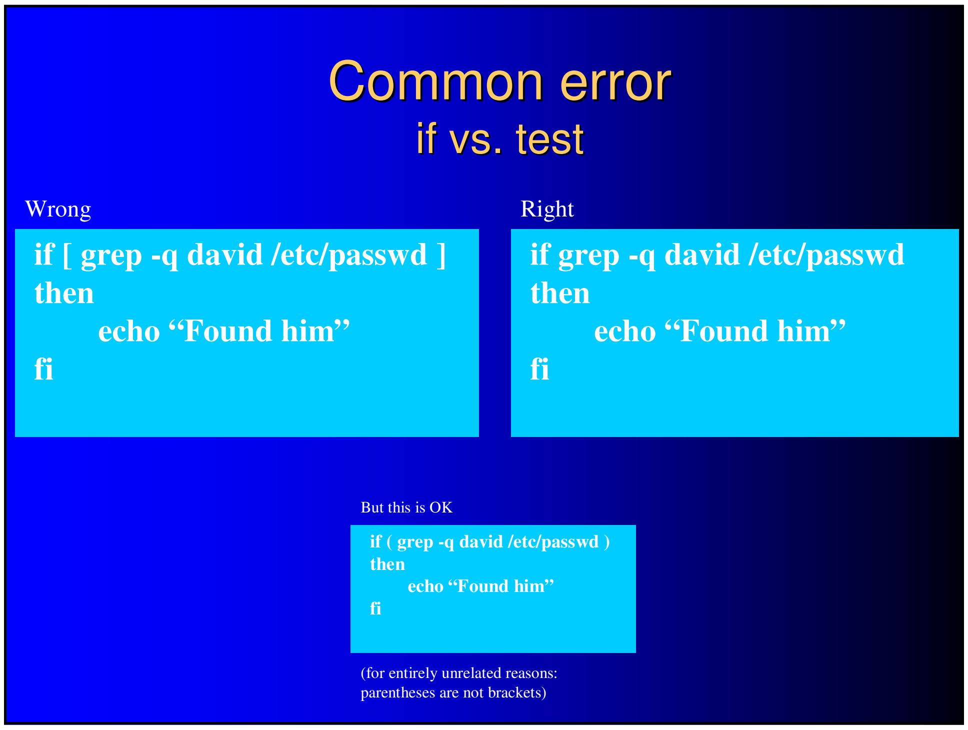
Reality

```
if [ -f fred.c ]
then
    do something
fi
```

Right: [ ] belong to (are implicit form of) “test” syntax

[ ] is *not* a syntax demarcator

it is a command



Common error
if vs. test

Wrong

```
if [ grep -q david /etc/passwd ]
then
        echo “Found him”
fi
```

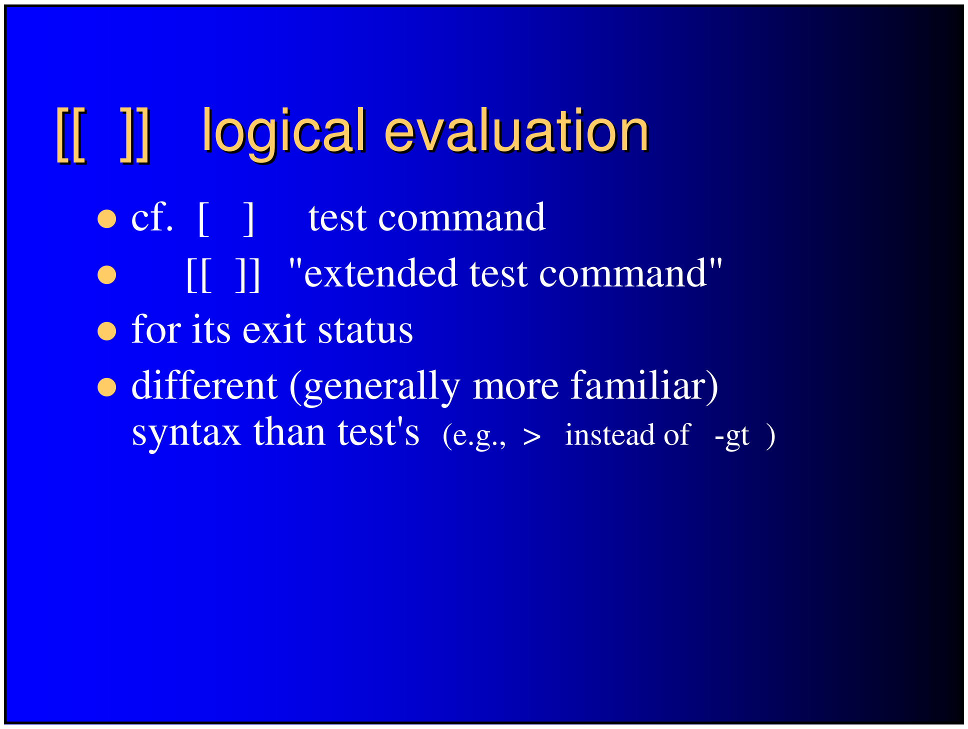
Right

```
if grep -q david /etc/passwd
then
        echo “Found him”
fi
```

But this is OK

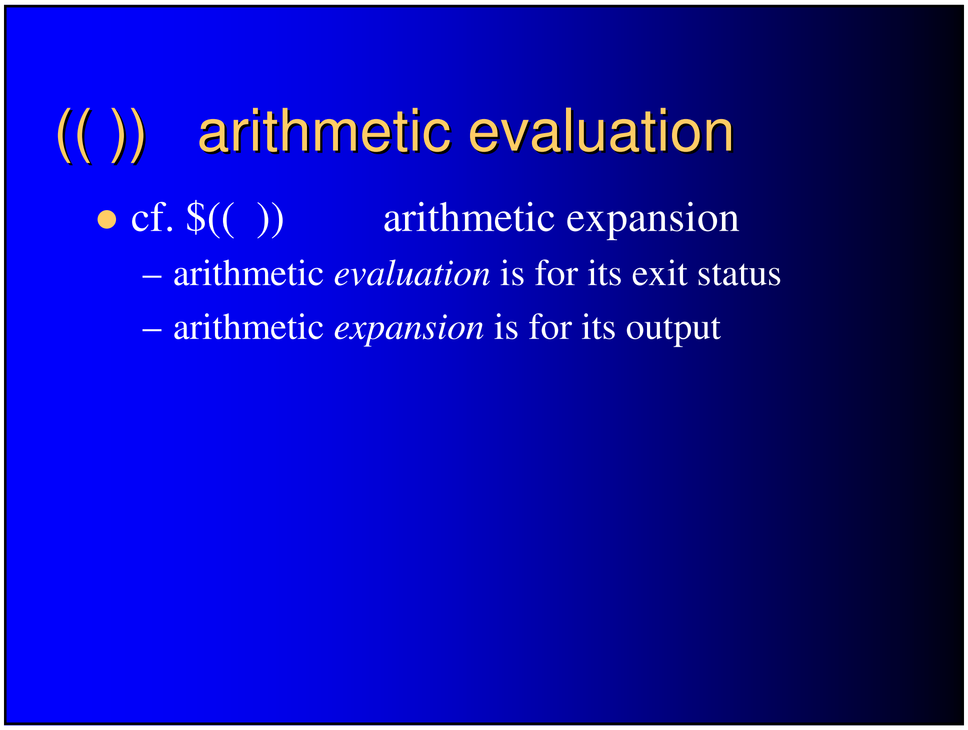
```
if ( grep -q david /etc/passwd )
then
        echo “Found him”
fi
```

(for entirely unrelated reasons: parentheses are not brackets)



[[ ]] logical evaluation

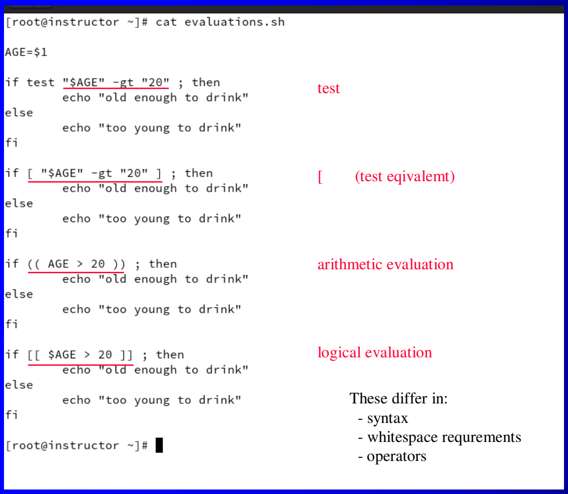
- cf. [ ] test command
- [[ ]] "extended test command"
- for its exit status
- different (generally more familiar) syntax than test's (e.g., > instead of -gt)



(( )) arithmetic evaluation

- cf. $(( )) arithmetic expansion
  - arithmetic *evaluation* is for its exit status
  - arithmetic *expansion* is for its output

Different forms of evaluation



```
[root@instructor ~]# cat evaluations.sh

AGE=$1

if test "$AGE" -gt "20" ; then
        echo "old enough to drink"
else
        echo "too young to drink"
fi

if [ "$AGE" -gt "20" ] ; then
        echo "old enough to drink"
else
        echo "too young to drink"
fi

if (( AGE > 20 )) ; then
        echo "old enough to drink"
else
        echo "too young to drink"
fi

if [[ $AGE > 20 ]] ; then
        echo "old enough to drink"
else
        echo "too young to drink"
fi

[root@instructor ~]#
```

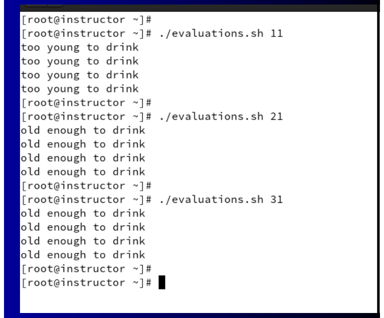
test

[(test eqivalemt)

arithmetic evaluation

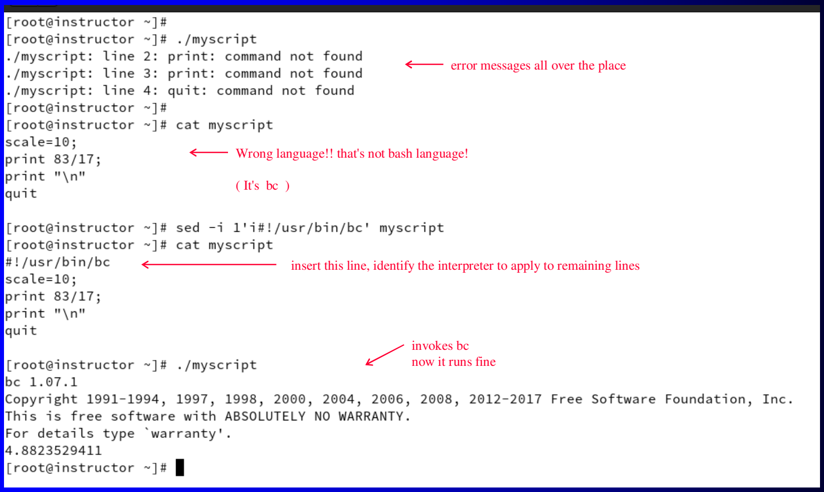
logical evaluation

These differ in:
- syntax
- whitespace requrements
- operators



```
[root@instructor ~]#
[root@instructor ~]# ./evaluations.sh 11
too young to drink
too young to drink
too young to drink
too young to drink
[root@instructor ~]#
[root@instructor ~]# ./evaluations.sh 21
old enough to drink
old enough to drink
old enough to drink
old enough to drink
[root@instructor ~]#
[root@instructor ~]# ./evaluations.sh 31
old enough to drink
old enough to drink
old enough to drink
old enough to drink
[root@instructor ~]#
[root@instructor ~]#
```

interpreter scripts - shebang #!



```
[root@instructor ~]#
[root@instructor ~]# ./myscript
./myscript: line 2: print: command not found
./myscript: line 3: print: command not found
./myscript: line 4: quit: command not found
[root@instructor ~]#
[root@instructor ~]# cat myscript
scale=10;
print 83/17;
print "\n"
quit

[root@instructor ~]# sed -i 1'i#!/usr/bin/bc' myscript
[root@instructor ~]# cat myscript
#!/usr/bin/bc
scale=10;
print 83/17;
print "\n"
quit

[root@instructor ~]# ./myscript
bc 1.07.1
Copyright 1991-1994, 1997, 1998, 2000, 2004, 2006, 2008, 2012-2017 Free Software Foundation, Inc.
This is free software with ABSOLUTELY NO WARRANTY.
For details type `warranty'.
4.8823529411
[root@instructor ~]#
```

$\longleftarrow$ error messages all over the place

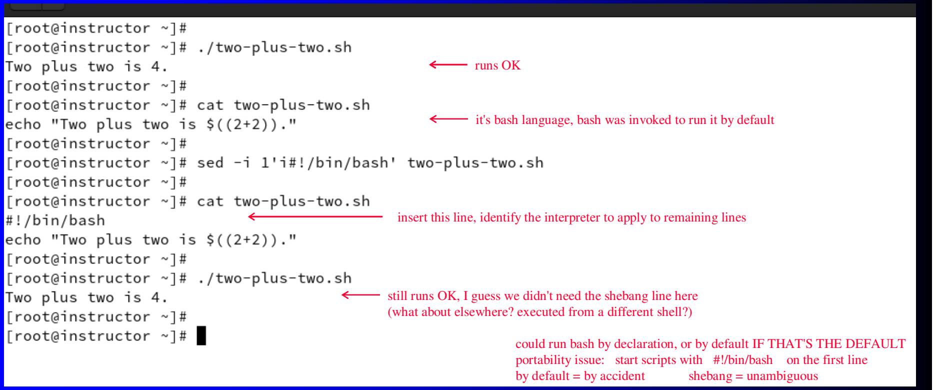
$\longleftarrow$ Wrong language!! that's not bash language!

(It's bc)

$\longleftarrow$ insert this line, identify the interpreter to apply to remaining lines

$\swarrow$ invokes bc
now it runs fine

interpreter scripts - #!/bin/bash



```
[root@instructor ~]#
[root@instructor ~]# ./two-plus-two.sh
Two plus two is 4.
[root@instructor ~]#
[root@instructor ~]# cat two-plus-two.sh
echo "Two plus two is $((2+2))."
[root@instructor ~]#
[root@instructor ~]# sed -i 1'i#!/bin/bash' two-plus-two.sh
[root@instructor ~]#
[root@instructor ~]# cat two-plus-two.sh
#!/bin/bash
echo "Two plus two is $((2+2))."
[root@instructor ~]#
[root@instructor ~]# ./two-plus-two.sh
Two plus two is 4.
[root@instructor ~]#
[root@instructor ~]# █
```

⟵ runs OK

⟵ it's bash language, bash was invoked to run it by default

⟵ insert this line, identify the interpreter to apply to remaining lines

⟵ still runs OK, I guess we didn't need the shebang line here
(what about elsewhere? executed from a different shell?)

could run bash by declaration, or by default IF THAT'S THE DEFAULT
portability issue: start scripts with #!/bin/bash on the first line
by default = by accident shebang = unambiguous

Arithmetic evaluation by bash

- computationally expensive, inefficient

$((22+33)

$$\overbrace{00110010}^{2}\ \overbrace{00110010}^{2}\ \overbrace{00101011}^{+}\ \overbrace{00110011}^{3}\ \overbrace{00110011}^{3}$$

↓ *expensive!*

$$\begin{array}{r} 00010110 \\ +\ 00100001 \\ \hline 00110111 \end{array}$$

↓ *sheesh!*

$$\underbrace{00110101}_{5}\ \underbrace{00110101}_{5}$$

looping – conditional repetition

while

```
while condition do
      commands
done
```

while

```
read trythis
while [ “$trythis” != “secret” ]
do
      echo “Sorry, try again“
      read trythis
done
```

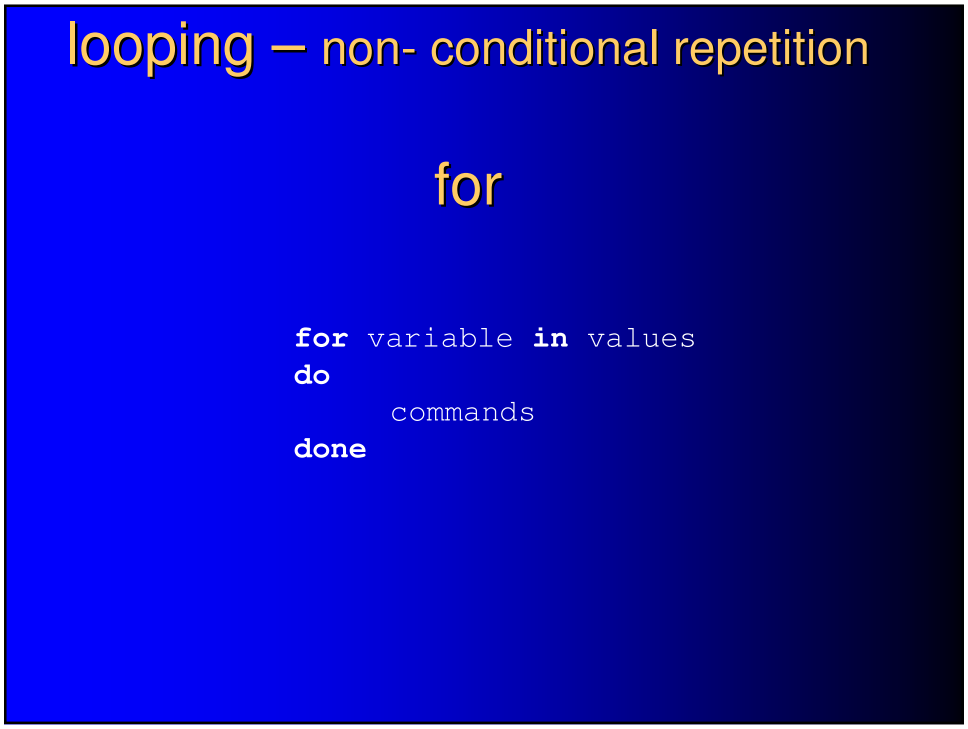
looping – conditional repetition

until

```
until condition
do
        commands
done
```

until

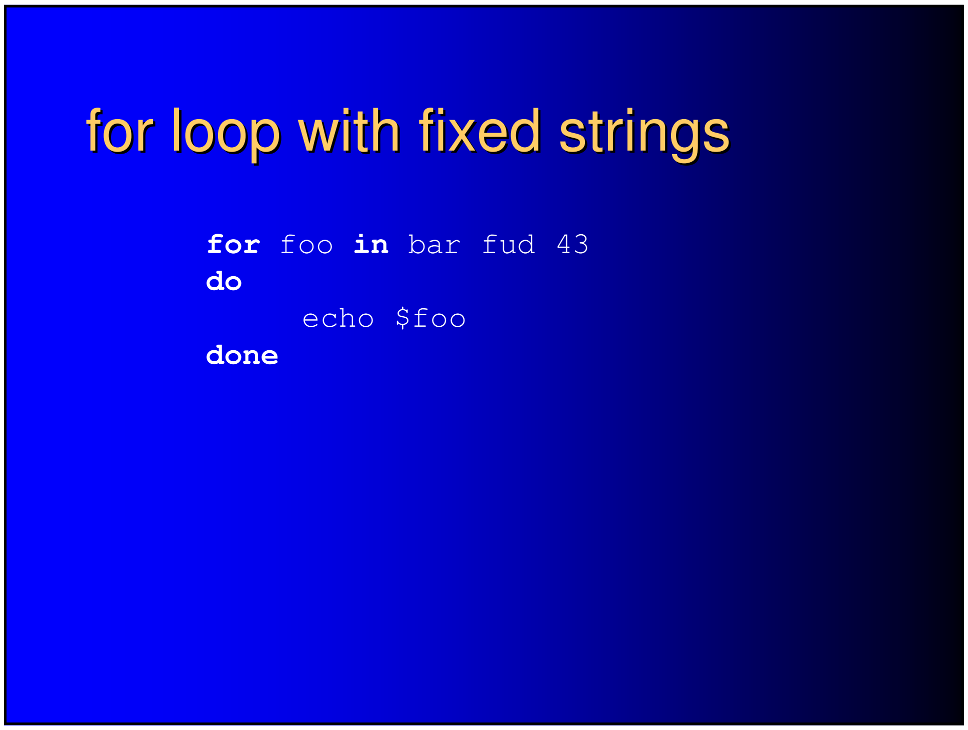
```
until who | grep "$1" > /dev/null
do
        sleep 5
done
echo "*** $1 has just logged in ***"
```



looping – non- conditional repetition

for

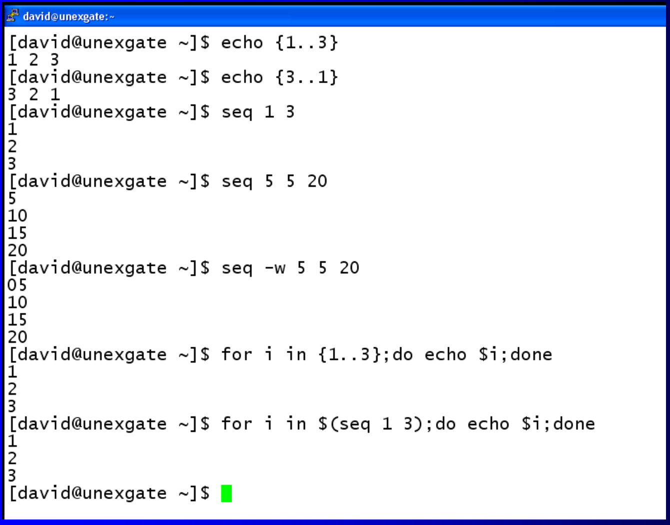
```
for variable in values
do
        commands
done
```



for loop with fixed strings

```
for foo in bar fud 43
do
        echo $foo
done
```

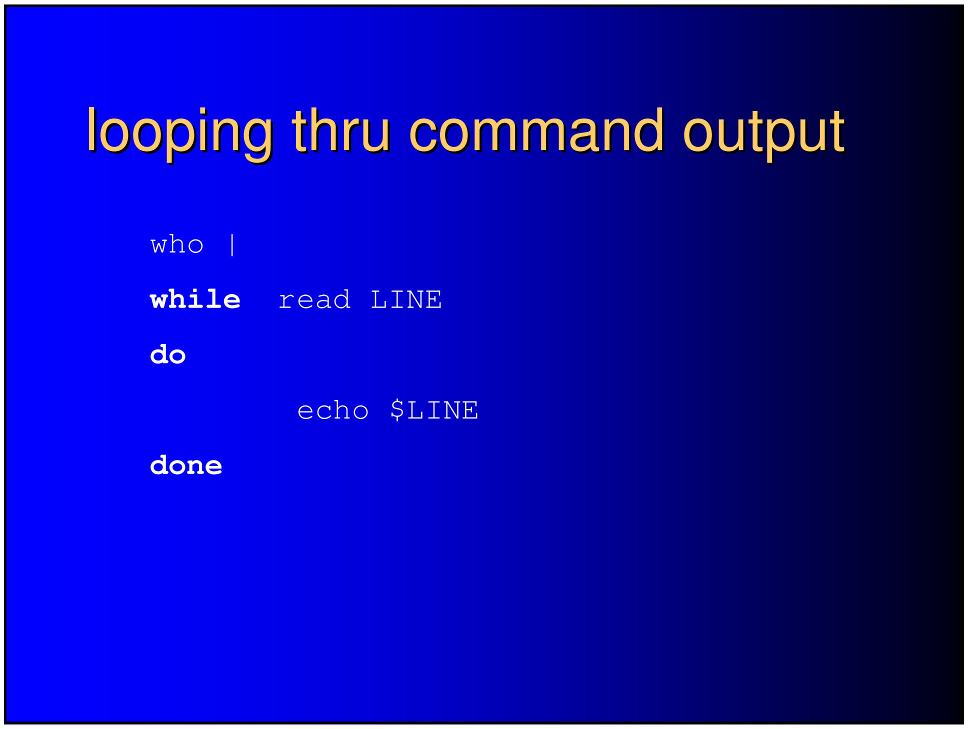
Manufacture step values with {x..y} or seq x y



```
[david@unexgate ~]$ echo {1..3}
1 2 3
[david@unexgate ~]$ echo {3..1}
3 2 1
[david@unexgate ~]$ seq 1 3
1
2
3
[david@unexgate ~]$ seq 5 5 20
5
10
15
20
[david@unexgate ~]$ seq -w 5 5 20
05
10
15
20
[david@unexgate ~]$ for i in {1..3};do echo $i;done
1
2
3
[david@unexgate ~]$ for i in $(seq 1 3);do echo $i;done
1
2
3
[david@unexgate ~]$
```

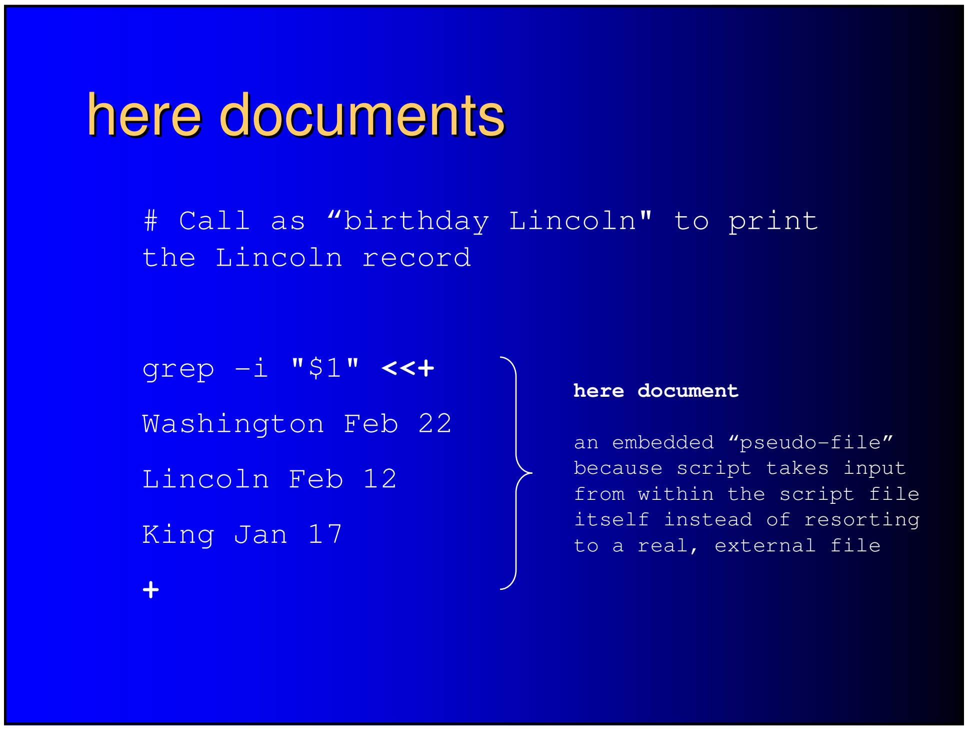
looping thru a file

```
while read LINE
do
        echo $LINE

done < /home/joe/myfile
```



looping thru command output

```
who |
while  read LINE
do
        echo $LINE
done
```



here documents

```
# Call as "birthday Lincoln" to print
the Lincoln record

grep -i "$1" <<+
Washington Feb 22
Lincoln Feb 12
King Jan 17
+
```

here document

an embedded "pseudo-file" because script takes input from within the script file itself instead of resorting to a real, external file