Unix Programming

Shell Programming

Nguyen Thanh Hung
Software Engineering Department
Hanoi University of Science and Technology

Outline

- What a shell is
- Basic considerations
- The subtleties of syntax: variables, conditions, and program control
- Lists
- Functions
- Commands and command execution
- grep and regular expressions
- find

Why Program with a Shell?

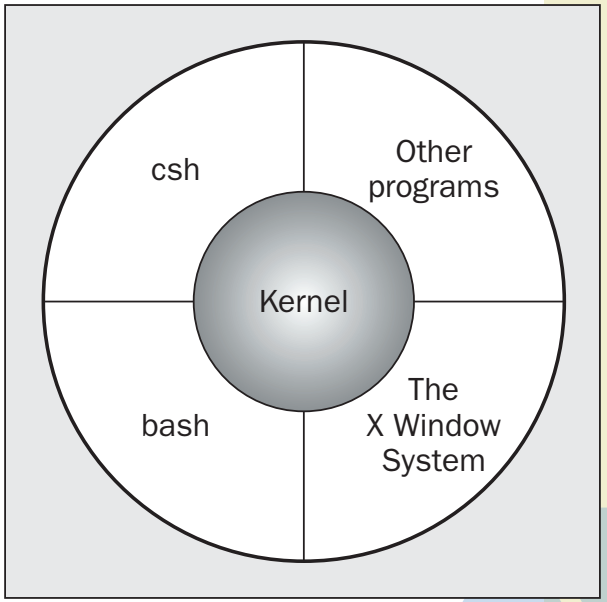
- You can program the shell quickly and simply
- A shell is always available even on the most basic Linux installation
- The shell is also ideal for any small utilities that perform some relatively simple task for which efficiency is less important than easy configuration, maintenance, and portability
- It's much more powerful than Windows command prompt, capable of running reasonably complex programs in its own right
- The shell executes shell programs, often referred to as scripts, which are interpreted at runtime. This generally makes debugging easier because you can easily execute single lines, and there's no recompile time

Shell

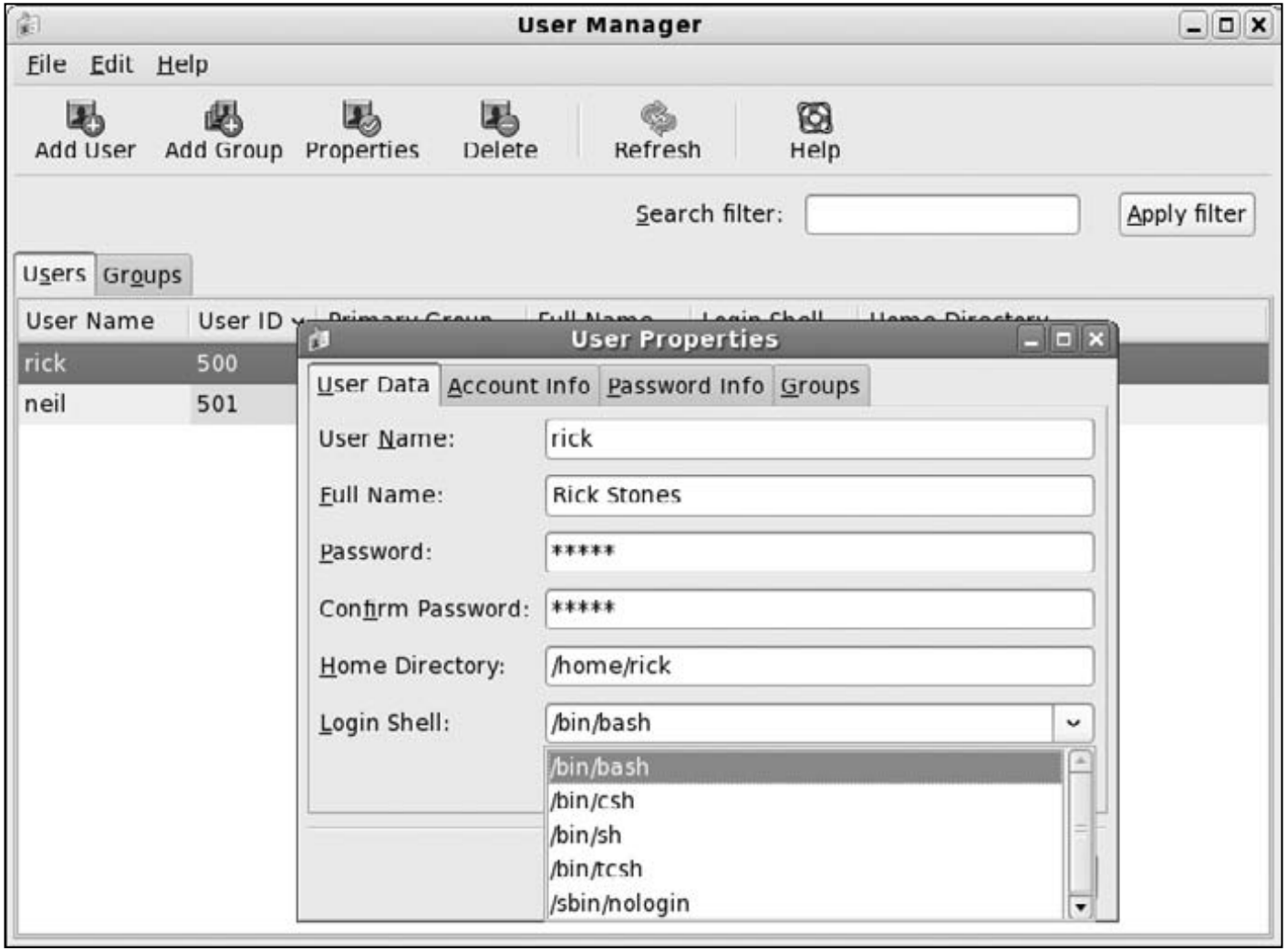
What is a shell?

- A program that acts as the interface between you and the Linux system, enabling you to enter commands for the operating system to execute

```
$ /bin/bash --version
GNU bash, version 3.2.9(1)-release (i686-pc-linux-gnu)
Copyright (C) 2005 Free Software Foundation, Inc.
```



Shell


User Manager
File Edit Help
Add User
Add Group
Properties
Delete
Refresh
Help
Search filter:
Apply filter
Users
Groups
User Name
User ID
rick
500
neil
501
User Properties
User Data
Account Info
Password Info
Groups
User Name:
rick
Full Name:
Rick Stones
Password:
*****
Confirm Password:
*****
Home Directory:
/home/rick
Login Shell:
/bin/bash
/bin/bash
/bin/csh
/bin/sh
/bin/tcsh
/sbin/nologin

Shell

- **Many other shells are available, either free or commercially**

Shell Name	A Bit of History
`sh` (Bourne)	The original shell from early versions of UNIX
`csh`, `tcsh`, `zsh`	The C shell, and its derivatives, originally created by Bill Joy of Berkeley UNIX fame. The C shell is probably the third most popular type of shell after bash and the Korn shell.
`ksh`, `pdksh`	The Korn shell and its public domain cousin. Written by David Korn, this is the default shell on many commercial UNIX versions.
`bash`	The Linux staple shell from the GNU project. bash, or Bourne Again SHell, has the advantage that the source code is freely available, and even if it's not currently running on your UNIX system, it has probably been ported to it. bash has many similarities to the Korn shell.

Pipes and Redirection

Redirecting Output

- $ **ls -l > lsoutput.txt:** saves the output of the ls command into a file called **lsoutput.txt**
- To append to the file, use the >> operator. For example,

  $ ps >> lsoutput.txt: will append the output of the ps command to the end of the specified file

Pipes and Redirection

- **Redirecting Input**
  - more < killout.txt

Pipes and Redirection

Pipes

- You can connect processes using the pipe operator (|)
- In Linux, processes connected by pipes can run simultaneously and are automatically rescheduled as data flows between them

```
$ ps > psout.txt
$ sort psout.txt > pssort.out
```

```
$ ps | sort > pssort.out
```

The Shell as a Programming Language

Interactive Programs

- Suppose you have a large number of C files and wish to examine the files that contain the string POSIX

```
$ for file in *
> do
> if grep -l POSIX $file
> then
> more $file
> fi
> done
posix
This is a file with POSIX in it - treat it well
$
```

Script

Creating a Script

- Comments start with a # and continue to the end of a line
- the first line, #!/bin/sh, is a special form of comment; the #! characters tell the system that the argument that follows on the line is the program to be used to execute this file.

```
#!/bin/sh

# first
# This file looks through all the files in the current
# directory for the string POSIX, and then prints the names of
# those files to the standard output.

for file in *
do
  if grep -q POSIX $file
  then
    echo $file
  fi
done

exit 0
```

Script

Making a Script Executable

```
$ /bin/sh first
```

- This should work, but it would be much better if you could simply invoke the script by typing its name

```
$ chmod +x first
```

```
$ first
```

Shell Syntax

- Variables: strings, numbers, environments, and parameters
- Conditions: shell Booleans
- Program control: if, elif, for, while, until, case
- Lists
- Functions
- Commands built into the shell
- Getting the result of a command

Shell Syntax

Variables

- You don't usually declare variables in the shell before using them.
- You create them by simply using them (for example, when you assign an initial value to them)
- By default, all variables are considered and stored as strings, even when they are assigned numeric values.
- Within the shell you can access the contents of a variable by preceding its name with a $.
- When you assign a value to a variable, just use the name of the variable

Shell Syntax

Variables

```
$ salutation=Hello
$ echo $salutation
Hello
$ salutation="Yes Dear"
$ echo $salutation
Yes Dear
$ salutation=7+5
$ echo $salutation
7+5
```

Shell Syntax

Variables: Read command

- You can assign user input to a variable by using the **read** command.
- This takes one parameter, the name of the variable to be read into, and then waits for the user to enter some text

```
$ read salutation
Wie geht's?
$ echo $salutation
Wie geht's?
```

Shell Syntax: Variables

Examples

```
#!/bin/sh

myvar="Hi there"

echo $myvar
echo "$myvar"
echo '$myvar'
echo \$myvar

echo Enter some text
read myvar

echo '$myvar' now equals $myvar
exit 0
```

```
$ ./variable
Hi there
Hi there
$myvar
$myvar

Enter some text
Hello World
$myvar now equals Hello World
```

- The variable myvar is created and assigned the string Hi there.
- Using double quotes doesn't affect the substitution of the variable, while single quotes and the backslash do.
- Use the read command to get a string from the user.

Shell Syntax: Variables

Environment Variables

- When a shell script starts, some variables are initialized from values in the environment

Environment Variable	Description
`$HOME`	The home directory of the current user
`$PATH`	A colon-separated list of directories to search for commands
`$IFS`	An input field separator. This is a list of characters that are used to separate words when the shell is reading input, usually space, tab, and newline characters.
`$0`	The name of the shell script
`$#`	The number of parameters passed
`$$`	The process ID of the shell script, often used inside a script for generating unique temporary filenames; for example `/tmp/tmpfile_$$`

Shell Syntax

Parameter environment

- If your script is invoked with parameters, some additional variables are created.
- If no parameters are passed, the environment variable $# still exists but has a value of 0.

Parameter Variable	Description
`$1, $2, …`	The parameters given to the script
`$*`	A list of all the parameters, in a single variable, separated by the first character in the environment variable `IFS`. If `IFS` is modified, then the way `$*` separates the command line into parameters will change.
`$@`	A subtle variation on `$*`; it doesn't use the `IFS` environment variable, so parameters are not run together even if `IFS` is empty.

Shell Syntax

- It's easy to see the difference between $@ and $* by trying them out:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

- Within double quotes, $@ expands the positional parameters as separate fields, regardless of the IFS value.
- In general, if you want access to the parameters, $@ is the sensible choice.

Shell Syntax: Variables

Example

```
#!/bin/sh

salutation="Hello"
echo $salutation
echo "The program $0 is now running"
echo "The second parameter was $2"
echo "The first parameter was $1"
echo "The parameter list was $*"
echo "The user's home directory is $HOME"

echo "Please enter a new greeting"
read salutation

echo $salutation
echo "The script is now complete"
exit 0
```

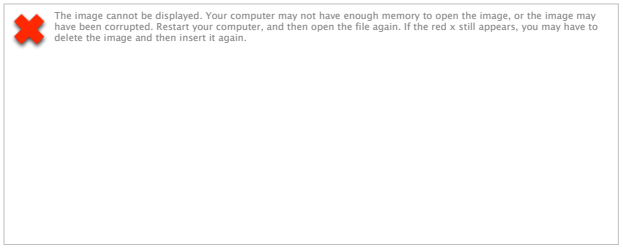
```
$ ./try_var foo bar baz
Hello
The program ./try_var is now running
The second parameter was bar
The first parameter was foo
The parameter list was foo bar baz
The user's home directory is /home/rick
Please enter a new greeting
Sire
Sire
The script is now complete
$
```

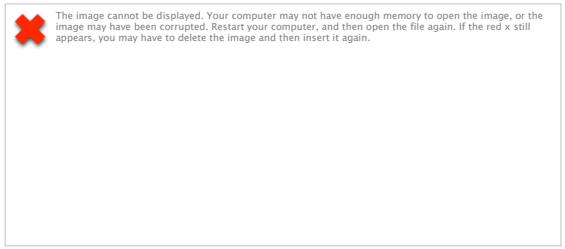
- This script creates the variable salutation, displays its contents, and then shows how various parameter variables and the environment variable $HOME already exist and have appropriate values.

Conditions

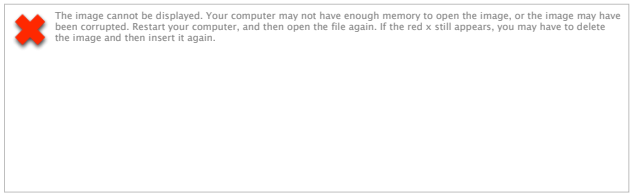
The test or [Command

- Is the shell's Boolean check
- Checking to see whether a file exists.
- The command for this is test -f <filename>





- Note that you must put spaces between the [braces and the condition being checked
- If you prefer putting then on the same line as if, you must add a semicolon to separate the test from the then:



Conditions

String comparison

String Comparison	Result
`string1 = string2`	True if the strings are equal
`string1 != string2`	True if the strings are not equal
`-n string`	True if the string is not null
`-z string`	True if the string is null (an empty string)

Conditions

Arithmetic Comparison

Arithmetic Comparison	Result
`expression1 -eq expression2`	True if the expressions are equal
`expression1 -ne expression2`	True if the expressions are not equal
`expression1 -gt expression2`	True if `expression1` is greater than `expression2`
`expression1 -ge expression2`	True if `expression1` is greater than or equal to `expression2`
`expression1 -lt expression2`	True if `expression1` is less than `expression2`
`expression1 -le expression2`	True if `expression1` is less than or equal to `expression2`
`! expression`	True if the expression is false, and vice versa

Conditions

File conditional

File Conditional	Result
`-d file`	True if the file is a directory
`-e file`	True if the file exists. Note that historically the `-e` option has not been portable, so `-f` is usually used.
`-f file`	True if the file is a regular file
`-g file`	True if `set-group-id` is set on `file`
`-r file`	True if the file is readable
`-s file`	True if the file has nonzero size
`-u file`	True if `set-user-id` is set on `file`
`-w file`	True if the file is writable
`-x file`	True if the file is executable

Conditions

Example

```
#!/bin/sh

if [ -f /bin/bash ]
then
  echo "file /bin/bash exists"
fi

if [ -d /bin/bash ]
then
  echo "/bin/bash is a directory"
else
  echo "/bin/bash is NOT a directory"
fi
```

Control Structure

if

```
if condition
then
   statements
else
   statements
fi
```

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]; then
  echo "Good morning"
else
  echo "Good afternoon"
fi

exit 0
```

```
Is it morning? Please answer yes or no
yes
Good morning
$
```

Control Structure

elif

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ $timeofday = "yes" ]
then
  echo "Good morning"

elif [ $timeofday = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

Control Structure

A Problem with Variables

- just press Enter (or Return on some keyboards), rather than answering the question. You'll get this error message:

  [: =: unary operator expected

- When the variable timeofday was tested, it consisted of a blank string. Therefore, the if clause looks like

```
if [ = "yes" ]
```

- which isn't a valid condition. To avoid this, you must use quotes around the variable:

```
if [ "$timeofday" = "yes" ]
```

Control Structure

New Script

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

if [ "$timeofday" = "yes" ]
then
  echo "Good morning"
elif [ "$timeofday" = "no" ]; then
  echo "Good afternoon"
else
  echo "Sorry, $timeofday not recognized. Enter yes or no"
  exit 1
fi

exit 0
```

Control Structure

for

```
for variable in values
do
  statements
done
```

```
#!/bin/sh

for foo in bar fud 43
do
  echo $foo
done
exit 0
```

```
bar
fud
43
```

Control Structure

Example

```
#!/bin/sh

for file in $(ls f*.sh); do
  lpr $file
done
exit 0
```

Control Structure

While

```
while condition do
   statements
done
```

```
#!/bin/sh

echo "Enter password"
read trythis

while [ "$trythis" != "secret" ]; do
  echo "Sorry, try again"
  read trythis
done
exit 0
```

```
Enter password
password
Sorry, try again
secret
$
```

Control Structure

until

```
until condition
do
   statements
done
```

```
#!/bin/bash

until who | grep "$1" > /dev/null
do
   sleep 60
done

# now ring the bell and announce the expected user.

echo -e '\a'
echo "**** $1 has just logged in ****"

exit 0
```

you can set up an alarm that is initiated when another user, whose login name you pass on the command line, logs on

Control Structure

Case

```
case variable in
  pattern [ | pattern] ...) statements;;
  pattern [ | pattern] ...) statements;;
  ...
esac
```

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes)  echo "Good Morning";;
    no )  echo "Good Afternoon";;
    y  )  echo "Good Morning";;
    n  )  echo "Good Afternoon";;
    *  )  echo "Sorry, answer not recognized";;
esac

exit 0
```

Control Structure

Case

```
#!/bin/sh

echo "Is it morning? Please answer yes or no"
read timeofday

case "$timeofday" in
    yes | y | Yes | YES )      echo "Good Morning";;
    n* | N* )                  echo "Good Afternoon";;
    * )                        echo "Sorry, answer not recognized";;
esac

exit 0
```

Notice that each pattern line is terminated with double semicolons (;;). You can put multiple statements between each pattern and the next, so a double semicolon is needed to mark where one statement ends and the next pattern begins.

Control Structure

List

- Sometimes you want to connect commands in a series. For instance, you may want several different conditions to be met before you execute a statement

```
if [ -f this_file ]; then
    if [ -f that_file ]; then
        if [ -f the_other_file ]; then
            echo "All files present, and correct"
        fi
    fi
fi
```

```
if [ -f this_file ]; then
   foo="True"
elif [ -f that_file ]; then
   foo="True"
elif [ -f the_other_file ]; then
   foo="True"
else
   foo="False"
fi
if [ "$foo" = "True" ]; then
   echo "One of the files exists"
fi
```

Control Structure

AND List

- The AND list construct enables you to execute a series of commands, executing the next command only if all the previous commands have succeeded.

```
statement1 && statement2 && statement3 && ...
```

```
#!/bin/sh

touch file_one
rm -f file_two

if [ -f file_one ] && echo "hello" && [ -f file_two ] && echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

```
hello
in else
```

Control Structure

OR List

- The OR list construct enables us to execute a series of commands until one succeeds, and then not execute any more

```
statement1 || statement2 || statement3 || ...
```

```
#!/bin/sh

rm -f file_one

if [ -f file_one ] || echo "hello" || echo " there"
then
    echo "in if"
else
    echo "in else"
fi

exit 0
```

```
hello
in if
```

Control Structure

Function

```
function_name () {
  statements
}
```

```
#!/bin/sh

foo() {
    echo "Function foo is executing"
}

echo "script starting"
foo
echo "script ended"

exit 0
```

```
script starting
Function foo is executing
script ending
```

To call a function with arguments:

function_name $arg1 $arg2

Control Structure

Local variable

```
#!/bin/sh

sample_text="global variable"

foo() {

    local sample_text="local variable"
    echo "Function foo is executing"
    echo $sample_text
}

echo "script starting"
echo $sample_text

foo

echo "script ended"
echo $sample_text

exit 0
```

Function

Returning value

```
#!/bin/sh

yes_or_no() {
  echo "Is your name $* ?"
  while true
  do
    echo -n "Enter yes or no: "
    read x
    case "$x" in
      y | yes ) return 0;;
      n | no )  return 1;;
      * )       echo "Answer yes or no"
    esac
  done
}
```

```
echo "Original parameters are $*"

if yes_or_no "$1"
then
  echo "Hi $1, nice name"
else
  echo "Never mind"
fi
exit 0
```

```
$ ./my_name Rick Neil
Original parameters are Rick Neil
Is your name Rick ?
Enter yes or no: yes
Hi Rick, nice name
$
```

Exercises

- 1. Write a script to calculate the sum of two integers
- 2. Write a script to find out the biggest number in 3 integers
- 3. Write a script to sort the given five integer numbers in ascending order (using array)
- 4. Write a script to calculate average of given integer numbers on command line arguments

Commands

: Command

- is a null command
- useful to simplify the logic of conditions, being an alias for true

```
#!/bin/sh

rm -f fred
if [ -f fred ]; then
    :
else
    echo file fred did not exist
fi

exit 0
```

Commands

- **. Command**
  - executes the command in the current shell
- **echo Command**
- **eval Command**
  - enables you to evaluate arguments

```
foo=10
x=foo
y='$'$x
echo $y
```

```
foo=10
x=foo
eval y='$'$x
echo $y
```

Commands

- **export Command**
  - makes the variable named as its parameter available in subshells

export bar=”The second meta-syntactic variable”

- **setenv Command**

setenv bar ”The second meta-syntactic variable”

Commands

expr Command

- evaluates its arguments as an expression

x=`expr $x + 1`

- The `` (backtick) characters make x take the result of executing the command expr $x + 1
- You could also write it using the syntax $( ) rather than backticks, like this:

x=$(expr $x + 1)

Commands

printf Command

- format string is very similar to that used in C or C++

printf “format string“ parameter1 parameter2 ...

```
$ printf "%s\n" hello
hello
$ printf "%s %d\t%s" "Hi There" 15 people
Hi There 15   people
```

Commands

find Command

```
# find / -name test -print
/usr/bin/test
#
```

```
# find / -mount -name test -print
/usr/bin/test
#
```

Not to search mounted directories

```
find [path] [options] [tests] [actions]
```

Option	Meaning
`-depth`	Search the contents of a directory before looking at the directory itself.
`-follow`	Follow symbolic links.
`-maxdepths N`	Search at most *N* levels of the directory when searching.
`-mount` (or `-xdev`)	Don't search directories on other file systems.

Commands

find Command

Test	Meaning
`-atime N`	The file was last accessed *N* days ago.
`-mtime N`	The file was last modified *N* days ago.
`-name pattern`	The name of the file, excluding any path, matches the pattern provided. To ensure that the pattern is passed to `find`, and not evaluated by the shell immediately, the pattern must always be in quotes.
`-newer otherfile`	The file is newer than the file `otherfile`.
`-type C`	The file is of type *C*, where *C* can be of a particular type; the most common are "d" for a directory and "f" for a regular file. For other types consult the manual pages.
`-user username`	The file is owned by the user with the given name.

Try searching in the current directory for files modified more recently than the file while2:

```
$ find . -newer while2 -print
.
./elif3
./words.txt
./words2.txt
./_trap
$
```

Commands

find Command

Try searching in the current directory for files modified more recently than the file while2:

```
$ find . -newer while2 -print
.
./elif3
./words.txt
./words2.txt
./_trap
$
```

only in regular files, so you add an additional test, -type f

```
$ find . -newer while2 -type f -print
./elif3
./words.txt
./words2.txt
./_trap
$
```

Commands

grep Command

```
grep [options] PATTERN [FILES]
```

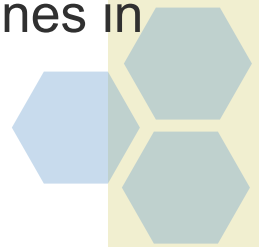
Option	Meaning
`-c`	Rather than print matching lines, print a count of the number of lines that match.
`-E`	Turn on extended expressions.
`-h`	Suppress the normal prefixing of each output line with the name of the file it was found in.
`-i`	Ignore case.
`-l`	List the names of the files with matching lines; don't output the actual matched line.
`-v`	Invert the matching pattern to select nonmatching lines, rather than matching lines.

Commands

grep Command

```
$ grep in words.txt
When shall we three meet again.  In thunder, lightning, or in rain?
I come, Graymalkin!
$ grep -c in words.txt words2.txt
words.txt:2
words2.txt:14
$ grep -c -v in words.txt words2.txt
words.txt:9
words2.txt:16
$
```

- The first example searches for the string “in” in the file words.txt and prints out any lines that match.
- The second example counts the number of matching lines in two different files. In this case, the filenames are printed out.
- Finally, use the -v option to invert the search and count lines in the two files that don’t match.



Commands

grep Command

Start by looking for lines that end with the letter *e*. You can probably guess you need to use the special character $:

```
$ grep  e$ words2.txt
Art thou not, fatal vision, sensible
I see thee yet, in form as palpable
Nature seems dead, and wicked dreams abuse
$
```

Now suppose you want to find words that end with the letter *a*. To do this, you need to use the special match characters in braces. In this case, you use [[:blank:]], which tests for a space or a tab:

```
$ grep a[[:blank:]] words2.txt
Is this a dagger which I see before me,
A dagger of the mind, a false creation,
Moves like a ghost. Thou sure and firm-set earth,
$
```

Now look for three-letter words that start with *Th*. In this case, you need both [[:space:]] to delimit the end of the word and . to match a single additional character:

```
$ grep Th.[[:space:]] words2.txt
The handle toward my hand? Come, let me clutch thee.
The curtain'd sleep; witchcraft celebrates
Thy very stones prate of my whereabout,
$
```

Finally, use the extended grep mode to search for lowercase words that are exactly 10 characters long. Do this by specifying a range of characters to match *a* to *z*, and a repetition of 10 matches:

```
$ grep -E [a-z]\{10\} words2.txt
Proceeding from the heat-oppressed brain?
And such an instrument I was to use.
The curtain'd sleep; witchcraft celebrates
Thy very stones prate of my whereabout,
$
```