Linux, R and Awk Tutorial

EMBL Heidelberg

Course Materials

Tobias Rausch
June 2011

Contents

1 Linux Shell Commands **4**
1.1 Introduction 4
1.2 Linux Directories 4
1.3 Linux Command Syntax 5
1.4 Input / Output Redirection and Pipes 6
1.5 What the Shell can do for you 7
1.6 Files 9

2 R Statistics **11**
2.1 Introduction 11
2.2 Summary statistics 11
2.3 Graphics 12
2.4 Data Input / Output 13
2.5 Exiting R 14

3 Awk **15**
3.1 Introduction 15
3.2 Awk Basics 15
3.3 Awk without a Search Pattern 15
3.4 Awk with Program Actions 16
3.5 More than you ever wanted to know about Awk 17

4 Regular Expression **18**
4.1 Introduction 18
4.2 Meta-Characters and Literals 18
4.3 Basic Matching 18
4.3.1 Matching line begin and line end 18
4.3.2 Character classes 19
4.3.3 Matching any Character 19
4.3.4 Matching several Alternatives 19
4.4 Quantifiers 20
4.4.1 Optional matching 20

4.4.2 At least one match 20
4.4.3 Any number of matches, including none 20
4.5 Some Food for Thought 20

Linux Shell Commands

1.1 Introduction

This is a brief Linux command line tutorial. It serves two purposes. First, to get you started on the data analysis of the course and second, to give you a glimpse of the power of linux shell programs. The only prerequisite for this tutorial is that you have access to a linux shell prompt.

```
/home/rausch>
```

1.2 Linux Directories

The Linux filesytem is organized as a tree. The root of the tree is simply labeled as /. A first level subdirectory of / is, for instance, /home or /user. The home directory contains all user directories, such as /home/garfield or /home/snoopy (see Figure 1.1). The command that shows the directory you are currently in is

```
pwd
```

pwd stands for print working directory. Let us assume garfield is in its home directory /home/garfield and wants to create two subdirectories work and freetime. The commands to achieve that are

```
mkdir work
mkdir freetime
```

These commands use so-called relative paths because garfield creates the directories where he is currently in, which is hopefully /home/garfield. Alternatively, garfield can use absolute paths.

```
mkdir /home/garfield/work
mkdir /home/garfield/freetime
```

An absolute path descibes the whole path starting from the root through all subdirectories. Since our workaholic garfield has no freetime we only create two further subdirectories underneath /home/garfield/work.

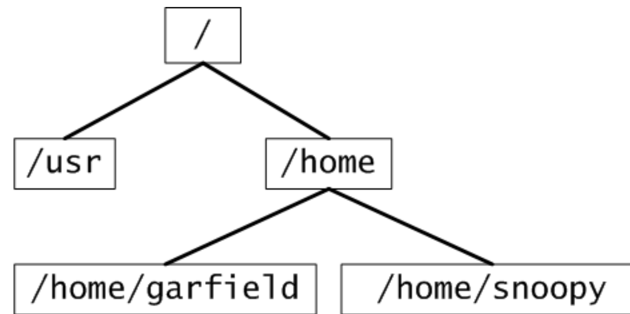

Figure 1.1: Linux directory tree.

```
mkdir ./work/sleep
mkdir /home/garfield/work/food
```

The former command uses a relative path whereas the later command uses an absolute path. Note that a simple . always specifies your current directory. The command cd allows you to move through the directory tree.

```
cd ./work/
pwd
cd /home/garfield/freetime
pwd
cd ..
pwd
```

Whereas a single . specifies your current directory a double .. specifies the parent directory. With these operations one can specify a very complicated command that does absolutely nothing.

```
cd ./work/./sleep/../sleep/../../
```

Finally, there is the ls command that shows the directory content and the rmdir command that removes a directory.

```
ls
ls /home/garfield/work
rmdir /home/garfield/freetime
ls /home/garfield
```

1.3 Linux Command Syntax

All Linux commands share a common structure.

```
<command> <option(s)> <argument(s)>
```

Here are some examples using the ls command.

```
ls -l /home/garfield/
ls -l -R /home/garfield/
ls -lR /home/garfield/
ls /home/garfield/freetime /home/garfield/
ls -I "*rk" /home/garfield
ls --help
ls
```

Depending on the command, arguments and options might be present or absent as shown above. All options are either flags or options taking a value. For example, −R is a flag that tells ls to list the directory content recursively or not. However, −I is an option requiring a value, which specifies a pattern. In the above case, the pattern excludes all files and directories from the listing that end with the letters ”rk”. In this Shell commands tutorial we do not cover regular expression patterns such as "*rk". However, these regular expressions (Friedl, 1997) are immensely powerful and hence, we discuss them in Chaper 4.

1.4 Input / Output Redirection and Pipes

The command echo simply shows something on the screen.

```
echo "Hi"
echo "Hi"
echo -e "Hi\nGarfield"
echo -e "Hi\n\nPrince\nGarfield"
```

The shell understands some meta-characters that, for instance, insert a new line \n or tab \t. These meta-characters are always escaped by \. To tell echo to interpret these meta-characters we switch on the −e option.

```
echo -e "chrX\t3\nchrY\t5"
```

The > symbol redirects the output of a command to a file and the command cat lists the content of a file. Try the following:

```
echo "Hi" > talk.txt
cat talk.txt
echo "Hi Garfield" > talk.txt
cat talk.txt
echo "Bye" >> talk.txt
cat talk.txt
```

Note that a single > redirects the output and deletes all previous contents of that file. To keep previous output you need to take double >> symbols.

The wc command shows the line, word and byte count for a file.

```
wc talk.txt
wc --help
wc -l talk.txt
```

Instead of redirecting the output we can also redirect the input. The < symbol redirects the input.

```
wc -l talk.txt
wc -l < talk.txt
wc -l < talk.txt > linesOfTalk.txt
cat linesOfTalk.txt
```

However, the most powerful feature is the pipe | symbol that "pipes" the ouput of one command into another command.

```
echo "Bla" | wc -l
echo -e "Bla\nBla" | wc -l
```

Almost all linux commands support the --help command line switch. Use it whenever you want to find out what options a command supports.

```
wc --help
```

Unfortunately, the help output of some commands is too large for the screen. But here is the beautiful pipe again.

```
ls --help | less
```

The command less allows you to page through the output of a command. Press q to exit the program less.

1.5 What the Shell can do for you

All the commands you typed so far are interpreted by the shell. The shell offers quite a large number of commands such as cat, ls or wc. In this Section, we want to get to know some very powerful commands such as grep, sort or cut using a simple biological example: snp calling. Let us first create two files with snp calls. You might want to cut and paste the commands below from the pdf.

```
echo -e "chr3\t50\nchr1\t104\nchr1\t80" > indA.txt
echo -e "chr5\t30\nchr1\t104\nchr2\t5" > indB.txt
cat indA.txt
cat indB.txt
```

The command grep searches each line of a file for a certain pattern and outputs all matching lines.

```
grep "chr3" indA.txt
grep "chr5" indA.txt
```

The command sort can be used to numerically or alphanumerically sort a file. The −k option can be used to specify a sorting key. If we use −k2,2 we sort the lines according to column 2. We also need to tell sort to order the lines according to numerical values or dictionary order. The later is the default. If we use −k2,2g we sort the lines according to column 2 in numerical order.

```
sort indA.txt
sort -k2,2g indA.txt
sort -k2,2 indA.txt
sort -k1,1 -k2,2g indA.txt
```

Note that we can define several sorting keys. The first key is the primary sorting key, the second key is the secondary sorting key, and so on. Here is a small example illustrating the difference.

```
echo -e "b\t300\na\t2\na\t300\nb\t31" > s.txt
cat s.txt
sort -k1,1 s.txt
sort -k1,1 -k2,2g s.txt
sort -k2,2g -k1,1 s.txt
```

The command cut can be used to extract columns from a file.

```
cut -f 1 indA.txt
cut -f 2 indA.txt
cut -f 1,2 indA.txt
```

The command uniq can be used to extract unique lines, duplicate lines or to count the number of occurences.

```
cut -f 1 indA.txt | sort | uniq
cut -f 1 indA.txt | sort | uniq -d
cut -f 1 indA.txt | sort | uniq -c
```

The default running mode of uniq is to discard all but one of successive identical lines from the input. Using the −d option it prints only duplicated lines. Using the −c option it counts how often each line occurs. Last but not least, there is the −u option that prints only unique lines.

With this set of tools we can answer quite a number of questions related to our snp example. Here are a few examples.

- How many SNPs are called for each individual?

```
wc -l indA.txt indB.txt
```

- How many SNPs per chromosome?

```
cut -f 1 indA.txt | sort | uniq -c
```

- All SNP coordinates shared by indA and indB?

```
sort indA.txt indB.txt | uniq -d
```

- All SNP coordinates only present in indA or indB?

```
sort indA.txt indB.txt | uniq -u
```

- All SNP coordinates that are only present in indA?

```
sort indB.txt indB.txt indA.txt | uniq -u
```

1.6 Files

The easiest way to create a new file is touch.

```
touch test.txt
```

Files can be edited using an editor. Popular editors for Linux are vi (Lamb and Robbins, 1998) and emacs (Cameron et al., 1996). You can open emacs using

```
emacs -nw test.txt
```

Emacs commands are entered using the Ctrl-Key, which is the Strg-Key on German keyboards. To save your file, for instance, you hold the Ctrl-Key and then press x and s. This is abbreviated as

```
C-x C-s
```

where C stands for holding the Ctrl-Key. To exit emacs use

```
C-x C-c
```

Please enter a couple of text lines in the file test.txt using emacs or an editor of your choice. Then save the file and try the following commands. Remember to press q to exit the less command.

```
cat test.txt
head test.txt
head -n 1 test.txt
head -n 2 test.txt
tail test.txt
tail -n 1 test.txt
head --help
tail --help
less test.txt
```

To move and copy files, Linux offers the mv and cp command.

```
ls
mv test.txt mov.txt
ls
cp mov.txt cop.txt
ls
mv --help | less
cp --help | less
```

Last but not least you can delete your files using rm.

```
ls
rm mov.txt cop.txt
ls
```

So far so good for this brief Linux tutorial. It is certainly worth to explore the shell tools deeper using the Internet or the vast literature (Newham, 2005; Newham et al., 2007; Robbins, 2006).

R Statistics

2.1 Introduction

This is a brief tutorial about using R (www.r-project.org), a free software environment for statistical computing and graphics. The only prerequisite for this tutorial is that you have access to R. For Linux, you just type R and press Enter.

```
/home/rausch> R
```

2.2 Summary statistics

Suppose you have thrown a dice for six times. Then you can store the data in R in a vector called x.

```
x = c(3,2,5,4,5,1)
x
```

Some very handy summary statistics are the mean, the median, the standard deviation, the variance and the range of values.

```
mean(x)
median(x)
sd(x)
var(x)
range(x)
```

Likewise, the minimum and maximum values can be easily computed.

```
min(x)
max(x)
```

If you need help for any of the commands use

```
help(min)
help(sd)
```

You need to press q to exit the help page. Since R is vector-based most functions operate element-wise.

```
x = c(3,2,5,4,5,1)
x
x+3
x / 10
x + x
x==5
```

My advice is to think twice whenever you want to write a loop in R. Usually, a loop is unnecessary and a simple function on the vector works equally well (and is much faster).

2.3 Graphics

To simulate some data we can play with, I briefly introduce the rnorm function here. The rnorm function randomly samples values from a normal distribution using a given mean and standard deviation.

```
samplesize = 10
rnorm(samplesize, mean = 200, sd = 20)
samplesize = 10000
x = rnorm(samplesize, mean = 200, sd = 20)
```

For a good impression of a novel data set you need a picture. A good way to summarize your data is usually a histogram. A histogram bins the values and plots the frequencies of each bin.

```
hist(x)
```

Not suprisingly, the picture looks like a normal distribution. However, be aware that the shape of the histogram largely depends on the bin size.

```
hist(x, breaks=c(0,100,200,300,400))
help(hist)
```

A bit more telling than histograms are boxplots. A boxplot shows the range, the median, the quartiles and the outliers of a data set.

```
boxplot(x)
```

The real power of boxplots becomes apparent in comparative data analyses (see Figure 2.1).

```
samplesize = 10000
x = rnorm(samplesize, mean = 200, sd = 20)
y = rnorm(samplesize, mean = 200, sd = 5)
z = rnorm(samplesize, mean = 400, sd = 10)
boxplot(x,y,z)
```

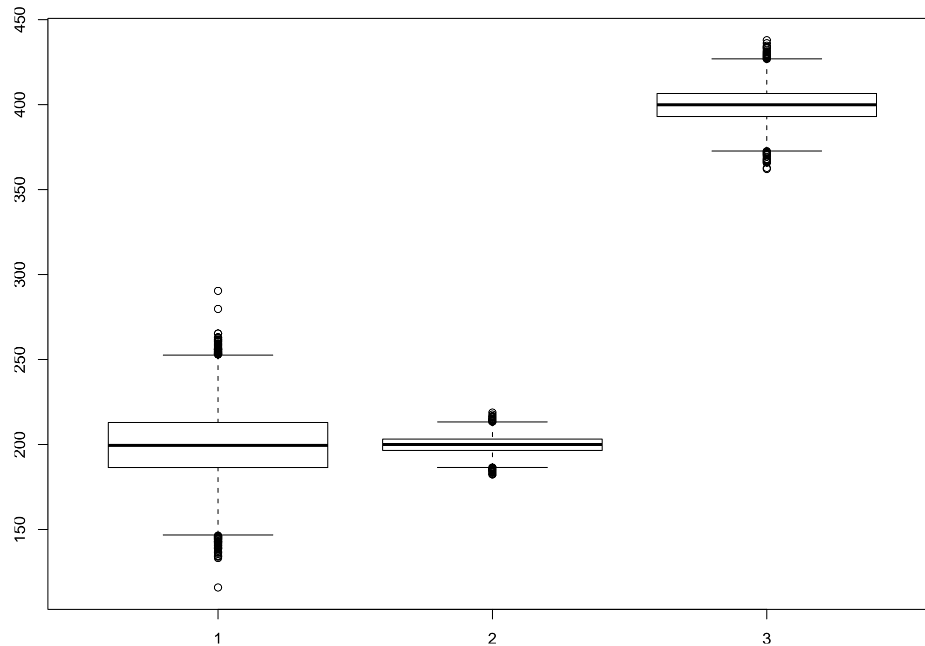

Figure 2.1: Boxplot of three data sets.

For two-dimensional data scatter plots are useful. For instance, given a group of people with a specific height and weight we can plot height against weight.

```
height=c(160,167,202,180)
weight=c(60,70,110, 80)
plot(height, weight)
```

The best thing about R is that you can extend it by your own functions. For example, you can write a function that computes the cumulative frequency and plots it against the data values.

```
edf=function(x) {
plot(sort(x), (1:length(x))/length(x), type="l")
}
edf(x)
```

2.4 Data Input / Output

To export the output of R, we simply redirect the screen to a file.

```
samplesize = 10000
x = rnorm(samplesize, mean = 200, sd = 20)
y = rnorm(samplesize, mean = 200, sd = 5)
z = rnorm(samplesize, mean = 400, sd = 5)
postscript("test.ps")
boxplot(x,y,z)
dev.off()
```

The postscript command opens a new device, which is in this case a file. To enable afterwards printing to the screen again we need to close the postscript device. This is done using the command dev.off(). To read in a file of numbers you can use scan.

```
x = scan("position.txt")
```

To read in a tab-delimited table of numbers you can use read.table.

```
x = read.table("position.txt")
```

2.5 Exiting R

Well, as usual quit does the job.

```
q()
```

For more information about R you can either use the web or one of the following books (Gentleman et al., 2005; Crawley, 2007).

Awk

3.1 Introduction

Similar to the Perl (www.perl.org) programming language Awk is a language for processing files of text. Awk is a full-featured programming language that can be used to write well-structured large programs. However, this tutorial only addresses so-called handy one-liner programs written in Awk. The great benefit of these one-liners is that they can be integrated into a unix pipe. That is why you should read Chapter 1 first before you read this Chapter. Since implementations of Awk exist on almost all Unix-like operating systems you just need access to a Linux command prompt for this tutorial.

```
/home/rausch> awk
```

3.2 Awk Basics

The good thing about Awk is that there is only one syntactic construct to understand.

```
awk <search pattern> {<program actions>}
```

Each line of the input data is tested against the search pattern. If the line matches the search pattern the commands in parantheses are executed. There are two special search patterns BEGIN and END. The program actions after BEGIN are executed before any line of the input data has been processed and the program actions after END are executed after all lines of the input data have been processed.

3.3 Awk without a Search Pattern

Surprisingly it makes sense to use Awk without a search pattern. This means every input line causes the program actions to be executed. To have some example data we create a simple coverage file. The file contains for non-

overlapping 100bp windows the number of reads falling into that window. You might want to copy and paste the following command-line to avoid typos.

```
echo -e "X\t0\t100\t2\nX\t100\t200\t4\nY\t0\t100\t3" > cov
```

Some simple awk one-liners without a search pattern would be.

```
awk '{print $1;}' cov
awk '{print $2;}' cov
awk '{print $4;}' cov
awk '{print $0;}' cov
```

As you can see Awk splits a tab-delimited file into variables. $0 contains the full input line, $1 column 1, $2 column 2, and so on. So to switch column 1 and 4 we simply type.

```
awk '{print $4"\t"$2"\t"$3"\t"$1;}' cov
```

The great thing about awk is that you can use it within pipes. The following command first sorts the file according to column 4 and then adds a line number as the first column. NR is a special variable that holds the line number of the input data.

```
sort -k4,4g cov
sort -k4,4g cov | awk '{print NR":"$0;}'
```

With that information you can now easily compute the sum and the mean of all coverage values.

```
awk '{SUM+=$4;} END {print SUM;}' cov
awk '{SUM+=$4;} END {print SUM / NR;}' cov
```

For each line we add the fourth column to the variable SUM and when all lines have been process we simply output the sum. For the mean we have to divide by the number of lines, which is the final value of NR. The default initialization of SUM is 0 but we can make this also explicit using the BEGIN keyword.

```
awk 'BEGIN {SUM=0} {SUM+=$4;} END {print SUM;}' cov
awk 'BEGIN {SUM=1} {SUM+=$4;} END {print SUM;}' cov
```

3.4 Awk with Program Actions

You might guess it already it also makes sense to execute an Awk one-liner without any program actions. The default program action is to print the full input line. Here are some examples.

```
awk '/X/' cov
awk '/[Y4]/' cov
awk '$1=="X"' cov
awk '$4>3' cov
```

The first two examples use again simple regular expressions (see Chapter 4). The pattern is enclosed in / characters. The first pattern simply looks for an X somewhere on the input line. The second pattern looks for a Y or a 4 somewhere in the input line. The pattern search either returns true if the pattern was found or false otherwise. Because of that we can also use a comparison as a pseudo search pattern. If the comparison is true for the input line the line is printed otherwise it is discarded. So the last example prints all lines where the fourth column is greater 3.

3.5 More than you ever wanted to know about Awk

Obviously, we can now easily count all lines containing X or Y. Here are the two commands.

```
awk 'BEGIN {c=0} /X/ {c++} END {print c}' cov
awk 'BEGIN {c=0} /Y/ {c++} END {print c}' cov
```

Another example would be to extract the window of maximum coverage.

```
awk '$4>max {max=$4; maxline=$0} END {print maxline}' cov
```

Alright, so here is a last, challenging one. Do not worry if you do not understand what is going on. It is actually a very complicated one and it is just meant to catch your interest to explore the features of Awk a bit more on your own (Aho et al., 1987).

```
awk '{a[i++]=$0} END {while (i--) print a[i] }' cov
```

Regular Expression

4.1 Introduction

This is a brief tutorial on regular expressions (Friedl, 1997), which are often abbreviated as regex. I guess, most of you have used them occasionally already when writing commands such as ls *.txt to list all text files. In this Chapter, we are going to explore regular expressions in the context of Awk. So please make sure you read Chapter 3 first. The only prerequisite for this tutorial is a Unix-like command line and Awk.

```
/home/rausch> awk
```

4.2 Meta-Characters and Literals

Regular expressions provide a flexible way to match certain substrings of a text. The expression itself is written in a formal language consisting of so-called meta-characters and so-called literals. The later are normal text characters whereas the former are a sort of grammar, stating where and in what number the literals have to occur.

4.3 Basic Matching

4.3.1 Matching line begin and line end

Let us create a simple text file and execute the following awk commands.

```
echo -e "mad dog\ndog\nDog\ndoggy" > s.txt
awk '/^dog/' s.txt
awk '/dog$/' s.txt
awk '/^dog$/' s.txt
```

The metacharacter ^ matches the line beginning. Hence, ^dog matches all lines beginning with dog. Likewise, the dollar sign matches all lines ending

with dog and the last awk line simply matches all lines simply containing dog and nothing else.

4.3.2 Character classes

You noticed already that we missed the Dog with a capital D. To list multiple characters that can occur at a certain position one uses so-called character classes.

```
awk '/^[Dd]og/' s.txt
```

Character classes are enclosed in brackets [<characters>]. Sometimes it is much easier to state what should not match instead of what should match. For that reason one can negate the character class with the ^ symbol within the character class.

```
awk '/^[^Dd]/' s.txt
awk '/^[^d]/' s.txt
```

4.3.3 Matching any Character

Last but not least, there is the . metacharacter. The dot matches any character.

```
awk '/^.og/' s.txt
awk '/./' s.txt
```

The last awk command does match any character but not an empty line. How can one match the actual dot then? Well, one needs to tell the regular expression engine that this dot is not a metacharacter. This is done by means of escaping using the \ symbol.

```
echo "bla.txt" | awk '/bla\.txt/'
```

4.3.4 Matching several Alternatives

Sometimes you do not want to match one specific instance but several different alternatives. The metacharacter for that purpose is the | metacharacter.

```
awk '/(dog|mad)/' s.txt
```

You can already see that there is usually more than one alternative to write a regular expression. For example, an expression to match the British and American version of grey / gray, you could use (grey|gray), gr[ea]y or gr(e|a)y.

4.4 Quantifiers

4.4.1 Optional matching

Another example for British and American word confusion is colour / color. The metacharacter ? makes the previous character optional.

```
echo "color" | awk '/colour/'
echo "color" | awk '/colou?r/'
echo "colour" | awk '/color/'
echo "colour" | awk '/colou?r/'
```

4.4.2 At least one match

To specify that something needs to match at least once you can use the + metacharacter. For instance, if you have a list of people and their age you do not know if the age is one, two or three digits long.

```
echo -e "12\nNoAge\n5\n101" | awk '/[0123456789]+/'
```

4.4.3 Any number of matches, including none

The star metacharacter * means that the preceding item can occur any number of times. If we apply it to the last example you can see how the "NoAge" slips through now.

```
echo -e "12\nNoAge\n5\n101" | awk '/[0123456789]*/'
```

4.5 Some Food for Thought

Unfortunately, not all tools support the same set of regular expressions. However, the ones I discussed here are fairly standard and are supported by both awk and perl. Also egrep (a version of grep supporting regular expressions) supports this set of expressions. We can, for instance, rewrite the above example using egrep like this.

```
echo -e "12\nNoAge\n5\n101" | egrep '[0123456789]+'
```

Perl also supports a mechanism called back-referencing. This allows one to use the matched substring later on because it is stored in a variable.
Anyway, to get you started here are a couple of regular expressions. Try to understand what they are trying to match.

```
.*\.txt
[0123456789]?[0123456789]:[0123456789][0123456789](am|pm)
[0123456789]+(\.[0123456789][0123456789])?
3[,\.]14[0123456789]*
```

Bibliography

A. V. Aho, B. W. Kernighan, and P. J. Weinberger. *The AWK programming language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.

D. Cameron, B. Rosenblatt, and E. Raymond. *Learning GNU Emacs (2nd ed.)*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1996.

M. J. Crawley. *The R Book*. Wiley Publishing, 2007.

J. E. F. Friedl. *Mastering Regular Expressions*. O'Reilly & Associates, Sebastopol, California, Jan. 1997.

R. Gentleman, V. Carey, W. Huber, R. Irizarry, and S. Dudoit. *Bioinformatics and Computational Biology Solutions Using R and Bioconductor (Statistics for Biology and Health)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

L. Lamb and A. Robbins. *Learning the vi Editor*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1998.

C. Newham. *Learning The Bash Shell (Nutshell Handbooks)*. O'Reilly & Associates, Inc., 2005.

C. Newham, J. Vossen, C. Albing, and J. Vossen. *Bash Cookbook: Solutions and Examples for Bash Users (Cookbooks (O'Reilly))*. O'Reilly Media, Inc., 2007.

A. Robbins. *Bash Quick Reference*. Oreilly & Associates Inc, 2006.

Index

Awk, 15
Awk Commands
  print, 16
Awk Variables
  BEGIN, 15
  END, 15
  NR, 16

Bash command, 5
  cat, 6
  cd, 5
  cp, 10
  cut, 8
  echo, 6
  grep, 8
  head, 10
  ls, 5
  mkdir, 4
  mv, 10
  pwd, 4
  rm, 10
  sort, 8
  tail, 10
  touch, 9
  uniq, 8
  wc, 7

Command syntax, 5

Directories, 4
  Changing, 5
  Creating, 4
  Listing, 5

Files, 9
  Copying, 10
  Editing, 9
  Listing, 10
  Moving, 10

Input redirection, 7

Output redirection, 6

Pipe, 7
Postscript, 13

R Commands
  boxplot, 12
  hist, 12
  max, 11
  mean, 11
  median, 11
  min, 11
  plot, 13
  postscript, 13
  q, 14
  range, 11
  read.table, 14
  rnorm, 12
  scan, 14
  sd, 11
  var, 11
R Graphics, 12
R Statistics, 11
Regular Expression, 18

Shell commands, 4
Summary statistics, 11