Become Efficient at Linux Command Line -Part 2

Table of Contents

Expanding Skills on Linux

Linux systems come with thousands of command-line programs. Experienced users typically rely on a smaller subset—a toolbox of sorts—that they return to again and again.

Overall, the commands in this chapter serve four common, practical needs for pipelines and other complex commands:

Producing text

- Printing dates, times, sequences of numbers and letters, file paths, repeated strings, and other text to jumpstart your pipelines.

Isolating text

- Extracting any part of a text file with a combination of `grep`, `cut`, `head`, `tail`, and one handy feature of `awk`.

Combining text

- Combining files from top to bottom with `cat` and `tac`, or side by side with `echo` and `paste`. You can also interleave files with `paste` and `diff`.

Transforming text

- Converting text into other text using simple commands such as `tr` and `rev`, or more powerful commands such as `awk` and `sed`.

PRODUCING TEXT

Every pipeline begins with a simple command that prints to stdout. Sometimes it's a command like `grep` or `cut` that pulls selected data from a file:

```
#Print all usernames and sort them
$ cut -d: -f1 /etc/passwd | sort
```

or even `cat`, which is convenient for piping the full contents of multiple files to other commands:

```
# Total the number of lines
$ cat *.txt | wc -l
```

Other times, the initial text in a pipeline comes from other sources. You already know one such command, `ls`, which prints file and directory names and associated information. Let's take a look at some other text-producing commands and techniques:

date

- Prints dates and times in various formats

seq

- Prints a sequence of numbers

Brace expansion

- A shell feature that prints a sequence of numbers or characters

find

- Prints file paths

yes

- Prints the same line repeatedly

The date Command

The `date` command prints the current date and/or time in various formats:

```
# Default formatting
$ date
Mon Jun 28 16:57:33 EDT 2021
# Year-Month-Day format
$ date +%Y-%m-%d
2021-06-28
# Hour:Minute:Seconds format
$ date +%H:%M:%S
16:57:33
```

To control the output format, provide an argument that begins with a plus sign (`+`) followed by any text. The text may contain special expressions that begin with a percent sign (`%`), such as `%Y` for the current four-digit year and `%H` for the current hour on a 24-hour clock. A full list of expressions is on the manpage for `date`.

```
$ date +"I cannot believe it's already %A!"
I cannot believe it's already Tuesday!
```

The seq Command

The `seq` command prints a sequence of numbers in a range. Provide two arguments, the low and high values of the range, and `seq` prints the whole range:

```
#Print all integers from 1 to 5, inclusive
$ seq 1 5
1
2
3
4
5
```

If you provide three arguments, the first and third define the range, and the middle number is the increment:

```
Increment by 2 instead of 1
$ seq 1 2 10
1
3
5
7
9
```

Use a negative increment such as `-1` to produce a descending sequence:

```
$ seq 3 -1 0
3
2
1
0
```

or a decimal increment to produce floating-point numbers:

```
#Increment by 0.1
$ seq 1.1 0.1 2
1.1
1.2
1.3
⋮
2.0
```

By default, values are separated by a newline character, but you can change the separator with the `-s` option followed by any string:

```
#Separate values with forward slashes
$ seq -s/ 1 5
1/2/3/4/5
```

The option `-w` makes all values the same width (in characters) by adding leading zeros as needed:

```
$ seq -w 8 10
08
09
10
```

Brace Expansion (A Shell Feature)

The shell provides its own way to print a sequence of numbers, known as *brace expansion*. Start with a left curly brace, add two integers separated by two dots, and end with a right curly brace:

```
# Forward from 1
$ echo {1..10}
1 2 3 4 5 6 7 8 9 10

# Backward from 10
$ echo {10..1}
10 9 8 7 6 5 4 3 2 1

# With leading zeros (equal width)
$ echo {01..10}
01 02 03 04 05 06 07 08 09 10
```

More generally, the shell expression `{x..y..z}` generates the values *x* through *y*, incrementing by *z*:

```
# Count by hundreds from 1
$ echo {1..1000..100}
1 101 201 301 401 501 601 701 801 901

# Backward from 1000
$ echo {1000..1..100}
1000 900 800 700 600 500 400 300 200 100

# With leading zeros
$ echo {01..1000..100}
0001 0101 0201 0301 0401 0501 0601 0701 0801 0901
```

Brace expansion also can produce sequences of letters, which `seq` cannot:

```
$ echo {A..Z}
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

Brace expansion always produces output on a single line separated by space characters. Change this by piping the output to other commands, such as `tr`

```
# Delete spaces
$ echo {A..Z} | tr -d ' '
ABCDEFGHIJKLMNOPQRSTUVWXYZ

# Change spaces into newlines
$ echo {A..Z} | tr ' ' '\n'
A
B
C
⋮
Z
```

Create an alias that prints the *n*th letter of the English alphabet:

```
$ alias nth="echo {A..Z} | tr -d ' ' | cut -c"
$ nth 10
J
```

The find Command

The `find` command lists files in a directory recursively, descending into subdirectories and printing full paths. Results are not alphabetical (pipe the output to `sort` if needed):

```
# List all of /etc recursively
$ find /etc -print
/etc
/etc/issue.net
/etc/nanorc
/etc/apache2
/etc/apache2/sites-available
/etc/apache2/sites-available/default.conf
⋮
```

`find` has numerous options that you can combine. Here are a few highly useful ones. Limit the output only to files or directories with the option `-type`:

```
# Files only
$ find . -type f -print
# Directories only
$ find . -type d -print
```

Limit the output to names that match a filename pattern with the option `-name`. Quote or escape the pattern so the shell doesn't evaluate it first:

```
# Files ending with .conf
$ find /etc -type f -name "*.conf" -print
/etc/logrotate.conf
/etc/systemd/logind.conf
/etc/systemd/timesyncd.conf
⋮
```

Make the name-matching case insensitive with the option `-iname`:

```
$ find . -iname "*.txt" -print
```

`find` has sensible defaults. The default directory is the current directory, and the default action is `-print`, so the following commands are equivalent:

```
$ find . -iname "*.txt" -print
$ find -iname "*.txt"
```

`find` can also execute a Linux command for *each file path* in the output, using `-exec`. The syntax is a bit wonky:

1. Construct a `find` command and omit `-print`.
2. Append `-exec` followed by the command to execute. Use the expression `{}` to indicate where the file path should appear in the command.
3. End with a quoted or escaped semicolon, such as `";"` or `\;`.

Here's a toy example to print an `@` symbol on either side of the file path:

```
$ find /etc -exec echo @ {} @ ";"
@ /etc @
@ /etc/issue.net @
@ /etc/nanorc @
⋮
```

A more practical example performs a long listing (`ls -l`) for all *.conf* files in */etc* and its subdirectories:

```
$ find /etc -type f -name "*.conf" -exec ls -l {} ";"
-rw-r--r-- 1 root root 703  Aug 21  2017 /etc/logrotate.conf
-rw-r--r-- 1 root root 1022 Apr 20  2018 /etc/systemd/logind.conf
-rw-r--r-- 1 root root 604  Apr 20  2018
/etc/systemd/timesyncd.conf
⋮
```

`find -exec` works well for mass deletions of files throughout a directory hierarchy (but be careful!). Let's delete files with names ending in a tilde (~) within the directory *$HOME/tmp* and its subdirectories. For safety, first run the command `echo rm` to see which files would be deleted, then remove `echo` to delete for real:

```
# echo for safety
$ find $HOME/tmp -type f -name "*~" -exec echo rm {} ";"
rm /home/smith/tmp/file1~
rm /home/smith/tmp/junk/file2~
rm /home/smith/tmp/vm/vm-8.2.0b/lisp/vm-cus-load.el~
# Delete for real
$ find $HOME/tmp -type f -name "*~" -exec rm {} ";"
```

`find` also has some built-in actions, such as `-ls` to list files and `-delete` to remove them, that are more efficient than `-exec ls` or `-exec rm`:

```
# List .conf files like "ls -l"
$ find /etc -type f -name "*.conf" -ls
# Delete tilde files
$ find $HOME/tmp -type f -name "*~" -delete
```

The yes Command

The `yes` command prints the same string over and over until terminated:

```
# Repeats "y" by default
$ yes
y
y
y ^C            Kill the command with Ctrl-C
$ yes woof!     Repeat any other string
woof!
woof!
woof! ^C
```

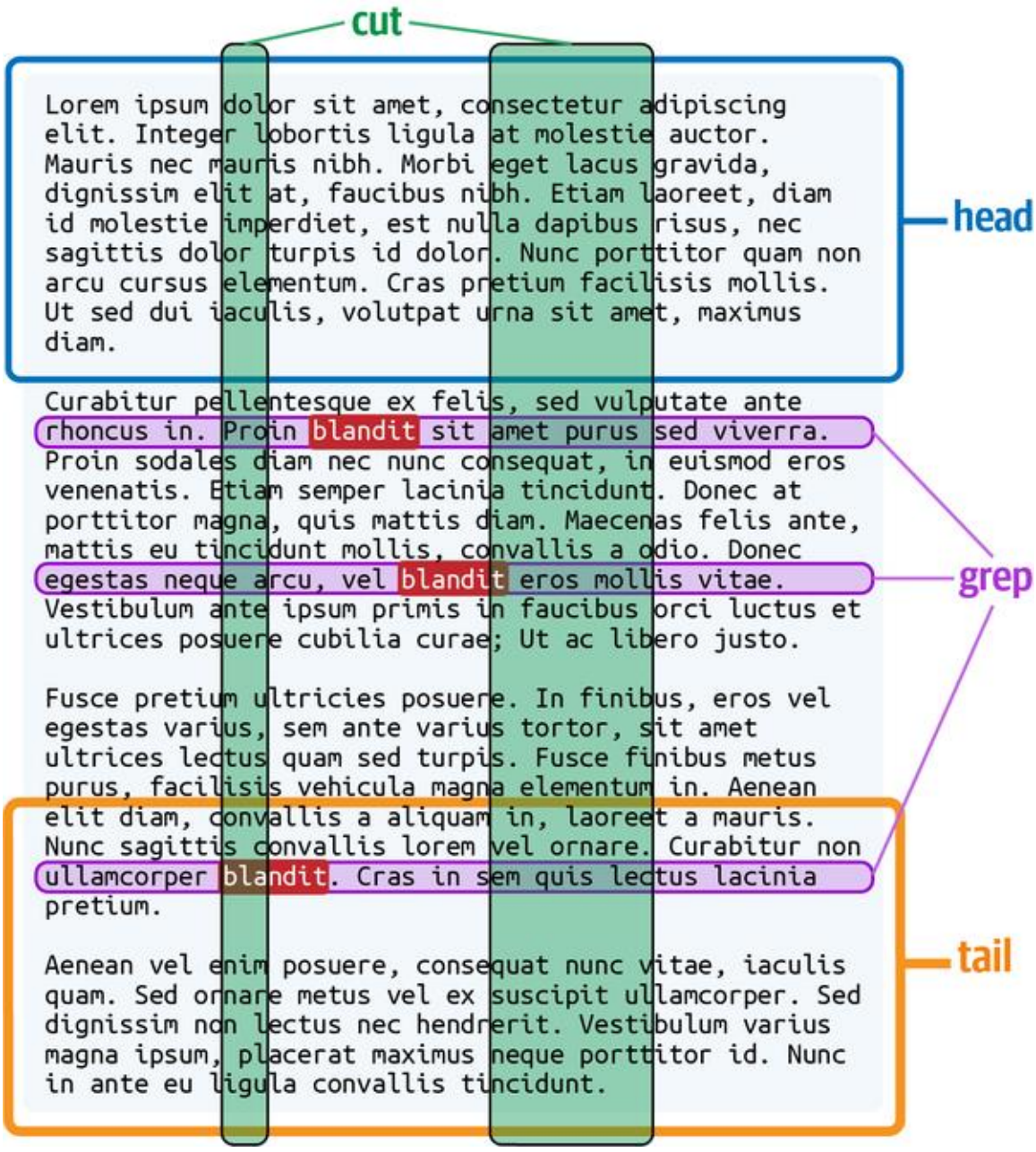
What's the use of this curious behavior? `yes` can supply input to interactive programs so they can run unattended. For example, the program `fsck`, which checks a Linux filesystem for errors, may prompt the user to continue and wait for a response of `y` or `n`. The output of the `yes` command, when piped to `fsck`, responds to every prompt on your behalf, so you can walk away and let `fsck` run to completion

The main use of `yes` for our purposes is printing a string a specific number of times by piping `yes` to `head`

```
# Print a string 3 times
$ yes "Efficient Linux" | head -n3
Efficient Linux
Efficient Linux
Efficient Linux
```

ISOLATING TEXT

When you need just part of a file, the simplest commands to combine and run are `grep`, `cut`, `head`, and `tail`.



In this section, I dive more deeply into `grep`, which does a lot more than match plain strings, and explain `tail` more formally. I also preview one feature of the command `awk` for extracting columns in a way that `cut` cannot. These five commands in combination can isolate pretty much any text using a single pipeline.

You've already seen `grep` print lines from a file that match a given string:

```
$ cat frost
Whose woods these are I think I know.
```

```
His house is in the village though;
He will not see me stopping here
To watch his woods fill up with snow.
This is not the end of the poem.

# Print lines containing "his"
$ grep his frost
To watch his woods fill up with snow.
This is not the end of the poem.              "This" matches "his"
```

`grep` also has some highly useful options. Use the `-w` option to match whole words only:

```
# Match the word "his" exactly
$ grep -w his frost
To watch his woods fill up with snow.
```

Use the `-i` option to ignore the case of letters:

```
$ grep -i his frost
His house is in the village though;              Matches "His"
To watch his woods fill up with snow.            Matches "his"
This is not the end of the poem.               "This" matches "his"
```

Use the `-l` option to print only the names of the files that contain matching lines, but not the matching lines themselves:

```
# Which files contain the string "his"?
$ grep -l his *
frost
```

The real power of `grep`, however, appears when you move beyond matching simple strings to matching patterns, called *regular expressions*

To match this:	Use this syntax:	Example
Beginning of a line	`^`	`^a` = Line beginning with a
End of a line	`$`	`!$` = Line ending with an exclamation point
Any single character (except newline)	`.`	`...` = Any three consecutive characters
A literal caret, dollar sign, or any other special character *c*	`\c`	`\$` = A literal dollar sign
Zero or more occurrences of expression *E*	*`E`*`*`	`_*` = Zero or more underscores
Any single character in a set	`[`*`characters`*`]`	`[aeiouAEIOU]` = Any vowel

Any single character *not* in a set	`[^characters]`	`[^aeiouAEIOU]` = Any nonvowel
Any character in a given range between c_1 and c_2	`[`c_1`-`c_2`]`	`[0-9]` = Any digit
Any character *not* in a given range between c_1 and c_2	`[^`c_1`-`c_2`]`	`[^0-9]` = Any nondigit
Either of two expressions E_1 or E_2	E_1`|`E_2 for grep and sed	`one|two` = Either `one` or `two`
	E_1`|`E_2 for awk	`one|two` = Either `one` or `two`
Grouping expression E for precedence	`\(`E`\)` for grep and sed	`\(one|two\)*` = Zero or more occurrences of `one` or `two`
	`(`E`)` for awk	`(one|two)*` = Zero or more occurrences of `one` or `two`

Here are some example `grep` commands with regular expressions. Match all lines that begin with a capital letter:

```
$ grep '^[A-Z]' myfile
```

Match all nonblank lines (i.e., match blank lines and use `-v` to omit them):

```
$ grep -v '^$' myfile
```

Match all lines that contain either *cookie* or *cake*:

```
$ grep 'cookie\|cake' myfile
```

Match all lines at least five characters long:

```
$ grep '.....' myfile
```

Match all lines in which a less-than symbol appears somewhere before a greater-than symbol, such as lines of HTML code:

```
$ grep '<.*>' page.html
```

Regular expressions are great, but sometimes they get in the way. Suppose you want to search for the two lines in the *frost* file that contain a `w` followed by a period. The following command produces the wrong results, because a period is a regular expression meaning “any character”:

```
$ grep w. frost
Whose woods these are I think I know.
He will not see me stopping here
To watch his woods fill up with snow.
```

To work around this problem, you can escape the special character:

```
$ grep 'w\.' frost
Whose woods these are I think I know.
To watch his woods fill up with snow.
```

but this solution becomes cumbersome if you have many special characters to escape. Fortunately, you can force `grep` to forget about regular expressions and search for every character literally in the input by using the `-F` ("fixed") option. (You may also run the equivalent `fgrep` command, but it is officially deprecated.)

```
$ grep -F w. frost
Whose woods these are I think I know.
To watch his woods fill up with snow.
$ fgrep w. frost
Whose woods these are I think I know.
To watch his woods fill up with snow.
```

`grep` has many other options; I'll present just one more that solves a common problem. Use the `-f` option (lowercase; don't confuse it with `-F`) to match against a set of strings rather than a single string. As a practical example, let's list all shells found in the file */etc/passwd*. Each line in */etc/passwd* contains information about a user, organized as colon-separated fields. The final field on each line is the program launched when the user logs in. This program is often but not always a shell:

```
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash                    7th field is a
shell
daemon:x:1:1:daemon:/usr/sbin:/usr/sbin/nologin 7th field is not
a shell
⋮
```

How can you tell if a program is a shell? Well, the file */etc/shells* lists all valid login shells on a Linux system:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/bin/csh
```

So, you can list all valid shells in */etc/passwd* by extracting the seventh field with `cut`, eliminating duplicates with `sort -u`, and checking the results against */etc/shells* with `grep -f`. I also add the `-F` option to be cautious, so all lines in */etc/shells* are taken literally, even if they contain special characters:

```
$ cut -d: -f7 /etc/passwd | sort -u | grep -f /etc/shells -F
/bin/bash
/bin/sh
```

The tail Command

The `tail` command prints the last lines of a file—10 lines by default. It's a partner to the `head` command. Suppose you have a file named *alphabet* containing 26 lines, one per letter:

```
$ cat alphabet
A is for aardvark
B is for bunny
C is for chipmunk
⋮
X is for xenorhabdus
Y is for yak
Z is for zebu
```

Print the last three lines with `tail`. The option `-n` sets the number of lines to be printed, just as it does for `head`:

```
$ tail -n3 alphabet
X is for xenorhabdus
Y is for yak
Z is for zebu
```

If you precede the number with a plus sign (+), printing begins at that line number and proceeds to the end of the file. The following command begins at the 25th line of the file:

```
$ tail -n+25 alphabet
Y is for yak
Z is for zebu
```

Combine `tail` and `head` to print any range of lines from a file. To print the fourth line alone, for example, extract the first four lines and isolate the last one:

```
$ head -n4 alphabet | tail -n1
D is for dingo
```

In general, to print lines *M* through *N*, extract the first *N* lines with `head`, then isolate the last *N-M+1* lines with `tail`. Print lines six through eight of the *alphabet* file:

```
$ head -n8 alphabet | tail -n3
F is for falcon
```

```
G is for gorilla
H is for hawk
```

The awk {print} Command

The command `awk` is a general-purpose text processor with hundreds of uses. Let's preview one small feature, `print`, that extracts columns from a file in ways that `cut` cannot. Consider the system file */etc/hosts*, which includes IP addresses and hostnames separated by any amount of whitespace:

```
$ less /etc/hosts
127.0.0.1       localhost
127.0.1.1           myhost      myhost.example.com
192.168.1.2       frodo
192.168.1.3     gollum
192.168.1.28        gandalf
```

Suppose you want to isolate hostnames by printing the second word on each line. The challenge is that each hostname is preceded by an arbitrary amount of whitespace. `cut` needs its columns either lined up neatly by column number (`-c`) or separated by a single consistent character (`-f`). You need a command to print the second word on each line, which `awk` provides with ease:

```
$ awk '{print $2}' /etc/hosts
localhost
myhost
frodo
gollum
gandalf
```

`awk` refers to any column by a dollar sign followed by the column number: for example, `$7` for the seventh column. If the column number has more than one digit, surround the number with parentheses: for example, `$(25)`. To refer to the final field, use `$NF` ("number of fields"). To refer to the entire line, use `$0`.

`awk` does not print whitespace between values by default. If you want whitespace, separate the values with commas:

```
# No whitespace
$ echo Efficient fun Linux | awk '{print $1 $3}'
EfficientLinux

# Whitespace
$ echo Efficient fun Linux | awk '{print $1, $3}'
Efficient Linux
```

`awk`'s `print` statement is great for processing the output of commands that strays outside tidy columns. An example is `df`, which prints the amount of free and used disk space on a Linux system:

```
$ df / /data
Filesystem      1K-blocks       Used  Available Use% Mounted on
/dev/sda1      1888543276  902295944  890244772  51% /
/dev/sda2      7441141620 1599844268 5466214400  23% /data
```

The column locations may vary depending on the length of the `Filesystem` paths, the disk sizes, and the options you pass to `df`, so you can't reliably extract values with `cut`. With `awk`, however, you can easily isolate (say) the fourth value on each line, representing available disk space:

```
$ df / /data | awk '{print $4}'
Available
890244772
5466214400
```

and even remove the first line (the header) at the same time with a little `awk` magic, printing only line numbers greater than 1:

```
$ df / /data | awk 'FNR>1 {print $4}'
890244772
5466214400
```

If you encounter input separated by something other than space characters, `awk` can change its field separator to any regular expression with the `-F` option:

```
$ echo efficient:::::linux | awk -F':*' '{print $2}'
linux
```

COMBINING TEXT

You already know several commands that combine text from different files. The first is `cat`, which prints the contents of multiple files to stdout. It's a joiner of files top-to-bottom. That's where its name comes from—it con*cat*enates files:

```
$ cat poem1
It is an ancient Mariner,
And he stoppeth one of three.
$ cat poem2
'By thy long grey beard and glittering eye,
$ cat poem3
Now wherefore stopp'st thou me?
$ cat poem1 poem2 poem3
```

```
It is an ancient Mariner,
And he stoppeth one of three.
'By thy long grey beard and glittering eye,
Now wherefore stopp'st thou me?
```

The second command you've seen for combining text is `echo`, the shell builtin that prints whatever arguments you give it, separated by a single space character. It combines strings side by side:

```
$ echo efficient             linux      in      $HOME
efficient linux in /home/smith
```

Let's examine some more commands that combine text:

tac

- A bottom-to-top combiner of text files

paste

- A side-by-side combiner of text files

diff

- A command that interleaves text from two files by printing their differences

The tac Command

The `tac` command reverses a file line by line. Its name is *cat* spelled backward.

```
$ cat poem1 poem2 poem3 | tac
Now wherefore stopp'st thou me?
'By thy long grey beard and glittering eye,
And he stoppeth one of three.
It is an ancient Mariner,
```

Notice I concatenated three files before reversing the text. If I instead provide multiple files to `tac` as arguments, it reverses the lines of each file in turn, producing different output:

```
$ tac poem1 poem2 poem3
And he stoppeth one of three.
It is an ancient Mariner,
'By thy long grey beard and glittering eye,
Now wherefore stopp'st thou me?
```

`tac` is great for processing data that is already in chronological order but not reversible with the `sort -r` command. A typical case is reversing a web-server log file to process its lines from newest to oldest:

```
192.168.1.34 - - [30/Nov/2021:23:37:39 -0500] "GET / HTTP/1.1"
...
192.168.1.10 - - [01/Dec/2021:00:02:11 -0500] "GET /notes.html
HTTP/1.1" ...
192.168.1.8 - - [01/Dec/2021:00:04:30 -0500] "GET /stuff.html
HTTP/1.1" ...
⋮
```

The lines are in chronological order with timestamps, but they aren't in alphabetical or numeric order, so the `sort -r` command isn't helpful. The `tac` command can reverse these lines without needing to consider the timestamps.

The paste Command

The `paste` command combines files side by side in columns separated by a single tab character. It's a partner to the `cut` command, which extracts columns from a tab-separated file:

```
$ cat title-words1
EFFICIENT
AT
COMMAND
$ cat title-words2
linux
the
line
$ paste title-words1 title-words2
EFFICIENT  linux
AT    the
COMMAND line
$ paste title-words1 title-words2 | cut -f2
linux
the
line
```

Change the separator to another character, such as a comma, with the option `-d` (meaning "delimiter"):

```
$ paste -d, title-words1 title-words2
EFFICIENT,linux
AT,the
COMMAND,line
```

Transpose the output, producing pasted rows instead of pasted columns, with the `-s` option:

```
$ paste -d, -s title-words1 title-words2
EFFICIENT,AT,COMMAND
linux,the,line
```

`paste` also interleaves data from two or more files if you change the separator to a newline character (`\n`):

```
$ paste -d "\n" title-words1 title-words2
EFFICIENT
linux
AT
the
COMMAND
line
```

The diff Command

`diff` compares two files line by line and prints a terse report about their differences:

```
$ cat file1
Linux is all about efficiency.
I hope you will enjoy this book.
$ cat file2
MacOS is all about efficiency.
I hope you will enjoy this book.
Have a nice day.
$ diff file1 file2
1c1
< Linux is all about efficiency.
---
> MacOS is all about efficiency.
2a3
> Have a nice day.
```

The notation `1c1` represents a change or difference between the files. It means that line 1 in the first file differs from line 1 in the second file. This notation is followed by the relevant line from *file1*, a three-dash separator (`---`), and the relevant line from *file2*. The leading symbol < always indicates a line from the first file, and > indicates a line from the second file.

The notation `2a3` represents an addition. It means that *file2* has a third line not present after the second line of *file1*. This notation is followed by the extra line from *file2*, "Have a nice day."

`diff` output may contain other notation and can take other forms. This short explanation is enough for our main purpose, however, which is to use `diff` as a text processor that interleaves lines from two files. Many users don't think of `diff` this way, but it's great for forming pipelines to solve certain kinds of problems. For example, you can isolate the differing lines with `grep` and `cut`:

```
$ diff file1 file2 | grep '^[<>]'
< Linux is all about efficiency.
> MacOS is all about efficiency.
> Have a nice day.
$ diff file1 file2 | grep '^[<>]' | cut -c3-
Linux is all about efficiency.
MacOS is all about efficiency.
Have a nice day.
```

TRANSFORMING TEXT

We introduced several commands that read text from stdin and transform it into something else on stdout. `wc` prints a count of lines, words, and characters; `sort` arranges lines into alphabetical or numeric order; and `uniq` consolidates duplicate lines. Let's discuss several more commands that transform their input:

tr

- Translates characters into other characters

rev

- Reverses characters on a line

awk and sed

- General-purpose transformers

The tr Command

`tr` translates one set of characters into another.

```
#Translate colons into newlines
$ echo $PATH | tr : "\n"
/home/smith/bin
/usr/local/bin
/usr/bin
/bin
/usr/games
/usr/lib/java/bin
```

`tr` takes two sets of characters as arguments, and it translates members of the first set into the corresponding members of the second. Common uses are converting text to uppercase or lowercase:

```
#Translate a into A, b into B, etc.
$ echo efficient | tr a-z A-Z
EFFICIENT
$ echo Efficient | tr A-Z a-z
efficient
```

converting spaces into newlines:

```
$ echo Efficient Linux | tr " " "\n"
Efficient
Linux
```

and deleting whitespace with the `-d` (delete) option:

```
# Remove spaces and tabs
$ echo efficient linux | tr -d ' \t'
efficientlinux
```

The rev Command

The `rev` command reverses the characters of each line of input

```
$ echo Efficient Linux! | rev
!xuniL tneiciffE
```

Beyond the obvious entertainment value, `rev` is handy for extracting tricky information from files. Suppose you have a file of celebrity names:

```
$ cat celebrities
Jamie Lee Curtis
Zooey Deschanel
Zendaya Maree Stoermer Coleman
Rihanna
```

and you want to extract the final word on each line (Curtis, Deschanel, Coleman, Rihanna). This would be easy with `cut -f` if each line had the same number of fields, but the number varies. With `rev`, you can reverse all the lines, cut the *first* field, and reverse again to achieve your goal

```
$ rev celebrities
sitruC eeL eimaJ
lenahcseD yeooZ
nameloC remreotS eeraM ayadneZ
```

```
annahiR
$ rev celebrities | cut -d' ' -f1
sitruC
lenahcseD
nameloC
annahiR
$ rev celebrities | cut -d' ' -f1 | rev
Curtis
Deschanel
Coleman
Rihanna
```

The awk and sed Commands

`awk` and `sed` are general-purpose "supercommands" for processing text. They can do most everything that the other commands in this chapter do, but with more cryptic-looking syntax. As a simple example, they can print the first 10 lines of a file like `head` does:

```
# Print 10 lines and quit (q)
$ sed 10q myfile
#Print while line number is ≤ 10
$ awk 'FNR<=10' myfile
```

They can also do things that our other commands cannot, like replace or swap strings:

```
# Replace .jpg by .png
$ echo image.jpg | sed 's/\.jpg/.png/'
image.png
# Swap two words
$ echo "linux efficient" | awk '{print $2, $1}'
efficient linux
```

Success with these commands really means:

- Understanding the *kinds* of transformations, they make possible, so you can think, "Ah! This is a job for `awk` (or `sed`)!" and apply them in your time of need
- Learning to read their manpages

awk essentials

`awk` transforms lines of text from files (or stdin) into any other text, using a sequence of instructions called an *awk program*. You can supply the awk program on the command line:

```
$ awk program input-files
```

You can also store one or more awk programs in files and refer to them with the `-f` option, and the programs run in sequence:

```
$ awk -f program-file1 -f program-file2 -f program-file3 input-
files
```

An awk program includes one or more *actions*, such as calculating values or printing text, that run when an input line matches a *pattern*. Each instruction in the program has the form:

```
pattern {action}
```

Typical patterns include:

The word BEGIN

- Its action runs just once, before `awk` processes any input.

The word END

- Its action runs just once, after `awk` has processed all the input.

A regular expression

- An example is `/^[A-Z]/` to match lines that begin with a capital letter.

Other expressions specific to `awk`

- For example, to check whether the third field on an input line (`$3`) begins with a capital letter, a pattern would be `$3~/^[A-Z]/`. Another example is `FNR>5`, which tells `awk` to skip the first five lines of input.

An action with no pattern runs for every line of input. As an example, `awk` elegantly solves the “print the celebrity’s last name”

```
$ awk '{print $NF}' celebrities
Curtis
Deschanel
Coleman
Rihanna
```

A pattern with no action runs the default action `{print}`, which just prints any matching input lines unchanged:

```
$ echo efficient linux | awk '/efficient/'
efficient linux
```

To produce a tidy bibliography, converting lines from this format

```
python      Programming Python     2010  Lutz, Mark
```

to this format:

```
Lutz, Mark (2010). "Programming Python"
```

animals.txt content

```
python      Programming Python     2010  Lutz, Mark
snail SSH, The Secure Shell 2005  Barrett, Daniel
alpaca      Intermediate Perl      2012  Schwartz, Randal
robin MySQL High Availability      2014  Bell, Charles
horse Linux in a Nutshell    2009  Siever, Ellen
donkey      Cisco IOS in a Nutshell     2005  Boney, James
oryx  Writing Word Macros    1999  Roman, Steven
```

This feat requires rearranging three fields and adding some characters like parentheses and double quotes. The following awk program does the trick, employing the option `-F` to change the input separator from spaces to tabs (`\t`):

```
$ awk -F'\t' '{print $4, "(" $3 ").", "\"" $2 "\""}' animals.txt
Lutz, Mark (2010). "Programming Python"
Barrett, Daniel (2005). "SSH, The Secure Shell"
Schwartz, Randal (2012). "Intermediate Perl"
Bell, Charles (2014). "MySQL High Availability"
Siever, Ellen (2009). "Linux in a Nutshell"
Boney, James (2005). "Cisco IOS in a Nutshell"
Roman, Steven (1999). "Writing Word Macros"
```

Add a regular expression to process only the “horse” book:

```
$ awk -F'\t' '/^horse/{print $4, "(" $3 ").", "\"" $2 "\""}'
animals.txt
Siever, Ellen (2009). "Linux in a Nutshell"
```

Or process only books from 2010 or later, by testing whether field `$3` matches `^201`:

```
$ awk -F'\t' '$3~/^201/{print $4, "(" $3 ").", "\"" $2 "\""}'
animals.txt
Lutz, Mark (2010). "Programming Python"
```

```
Schwartz, Randal (2012). "Intermediate Perl"
Bell, Charles (2014). "MySQL High Availability"
```

Finally, add a `BEGIN` instruction to print a friendly heading, some dashes for indenting, and an `END` instruction to direct the reader to further information:

```
$ awk -F'\t' \
  'BEGIN {print "Recent books:"} \
  $3~/^201/{print "-", $4, "(" $3 ").", "\"" $2 "\""} \
  END {print "For more books, search the web"}' \
  animals.txt
Recent books:
- Lutz, Mark (2010). "Programming Python"
- Schwartz, Randal (2012). "Intermediate Perl"
- Bell, Charles (2014). "MySQL High Availability"
For more books, search the web
```

`awk` does much more than print—it can also perform calculations, like summing the numbers 1 to 100:

```
$ seq 1 100 | awk '{s+=$1} END {print s}'
5050
```

To learn more about refer the rip tutorial https://riptutorial.com/awk

Improving the duplicate file detector

In "Detecting Duplicate Files", you constructed a pipeline that detects and counts duplicate JPEG files by checksum, but it was not powerful enough to print the filenames:

```
$ md5sum *.jpg | cut -c1-32 | sort | uniq -c | sort -nr | grep -v
"      1 "
      3 f6464ed766daca87ba407aede21c8fcc
      2 c7978522c58425f6af3f095ef1de1cd5
      2 146b163929b6533f02e91bdf21cb9563
```

Now that you know `awk`, you have the tools to print the filenames as well. Let's construct a new command that reads each line of `md5sum` output:

```
$ md5sum *.jpg
146b163929b6533f02e91bdf21cb9563  image001.jpg
63da88b3ddde0843c94269638dfa6958  image002.jpg
146b163929b6533f02e91bdf21cb9563  image003.jpg
⋮
```

and not only counts occurrences of each checksum but also stores the filenames for printing. You'll need two additional `awk` features called *arrays* and *loops*.

An *array* is a variable that holds a collection of values. If the array is named `A` and holds seven values, then the values could be accessed as `A[1]`, `A[2]`, `A[3]`, up to `A[7]`. The values 1 through 7 are called the *keys* of the array, and `A[1]` through `A[7]` are called the array's *elements*. You can create any keys you want, however. If you'd rather access the seven elements of your array using the names of Disney characters, go ahead and name them `A["Doc"]`, `A["Grumpy"]`, `A["Bashful"]`, all the way to `A["Dopey"]`.

To count duplicate images, create an array called `counts` with one element for each checksum. Each array key is a checksum, and the associated element holds the number of times that checksum occurs in the input. For example, the array element `counts["f6464ed766daca87ba407aede21c8fcc"]` could have value 3. The following awk script examines each line of `md5sum` output, isolates the checksum (`$1`), and uses it as a key for the `counts` array. The operator `++` increments an element by 1 each time `awk` encounters its associated checksum:

```
$ md5sum *.jpg | awk '{counts[$1]++}'
```

o far, the awk script produces no output—it just counts each checksum and exits. To print the counts, you need a second `awk` feature called a `for` loop. A `for` loop steps through an array, key by key, and processes each element in sequence, using this syntax:

```
for (variable in array) do something with array[variable]
```

For example, print each array element by its key:

```
for (key in counts) print array[key]
```

Place this loop in the `END` instruction so it runs after all the counts are calculated.

```
$ md5sum *.jpg \
  | awk '{counts[$1]++} \
         END {for (key in counts) print counts[key]}'
1
2
2
⋮
```

Next, add the checksums to the output. Each array key is a checksum, so just print the key after the count:

```
$ md5sum *.jpg \
  | awk '{counts[$1]++} \
         END {for (key in counts) print counts[key] " " key}'
1 714eceeb06b43c03fe20eb96474f69b8
2 146b163929b6533f02e91bdf21cb9563
```

```
2 c7978522c58425f6af3f095ef1de1cd5
⋮
```

To collect and print filenames, use a second array, `names`, also with checksums as its keys. As `awk` processes each line of output, append the filename (`$2`) to the corresponding element of the `names` array, along with a space as a separator. In the `END` loop, after printing the checksum (`key`), print a colon and the collected filenames for that checksum:

```
$ md5sum *.jpg \
  | awk '{counts[$1]++; names[$1]=names[$1] " " $2} \
         END {for (key in counts) print counts[key] " " key ":"
names[key]}'
1 714eceeb06b43c03fe20eb96474f69b8: image011.jpg
2 146b163929b6533f02e91bdf21cb9563: image001.jpg image003.jpg
2 c7978522c58425f6af3f095ef1de1cd5: image019.jpg image020.jpg
⋮
```

Lines that begin with 1 represent checksums that occur only once, so they are not duplicates. Pipe the output to `grep -v` to remove these lines, then sort the results numerically, high to low, with `sort -nr` and you have your desired output:

```
$ md5sum *.jpg \
  | awk '{counts[$1]++; names[$1]=names[$1] " " $2} \
         END {for (key in counts) print counts[key] " " key ":"
names[key]}' \
  | grep -v '^1 ' \
  | sort -nr
3 f6464ed766daca87ba407aede21c8fcc: image007.jpg image012.jpg
image014.jpg
2 c7978522c58425f6af3f095ef1de1cd5: image019.jpg image020.jpg
2 146b163929b6533f02e91bdf21cb9563: image001.jpg image003.jpg
```

sed essentials

`sed`, like `awk`, transforms text from files (or stdin) into any other text, using a sequence of instructions called a *sed script*. sed scripts are pretty cryptic on first glance. An example is `s/Windows/Linux/g`, which means to replace every occurrence of the string `Windows` with `Linux`. The word *script* here does not mean a file (like a shell script) but a string. Invoke `sed` with a single script on the command line:

```
$ sed script input-files
```

or use the `-e` option to supply multiple scripts that process the input in sequence:

```
$ sed -e script1 -e script2 -e script3 input-files
```

You can also store sed scripts in files and refer to them with the `-f` option, and they run in sequence:

```
$ sed -f script-file1 -f script-file2 -f script-file3 input-files
```

As with `awk`, the utility of `sed` depends on your skill in creating sed scripts. The most common type of script is a substitution script that replaces strings with other strings. The syntax is:

```
s/regexp/replacement/
```

where *`regexp`* is a regular expression to match against each input line. and *`replacement`* is a string to replace the matched text. As a simple example, change one word into another:

```
$ echo Efficient Windows | sed "s/Windows/Linux/"
Efficient Linux
```

`sed` easily solves the "print the celebrity's last name" problem from "The rev Command" with a regular expression. Just match all characters (`.*`) up to the last space and replace them with nothing:

```
$ sed 's/.* //' celebrities
Curtis
Deschanel
Coleman
Rihanna
```

You may follow a substitution with several options to affect its behavior. The option `i` makes matches case insensitive:

```
# Case sensitive; no match
$ echo Efficient Stuff | sed "s/stuff/linux/"
Efficient Stuff
# Case-insensitive match
$ echo Efficient Stuff | sed "s/stuff/linux/i"
Efficient linux
```

The option `g` ("global") replaces all occurrences of the regular expression instead of just the first one:

```
# Replaces just the first "f"
$ echo efficient stuff | sed "s/f/F/"
eFficient stuff
# Replaces all occurrences of "f"
```

```
$ echo efficient stuff | sed "s/f/F/g"
eFFicient stuFF
```

Another common type of sed script is a deletion script. It removes lines by their line number:

```
# Remove the 4th line
$ seq 10 14 | sed 4d
10
11
12
14
```

or lines that match a regular expression:

```
# Delete lines ending in an odd digit
$ seq 101 200 | sed '/[13579]$/d'
102
104
106
⋮
200
```

Matching subexpressions with sed

Suppose you have some filenames:

```
$ ls
image.jpg.1  image.jpg.2  image.jpg.3
```

and want to produce new names, *image1.jpg*, *image2.jpg*, and *image3.jpg*. `sed` can split the filenames into parts and rearrange them via a feature called *subexpressions*. First, create a regular expression that matches the filenames:

```
image\.jpg\.[1-3]
```

You want to move the final digit earlier in the filename, so isolate that digit by surrounding it with the symbols `\(` and `\)`. This defines a subexpression—a designated part of a regular expression:

```
image\.jpg\.\([1-3]\)
```

`sed` can refer to subexpressions by number and manipulate them. You created only one subexpression, so its name is `\1`. A second subexpression would be `\2`, and so on, up to a maximum of `\9`. Your new filenames would have the form `image\1.jpg`. Therefore, your sed script would be:

```
$ ls | sed "s/image\.jpg\.\([1-3]\)/image\1.jpg/"
image1.jpg
image2.jpg
image3.jpg
```

To make things more complicated, suppose the filenames had more variation, consisting of lowercase words:

```
$ ls
apple.jpg.1  banana.png.2  carrot.jpg.3
```

Create three subexpressions to capture the base filename, extension, and final digit:

```
# \1 = Base filename of one letter or more
\([a-z][a-z]*\)
#\2 = File extension of three letters
\([a-z][a-z][a-z]\)
# \3 = A digit
\([0-9]\)
```

Connect them with escaped dots (`\.`) to form this regular expression:

```
\([a-z][a-z]*\)\.\([a-z][a-z][a-z]\)\.\([0-9]\)
```

Represent the newly transformed filenames to `sed` as `\1\3.\2`, and the final substitution with `sed` becomes:

```
$ ls | sed "s/\([a-z][a-z]*\)\.\([a-z][a-z][a-z]\)\.\([0-
9]\)/\1\3.\2/"
apple1.jpg
banana2.png
carrot3.jpg
```

This command does not rename files—it just prints the new names

Refer https://www.grymoire.com/Unix/Sed.html for more on sed

TOWARD AN EVEN LARGER TOOLBOX

Most Linux systems come with thousands of command-line programs, and most of them have numerous options that change their behavior. You're not likely to learn and remember them all. So, in a moment of need, how do you locate a new program—or tailor a program that you already know—to accomplish your goals?

Your first (obvious) step is a web search engine. For example, if you need a command that limits the width of lines in a text file, wrapping any lines that are too long, search the web for (say) "Linux command wrap lines" and you'll be pointed to the `fold` command:

```
$ cat title.txt
This book is titled "Efficient Linux at the Command Line"
$ fold -w40 title.txt
This book is titled "Efficient Linux at
the Command Line"
```

To discover commands that are already installed on your Linux system, run the command `man -k` (or equivalently, `apropos`). Given a word, `man -k` searches for that word in the brief descriptions at the top of manpages:

```
$ man -k width
DisplayWidth (3)      - image format functions and macros
DisplayWidthMM (3)    - image format functions and macros
fold (1)              - wrap each input line to fit in specified
width
⋮
```

`man -k` accepts `awk`-style regular expressions in search strings

```
$ man -k "wide|width"
```

A command that's not installed on your system might still be installable through your system's package manager. A package manager is software for installing Linux programs that are supported for your system. Some popular package managers include `apt`, `dnf`, `emerge`, `pacman`, `rpm`, `yum`, and `zypper`. Use the `man` command to figure out which package manager is installed on your system and learn how to search for uninstalled packages. Often it's a two-command sequence: one command to copy the latest data about available packages ("metadata") from the internet onto your system, and another to search the metadata. For example, for Ubuntu or Debian Linux-based systems, the commands are:

```
# Download the latest metadata
$ sudo apt update
# Search for a string
$ apt-file search string
```

f, after much searching, you cannot locate or construct an appropriate command to meet your needs, consider asking for help in an online forum. A great starting point for asking effective questions is Stack Overflow's "How Do I Ask a Good Question?" help page. In general, present your questions in a way that is respectful of other people's time, and experts will be more inclined to answer. That means making your question short and to the point, including any error messages or other output word for word, and explaining what you have tried so far on your own. Spend quality time to ask a quality question: you'll not only increase your chances of a helpful answer, but also, if the forum is public and searchable, a clear question and answer may aid others who have a similar problem.

Parents, Children, and Environments

The purpose of the shell—to run commands—is so fundamental to Linux that you might think the shell is built into Linux in some special way. It is not. A shell is just an ordinary program like `ls` or `cat`. It is programmed to repeat the following steps over and over and over and over...

- Print a prompt.
- Read a command from stdin.
- Evaluate and run the command.

Linux does a great job of hiding the fact that a shell is an ordinary program. When you log in, Linux automatically runs an instance of the shell for you, known as your *login shell*. It launches so seamlessly that it appears to *be* Linux, when really, it's just a program launched on your behalf to interact with Linux.

The more you understand about the shell, the more effectively you can work with Linux and the fewer superstitions you'll develop about its inner workings.

- Where shell programs are located
- How different shell instances may be related to each other
- Why different shell instances may have the same variables, values, aliases, and other context
- How to change a shell's default behavior by editing configuration files

SHELLS ARE EXECUTABLE FILES

The default shell on most Linux systems is `bash` and it's an ordinary program—an executable file—located in the system directory */bin* alongside `cat`, `ls`, `grep`, and other familiar commands:

```
$ cd /bin
$ ls -l bash cat ls grep
-rwxr-xr-x 1 root root 1113504 Jun  6  2019 bash
-rwxr-xr-x 1 root root   35064 Jan 18  2018 cat
-rwxr-xr-x 1 root root  219456 Sep 18  2019 grep
-rwxr-xr-x 1 root root  133792 Jan 18  2018 ls
```

`bash` is also not the only shell on your system, most likely. Valid shells are usually listed, one per line, in the file */etc/shells*:

```
$ cat /etc/shells
/bin/sh
/bin/bash
/bin/csh
/bin/zsh
```

To see which shell you're running, `echo` the shell variable `SHELL`:

```
$ echo $SHELL
/bin/bash
```

In theory, a Linux system can treat *any program* as a valid login shell, if a user account is configured to invoke it on login and it's listed in */etc/shells* (if required on your system). With superuser privileges, you can even write and install your own shell, like the script in Example Below

```
#!/bin/bash
# Print a prompt
echo -n '$ '
# Read the user's input in a loop. Exit when the user presses
Ctrl-D.
while read line; do
 # Ignore the input $line and print a message
 echo "I'm sorry, I'm afraid I can't do that"
 # Print the next prompt
 echo -n '$ '
done
```

It reads any command and responds, "I'm sorry, I'm afraid I can't do that." This custom shell is intentionally silly, but it demonstrates that other programs can be just as legitimate a shell as */bin/bash*

Since `bash` is just a program, you can run it manually like any other command:

```
$ bash
```

If you do so, you'll just see another prompt, as if your command had no effect:

```
$
```

But really, you have run a new instance of `bash`. This new instance prints a prompt and awaits your command. To make the new instance more visible, change its prompt (say, to `%%`) by setting the shell variable `PS1`, and run some commands:

```
$ PS1="%% "
%% ls                          # The prompt has changed
animals.txt
```

```
%% echo "This is a new shell"
This is a new shell
```

Now run `exit` to terminate the new instance of `bash`. You'll return to the original shell, which has a dollar-sign prompt:

```
%% exit
$
```

I must emphasize that the change from `%%` back to `$` was not a prompt change. It was a whole shell change. The new instance of `bash` has ended, so the original shell prompts you for the next command.

PARENT AND CHILD PROCESSES

When one instance of the shell invokes another, as I just demonstrated, the original shell is called the *parent* and the new instance is called the *child*. The same is true for any Linux program that invokes any other Linux program. The invoking program is the parent, and the invoked program is its child. A running Linux program is known as a *process*, so you'll also see the terms *parent process* and *child process*. A process can invoke any number of children, but each child has only one parent.

Every process has its own environment which includes a current directory, search path, shell prompt, and other important information held in shell variables. When a child is created, its environment is largely a copy of its parent's environment. *Every time you run a simple command; you create a child process*. This is such an important point for understanding Linux that I'll say it again: even when you run a simple command like `ls`, that command secretly runs inside a new child process with its own (copied) environment. That means any changes you make to a child, like changing the prompt variable `PS1` in a child shell, affect only the child and are lost when the child exits. Likewise, any changes to the parent won't affect its children that are already running. Changes to the parent *can* affect its *future* children, however, because each child's environment is copied from its parent's environment on startup.

Why does it matter that commands run in child processes? For one thing, it means that any program you run can `cd` all over the filesystem, but when it exits, your current shell (the parent) has not changed its current directory. Here's a quick experiment to prove it. Create a shell script called `cdtest` in your home directory containing a `cd` command:

```
#!/bin/bash
cd /etc
echo "Here is my current directory:"
pwd
```

Make it executable:

```
$ chmod +x cdtest
```

Print your current directory name and then run the script:

```
$ pwd
/home/smith
$ ./cdtest
Here is my current directory:
/etc
```

Now check your current directory:

```
$ pwd
/home/smith
```

Your current directory hasn't changed, even though the `cdtest` script traveled to the */etc* directory. That's because `cdtest` ran inside a child process with its own environment. Changes to the child's environment cannot affect the parent's environment, so the parent's current directory did not change. The same thing happens when you run an executable program like `cat` or `grep`—it runs in a child process that exits after the program terminates, taking any environment changes with it.

Pipelines launch multiple child processes: one for each command in the pipeline.

```
$ cut -f1 grades | sort | uniq -c | sort -nr | head -n1 | cut -c9
```

ENVIRONMENT VARIABLES

Every instance of the shell has a collection of variables. Some variables are local to a single shell. They are called *local variables*. Other variables are automatically copied from a given shell to each child it invokes. These variables are called *environment variables*, and they collectively form the shell's environment. Some examples of environment variables and their uses are:

HOME

> The path to your home directory. Its value is set automatically by your login shell when you log in. Text editors like `vim` and `emacs` read the variable `HOME` so they can locate and read their configuration files (*$HOME/.vim* and *$HOME/.emacs*, respectively).

PWD

> Your shell's current directory. Its value is set and maintained automatically by the shell each time you `cd` to another directory. The command `pwd` reads the variable `PWD` to print the name of your shell's current directory.

EDITOR

> The name of (or path to) your preferred text editor. Its value is generally set by you in a shell configuration file. Other programs read this variable to launch an appropriate editor on your behalf.

View a shell's environment variables with the `printenv` command. The output is one variable per line, unsorted, and can be quite long, so pipe it though `sort` and `less` for friendlier viewing

```
$ printenv | sort -i | less
⋮
DISPLAY=:0
EDITOR=emacs
HOME=/home/smith
LANG=en_US.UTF-8
PWD=/home/smith/Music
SHELL=/bin/bash
TERM=xterm-256color
USER=smith
⋮
```

Local variables do not appear in the output of `printenv`. Display their values by preceding the variable name with a dollar sign and printing the result with `echo`:

```
$ title="Efficient Linux"
$ echo $title
Efficient Linux
#(produces no output)
$ printenv title
```

Creating Environment Variables

To turn a local variable into an environment variable, use the `export` command:

```
# A local variable
$ MY_VARIABLE=10
# Export it to become an environment variable
$ export MY_VARIABLE
# Or, set and export in a single command
$ export ANOTHER_VARIABLE=20
```

`export` specifies that the variable and its value will be copied from the current shell to any future children. Local variables are not copied to future children:

```
# Set an environment variable
$ export E="I am an environment variable"
# Set a local variable
$ L="I am just a local variable"
$ echo $E
I am an environment variable
$ echo $L
I am just a local variable
# Run a child shell
$ bash
# Environment variable was copied
$ echo $E
I am an environment variable
# Local variable was not copied
# Empty string is printed


$ echo $L

# Exit the child shell
$ exit
```

Remember, a child's variables are *copies*. Any changes to the copy do not affect the parent shell:

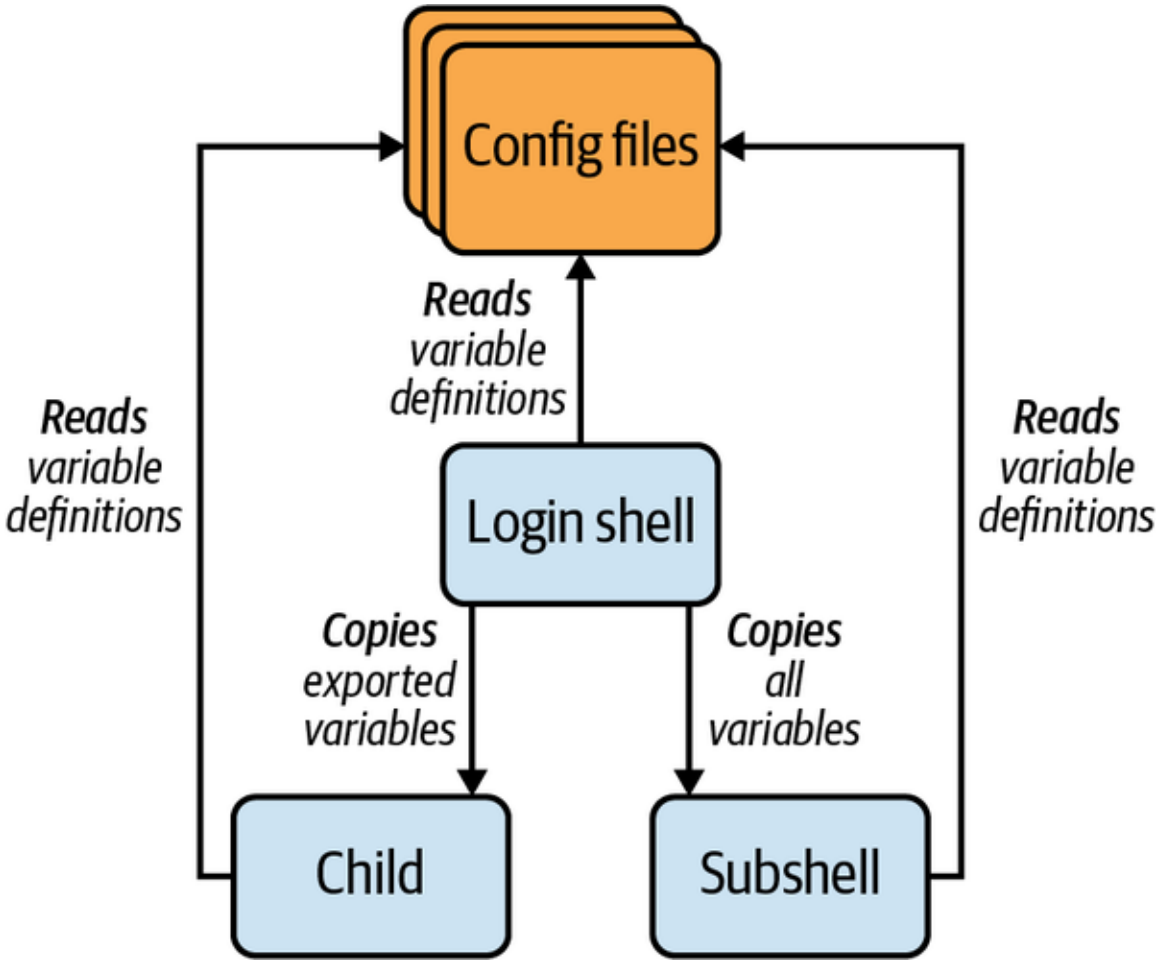
```
# Set an environment variable
$ export E="I am the original value"
# Run a child shell
$ bash
# Parent's value was copied
$ echo $E
I am the original value
# Change the child's copy
$ E="I was modified in a child"
$ echo $E
I was modified in a child
# Exit the child shell
$ exit
# Parent's value is unchanged
$ echo $E
I am the original value
```

Launch a new shell anytime and change anything in its environment, and all the changes disappear when you exit the shell. This means you can experiment with shell features safely—just run a shell manually, creating a child, and terminate it when finished.

Superstition Alert: “Global” Variables

Sometimes Linux hides its inner workings too well. A great example is the behavior of environment variables. Somehow, like magic, variables like `HOME` and `PATH` each have a consistent value in all your shell instances. They seem to be “global variables” in some sense. But an environment variable is *not global*. Each shell instance has its own copy. Modifying an environment variable in one shell cannot change the value in any other running shells. Modifications affect only that shell’s future children (not yet invoked).

If that’s the case, how does a variable like `HOME` or `PATH` seem to keep its value in all your shell instances? There are two avenues to make this happen, which are illustrated below



Children copy from their parents.

For variables like `HOME`, the values are usually set and exported by your login shell. All future shells (until you log out) are children of the login shell, so they receive a copy of

the variable and its value. These sorts of system-defined environment variables are so rarely modified in the real world that they seem global, but they are just ordinary variables that play by the ordinary rules. (You may even change their values in a running shell, but you might disrupt the expected behavior of that shell and other programs.)

Different instances read the same configuration files.

Local variables, which are not copied to children, can have their values set in a Linux configuration file such as *$HOME/.bashrc*. Each instance of the shell, on invocation, reads and executes the appropriate configuration files. As a result, these local variables appear to be copied from shell to shell. The same is true for other nonexported shell features such as aliases.

This behavior leads some users to believe that the `export` command creates a global variable. It does not. The command `export WHATEVER` simply declares that the variable `WHATEVER` will be copied from the current shell to any future children.

Child Shells Versus Subshells

A child is a partial copy of its parent. It includes copies of its parent's environment variables, for example, but not its parent's local (unexported) variables or aliases:

```
# List aliases
$ alias
alias gd='pushd'
alias l='ls -CF'
alias pd='popd'
# Run a child shell and ignore bashrc files
$ bash --norc
# List aliases - none are known
$ alias
# Environment variables are known
$ echo $HOME
/home/smith
# Exit the child shell
$ exit
```

If you've ever wondered why your aliases aren't available in shell scripts, now you know. Shell scripts run in a child, which does not receive copies of the parent's aliases.

A *subshell*, in contrast, is a complete copy of its parent. It includes all the parent's variables, aliases, functions, and more. To launch a command in a subshell, enclose the command in parentheses:

```
# Launches ls -l in a subshell
$ (ls -l)
```

```
-rw-r--r-- 1 smith smith 325 Oct 13 22:19 animals.txt
# View aliases in a subshell
$ (alias)
alias gd=pushd
alias l=ls -CF
alias pd=popd
⋮
Run an alias from the parent
$ (l)
animals.txt
```

To check if a shell instance is a subshell, print the variable `BASH_SUBSHELL`. The value is nonzero in subshells, zero otherwise:

```
$ echo $BASH_SUBSHELL          Check the current shell
0                              Not a subshell
$ bash                         Run a child shell
$ echo $BASH_SUBSHELL          Check the child shell
0                              Not a subshell
$ exit                         Exit the child shell
$ (echo $BASH_SUBSHELL)        Run an explicit subshell
1                              Yes, it's a subshell
```

Configuring Your Environment

When `bash` runs, it configures itself by reading a sequence of files, called *configuration files*, and executing their contents. These files define variables, aliases, functions, and other shell features, and they can include any Linux command. (They are like shell scripts that configure the shell.) Some configuration files are defined by the system administrator and apply to all users system-wide. They are found in the directory */etc*. Other configuration files are owned and changed by individual users. They are located in the user's home directory. Bash configuration files are of several types

Startup files

> Configuration files that execute automatically when you log in—that is, they apply only to your login shell. An example command in this file might set and export an environment variable. Defining an alias in this file would be less helpful, however, because aliases are not copied to children.

Initialization ("init") files

> Configuration files that execute for every shell instance that is not a login shell—for example, when you run an interactive shell by hand or a (noninteractive) shell

script. An example initialization file command might set a variable or define an alias.

Cleanup files

Configuration files that execute immediately before your login shell exits. An example command in this file might be `clear` to blank your screen on logout.

File type	Run by	System-wide location	Personal file locations (in order invoked)
Startup files	Login shells, on invocation	*/etc/profile*	*$HOME/.bash_profile*, *$HOME/.bash_login*, and *$HOME/.profile*
Init files	Interactive shells (nonlogin), on invocation	*/etc/bashrc* or */etc/bash.bashrc*, depending on distro (look in */etc* to check)	*$HOME/.bashrc*
	Shell scripts, on invocation	Set the variable `BASH_ENV` to the absolute path to an initialization file (example: `BASH_ENV=/usr/local/etc/bashrc`)	Set the variable `BASH_ENV` to the absolute path to an initialization file (example: `BASH_ENV=/usr/local/etc/bashrc`)
Cleanup files	Login shells, on exit	*/etc/bash.bash_logout*	*$HOME/.bash_logout*

Notice that you have three choices for personal startup files in your home directory (*.bash_profile*, *.bash_login*, and *.profile*). Most users can just pick one and stick with it. Your Linux distro probably supplies one of them already, prefilled with (ideally) useful commands. Things are a bit different if you happen to run other shells such as Bourne shell (*/bin/sh*) and Korn shell (*/bin/ksh*). These shells also read *.profile* and can fail if handed `bash`-specific commands to execute. Place `bash`-specific commands in *.bash_profile* or *.bash_login* instead (again, just pick one).

Users sometimes find the separation of personal startup files and the personal initialization file confusing. Why would you want your login shell to behave differently from other shells, say, that you open in multiple windows? The answer is, in many cases,

you don't need them to behave differently. Your personal startup file might do little more than source your personal initialization file, *$HOME/.bashrc*, so all interactive shells (login or nonlogin) would have largely the same configuration.

In other cases, you might prefer to split responsibilities between your startup and initialization files. For example, your personal startup file might set and export your environment variables to be copied to future children, whereas *$HOME/.bashrc* might define all your aliases (which are not copied to children).

In each of these cases, it's generally worthwhile to have your personal startup file source your personal initialization file:

```
# Place in $HOME/.bash_profile or other personal startup file
if [ -f "$HOME/.bashrc" ]
then
  source "$HOME/.bashrc"
fi
```

Whatever you do, try not to place identical configuration commands in two different configuration files. That's a recipe for confusion, and it's hard to maintain, because any change you make to one file you must remember to duplicate in the other (and you'll forget, trust me). Instead, source one file from the other as I've shown.

Rereading a Configuration File

When you change any startup or initialization file, you can force a running shell to reread it by sourcing the file

```
# Uses the builtin "source" command
$ source \~/.bash_profile
# Uses a dot
$ . ~/.bash_profile
```

Why do you source a configuration file instead of making it executable with `chmod` and running it like a shell script? Because a script runs in a child process. Any commands in the script would not affect your intended (parent) shell. They would affect only the child, which exits, leaving you with nothing changed.

11 More Ways to Run a Command

Now that you have lots of commands in your toolbox and a thorough understanding of the shell, it's time to learn...how to run commands. Wait a minute, haven't you been running

commands since the beginning of the book? Well, yes, but only in two ways. The first is the ordinary execution of a simple command:

```
$ grep Nutshell animals.txt
```

The second is a pipeline of simple commands

```
$ cut -f1 grades | sort | uniq -c | sort -nr
```

I'll show you 11 more ways to run a command and why you should care to learn them. Each technique has pros and cons, and the more techniques you know, the more flexibly and efficiently you can interact with Linux.

LIST TECHNIQUES

A list is a sequence of commands on a single command line. You've already seen one type of list—a pipeline—but the shell supports others with different behavior:

Conditional lists

: Each command depends on the success or failure of the previous one.

Unconditional lists

: Commands simply run one after the other.

Technique #11: Conditional Lists

Suppose you want to create a file *new.txt* in a directory *dir*. A typical sequence of commands might be:

```
# Enter the directory
$ cd dir
# Make the file
$ touch new.txt
```

Notice how the second command depends on the success of the first. If the directory *dir* doesn't exist, there is no point in running the `touch` command. The shell lets you make this dependency explicit. If you place the operator `&&` (pronounced "and") between the two commands on a single line:

```
$ cd dir && touch new.txt
```

then the second command (`touch`) runs only if the first command (`cd`) succeeds. The preceding example is a *conditional list* of two commands.

Very likely, you run commands every day that depend on previous ones. For example, have you ever made a backup copy of a file for safekeeping, modified the original, and deleted the backup when done?

```
# Make a backup copy
$ cp myfile.txt myfile.safe
# Change the original
$ nano myfile.txt
# Delete the backup
$ rm myfile.safe
```

Each of these commands makes sense only if the preceding command succeeds. Therefore, this sequence is a candidate for a conditional list:

```
$ cp myfile.txt myfile.safe && nano myfile.txt && rm myfile.safe
```

As another example, if you use the version-control system Git to maintain files, you're probably familiar with the following sequence of commands after you change some files: run `git add` to prepare files for a commit, then `git commit`, and finally `git push` to share your committed changes. If any of these commands failed, you wouldn't run the rest (until you fixed the cause of the failure). Therefore, these three commands work well as a conditional list:

```
$ git add . && git commit -m"fixed a bug" && git push
```

Just as the `&&` operator runs a second command only if the first succeeds, the related operator `||` (pronounced "or") runs a second command only if the first fails. For example, the following command tries to enter *dir*, and if it fails to do so, it creates *dir*

```
$ cd dir || mkdir dir
```

You'll commonly see the `||` operator in scripts, causing the script to exit if an error occurs:

```
# If a directory can't be entered, exit with an error code of 1
cd dir || exit 1
```

Combine the `&&` and `||` operators to set up more complicated actions for success and failure. The following command tries to enter directory *dir*, and if it fails, it creates the directory and enters it. If all fails, the command prints a failure message:

```
$ cd dir || mkdir dir && cd dir || echo "I failed"
```

The commands in a conditional list don't have to be simple commands; they can also be pipelines and other combined commands.

Exit Codes Indicate Success or Failure

What does it mean for a Linux command to succeed or fail? Every Linux command produces a result when it terminates, called an *exit code*. By convention, an exit code of zero means success and any nonzero value means failure. View the exit code of a shell's most recently completed command by printing the special shell variable whose name is a question mark (`?`):

```
$ ls myfile.txt
myfile.txt
# Print the value of the ? variable
$ echo $?
0
# cp failed
$ cp nonexistent.txt somewhere.txt
cp: cannot stat 'nonexistent.txt': No such file or directory
$ echo $?
1
```

Technique #2: Unconditional Lists

Commands in a list don't have to depend on one another. If you separate the commands with semicolons, they simply run in order. Success or failure of a command does not affect later ones in the list.

I like unconditional lists for launching ad hoc commands after I've left work for the day. Here's one that sleeps (does nothing) for two hours (7,200 seconds) and then backs up my important files:

```
$ sleep 7200; cp -a ~/important-files /mnt/backup_drive
```

Here's a similar command that functions as a primitive reminder system, sleeping for five minutes and then sending me an email

```
$ sleep 300; echo "remember to walk the dog" | mail -s reminder
$USER
```

Unconditional lists are a convenience feature: they produce the same results (mostly) as typing the commands individually and pressing Enter after each. The only significant difference relates to exit codes. In an unconditional list, the exit codes of the individual commands are thrown away except the last one. Only the exit code of the last command run in the list is assigned to the shell variable `?`:

```
$ mv file1 file2; mv file2 file3; mv file3 file4
# The exit code for "mv file3 file4"
```

```
$ echo $?
0
```

Substitution Techniques

Substitution means automatically replacing the text of a command with other text. I'll show you two types with powerful possibilities:

Command substitution: A command is replaced by its output.

Process substitution: A command is replaced by a file (sort of).

Technique #3: Command Substitution

Suppose you have a few thousand text files representing songs. Each file includes a song title, artist name, album title, and the song lyrics:

```
Title: Carry On Wayward Son
Artist: Kansas
Album: Leftoverture

Carry on my wayward son
There'll be peace when you are done
⋮
```

You'd like to organize the files into subdirectories by artist. To perform this task by hand, you could search for all song files by Kansas using `grep`:

```
$ grep -l "Artist: Kansas" *.txt
carry_on_wayward_son.txt
dust_in_the_wind.txt
belexes.txt
```

and then move each file to a directory *kansas*:

```
$ mkdir kansas
$ mv carry_on_wayward_son.txt kansas
$ mv dust_in_the_wind.txt kansas
$ mv belexes.txt kansas
```

Tedious, right? Wouldn't be great if you could tell the shell, "Move all files that contain the string *Artist: Kansas* to the directory *kansas*." In Linux terms, you'd like to take the list of names from the preceding `grep -l` command and hand it to `mv`. Well, you can do this easily with the help of a shell feature called *command substitution*:

```
$ mv $(grep -l "Artist: Kansas" *.txt) kansas
```

The syntax:

```
$(any command here)
```

executes the command inside the parentheses and replaces the command by its output. So on the preceding command line, the `grep -l` command is replaced by the list of filenames that it prints, as if you had typed the filenames like this:

```
$ mv carry_on_wayward_son.txt dust_in_the_wind.txt belexes.txt
kansas
```

Whenever you find yourself copying the output of one command into a later command line, you can usually save time with command substitution. You can even include aliases in command substitution, because its contents are run in a subshell, which includes copies of its parent's aliases.

In scripts, a common use of command substitution is to store the output of a command in a variable:

```
VariableName=$(some command here)
```

For example, to get the filenames containing Kansas songs and store them in a variable, use command substitution like so:

```
$ kansasFiles=$(grep -l "Artist: Kansas" *.txt)
```

The output might have multiple lines, so to preserve any newline characters, make sure you quote the value wherever you use it:

```
$ echo "$kansasFiles"
```

Technique #4: Process Substitution

Command substitution, which you just saw, replaces a command with its output in place, as a string. *Process substitution* also replaces a command with its output, but it treats the output as if it were stored in a file. This powerful difference may look confusing at first, so I'll explain it step-by-step.

Suppose you're in a directory of JPEG image files named *1.jpg* through *1000.jpg*, but some files are mysteriously missing and you want to identify them. Produce such a directory with the following commands:

```
$ mkdir /tmp/jpegs && cd /tmp/jpegs
$ touch {1..1000}.jpg
$ rm 4.jpg 981.jpg
```

A poor way to locate the missing files is to list the directory, sorted numerically, and look for gaps by eye:

```
# 4.jpg is missing
$ ls -1 | sort -n | less
1.jpg
2.jpg
3.jpg
5.jpg
⋮
```

A more robust, automated solution is to compare the existing filenames to a complete list of names from *1.jpg* to *1000.jpg*, using the `diff` command. One way to achieve this solution is with temporary files. Store the existing filenames, sorted, in one temporary file, *original-list*:

```
$ ls *.jpg | sort -n > /tmp/original-list
```

Then print a complete list of filenames from *1.jpg* to *1000.jpg* to another temporary file, *full-list*, by generating the integers 1 to 1000 with `seq`, and appending ".jpg" to each line with `sed`:

```
$ seq 1 1000 | sed 's/$/.jpg/' > /tmp/full-list
```

Compare the two temporary files with the `diff` command to discover that *4.jpg* and *981.jpg* are missing, then delete the temporary files:

```
$ diff /tmp/original-list /tmp/full-list
3a4
> 4.jpg
979a981
> 981.jpg
# Clean up afterwards
$ rm /tmp/original-list /tmp/full-list
```

That's a lot of steps. Wouldn't it be grand to compare the two lists of names directly and not bother with temporary files? The challenge is that `diff` can't compare two lists from stdin; it requires files as arguments. Process substitution solves the problem. It makes both lists appear to `diff` as files. Syntax

```
<(any command here)
```

runs the command in a subshell and presents its output as if it were contained in a file. For example, the following expression represents the output of `ls -1 | sort -n` as if it were contained in a file:

```
<(ls -1 | sort -n)
```

You can `cat` the file:

```
$ cat <(ls -1 | sort -n)
1.jpg
2.jpg
⋮
```

You can copy the file with `cp`:

```
$ cp <(ls -1 | sort -n) /tmp/listing
$ cat /tmp/listing
1.jpg
2.jpg
⋮
```

and as you'll now see, you can `diff` the file against another. Begin with the two commands that generated your two temporary files:

```
ls *.jpg | sort -n
seq 1 1000 | sed 's/$/.jpg/'
```

Apply process substitution so `diff` can treat them as files, and you get the same output as before, but without using temporary files:

```
$ diff <(ls *.jpg | sort -n) <(seq 1 1000 | sed 's/$/.jpg/')
3a4
> 4.jpg
979a981
> 981.jpg
```

Clean up the output by grepping for lines beginning with `>` and stripping off the first two characters with `cut`, and you have your missing files report:

```
$ diff <(ls *.jpg | sort -n) <(seq 1 1000 | sed 's/$/.jpg/') \
    | grep '>' \
    | cut -c3-
4.jpg
981.jpg
```

Process substitution transformed how I use the command line. Commands that read only from disk files suddenly could read from stdin. With practice, commands that previously seemed impossible became easy.

How Process Substitution Works

When the Linux operating system opens a disk file, it represents that file with an integer called a *file descriptor*. Process substitution mimicks a file by running a command and associating its output with a file descriptor, so the output appears to be in a disk file from the perspective of programs that access it. You can view the file descriptor with `echo`:

```
$ echo <(ls)
/dev/fd/63
```

In this case, the file descriptor for `<(ls)` is 63, and it's tracked in the system directory */dev/fd*.

Fun fact: stdin, stdout, and stderr are represented by the file descriptors 0, 1, and 2, respectively. That's why redirection of stderr has the syntax `2>`.

The expression `<(…)` creates a file descriptor for reading. The related expression `>(…)` creates a file descriptor for writing, but in 25 years I've never needed it.

Process substitution is a non-POSIX feature that might be disabled in your shell. To enable non-POSIX features in your current shell, run `set +o posix`.

COMMAND-AS-STRING TECHNIQUES

Every command is a string, but some commands are more "stringy" than others. I'll show you several techniques that construct a string, piece by piece, and then run the string as a command:

- Passing a command to `bash` as an argument
- Piping commands to `bash` on stdin
- Sending commands to another host with `ssh`
- Running a sequence of commands with `xargs`

Technique #5: Passing a Command as an Argument to bash

`bash` is a normal command like any other. So you can run it by name on the command line. By default, running `bash` launches an interactive shell for typing and executing commands, as you've seen. Alternatively, you can pass a command to `bash` as a string, via the `-c` option, and `bash` will run that string as a command and exit:

```
$ bash -c "ls -l"
-rw-r--r-- 1 smith smith 325 Jul  3 17:44 animals.txt
```

Why is this helpful? Because the new `bash` process is a child with its own environment, including a current directory, variables with values, and so on. Any changes to the child shell won't affect your currently running shell. Here's a `bash -c` command that changes directory to */tmp* just long enough to delete a file, then exits:

```
$ pwd
/home/smith
# Create a temporary file
```

```
$ touch /tmp/badfile
$ bash -c "cd /tmp && rm badfile"
# Current directory is unchanged
$ pwd
/home/smith
```

The most instructive and beautiful use of `bash -c`, however, arises when you run certain commands as the superuser. Specifically, the combination of `sudo` and input/output redirection produces an interesting (sometimes maddening) situation in which `bash -c` is the key to success.

Suppose you want to create a log file in the system directory */var/log*, which is not writable by ordinary users. You run the following `sudo` command to gain superuser privileges and create the log file, but it mysteriously fails:

```
$ sudo echo "New log file" > /var/log/custom.log
bash: /var/log/custom.log: Permission denied
```

Wait a minute—`sudo` should give you permission to create any file anywhere. How can this command possibly fail? Why didn't `sudo` even prompt you for a password? The answer is: because `sudo` didn't run. You applied `sudo` to the `echo` command but not to the output redirection, which ran first and failed. In detail:

- You pressed Enter.
- The shell began to evaluate the whole command, including redirection (`>`).
- The shell tried to create the file *custom.log* in a protected directory, */var/log*.
- You didn't have permission to write to */var/log*, so the shell gave up and printed the "Permission denied" message.

That's why `sudo` never ran. To solve this problem, you need to tell the shell, "Run the entire command, including output redirection, as the superuser." This is exactly the kind of situation that `bash -c` solves so well. Construct the command you want to run, as a string:

```
'echo "New log file" > /var/log/custom.log'
```

and pass it as an argument to `sudo bash -c`:

```
$ sudo bash -c 'echo "New log file" > /var/log/custom.log'
[sudo] password for smith: xxxxxxxx
$ cat /var/log/custom.log
New log file
```

This time, you've run `bash`, not just `echo`, as the superuser, and `bash` executes the entire string as a command. The redirection succeeds. Remember this technique whenever you pair `sudo` with redirection.

Technique #6: Piping a Command to bash

The shell reads every command that you type on stdin. That means `bash` the program can participate in pipelines. For example, print the string `"ls -l"` and pipe it to `bash`, and `bash` will treat the string as a command and run it:

```
$ echo "ls -l"
ls -l
$ echo "ls -l" | bash
-rw-r--r-- 1 smith smith 325 Jul  3 17:44 animals.txt
```

This technique is terrific when you need to run many similar commands in a row. If you can print the commands as strings, then you can pipe the strings to `bash` for execution. Suppose you're in a directory with many files, and you want to organize them into subdirectories by their first character. A file named *apple* would be moved to subdirectory *a*, a file named *cantaloupe* would move to subdirectory *c*, and so on.

First, list the files, sorted. We'll assume all the names are at least two characters long (matching the pattern `??*`) so our commands don't collide with the subdirectories *a* through *z*:

```
$ ls -1 ??*
apple
banana
cantaloupe
carrot
⋮
```

Create the 26 subdirectories you need via brace expansion:

```
$ mkdir {a..z}
```

Now generate the `mv` commands you'll need, as strings. Start with a regular expression for `sed` that captures the first character of the filename as expression #1 (`\1`):

```
^\(.\)
```

Capture the rest of the filename as expression #2 (`\2`):

```
\(.*\)$
```

Connect the two regular expressions:

```
^\(.\)\(.*\)$
```

Now form an `mv` command with the word *mv* followed by a space, the full filename (`\1\2`), another space, and the first character (`\1`):

```
mv \1\2 \1
```

The complete command generator is:

```
$ ls -1 ??* | sed 's/^\(.\)\(.*\)$/mv \1\2 \1/'
mv apple a
mv banana b
mv cantaloupe c
mv carrot c
⋮
```

Its output contains exactly the `mv` commands you need. Read the output to convince yourself it's correct, perhaps by piping it to `less` for page-by-page viewing:

```
$ ls -1 ??* | sed 's/^\(.\)\(.*\)$/mv \1\2 \1/' | less
```

When you're satisfied that your generated commands are correct, pipe the output to `bash` for execution:

```
$ ls -1 ??* | sed 's/^\(.\)\(.*\)$/mv \1\2 \1/' | bash
```

The steps you just completed are a repeatable pattern:

- Print a sequence of commands by manipulating strings.
- View the results with `less` to check correctness.
- Pipe the results to `bash`.

Technique #7: Executing a String Remotely with ssh

Disclaimer: this technique will make sense only if you're familiar with SSH, the secure shell, for logging into remote hosts.

In addition to the usual way of logging into a remote host:

```
$ ssh myhost.example.com
```

you also can execute a single command on the remote host—by passing a string to `ssh` on the command line. Simply append the string to the rest of the `ssh` command line:

```
$ ssh myhost.example.com ls
remotefile1
```

```
remotefile2
remotefile3
```

This technique is generally quicker than logging in, running a command, and logging out. If the command includes special characters, such as redirection symbols, that need to be evaluated on the remote host, then quote or escape them. Otherwise, they'll be evaluated by your local shell. Both of the following commands run `ls` remotely, but the output redirection occurs on different hosts:

```
# Creates outfile on local host
$ ssh myhost.example.com ls > outfile
# Creates outfile on remote host
$ ssh myhost.example.com "ls > outfile"
```

You can also pipe commands to `ssh` to run them on the remote host, much like you pipe them to `bash` to run locally:

```
$ echo "ls > outfile" | ssh myhost.example.com
```

When piping commands to `ssh`, the remote host might print diagnostic or other messages. These generally do not affect the remote command, and you can suppress them:

- If you see messages about pseudo-terminals or pseudo-ttys, such as "Pseudo-terminal will not be allocated because stdin is not a terminal," run `ssh` with the `-T` option to prevent the remote SSH server from allocating a terminal:

```
$ echo "ls > outfile" | ssh -T myhost.example.com
```

- If you see welcome messages that normally appear when you log in ("Welcome to Linux!") or other unwanted messages, try telling `ssh` explicitly to run `bash` on the remote host, and the messages should disappear:

```
$ echo "ls > outfile" | ssh myhost.example.com bash
```

Technique #8: Running a List of Commands with xargs

Many Linux users have never heard of the command `xargs`, but it's a powerful tool for constructing and running multiple, similar commands. Learning `xargs` was another transformative moment in my Linux education, and I hope yours as well.

`xargs` accepts two inputs:

- On stdin: A list of strings separated by whitespace. An example is file paths produced by `ls` or `find`, but any strings will do. I'll call them the *input strings*.
- On the command line: An incomplete command that's missing some arguments, which I'll call the *command template*.

`xargs` merges the input strings and the command template to produce and run new, complete commands, which I'll call the *generated commands*. I'll demonstrate this process with a toy example. Suppose you're in a directory with three files:

```
$ ls -1
apple
banana
cantaloupe
```

Pipe the directory listing to `xargs` to serve as its input strings, and provide `wc -l` to serve as the command template, like so:

```
$ ls -1 | xargs wc -l
3 apple
4 banana
1 cantaloupe
8 total
```

As promised, `xargs` applied the `wc -l` command template to the input strings and counts lines in each file. To print the same three files with `cat`, simply change the command template to "cat":

```
$ ls -1 | xargs cat
```

My toy examples with `xargs` have two shortcomings, one fatal and one practical. The fatal shortcoming is that `xargs` may do the wrong thing if an input string contains special characters, such as spaces

The practical shortcoming is that you don't need `xargs` here—you can accomplish the same tasks more simply with file pattern matching:

```
$ wc -l *
3 apple
4 banana
1 cantaloupe
8 total
```

Why use `xargs`, then? Its power becomes apparent when the input strings are more interesting than a simple directory listing. Suppose you want to count lines in all files in a directory *and all its subdirectories* (recursively), but only for Python source files with names ending in *.py*. It's easy to produce such a list of file paths with `find`:

```
$ find . -type f -name \*.py -print
fruits/raspberry.py
vegetables/leafy/lettuce.py
⋮
```

`xargs` can now apply the command template `wc -l` to each file path, producing a recursive result that would be difficult to obtain otherwise. For safety, I'll replace the option `-print` with `-print0`, and `xargs` with `xargs -0`

```
$ find . -type f -name \*.py -print0 | xargs -0 wc -l
6 ./fruits/raspberry.py
3 ./vegetables/leafy/lettuce.py
⋮
```

By combining `find` and `xargs`, you can empower any command to run recursively through the filesystem, affecting only files (and/or directories) that match your stated criteria. (In some cases, you can produce the same effect with `find` alone, using its option `-exec`, but `xargs` is often a cleaner solution.)

`xargs` has numerous options (see `man xargs`) that control how it creates and runs the generated commands. The most important ones in my view (other than `-0`) are `-n` and `-I`. The `-n` option controls how many arguments are appended by `xargs` onto each generated command. The default behavior is to append as many arguments as will fit within the shell's limits

```
$ ls | xargs echo          Fit as many input strings as possible:
apple banana cantaloupe carrot          echo apple banana
cantaloupe carrot
$ ls | xargs -n1 echo             One argument per echo command:
apple                                   echo apple
banana                                  echo banana
cantaloupe                              echo cantaloupe
carrot                                  echo carrot
$ ls | xargs -n2 echo            Two arguments per echo command:
apple banana                            echo apple banana
cantaloupe carrot                       echo cantaloupe carrot
$ ls | xargs -n3 echo           Three arguments per echo command:
apple banana cantaloupe                 echo apple banana
cantaloupe
carrot                                  echo carrot
```

PROCESS-CONTROL TECHNIQUES

So far, all commands I've discussed occupy the parent shell until they finish. Let's consider several techniques that forge a different relationship with the parent shell:

Background commands

- Immediately return the prompt and execute out of sight

Explicit subshells

- Can be launched in the middle of a combined command

Process replacement

- Supersedes the parent shell

Technique #9: Backgrounding a Command

So far, all our techniques run a command to completion while you wait, and then present the next shell prompt. But you don't have to wait, especially for commands that take a long time. You can launch commands in a special way so they disappear from sight (sort of) yet continue to run, freeing up the current shell immediately to run further commands. This technique is called *backgrounding* a command or *running a command in the background*. In contrast, commands that occupy the shell are called *foreground* commands. A shell instance runs at most one foreground command at a time plus any number of background commands.

To run a command in the background, simply append an ampersand (`&`). The shell responds with a cryptic-looking message indicating that the command is backgrounded and presents the next prompt:

```
# Count characters in a huge file
$ wc -c my_extremely_huge_file.txt &
[1] 74931
$
```

You can then continue running foreground commands (or more background commands) in this shell. Output from backgrounded commands may appear at any time, even while you are typing. If the backgrounded command finishes successfully, the shell will inform you with a *Done* message:

```
59837483748 my_extremely_huge_file.txt
[1]+  Done                    wc -c my_extremely_huge_file.txt
```

or if it fails, you'll see an *Exit* message with an exit code:

```
[1]+  Exit 1                  wc -c my_extremely_huge_file.txt
```

The ampersand is also a list operator, like `&&` and `||`:

```
$ command1 & command2 & command3 &
[1] 57351
[2] 57352
[3] 57353
$ command4 & command5 & echo hi
```

```
[1] 57431
[2] 57432
hi
```

Suspending a command and sending it to the background

A related technique is to run a foreground command, change your mind during execution, and send it to the background. Press Ctrl-Z to stop the command temporarily (called *suspending* the command) and return to the shell prompt; then type `bg` to resume running the command in the background.

Jobs and job control

Background commands are part of a shell feature called *job control* that manipulates running commands in various ways, like backgrounding, suspending, and resuming them. A *job* is a shell's unit of work: a single instance of a command running in a shell. Simple commands, pipelines, and conditional lists are all examples of jobs—basically anything you can run at the command line.

A job is more than a Linux process. A job may consist of one process, two processes, or more. A pipeline of six programs, for example, is a single job that includes (at least) six processes. Jobs are a construct of the shell. The Linux operating system doesn't keep track of jobs, just the underlying processes.

At any moment, a shell may have multiple jobs running. Each job in a given shell has a positive integer ID, called the job ID or job number. When you run a command in the background, the shell prints the job number and the ID of the first process it runs within the job. In the following command, the job number is 1 and the process ID is 74931:

```
$ wc -c my_extremely_huge_file.txt &
[1] 74931
```

Common job-control operations

The shell has built-in commands for controlling jobs, I'll demonstrate the most common job-control operations by running a bunch of jobs and manipulating them. To keep the jobs simple and predictable, I'll run the command `sleep`, which just sits there doing nothing ("sleeping") for a given number of seconds and then exits. For example, `sleep 10` sleeps for 10 seconds.

Command	Meaning
bg	Move the current suspended job into the background
bg %*n*	Move suspended job number *n* into the background (example: `bg %1`)
fg	Move the current background job into the foreground
fg %*n*	Move background job number *n* into the foreground (example: `fg %2`)

kill %*n*	Terminate background job number *n* (example: `kill %3`)
jobs	View a shell's jobs

Run a job in the background to completion:

```
# Run in the background
$ sleep 20 &
[1] 126288
# List this shell's jobs
$ jobs
[1]+  Running                 sleep 20 &
$
...eventually...
[1]+  Done                    sleep 20
```

When jobs complete, the *Done* message might not appear until the next time you press Enter.

Run a background job and bring it into the foreground:

```
# Run in the background
$ sleep 20 &
[1] 126362
# Bring into the foreground
$ fg
sleep 20
...eventually...
$
```

Run a foreground job, suspend it, and bring it back into the foreground:

```
$ sleep 20                              Run in the foreground
^Z                                      Suspend the job
[1]+  Stopped                 sleep 20
$ jobs                                  List this shell's jobs
[1]+  Stopped                 sleep 20
$ fg                                    Bring into the foreground
sleep 20
...eventually...
[1]+  Done                    sleep 20
```

Run a foreground job and send it to the background:

```
$ sleep 20                              Run in the foreground
^Z                                      Suspend the job
[1]+  Stopped                 sleep 20
$ bg                                    Move to the background
```

```
[1]+ sleep 20 &
$ jobs                                  List this shell's jobs
[1]+  Running          sleep 20 &
$
...eventually...
[1]+  Done             sleep 20
```

Work with multiple background jobs. Refer to a job by its job number preceded by a percent sign (`%1`, `%2`, and so on):

```
$ sleep 100 &                           Run 3 commands in the
background
[1] 126452
$ sleep 200 &
[2] 126456
$ sleep 300 &
[3] 126460
$ jobs                                  List this shell's jobs
[1]   Running          sleep 100 &
[2]-  Running          sleep 200 &
[3]+  Running          sleep 300 &
$ fg %2                                 Bring job 2 into the
foreground
sleep 200
^Z                                      Suspend job 2
[2]+  Stopped          sleep 200
$ jobs                                  See job 2 is suspended
("stopped")
[1]   Running          sleep 100 &
[2]+  Stopped          sleep 200
[3]-  Running          sleep 300 &
$ kill %3                               Terminate job 3
[3]+  Terminated       sleep 300
$ jobs                                  See job 3 is gone
[1]-  Running          sleep 100 &
[2]+  Stopped          sleep 200
$ bg %2                                 Resume suspended job 2 in
the background
[2]+ sleep 200 &
$ jobs                                  See job 2 is running again
[1]-  Running          sleep 100 &
[2]+  Running          sleep 200 &
$
```

Output and input in the background

A backgrounded command may write to stdout, sometimes at inconvenient or confusing times. Notice what happens if you sort the Linux dictionary file (100,000 lines long) and print the first two lines in the background. As expected, the shell immediately prints the job number (1), a process ID (81089), and the next prompt:

```
$ sort /usr/share/dict/words | head -n2 &
[1] 81089
$
```

If you wait until the job finishes, it prints two lines on stdout wherever your cursor happens to be at the time. In this case, the cursor is sitting at the second prompt, so you get this sloppy-looking output

```
$ sort /usr/share/dict/words | head -n2 &
[1] 81089
$ A
A's
```

Press Enter, and the shell will print a "job done" message:

```
[1]+  Done                    sort /usr/share/dict/words | head -n2
$
```

Screen output from a background job can appear at any time while the job runs. To avoid this sort of messiness, redirect stdout to a file, then examine the file at your leisure:

```
$ sort /usr/share/dict/words | head -n2 > /tmp/results &
[1] 81089
$
[1]+  Done                    sort /usr/share/dict/words | head -n2 >
/tmp/results
$ cat /tmp/results
A
A's
$
```

Other odd things happen when a background job attempts to read from stdin. The shell suspends the job, prints a *Stopped* message, and waits for input in the background. Demonstrate this by backgrounding `cat` with no arguments so it reads stdin:

```
$ cat &
[1] 82455
[1]+  Stopped                 cat
```

Jobs can't read input in the background, so bring the job into the foreground with `fg` and then supply the input:

```
$ fg
cat
Here is some input
Here is some input
⋮
```

After supplying all input, do any of the following:

- Continue running the command in the foreground until it completes.
- Suspend and background the command again by pressing Ctrl-Z followed by `bg`.
- End the input with Ctrl-D, or kill the command with Ctrl-C.

Backgrounding tips

Backgrounding is ideal for commands that take a long time to run, such as text editors during long editing sessions, or any program that opens its own windows. For example, programmers can save a lot of time by suspending their text editor rather than exiting. I've seen experienced engineers modify some code in their text editor, save and quit the editor, test the code, then relaunch the editor and hunt for the spot in the code where they'd left off. They lose 10–15 seconds to job-switching every time they quit the editor. If they instead suspend the editor (Ctrl-Z), test their code, and resume the editor (`fg`), they avoiding wasting time unnecessarily.

Backgrounding is also great for running a sequence of commands in the background using a conditional list. If any command within the list fails, the rest won't run and the job completes. (Just watch out for commands that read input, since they'll cause the job to suspend and wait for input.)

```
$ command1 && command2 && command3 &
```

Problem	Solution
Sending stdout from one program to stdin of another	Pipelines
Inserting output (stdout) into a command	Command substitution
Providing output (stdout) to a command that doesn't read from stdin, but does read disk files	Process substitution

Executing one string as a command	`bash -c`, or piping to `bash`
Printing multiple commands on stdout and executing them	Piping to `bash`
Executing many similar commands in a row	`xargs`, or constructing the commands as strings and piping them to `bash`
Managing commands that depend on one other's success	Conditional lists
Running several commands at a time	Backgrounding
Running several commands at a time that depend on one another's success	Backgrounding a conditional list
Running one command on a remote host	Run `ssh` *`host command`*
Changing directory in the middle of a pipeline	Explicit subshells
Running a command later	Unconditional list with `sleep` followed by the command
Redirecting to/from protected files	Run `sudo bash -c "`*`command > file`*`"`

Technique #10: Explicit Subshells

Each time you launch a simple command, it runs in a child process. Command substitution and process substitution create subshells. There are times, however, when it's helpful to launch an extra subshell explicitly. To do so, simply enclose a command in parentheses and it runs in a subshell:

```
$ (cd /usr/local && ls)
bin   etc   games   lib   man   sbin   share
$ pwd
/home/smith
```

When applied to a whole command, this technique isn't super useful, except maybe to save you from running a second `cd` command to return to your previous directory. However, if you place parentheses around one piece of a combined command, you can perform some useful tricks. A typical example is a pipeline that changes directory in the middle of execution. Suppose you have downloaded a compressed `tar` file, *package.tar.gz*, and you want to extract the files. A `tar` command to extract the files is:

```
$ tar xvf package.tar.gz
Makefile
src/
src/defs.h
src/main.c
⋮
```

The extraction occurs relative to the current directory. What if you want to extract them into a different directory? You could `cd` to the other directory first and run `tar` (and then `cd` back), but you can also perform this task with a single command. The trick is to pipe the tarred data to a subshell that performs directory operations and runs `tar` as it reads from stdin

```
$ cat package.tar.gz | (mkdir -p /tmp/other && cd /tmp/other &&
tar xzvf -)
```

This technique also works to copy files from one directory *dir1* to another existing directory *dir2* using two `tar` processes, one writing to stdout and one reading from stdin:

```
$ tar czf - dir1 | (cd /tmp/dir2 && tar xvf -)
```

The same technique can copy files to an existing directory on another host via SSH:

```
$ tar czf - dir1 | ssh myhost '(cd /tmp/dir2 && tar xvf -)'
```

Technique #11: Process Replacement

Normally when you run a command, the shell runs it in a separate process that is destroyed when the command exits. You can change this behavior with the `exec` command, which is a shell builtin. It *replaces* the running shell (a process) with another command of your choice (another process). When the new command exits, no shell prompt will follow because the original shell is gone.

To demonstrate this, run a new shell manually and change its prompt:

```
$ bash                    Run a child shell
$ PS1="Doomed> "          Change the new shell's prompt
Doomed> echo hello        Run any command you like
hello
```

Now `exec` a command and watch the new shell die:

```
Doomed> exec ls           ls replaces the child shell, runs, and
exits
animals.txt
$                         A prompt from the original (parent)
shell
```

Why would you ever run `exec`? One reason is to conserve resources by not launching a second process. Shell scripts sometimes make use of this optimization by running `exec` on the final command in the script. If the script is run many times (say, millions or billions of executions), the savings might be worth it.

`exec` has a second ability—it can reassign stdin, stdout, and/or stderr for the current shell. This is most practical in a shell script, such as this toy example that prints information to a file, */tmp/outfile*:

```
#!/bin/bash
echo "My name is $USER"> /tmp/outfile
echo "My current directory is $PWD" >> /tmp/outfile
echo "Guess how many lines are in the file /etc/hosts?" >>
/tmp/outfile
wc -l /etc/hosts >> /tmp/outfile
echo "Goodbye for now" >> /tmp/outfile
```

Instead of redirecting the output of each command to */tmp/outfile* individually, use `exec` to redirect stdout to */tmp/outfile* for the entire script. Subsequent commands can simply print to stdout:

```
#!/bin/bash
# Redirect stdout for this script
exec > /tmp/outfile2
# All subsequent commands print to /tmp/outfile2
echo "My name is $USER"
echo "My current directory is $PWD"
echo "Guess how many lines are in the file /etc/hosts?"
wc -l /etc/hosts
echo "Goodbye for now"
```

Run this script and examine the file */tmp/outfile2* to see the results:

```
$ cat /tmp/outfile2
My name is smith
My current directory is /home/smith
Guess how many lines are in the file /etc/hosts?
122 /etc/hosts
Goodbye for now
```

Problem	Solution
Sending stdout from one program to stdin of another	Pipelines
Inserting output (stdout) into a command	Command substitution
Providing output (stdout) to a command that doesn't read from stdin, but does read disk files	Process substitution
Executing one string as a command	`bash -c`, or piping to `bash`
Printing multiple commands on stdout and executing them	Piping to `bash`

Executing many similar commands in a row	`xargs`, or constructing the commands as strings and piping them to `bash`
Managing commands that depend on one other's success	Conditional lists
Running several commands at a time	Backgrounding
Running several commands at a time that depend on one another's success	Backgrounding a conditional list
Running one command on a remote host	Run `ssh` *`host command`*
Changing directory in the middle of a pipeline	Explicit subshells
Running a command later	Unconditional list with `sleep` followed by the command
Redirecting to/from protected files	Run `sudo bash -c "`*`command`*`>`*`file`*`"`