Algorithmic Methods of Data Mining

Data Science at the Command Line

Ioannis Chatzigiannakis

Sapienza University of Rome

Laboratory 4

What is a Shell?

- The user interface to the operating system
- Functionality:
  - Execute other programs
  - Manage files
  - Manage processes
- A program like any other
- Executed when you "open a Terminal"

Shell Interactive Use

- The # is called the "prompt"
- In the prompt we type the name of the command and press "Enter"
- The prompt allows
  - Command history
  - Command line editing
  - File expansion (tab completion)
  - Command expansion
  - Key bindings
  - Spelling correction
  - Job control

Prompt: The Command Line

```
# date
Sat Apr 21 16:47:30 GMT 2007
```

Error Handling

- If we type a wrong command, an error message appears

Prompt: The Command Line

```
# datee
datee: no such file or directory
```

- The error message states that either the file or the folder (directory) was not found
  - In the prompt all commands are assumed to be connected to a file ...
- The arrow keys ↑ ↓ allow to look-up previous commands
- The arrow keys ← → allow to move within the same command line

Terminating Command Execution

- We can interrupt the execution of a command by pressing *ctrl-c*
- We can "freeze" the output of the execution of a command by pressing *ctrl-s*
  - To "un-freeze" the output of a command we use *ctrl-q*
  - Note – only the output is frozen not the actual execution
- To close a terminal we use *ctrl-d*
  - We may need to press multiple times *ctrl-q*
  - All programs currently running will terminate

Manual Pages

- The command *man* allows to access the manual pages
- Manual pages are organized in categories
  1. Commands – *ls*, *cp*, *grep*
  2. System Calls – *fork*, *exit*
  3. Libraries
  4. I/O Files
  5. File Encoding Types
  6. Games
  7. Miscellaneous
  8. Administrator's Commands
  9. Documents
- We can request a page from a specific category
  man [category] [topic]

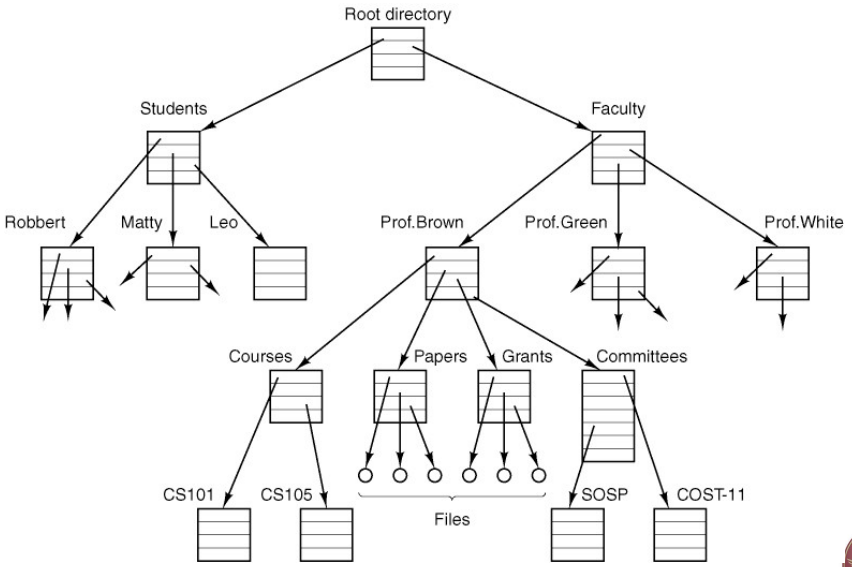
Manual Pages

```
FORK(2)                  Minix Programmer's Manual                  FORK(2)

NAME
     fork - create a new process

SYNOPSIS
     #include <sys/types.h>
     #include <unistd.h>

     pid_t fork(void)

DESCRIPTION
     Fork causes creation of a new process.  The new process  (child  process)
     is an exact copy of the calling process except for the following:

          The child process has a unique process ID.

          The child process has a  different  parent  process  ID  (i.e.,  the
          process ID of the parent process).

          The child process has its own  copy  of  the  parent's  descriptors.
standard-input, 1-24 (Top)
```

File System

- All system entities are abstracted as files
  - Folders and files
  - Commands and applications
  - I/O devices
  - Memory
  - Process communication
- The file system is hierarchical
  - Folders and files construct a tree structure
  - The root of the tree is represented using the /
- The actual structure of the tree depends on the distribution of Linux
  - Certain folders and files are standard across all Linux distributions

File System Example



Standard Folders

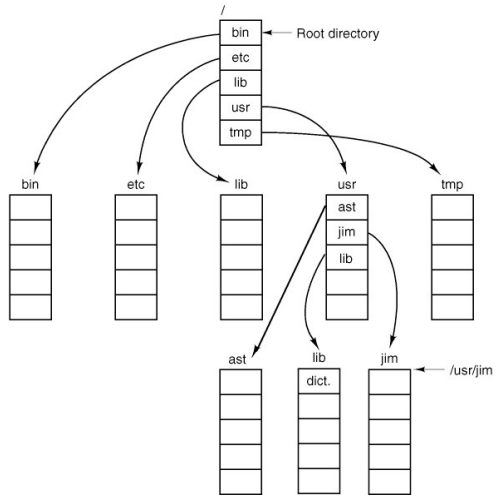
- `/bin` – Basic commands
- `/etc` – System settings
- `/usr` – Applications and Libraries
- `/usr/bin` – Application commands
- `/usr/local` – Applications installed by the local users
- `/sbin` – Administrator commands
- `/var` – Various system files
- `/tmp` – Temporary files
- `/dev` – Devices
- `/boot` – Files needed to start the system
- `/root` – Administrator's folder

Example of File Metadata

```
# ls -la
lrwxrwxrwx  1 bin    operator   2880 Jun  1  1993 bin
-r--r--r--  1 root   operator    448 Jun  1  1993 boot
drwxr-sr-x  2 root   operator  11264 May 11 17:00 dev
drwxr-sr-x 10 root   operator   2560 Jul  8 02:06 etc
drwxrwxrwx  1 bin    bin           7 Jun  1  1993 home
lrwxrwxrwx  1 root   operator      7 Jun  1  1993 lib
drwxr-sr-x  2 root   operator    512 Jul 23  1992 mnt
drwx------  2 root   operator    512 Sep 26  1993 root
drwxr-sr-x  2 bin    operator    512 Jun  1  1993 sbin
drwxrwxrwx  6 root   operator    732 Jul  8 19:23 tmp
drwxr-xr-x 27 bin    bin        1024 Jun 14  1993 usr
drwxr-sr-x 10 root   operator    512 Jul 23  1992 var
```

Navigating the File System

- Each folder contains two "virtual" folders
  `ls -la`
  `.  ..`
- The single dot represents the same folder
  `./myfile` ⇒ `myfile`
- The two dots represent the "parent" folder in the tree



File System Security

- For each file we have 16 bit to define authorization
  - 12 bit are used by the operator
  - They are split in 4 groups of 3 bit – 1 octal – each
- The first 4 bit cannot be changed
  - They characterize the type of the file (simple file, folder, symbolic link)
  - When we list the contents of a folder the first letter is used to signify:
    - – simple files
    d – folders
    l – symbolic links
- The next 3 bit are known as the s-bits and t-bit
- The last three groups are used to define the access writes for read 'r', write 'w' and execute 'x'
  - For the file owner, users of the same group, and all other users.

File System Permissions Examples

```
Type Owner Group Anyone
  d  rwx   r-x   ---
```

- Folder
- The owner has full access
- All users that belong to the group defined by the file can read and execute the file – but not modify the contents
- All other users cannot access the file or execute it
- To access a folder we use the command *cd* given that we have permission to execute 'x'

Changing the File Permissions

Examples of File Permissions

Binary	Octal	Text
001	1	x
010	2	w
100	4	r
110	6	rw-
101	5	r-x
-	644	rw-r--r--

- The command *chmod* allows to modify the permissions
- There are 2 way to define the new permissions
  1. Defining the 3 Octal – e.g., *644*
  2. By using text – e.g., *a+r*

Some Examples of *chmod*

```
make read/write-able for everyone
# chmod a+w myfile

add the 'execute' flag for directory
# chmod u+x mydir/

open all files for everyone
# chmod 755 *

make file readonly for group
# chmod g-w myfile

descend recursively into directory opening all files
# chmod -R a+r mydir/
```

Changing the Owner and Group of a File

- The command *chown* allows to change the owner of a file
- The command *chgrp* allows to change the group of a file

```
give ownership to ichatz
# chown ichatz myfile

set group to students
# chgrp students mydir/

give ownership to pcs and group to students
# chgrp pcs:students myfile mydir/

descend recursively into directory opening all files
# chown -R ichatz mydir/
```

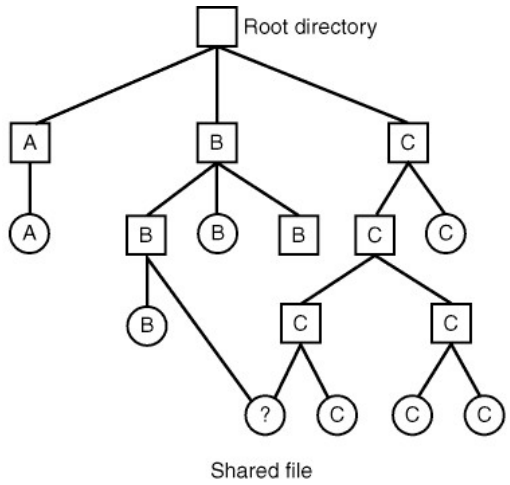
Symbolic Links

- The file system enables to create symbolic links
- Two types are provided
  - Symbolic link
  - Hard link
- The contents and metadata of the original file are used for all operations

```
create a symbolic link to a directory
# ln -s /var/log ./log
# ls -lg
lrwxrwxrwx 1 operator    8 Apr 25  log -> /var/log
```

- The contents and metadata of the original file are used for all operations
  - Except for deletion.

Examples of Symbolic Links



Shared file

Access Dates

- For each file the system keeps track of
  - Date of last usage/access
  - Date of last change

```
check last usage time
# ls -lu
drwxrwxrwx  1 bin     bin           7 Apr 25  1993 home
lrwxrwxrwx  1 root    operator      7 Apr 25  1993 lib
drwx------  2 root    operator    512 Mar 30  1993 root

check last change time
# ls -lc
drwxrwxrwx  1 bin     bin           7 Apr 25  1993 home
lrwxrwxrwx  1 root    operator      7 Oct 27  1993 lib
drwx------  2 root    operator    512 Oct 27  1993 root
```

Keep the software up-to-date

- Use the command line to update the software.
- Commands need to be executed as super user:

```
sudo ...
```

- Use apt tool to update software repository sources.

```
sudo apt update
```

- View updates available:

```
sudo apt list --upgradable
```

- Install updates:

```
sudo apt upgrade
```

Install Python - Jupyter ToolChain

- Use the command line to install the toolchain as super user.
- Install python3 pip using Ubuntu admin tool apt

```
sudo apt install python3-pip
```

- Use pip3 to install jupyter notebook.

```
sudo pip3 install notebook
```

- Modify Security Configuration
  - Allow traffic to port 8888.
- Identify Private IPv4 Address

```
jupyter notebook --ip=<private address>
```

Connect EC2 with S3

- To connect to your S3 buckets from your EC2 instances, you need to do the following:
  1. Create and attach an AWS Identity and Access Management (IAM) profile role to the instance that grants access to Amazon S3.
  2. Confirm that the S3 bucket policy doesn't have a policy denying access.
  3. Confirm network connectivity between the EC2 instance and Amazon S3.
- Install AWS CLI – AWS Command Line tool

```
sudo apt install awscli
```

- Access the S3 bucket

```
aws s3 ls
```

UNIX Shell

- The shell
  - Allows the execution of command scripts
  - Enables alternative methods to carry out complex tasks
  - Provides variables
- Various types of shells exist, e.g., korn, tcsh, zsh ...
- Every user has a preselected shell
  - The selection is stored in the file */etc/passwd*
    `ichatz:x:1000:1000:,,,:/home/ichatz:/bin/bash`
  - The command *chsh* allows to change the preselected shell
- Each shell uses a specific file for user settings

BASH Script Example

```
$ for dir in $PATH
>do
>  if [ -x $dir/gcc ]
>  then
>     echo Found $dir/gcc
>     break
>  else
>     echo Searching $dir/gcc
>  fi
>done
```

- For each folder within the variable $PATH
- Check if the folder contains the file gcc
  - If the file is found, print out the *path* and stop
  - Otherwise continue to the next folder.

Command line

```
# bash
bash-4.4.20#
```

- Left part of # can be changed.
- Right part of # is used to type in commands.
- Offers certain built-in commands
  - Implemented within the BASH source code
  - These commands are executed within the BASH process
- Allows to execute scripts
  - For this reason it is called a UNIX programming environment

Built-in Commands

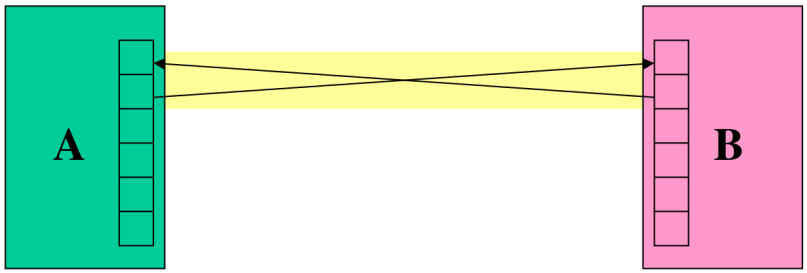
Command	Description	Exception
cd	Change Folder	`cd ..`
declare	Set a variable	`declare myvar`
echo	Print out a text to the standard output	`echo hello`
exec	Replace bash with another process	`exec ls`
exit	Terminate shell process	`exit`
export	Set a global variable	`export myvar=1`
history	List of command history	`history`
kill	Send a message to a process	`kill 1121`
let	Evaluate an arithmetic expression	`let myvar=3+5`

Built-in Commands

Command	Description	Exception
local	Declare a local variable	`local myvar=5`
pwd	The current folder	`pwd`
read	Read a value from standard input	`read myvar`
readonly	Lock the contents of a variable	`readonly myvar`
return	Complete a function call and return a value	`return 1`
set	List declared variables	`set`
shift	Shifts the command parameters	`shift 2`
test	Evaluate an expression	`test -d temp`
trap	Monitor a signal	`trap "echo Signal" 3`

UNIX Pipes

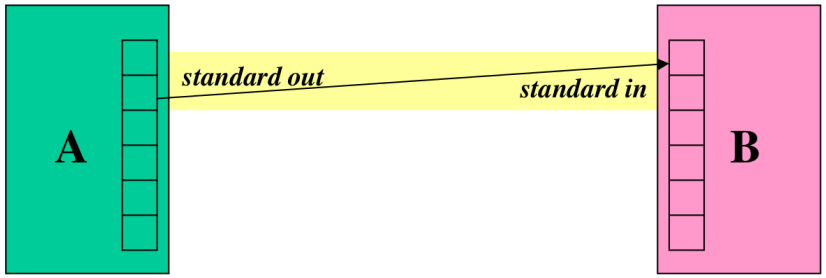
- General idea: The input of one program is the output of the other, and vice versa.



- Both programs run at the same time.

UNIX Pipes

- Often, only one end of the pipe is used.



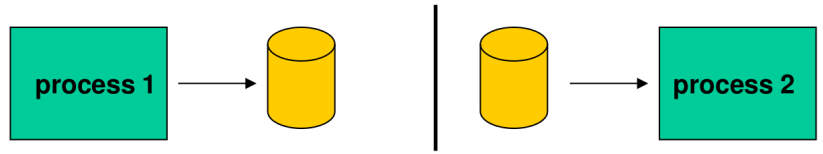
- This can be done using intermediate files.

UNIX Pipes

- Commands produce an output – using the descriptor > the output is redirected to a file
  `# ls > filelist`
- A new file is created under the name filelist
- If the file already exists, the new file will replace the old one.
- We can use the descriptor >> to redirect the output to an existing file
  `# ls -lt /root/doc >> /root/filelist`
- The commands that require input – using the descriptor < the input is redirected from a file
  `# sort < /root/filelist`

UNIX Pipes

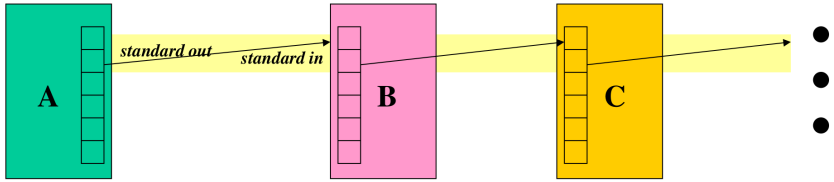
- File approach: Run first program, save output into file.
- Run second program, using file as input.



- Unnecessary use of the disk:
  - Slower,
  - Can take up a lot of space.
- Makes no use of multi-tasking.

UNIX Pipes

- The output of a process is redirected as input to another process.

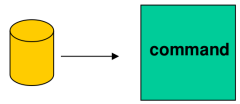


- The redirection is done using the descriptor |
  `# ls | sort` – *sorting the files of a folder*
  `# ls /root | wc -l` – *counting files*
- Multiple pipes are often chained together.

UNIX Pipes

- What's the difference?
- Both commands send input to command from a file instead of the terminal:
  `# cat file | command`
  `# command < file`
- An extra process !
  `# cat file | command`

`# command < file`



UNIX Pipes

- What if a process tries to read data but nothing is available?
  - UNIX puts the reader to sleep until data available.
- What if a process cannot keep up reading from the process that's writing?
  - UNIX keeps a buffer of unread data.
  - This is referred to as the pipe size.
  - If the pipe fills up, UNIX puts the writer to sleep until the reader frees up space (by doing a read).
- Multiple readers and writers possible with pipes.

UNIX Pipes

- Examples of filters:
  - Sort
    - Input: lines from a file.
    - Output: lines from the file sorted.
  - Grep
    - Input: lines from a file.
    - Output: lines that match the argument.
  - Sed
    - Programmable stream editor.

Processes

- We may execute commands in series by using the delimeter ;
  - Commands are executed one by one. When the first is completed, the next one starts. When the last command is completed, we get a new prompt
  - `# who | sort ; date`
- We may execute commands in the background using the delimeter &
  - The commands are executed and a new prompt is provided immediately
  - `# pr junk | lpr &`
- The execution of a command results to a new process
  - The command *ps* shows up in the list of active processes
  - The command *wait* is active until all the commands executed using the delimeter & complete.

List of processes

```
# ps -a
  PID TTY  TIME CMD
  106  c1  0:01 -sh
 4114  co  0:00 /bin/sh /usr/bin/packman
 2114  co  0:00 -sh
 6762  c1  0:00 ps -a
   87  c2  0:00 getty
   90  c3  0:00 getty
```

- Parameter a – list all the commands created by consoles
- Column PID – unique ID of the process
- Column TTY – the console ID that created the process
- Column TIME – total execution time
- Column CMD – the name of the command

Process management

- To terminate a process we use the command *kill [PID]*
- We may change the priority of a process
  - prefix *nice*
    `# nice pr junk | lpr &`
- We may delay the execution of a command
  - prefix *at*

```
# at 1500
ls -l / /root /dir | wc > allfiles
pr allfiles | lpr ; date > lpr-endtime &
date > lpr-starttime
^D
at: /usr/spool/at/07.111.1500.67 created
#
```

The echo command (1)

- Main way to produce output
- Prints out values of variables
- Recognizes special characters (or meta-characters)

```
bash-4.4.20# echo hello there
hello there
bash-4.4.20# let myvar=1; echo $myvar
1
bash-4.4.20# echo *
junk lpr-starttime temp
bash-4.4.20# echo print '*' "don't"
print * don't
```

The echo command (2)

- May contain more than 1 lines
- May also execute commands

```
bash-4.4.20# echo 'hello
there'
hello
there
bash-4.4.20# echo hello\
 there
hello there
bash-4.4.20# echo `date`
Mon Apr 30 16:12:21 GMT 2007
bash-4.4.20# echo -n `date` " "
Mon Apr 30 16:12:21 GMT 2007 bash-4.4.20#
```

Meta-characters

- The character *?* – defines any single character, e.g., `ls /etc/rc.????`
- The character * – defines multiple characters, e.g., `ls /etc/rc.*`
- The array *[...]* – defines a specific set of characters, e.g. `ls [abc].c`
- The use of the above meta-characters is also called filename substitution
- We may use these meta-characters in any combination within command execution
- The following command is disabled `mv *.x *.y`

Shell Variables

- The shell allows the declaration of variables
- Initial values of variables are defined in the user settings file
- The scope of the variables is connected with the session
  - Or until the user removes them
- The variables with UPPER-case letters are global – they are transfered to all processes executed by the shell
- The variables with LOWER-case letters are local – they are accessible only by the shell process

```
HOME            # The path to your home directory
term            # The terminal type
```

Shell Variables

- We may use variables at the command line
- We use the descriptor $

```
bash-4.4.20# myvar="hello"; echo $myvar
hello
bash-4.4.20# myvar="ls -la"
bash-4.4.20# $myvar
lrwxrwxrwx  1 bin    operator   2880 Jun  1  1993 bin
-r--r--r--  1 root   operator    448 Jun  1  1993 boot
drwxr-sr-x  2 root   operator  11264 May 11 17:00 dev
...
```

Special Variables

- Some special variables are provided

Variable	Description
USER	User name
HOME	Home folder of user
TERM	Type of terminal
SHELL	Name of shell
PATH	List of folders to look for commands
MANPATH	List of folders to look for manual pages
PWD	Active folder
OLDPWD	Previously active folder
HOSTNAME	Name of the system

Variable Handling

- The commands *env*, *printenv* provide a list of GLOBAL variables
- The command *set* provides a list of LOCAL variables
- To declare a new GLOBAL variable we use the command *export*
- Variable type is define by content type
  - String variables – `myvar = "value"`
  - Integer variables – `declare -i myvar`
  - Constant variables – `readonly me="ichatz"`
  - Array variables – `declare -a MYARRAY`
    `MYARRAY[0]="one"; MYARRAY[1]=5; echo ${MYARRAY[*]}`
- The names of the variables are case-sensitive
- The command *unset* removes a variable

Creation of scripts

- Scripts are used as if they were commands/applications
  - Defined by a source file
- We execute the script using the command *sh*
  - Or directly by setting execute access permissions

```
bash-4.4.20# echo 'who | wc -l' > nu
bash-4.4.20# cat nu
who | wc -l
bash-4.4.20# sh nu
      1
bash-4.4.20# chmod a+x nu
bash-4.4.20# nu
      1
```

Handling (1)

- We may pass parameters to a script at command-line
  - These are called the command-line arguments
- We use arguments as variables

Argument	Description
$0	The name of the script
$1 ... $9	The value of 1st ... 9th argument
$#	Number of arguments
$	All the arguments as string

```
bash-4.4.20# cat nu
echo Files found: `ls -la $1* | wc -l` "($1\*)"
bash-4.4.20# nu /b
Files found: 57 (/b*)
```

Handling Parameters (2)

- In order to access more than 9 parameters
  - We may not use *$10*
- We need to use command *shift x*
  - Shifts the parameters left-wise by *x* positions
  - Shifted parameters are lost (!)

```
bash-4.4.20# cat ten
shift 10
echo $1
echo $* " -- " $#
bash-4.4.20# ten 1 2 3 4 5 6 7 8 9 10
10
10 -- 1
```

Mathematical Expressions

- Allows the evaluation of mathematical expressions using integers
  - Similar with C programming language
  - No need to explicitely declare a variable as an integer
  - We use *expr* rather than *int*

```
((a = a + 1))
a=$((a+1))
a=$(($a+1))
let a = a + 1
let a++
a=`expr $a + 1`
```

If Expressions

```
if [ condition 1 ]; then
  if [[ condition 2 && condition 3]]; then
  ...
  fi
elif [ condition 4 ] || [ condition 5 ] ; then
  ...
else
  ...
fi
```

- The command *test* allows the evaluation of an expression
  - Returns either true or false
  - Supports broad range of expressions
  - e.g., we might check if we have write access to a given file

```
if test -w "$1"; then echo "File $1 is writable"
fi
```

Evaluation using test

Expression	Description
-gt	Greater or equal
-ge	Greater
-lt	Smaller
-le	Smaller or equal
-eg	Equal
-ne	Not Equal
-n str	Size of the string bigger than 0
-z str	Empty string
-d file	The file is a folder
-s file	A non empty file
-f file	The file exists
-r file	Read access to file
-w file	Write access to file
-x file	Execution access to file

Evaluation Example (1)

```
bash-4.4.20# cat check.sh
#!/bin/bash
read -p "Enter a filename: " filename
if [ ! -w "$filename" ]; then
  echo "File is not writeable"
  exit 1

elif [ ! -r "$filename"  ] ; then
  echo "File is not readable"
  exit 1
fi
...
```

Evaluation Example (2)

```
bash-4.4.20# cat check.sh
#!/bin/bash
TMPFILE = "diff.out"

diff $1 $2 > $TMPFILE

if [ ! -s "$TMPFILE" ]; then
  echo "Files are the same"

else
  more $TMPFILE

fi

if [ -f "$TMPFILE" ]; then
  rm -rf $TMPFILE
fi
```

Boolean expressions

```
if [ condition 1 && condition a]; then
  if [ condition 2 || condition b]; then
  ...
  fi
elif [ ! condition 3 ] ; then
  ...
else
  ...
fi
```

For Loop

```
for VAR in <list>
do
  ...
done


for i in 6 3 1 2
do
  echo $i
done | sort -n


for i in *.c
do
  echo $i
done
```

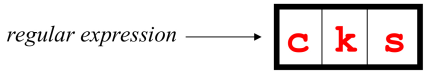
Introduction to Regular Expressions (1)

- A regular expression (regex) describes a set of possible input strings.
- Regular expressions descend from a fundamental concept, in Computer Science called finite automata theory
- Regular expressions are endemic to Unix
  - vi, ed, sed, and emacs
  - awk, tcl, perl and Python
  - grep, egrep, fgrep
  - compilers

Introduction to Regular Expressions (2)

- The simplest regular expressions are a string of literal characters to match.
- The string matches the regular expression if it contains the substring.

Introduction to Regular Expressions (3)



UNIX Tools rocks.

match

UNIX Tools sucks.

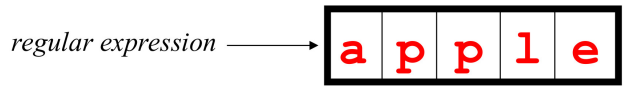
match

UNIX Tools is okay.

no match

Introduction to Regular Expressions (4)

- A regular expression can match a string in more than one place.

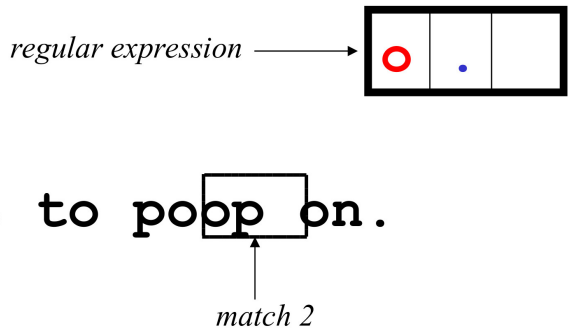


Scrapple from the apple.

match 1 match 2

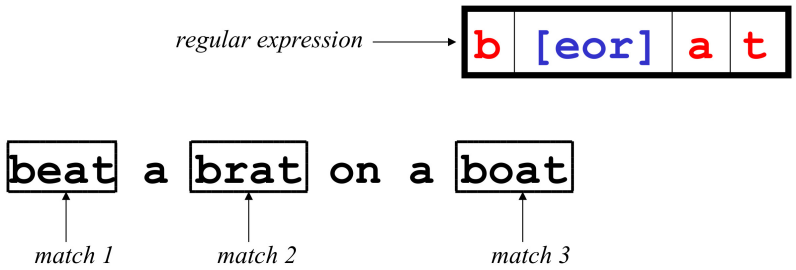
Introduction to Regular Expressions (5)

- The . regular expression can be used to match any character.



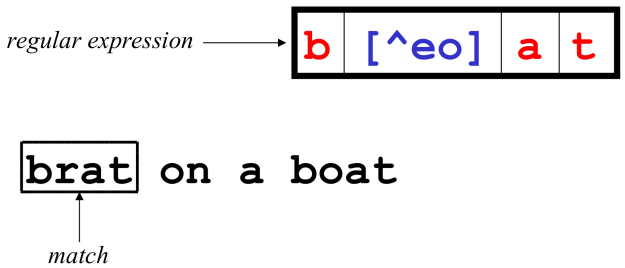
Character Classes (1)

- Character classes [] can be used to match any specific set of characters.



Character Classes (2)

- Character classes can be negated with the [^] syntax.



Character Classes (3)

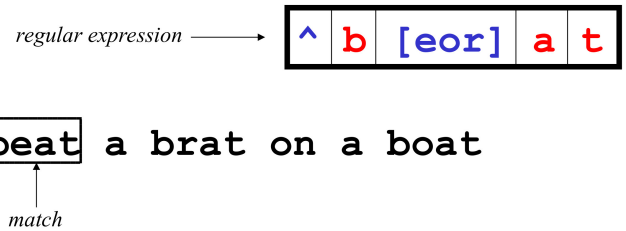
- [*aeiou*] will match any of the characters a, e, i, o, or u
- [*kK*]orn will match korn or Korn
- Ranges can also be specified in character classes
- [1 – 9] is the same as [123456789]
- [*abcde*] is equivalent to [*a* – *e*]
- You can also combine multiple ranges
- [*abcde*123456789] is equivalent to [*a* – *e*1 – 9]
- Note that the - character has a special meaning in a character class but only if it is used within a range,
- [–123] would match the characters -, 1, 2, or 3

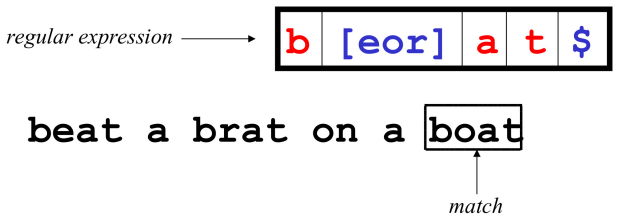
Named Character Classes

- Commonly used character classes can be referred to by name (alpha, lower, upper, alnum, digit, punct, cntrl)
- Syntax [: *name* :]
- $[a-zA-Z]$ is equivalent [[: *alpha* :]]
- $[a-zA-Z0-9]$ is equivalent [[: *alnum* :]]
- $[45a-z]$ is equivalent [45[: *lower* :]]
- Important for portability across languages

Anchor Characters

- Anchors are used to match at the beginning or end of a line (or both).
- ^ means beginning of the line
- $ means end of the line

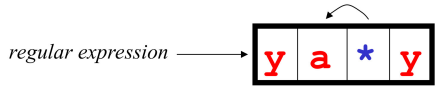




`^word$` `^$`

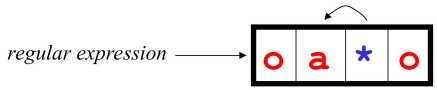
Repetition

- The * is used to define zero or more occurrences of the single regular expression preceding it.



I got mail, yaaaaaaaaaay!

match



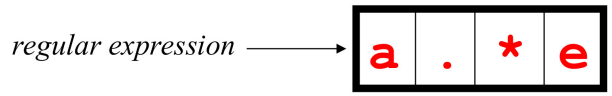
For me to poop on.

match

.*

Match Length

- A match will be the longest string that satisfies the regular expression.



Scrapple from the apple.

no no yes

Repetition Ranges

- Ranges can also be specified
- { } notation can specify a range of repetitions for the immediately preceding regex
- {n} means exactly n occurrences
- {n,} means at least n occurrences
- {n,m} means at least n occurrences but no more than m occurrences
- Example:
  .{0,} same as .*
  a{2,} same as aaa*

Subexpressions

- If you want to group part of an expression so that * or { } applies to more than just the previous character, use ( ) notation
- Subexpresssions are treated like a single character
- a* matches 0 or more occurrences of a
- abc* matches ab, abc, abcc, abccc, ...
- (abc)* matches abc, abcabc, abcabcabc, ...
- (abc)2,3 matches abcabc or abcabcabc

Global Regular Expressions Print – grep

- grep comes from the ed (Unix text editor) search command "global regular expression print" or g/re/p
- This was such a useful command that it was written as a standalone utility
- There are two other variants, egrep and fgrep that comprise the grep family
- grep is the answer to the moments where you know you want the file that contains a specific phrase but you can't remember its name

Syntax

- Regular expression concepts we have seen so far are common to grep
- grep: \(and \), \{ and \}