LINUX
ITC-18

Lab Codes

Submitted by -
Raunit Dalal
2017UIT2537

Chapter 1-What is The Shell?

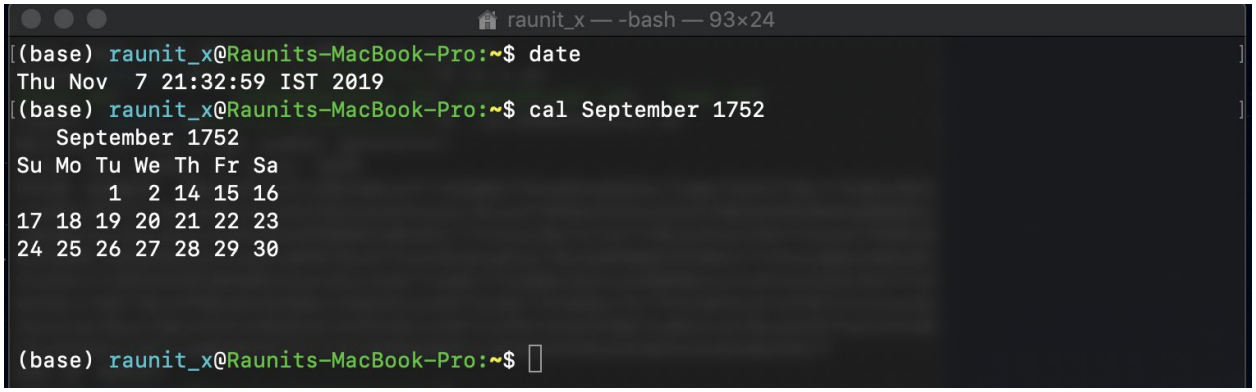
When we speak of the command line, we are really referring to the shell. The shell is a program that takes keyboard commands and passes them to the operating system to carry out. Almost all Linux distributions supply a shell program from the GNU Project called bash. The name “bash” is an acronym for “Bourne Again SHell”, a reference to the fact bash is an enhanced replacement for sh, the original Unix shell program written by Steve Bourne.

Command History

If we press the up-arrow key, we will see that the previous command “kaekfjaeifj” reappears after the prompt. This is called command history. Most Linux distributions remember the last 500 commands by default. Press the down-arrow key and the previous command disappears.

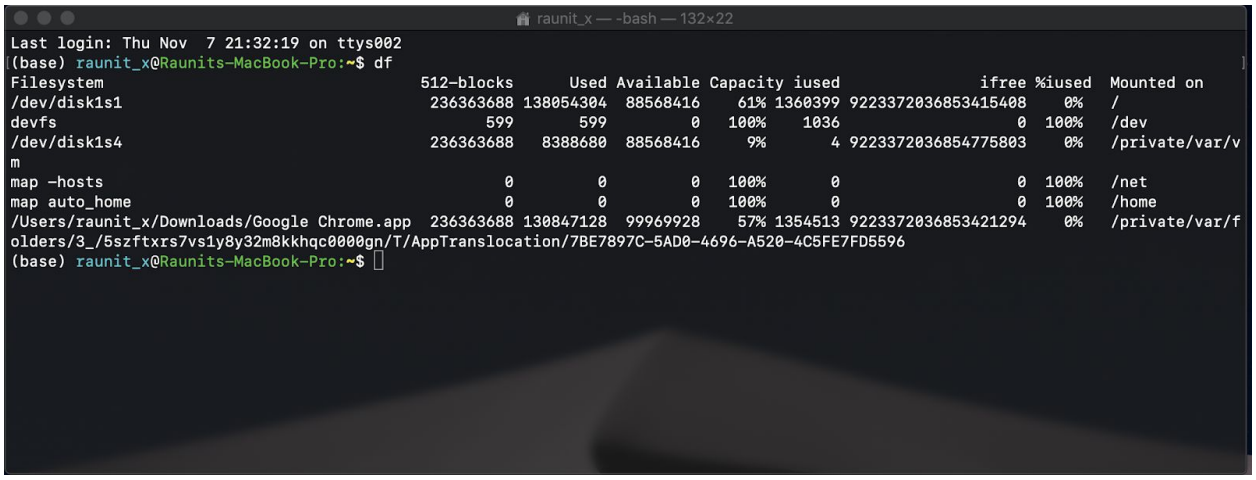
Simple commands

1) Date and cal
The first one is date. This command displays the current time and date.
A related command is cal which, by default, displays a calendar of the current month.



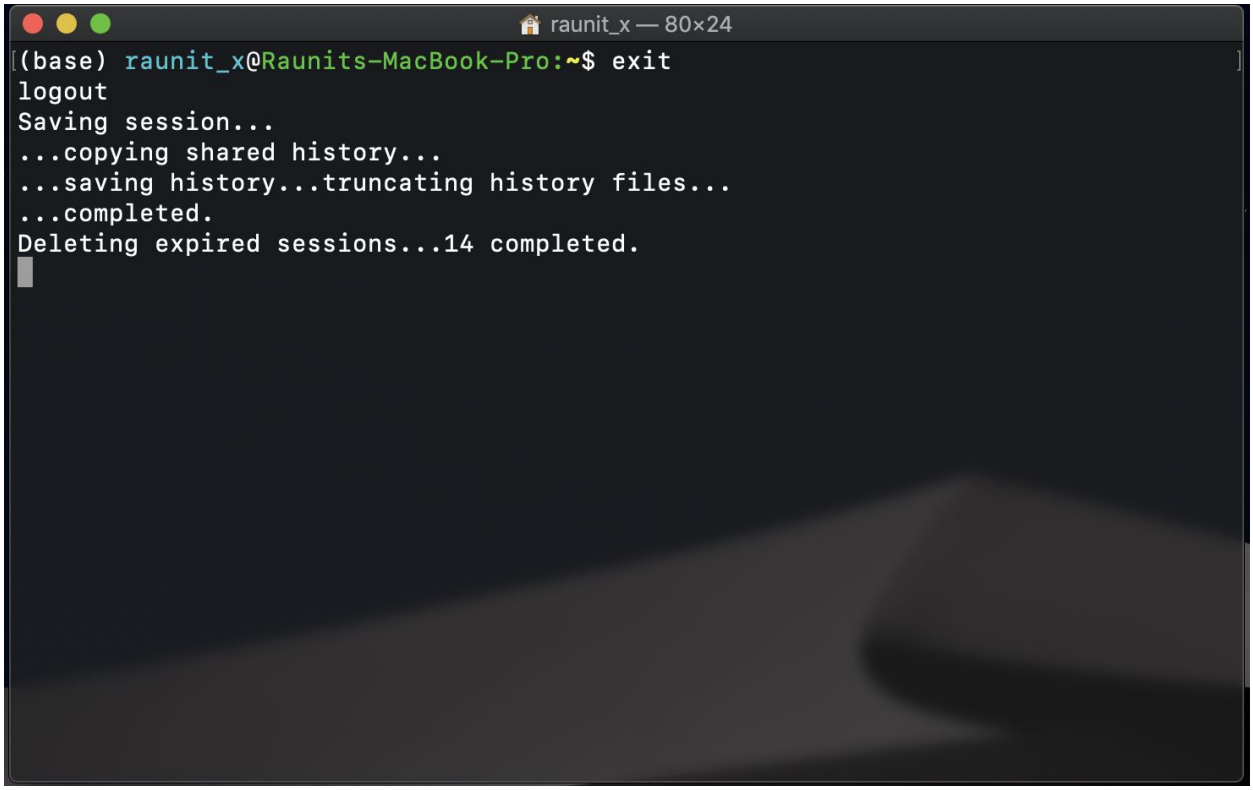
2) Df command

This command is used to see the current amount of free space on our disk drives.



4) Exit command

We can end a terminal session by either closing the terminal emulator window, or by entering the exit command at the shell prompt



Chapter 2 – Navigation

The first thing we need to learn (besides just typing) is how to navigate the file system on our Linux system. In this chapter we will introduce the following commands:

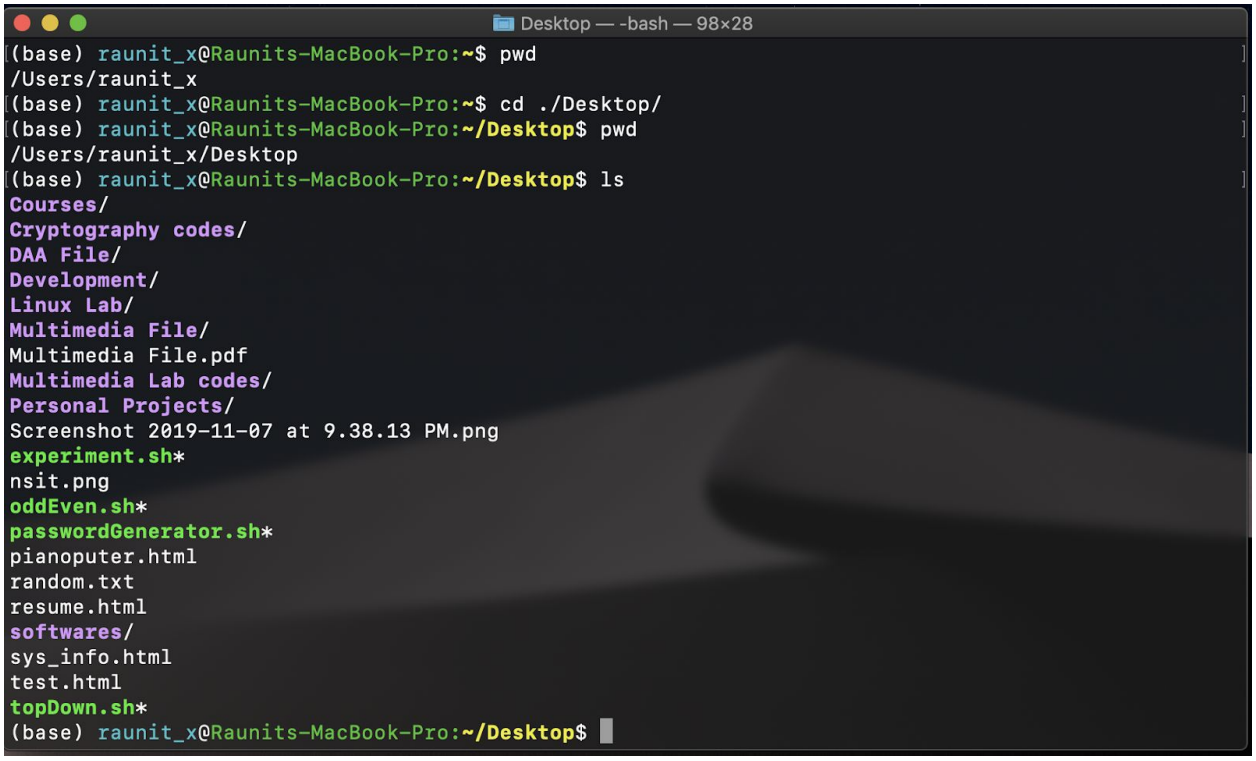
- pwd - Print name of current working directory
- cd - Change directory

● ls - List directory contents

1) PWD command
Print name of current working directory

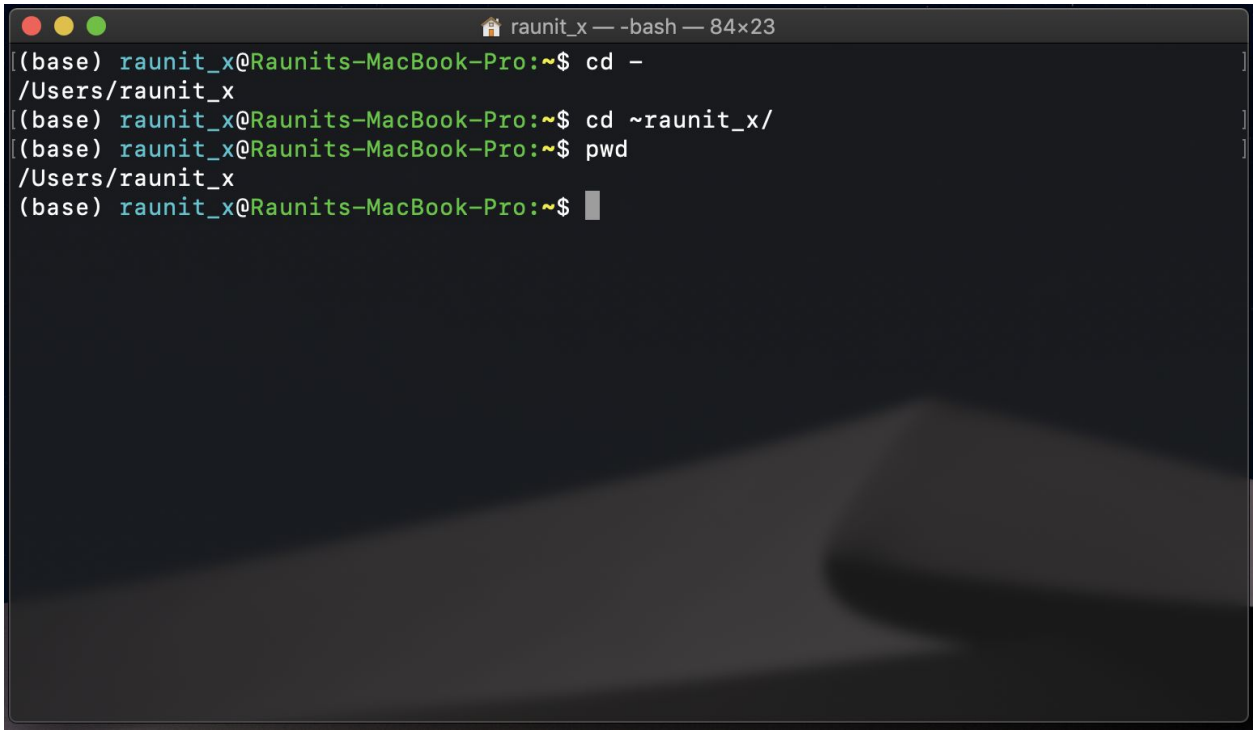
2) Cd command
change directory

3)Ls command
To list the files and directories in the current working directory, we use the ls command.



Changing The Current Working Directory

Shortcut	Result
`cd`	Changes the working directory to your home directory.
`cd -`	Changes the working directory to the previous working directory.
`cd ~user_name`	Changes the working directory to the home directory of *user_name*. For example, `cd ~bob` will change the directory to the home directory of user "bob."



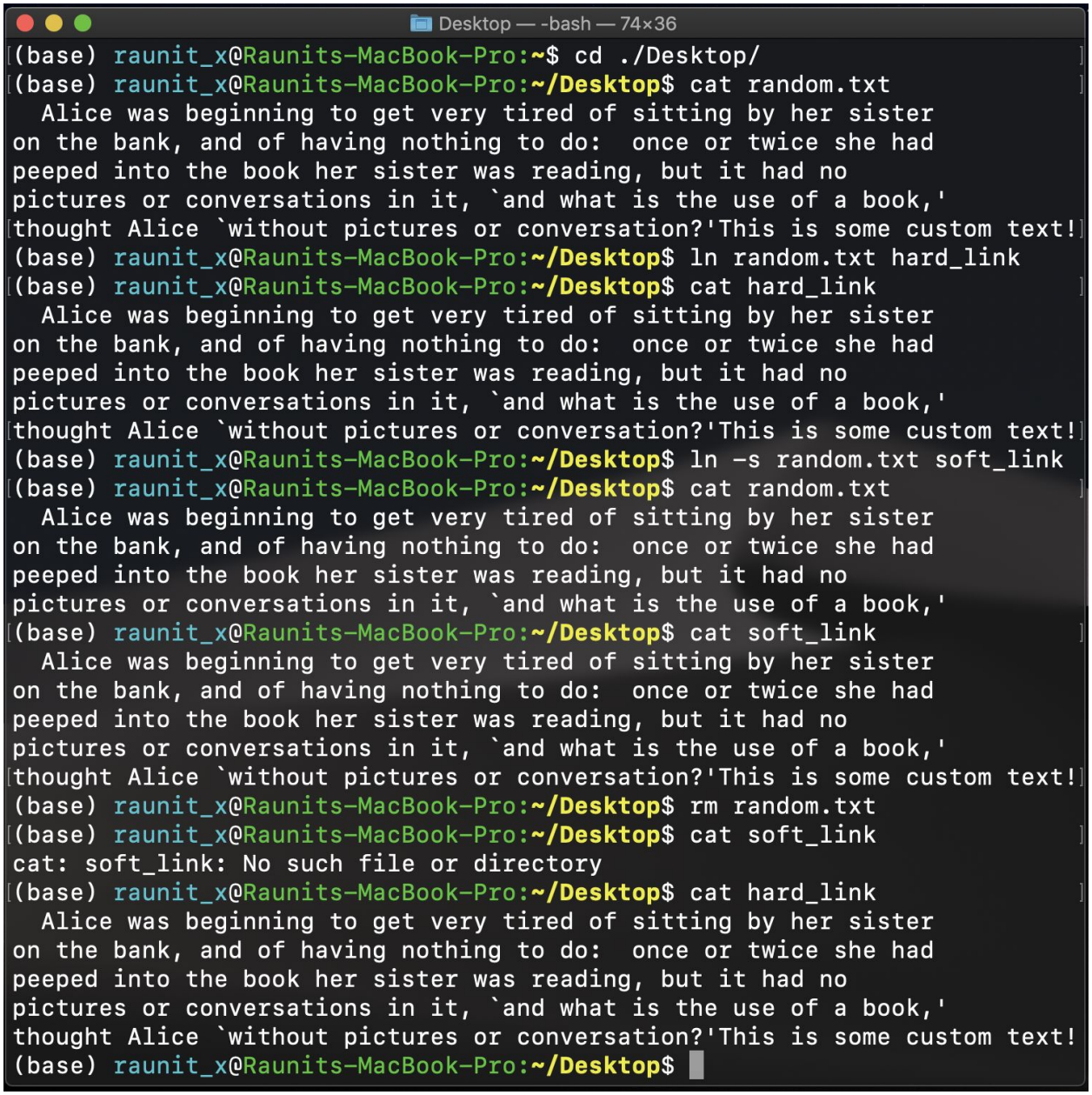
Chapter 4 - Manipulating Files and Directories

Hard and Soft Links

A link in UNIX is a pointer to a file. Like pointers in any programming languages, links in UNIX are pointers pointing to a file or a directory. Creating links is cca kind of shortcut to access a file. Links allow more than one file name to refer to the same file, elsewhere.
There are two types of links :

1. Soft Link or Symbolic links
2. Hard Links

A Symbolic or Soft link is an actual link to the original file, whereas a Hard link is a mirror copy of the original file. If you delete the original file, the soft link has no value, because it points to a non-existent file. But in the case of hard link, it is entirely the opposite. If you delete the original file, the hard link can still has the data of the original file. Because hard link acts as a mirror copy of the original file.

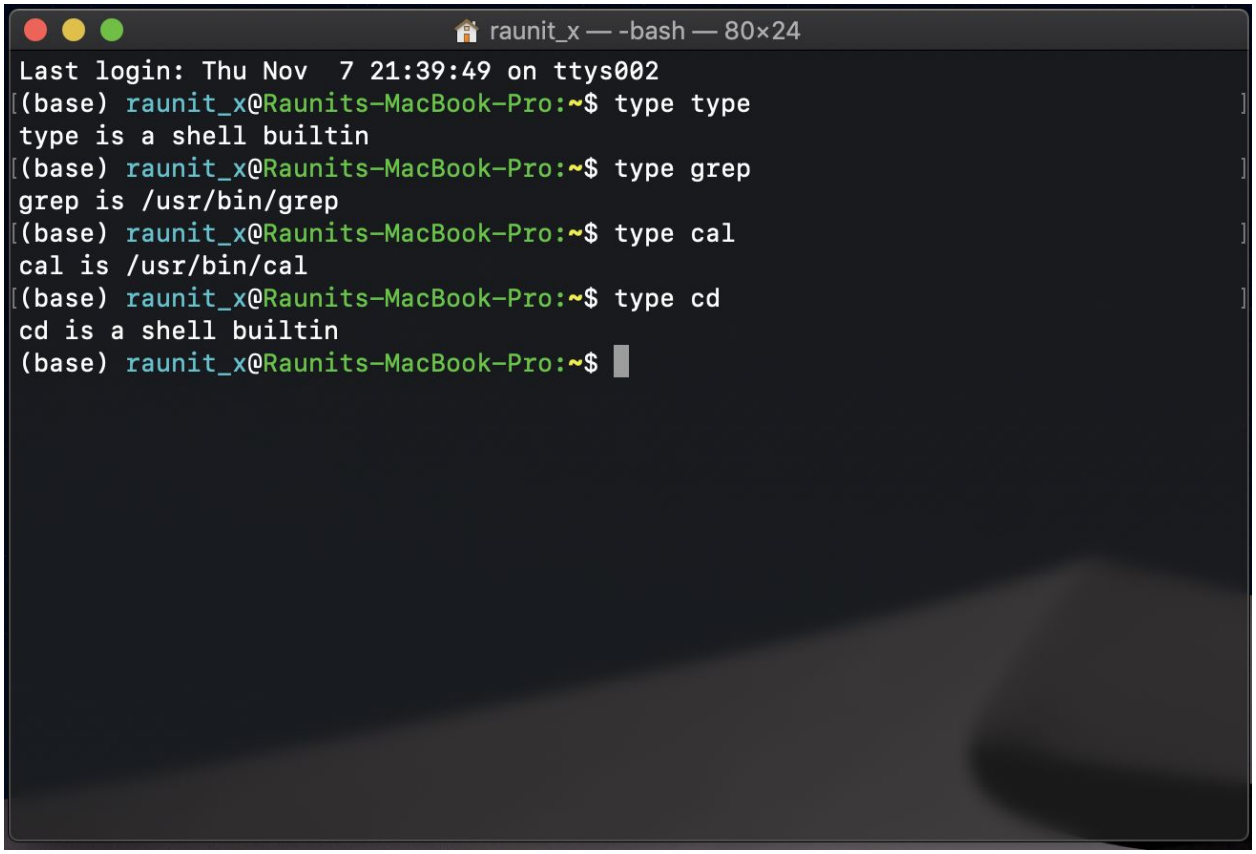

Desktop — -bash — 74×36
(base) raunit_x@Raunits-MacBook-Pro:~$ cd ./Desktop/
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ cat random.txt
Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
thought Alice `without pictures or conversation?'This is some custom text!
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ ln random.txt hard_link
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ cat hard_link
Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
thought Alice `without pictures or conversation?'This is some custom text!
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ ln -s random.txt soft_link
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ cat random.txt
Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ cat soft_link
Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
thought Alice `without pictures or conversation?'This is some custom text!
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ rm random.txt
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ cat soft_link
cat: soft_link: No such file or directory
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ cat hard_link
Alice was beginning to get very tired of sitting by her sister
on the bank, and of having nothing to do: once or twice she had
peeped into the book her sister was reading, but it had no
pictures or conversations in it, `and what is the use of a book,'
thought Alice `without pictures or conversation?'This is some custom text!
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$

BASIS FOR COMPARISON	HARD LINK	SOFT LINK
Definition	A file can be accessed through many different names known as hard links.	A file can be accessed through different references pointing to that file is known as a soft link.
Link validation, when the original file is deleted	Still valid and file can be accessed.	Invalid
Command	ln	ln -s
Can be linked	To its own partition.	To any other file system even networked.

Chapter 5 - Working with Commands

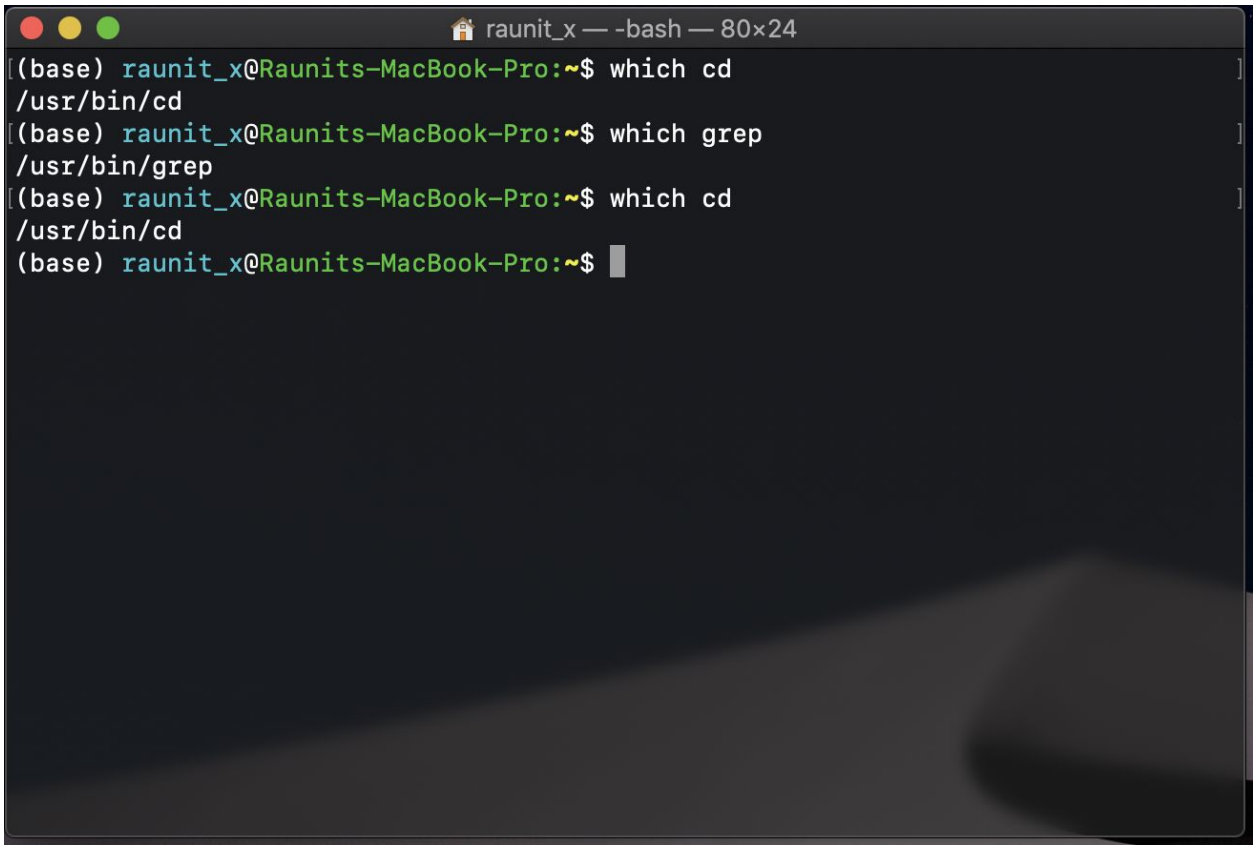
a. **type** – Display A Command's Type

The type command is a shell builtin that displays the kind of command the shell will execute, given a particular command name.



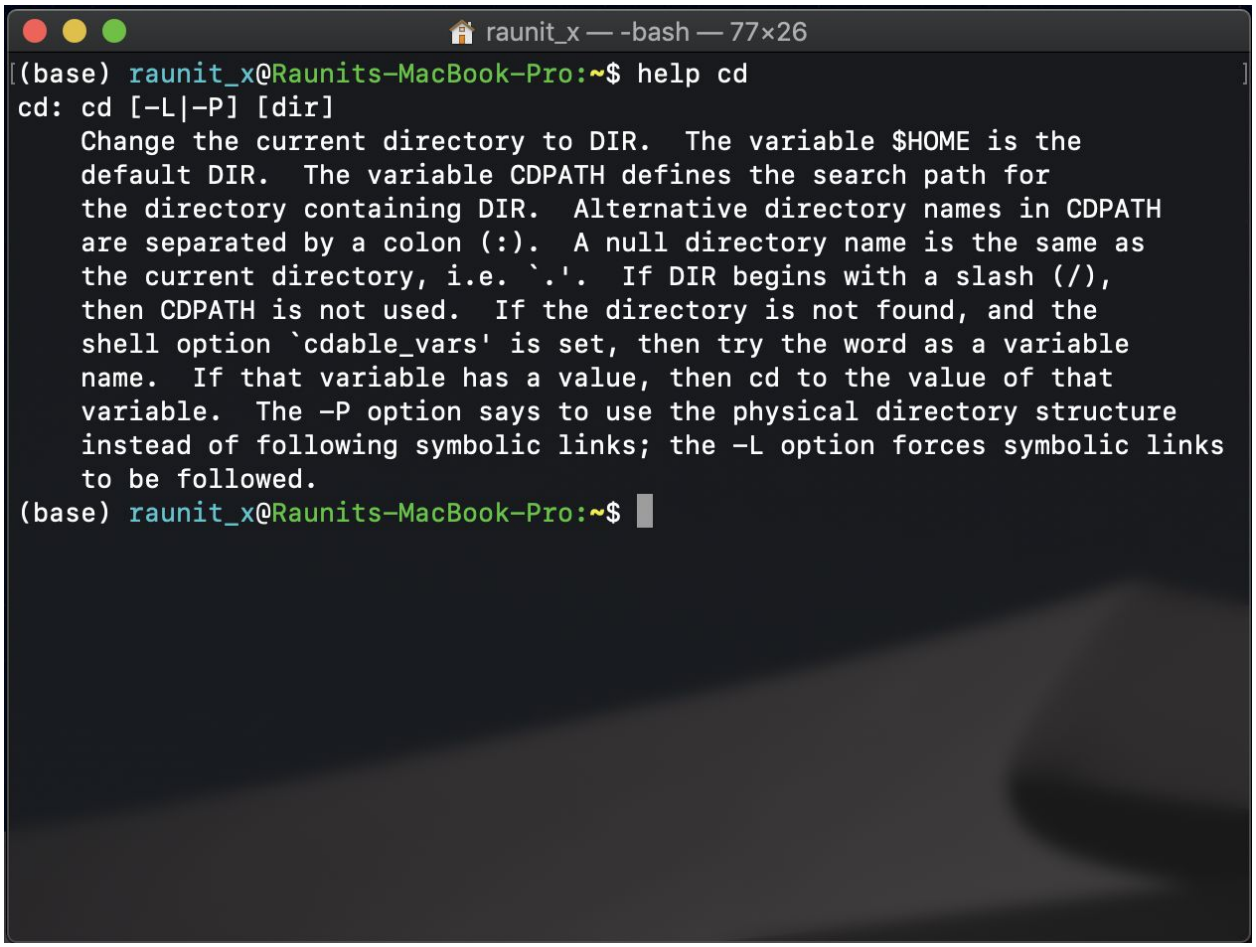
b. **which** – Display An Executable's Location

To determine the exact location of a given executable, the which command is used. which only works for executable programs, not builtins nor aliases that are substitutes for actual executable programs.



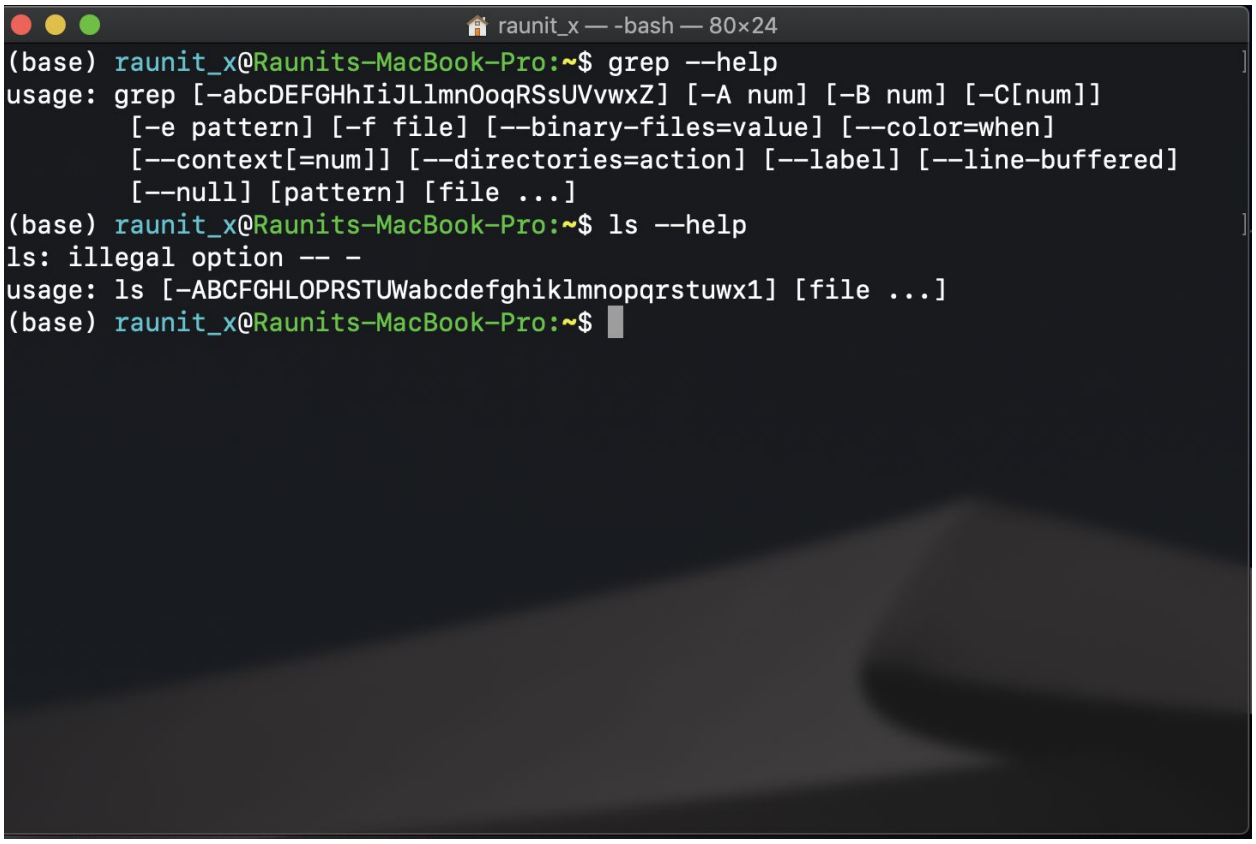
c. **help** – Get Help For Shell Builtins

bash has a **built-in help facility** available for each of the shell builtins



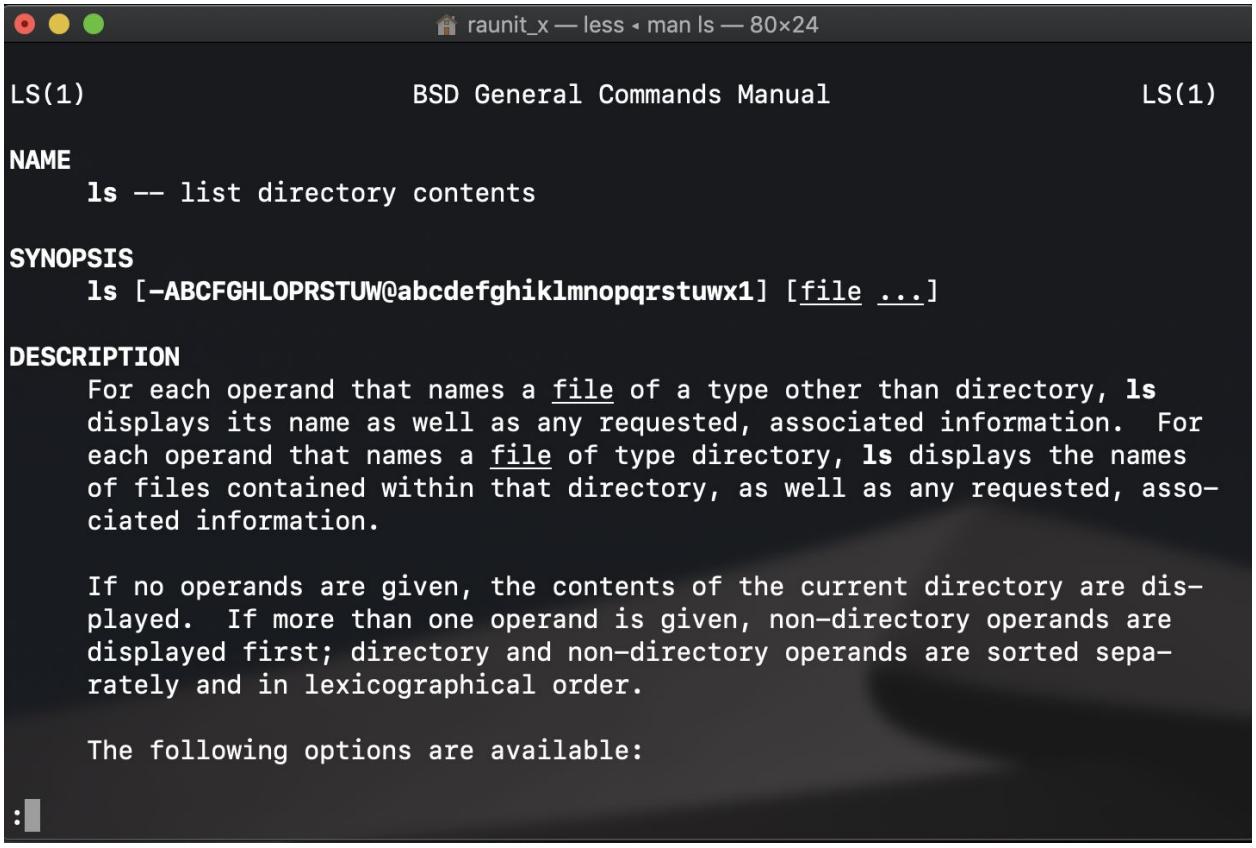
d. **--help** – Display Usage Information

Many executable programs support a “--help” option that displays a description of the command's supported syntax and options.



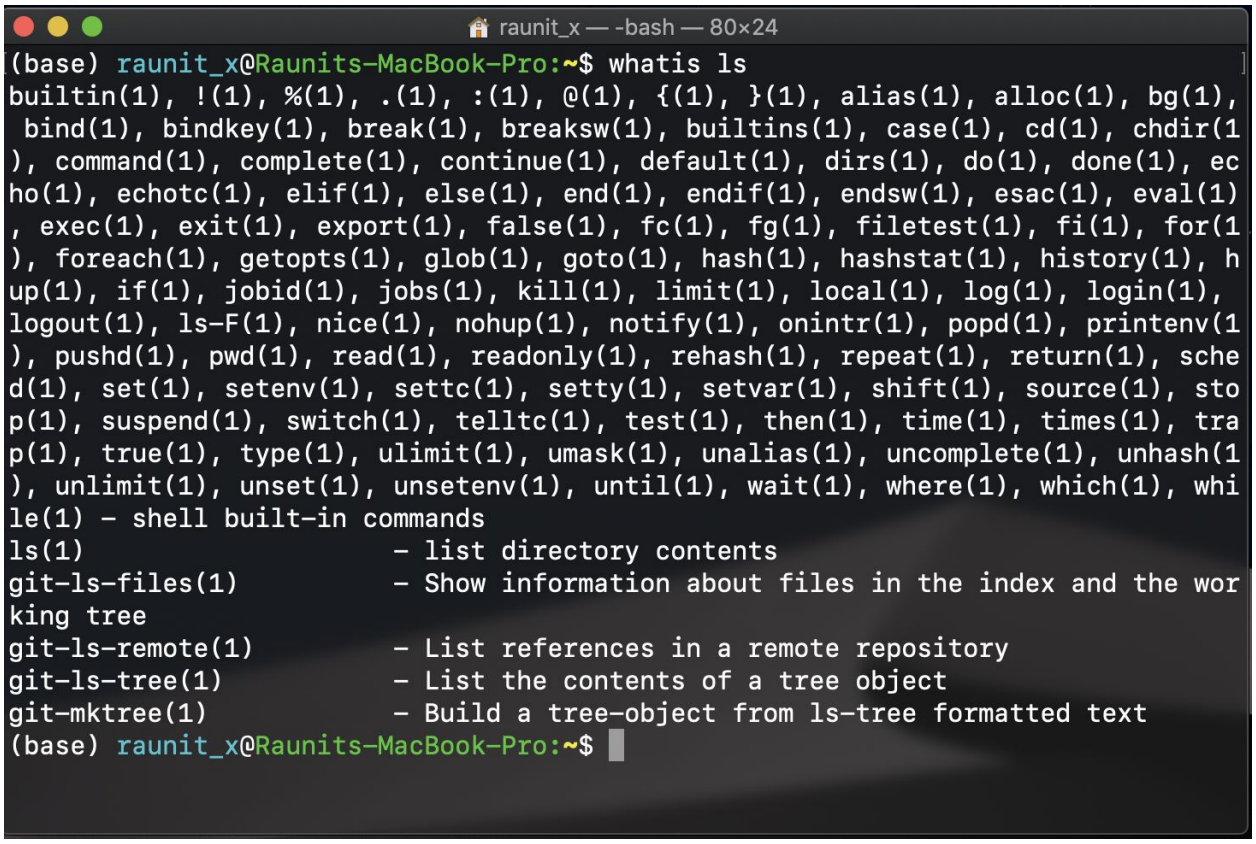
e. man – Display A Program's Manual Page

Most executable programs intended for command line use provide a formal piece of documentation called a manual or man page. A special paging program called man is used to view them.



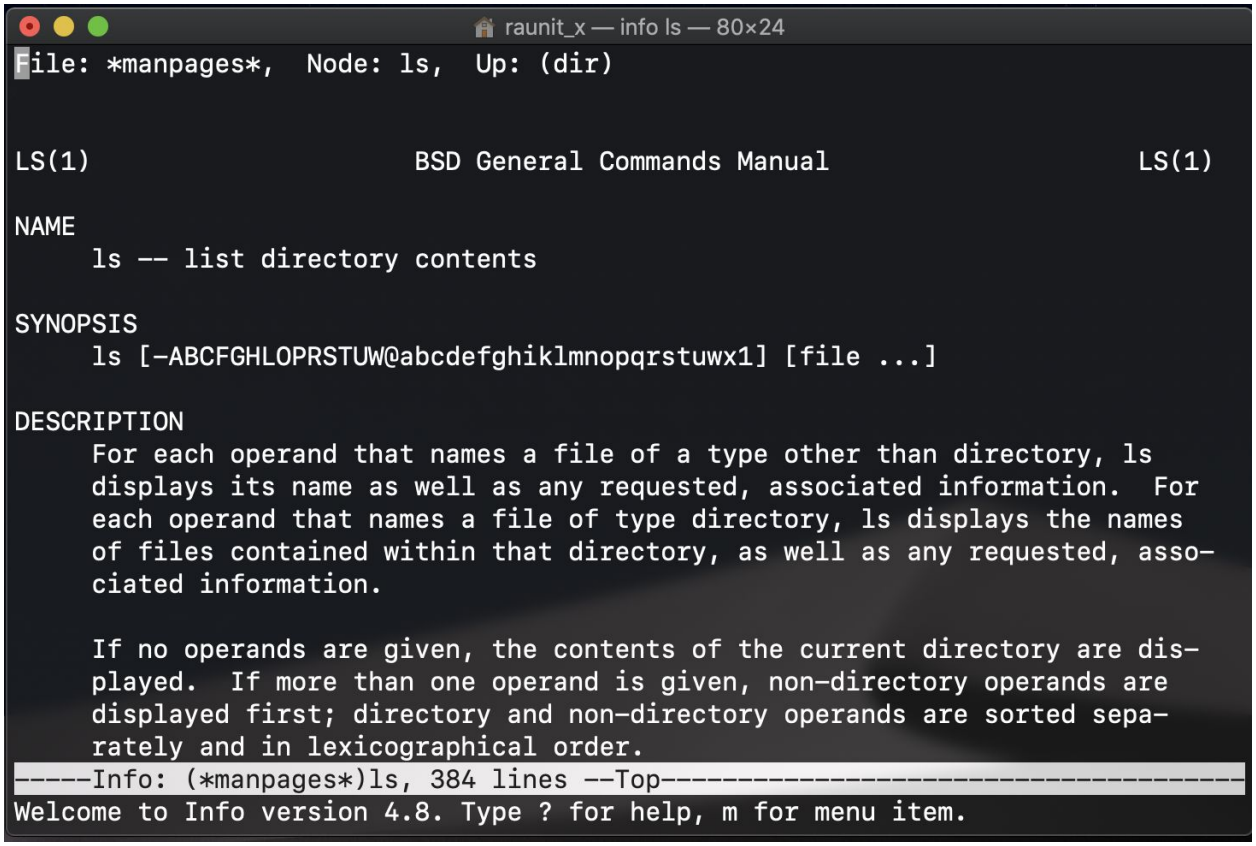
f. whatis – Display One-line Manual Page Descriptions

The whatis program displays the name and a one-line description of a man page matching a specified keyword:

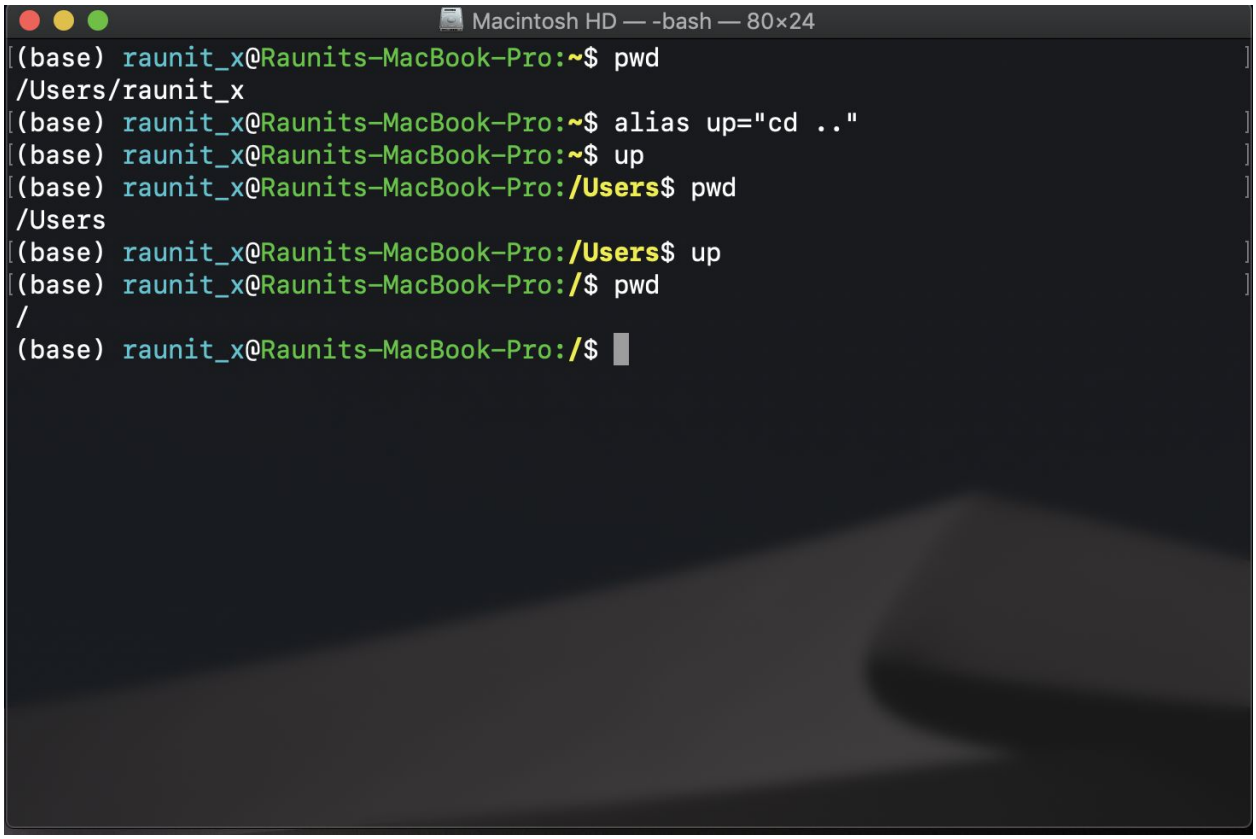


7. info – Display a Program's Info Entry

info command reads documentation in the info format. It will give detailed information for a command when compared with the man page.


raunit_x — info ls — 80×24
File: *manpages*, Node: ls, Up: (dir)
LS(1) BSD General Commands Manual LS(1)
NAME
ls -- list directory contents
SYNOPSIS
ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1] [file ...]
DESCRIPTION
For each operand that names a file of a type other than directory, ls displays its name as well as any requested, associated information. For each operand that names a file of type directory, ls displays the names of files contained within that directory, as well as any requested, associated information.
If no operands are given, the contents of the current directory are displayed. If more than one operand is given, non-directory operands are displayed first; directory and non-directory operands are sorted separately and in lexicographical order.
-----Info: (*manpages*)ls, 384 lines --Top-----
Welcome to Info version 4.8. Type ? for help, m for menu item.

8. alias



Chapter 6 – Redirection

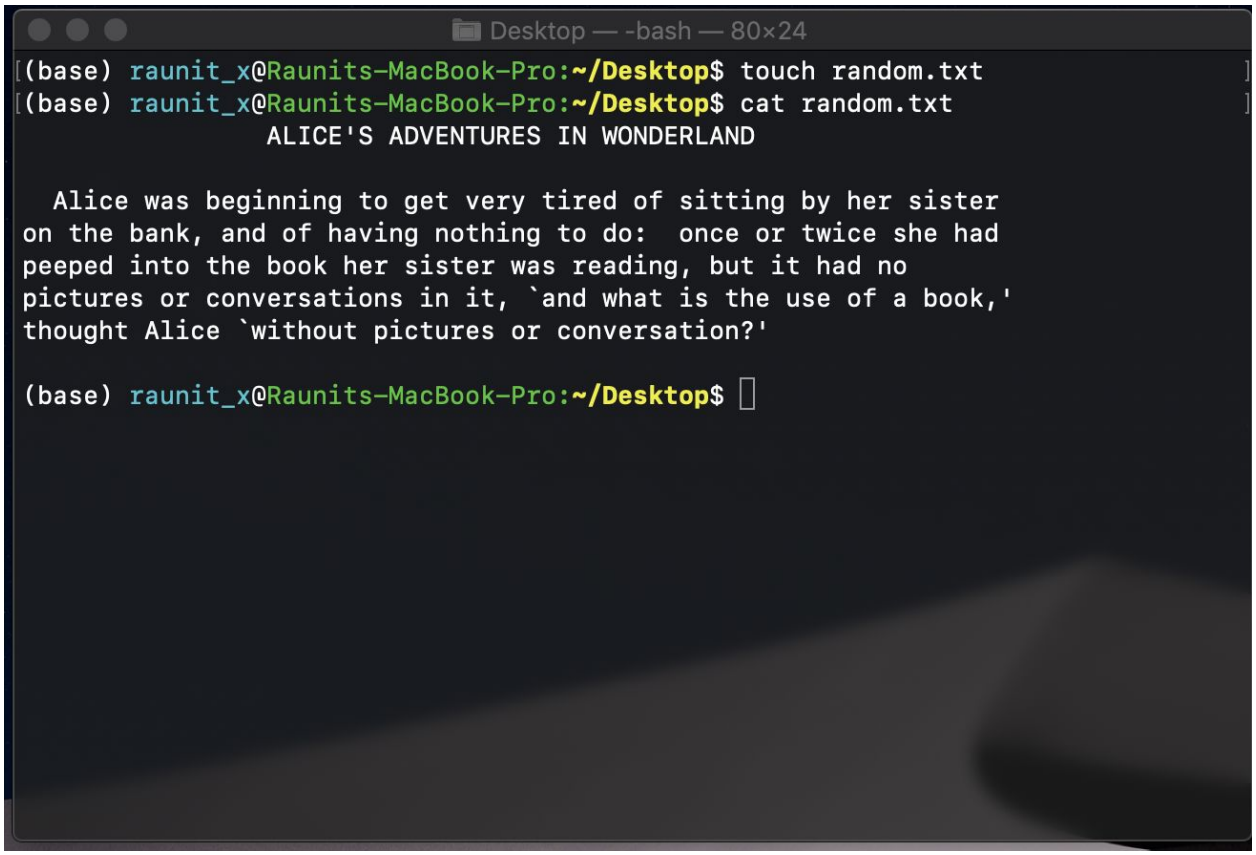
In this lesson we are going to unleash what may be the coolest feature of the command line. It's called I/O redirection. The “I/O” stands for input/output and with this facility you can redirect the input and output of commands to and from files, as well as connect multiple commands together into powerful command pipelines. To show off this facility, we will introduce the following commands:

- cat - Concatenate files
- sort - Sort lines of text
- uniq - Report or omit repeated lines
- grep - Print lines matching a pattern
- wc - Print newline, word, and byte counts for each file
- head - Output the first part of a file
- tail - Output the last part of a file

● tee - Read from standard input and write to standard output and files

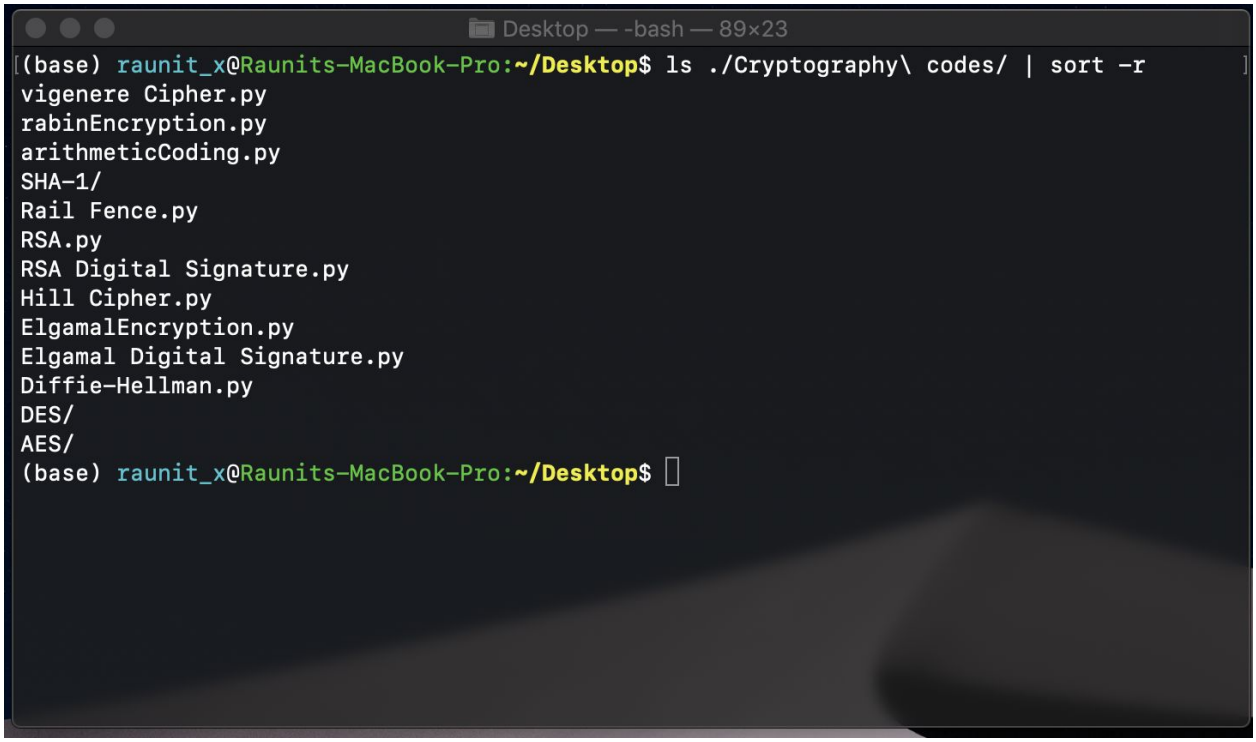
cat – Concatenate Files

The cat command reads one or more files and copies them to standard output like so:
It will display the contents of the file ls-output.txt. cat is often used to display short text files.



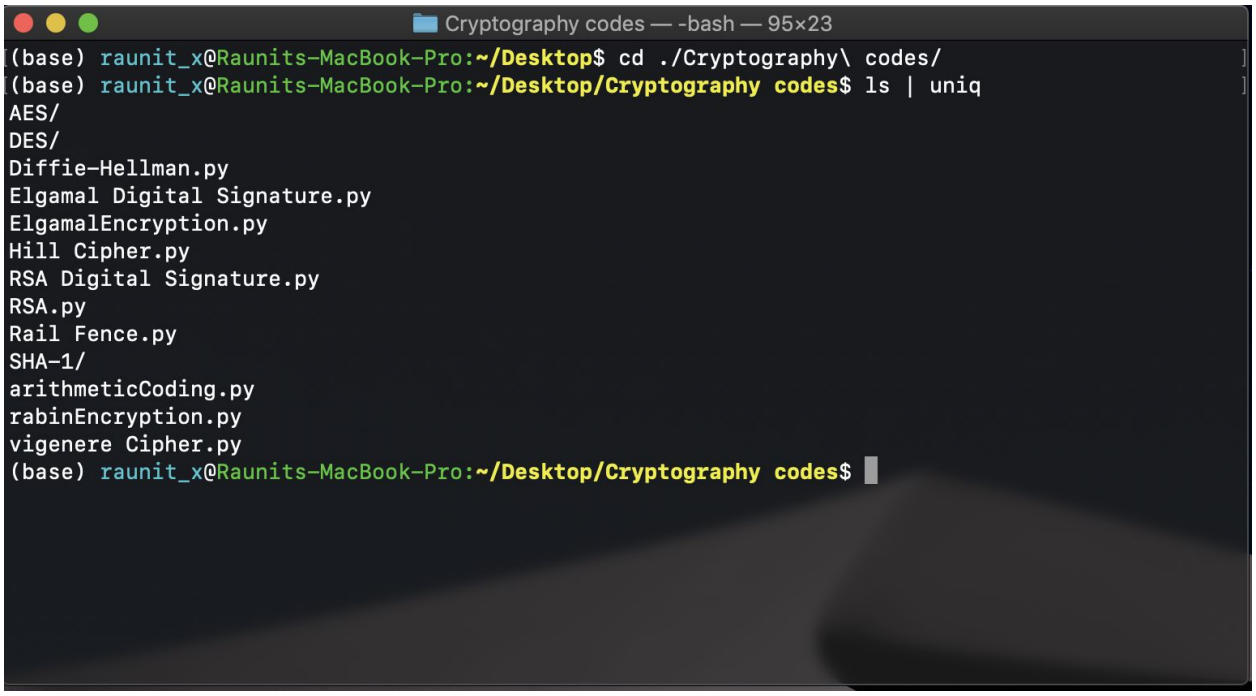
Pipelines

Pipelines are often used to perform complex operations on data. It is possible to put several commands together into a pipeline. Frequently, the commands used this way are referred to as filters. Filters take input, change it somehow and then output it. The first one we will try is sort. Imagine we wanted to make a combined list of all of the executable programs in /bin and /usr/bin, put them in sorted order and view it:

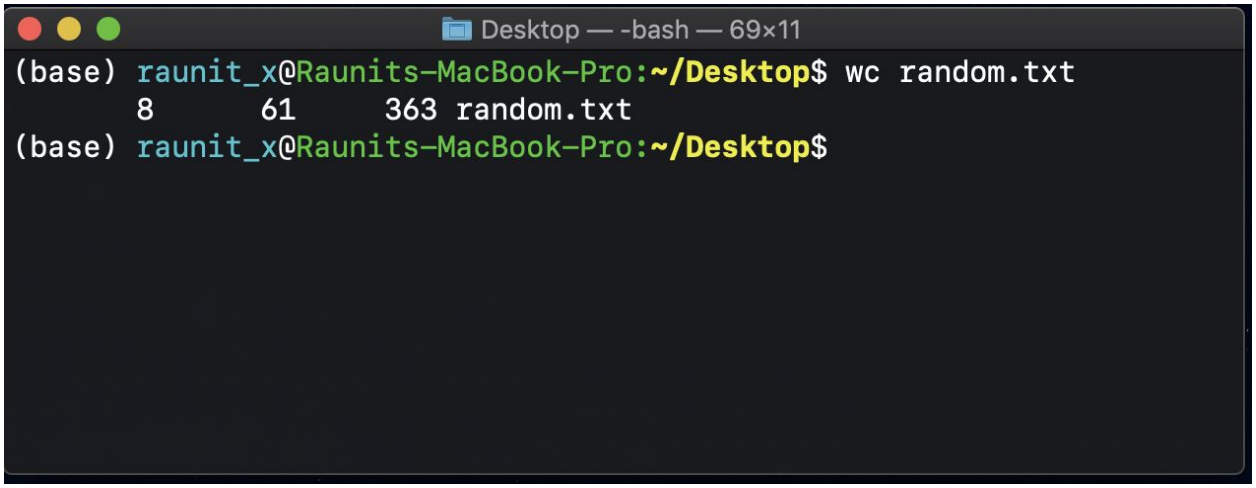


uniq - Report Or Omit Repeated Lines

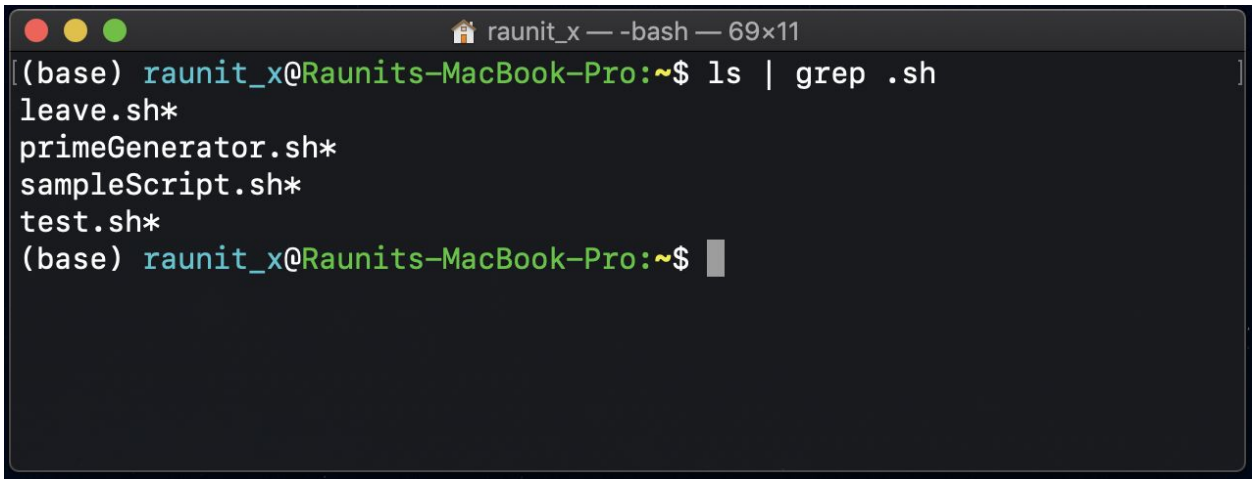
In this example, we use uniq to remove any duplicates from the output of the sort command. If we want to see the list of duplicates instead, we add the “-d” option to uniq like so:



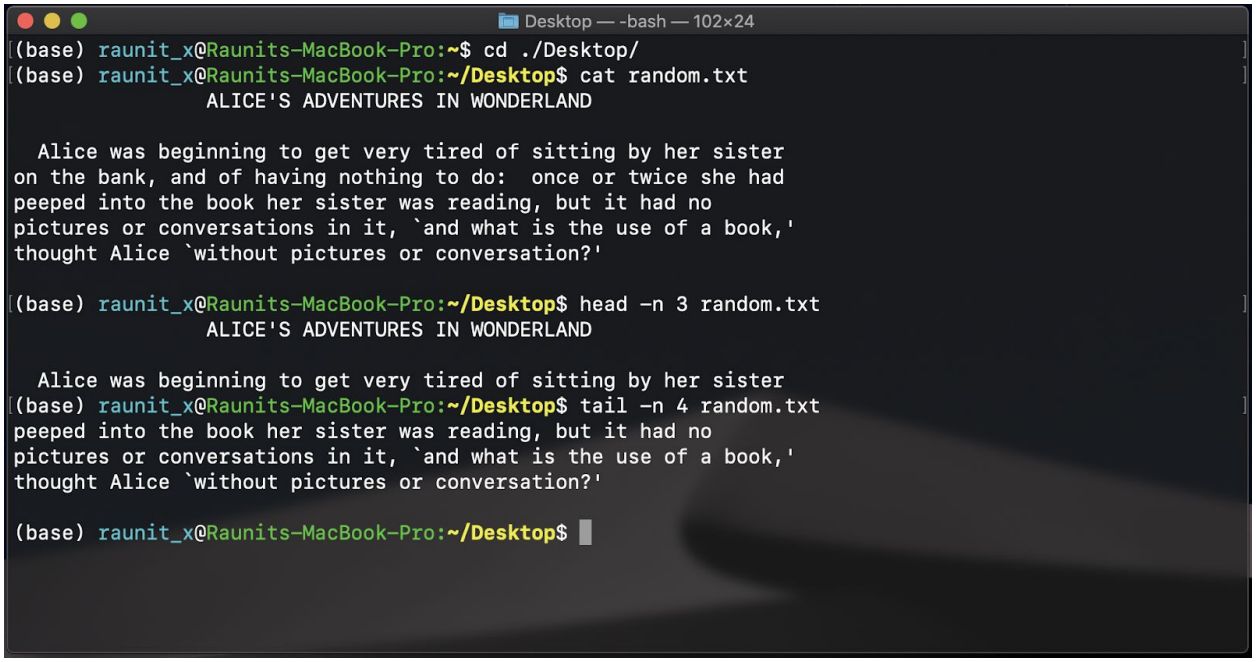
wc – Print Line, Word, And Byte Counts The wc (word count) command is used to display the number of lines, words, and bytes contained in files. For example:



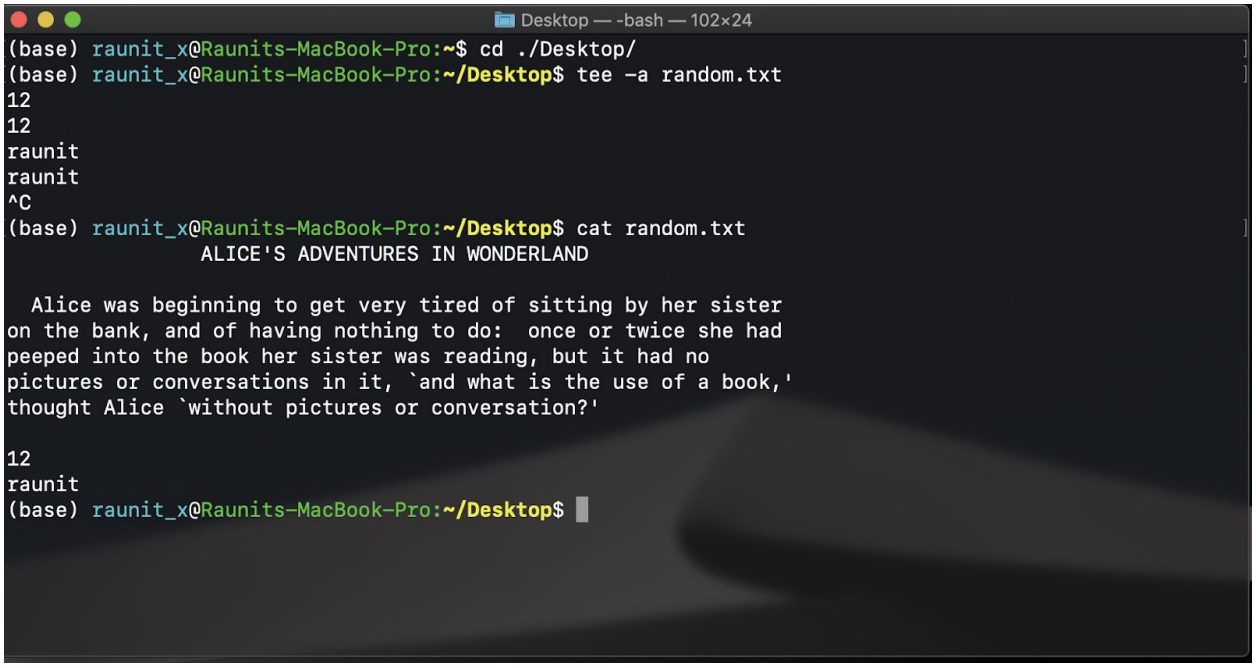
grep – Print Lines Matching A Pattern grep is a powerful program used to find text patterns within files. It's used like this:



head / tail – Print First / Last Part Of Files Sometimes you don't want all the output from a command. You may only want the first few lines or the last few lines. The head command prints the first ten lines of a file and the tail command prints the last ten lines. By default, both commands print ten lines of text, but this can be adjusted with the "-n" option:



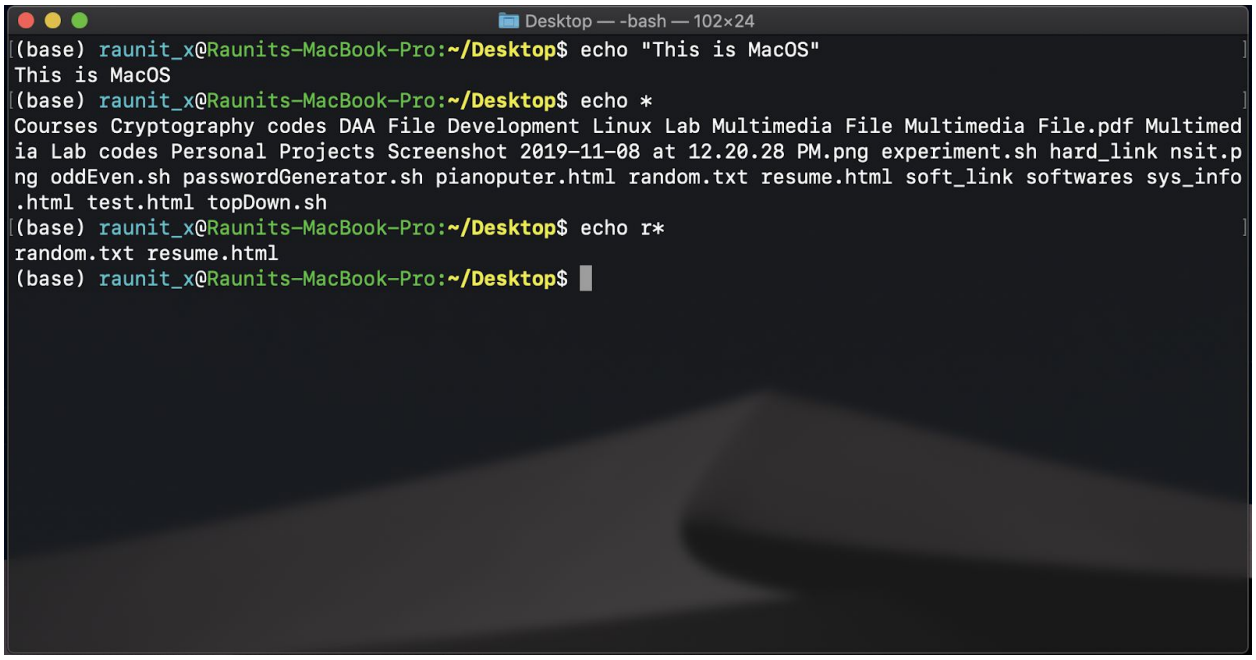
tee – Read From Stdin And Output To Stdout And Files



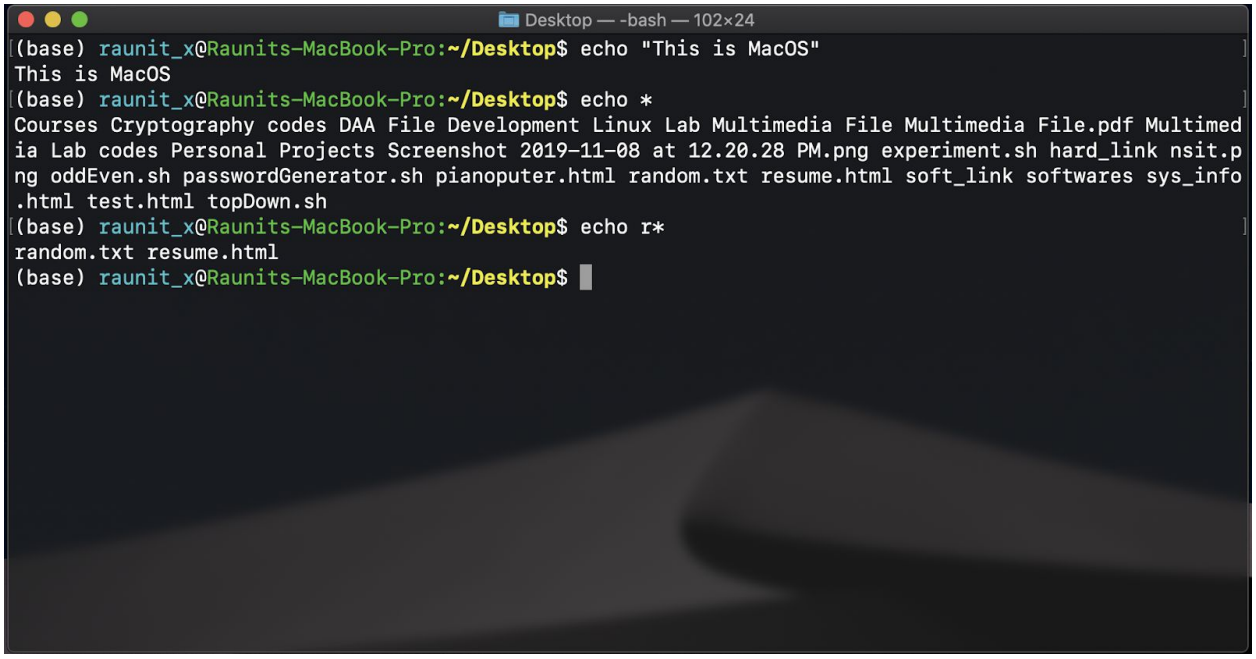
Chapter 7 – Seeing The World As The Shell Sees It

● **echo –** Display a line of text

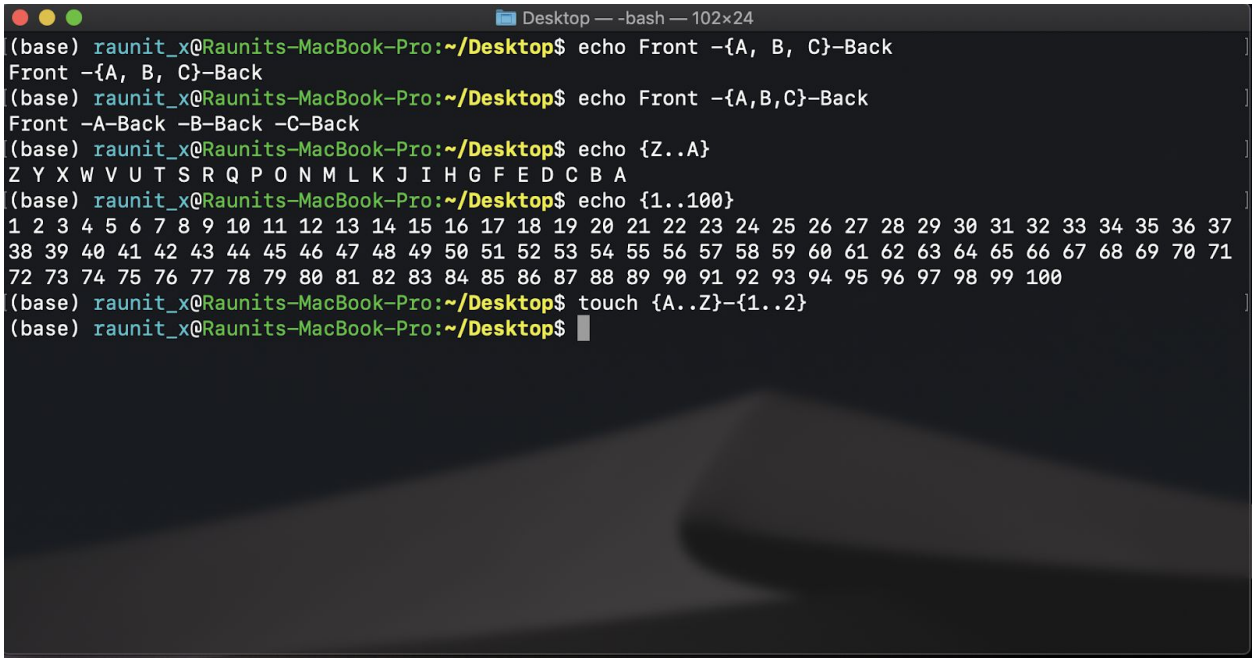
That's pretty straightforward. Any argument passed to echo gets displayed.



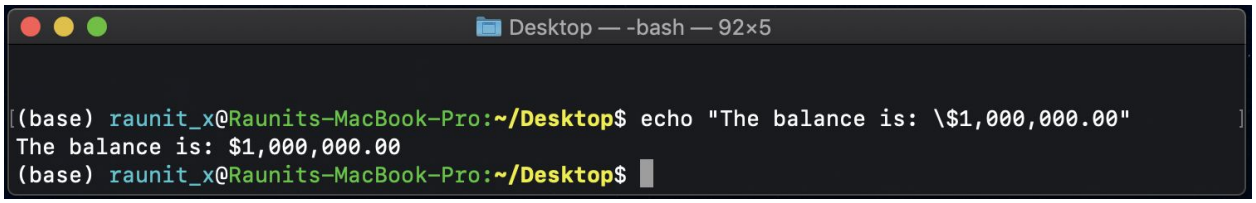
Arithmetic Expansion



Brace Expansion

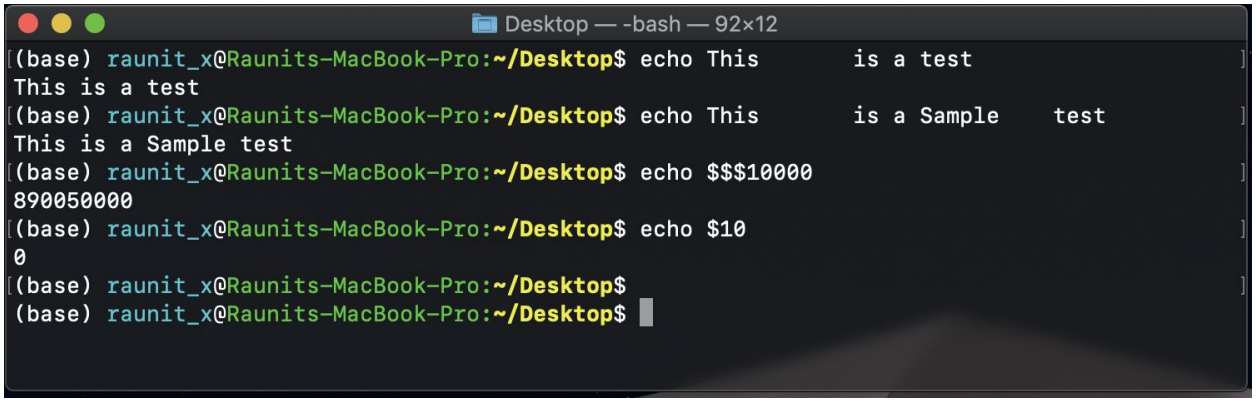


Parameter Expansion



Quoting

Now, we will see how we will control expansions.



In the first example, word-splitting by the shell removed extra whitespace from the echo command's list of arguments. In the second example, parameter expansion substituted an

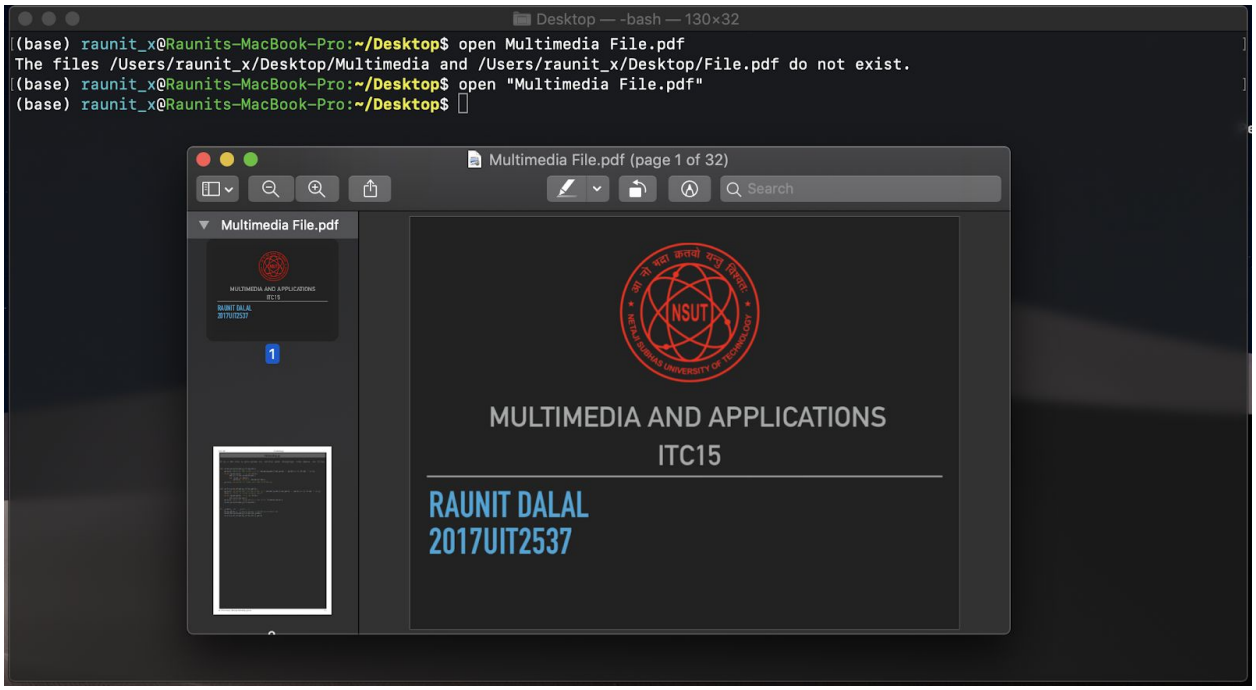
empty string for the value of “$1” because it was an undefined variable. The shell provides a mechanism called quoting to selectively suppress unwanted expansions.

Double Quotes

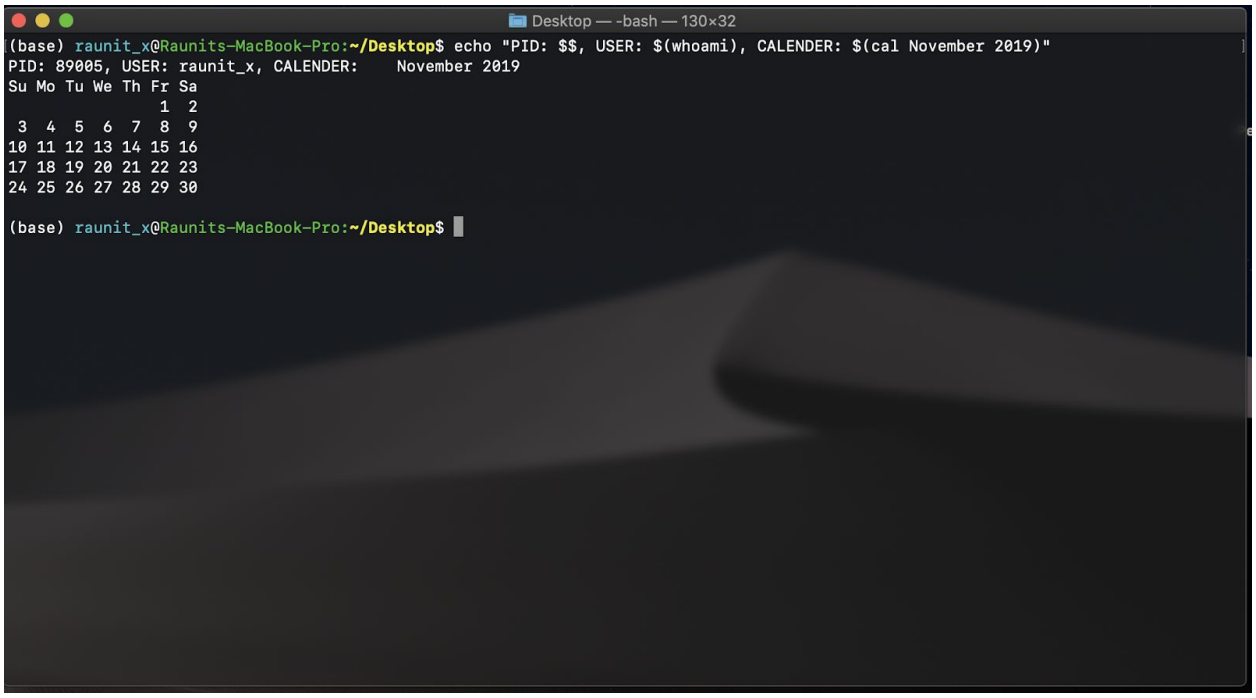
The first type of quoting we will look at is double quotes. If you place text inside double quotes, all the special characters used by the shell lose their special meaning and are treated as ordinary characters. The exceptions are “$”, “\” (backslash), and “`” (backquote). This means that word-splitting, pathname expansion, tilde expansion, and brace expansion are suppressed, but parameter expansion, arithmetic expansion, and command substitution are still carried out.

Using double quotes, we can cope with filenames containing embedded spaces. Say we were the unfortunate victim of a file called two words.txt. If we tried to use this on the command line, word-splitting would cause this to be treated as two separate arguments rather than the desired single argument:

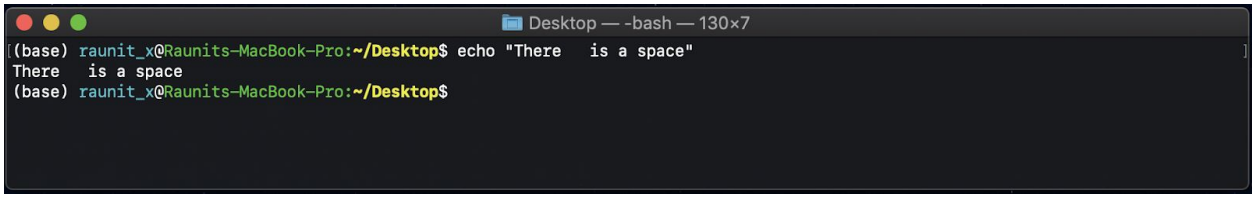
By using double quotes, we stop the word-splitting and get the desired result; further, we can even repair the damage:



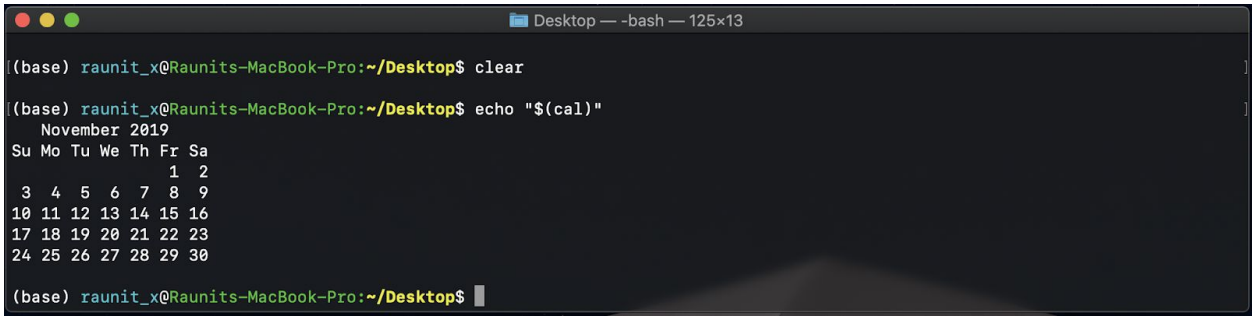
Remember, parameter expansion, arithmetic expansion, and command substitution still take place within double quotes:



By default, word-splitting looks for the presence of spaces, tabs, and newlines (linefeed characters) and treats them as delimiters between words.



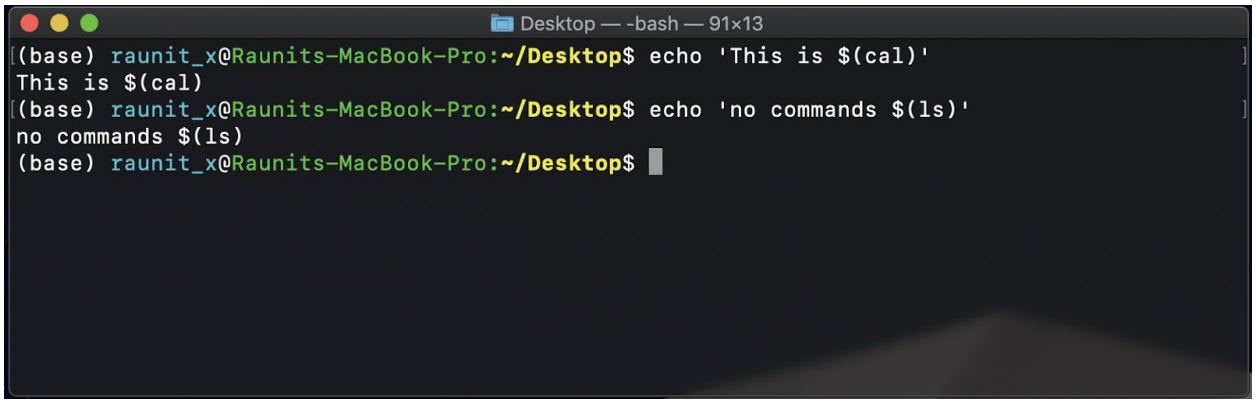
The fact that newlines are considered delimiters by the word-splitting mechanism causes an interesting, albeit subtle, effect on command substitution. Consider the following:



t

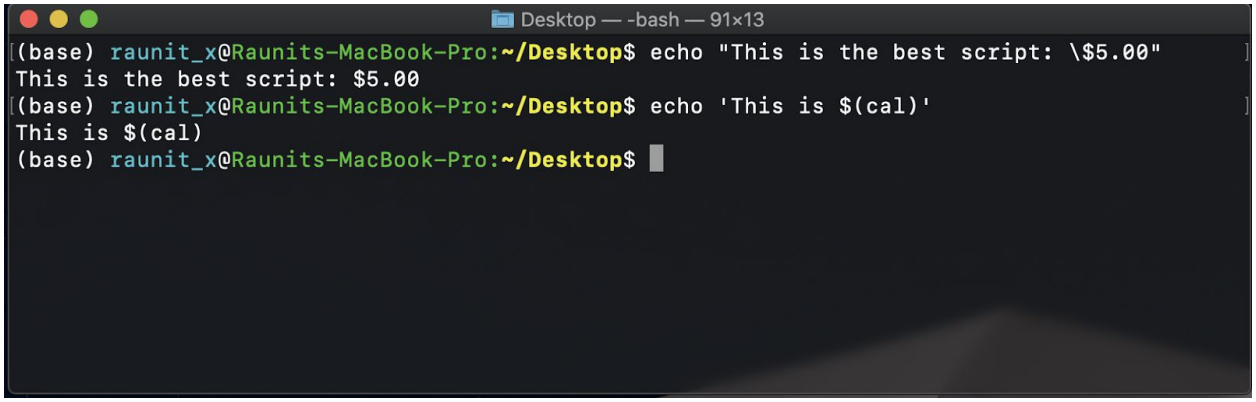
Single Quotes

If we need to suppress all expansions, we use single quotes. Here is a comparison of unquoted, double quotes, and single quotes:



Escaping Characters

Sometimes we only want to quote a single character. To do this, we can precede a character with a backslash, which in this context is called the escape character. Often this is done inside double quotes to selectively prevent an expansion:



Chapter 10-Permissions

While a typical computer will likely have only one keyboard and monitor, it can still be used by more than one user. For example, if a computer is attached to a network or the Internet, remote users can log in via ssh (secure shell) and operate the computer. In fact, remote users can execute graphical applications and have the graphical output appear on a remote display. The X Window System supports this aspart of its basic design.

n the Unix security model, a user may own files and directories. When a user owns a file or directory, the user has control over its access. Users can, in turn, belong to a group consisting of one or more users who are given access to files and directories by their owners. In addition to granting access to a group, an owner may also grant some set of access rights to everybody, which in Unix terms is referred to as the world.

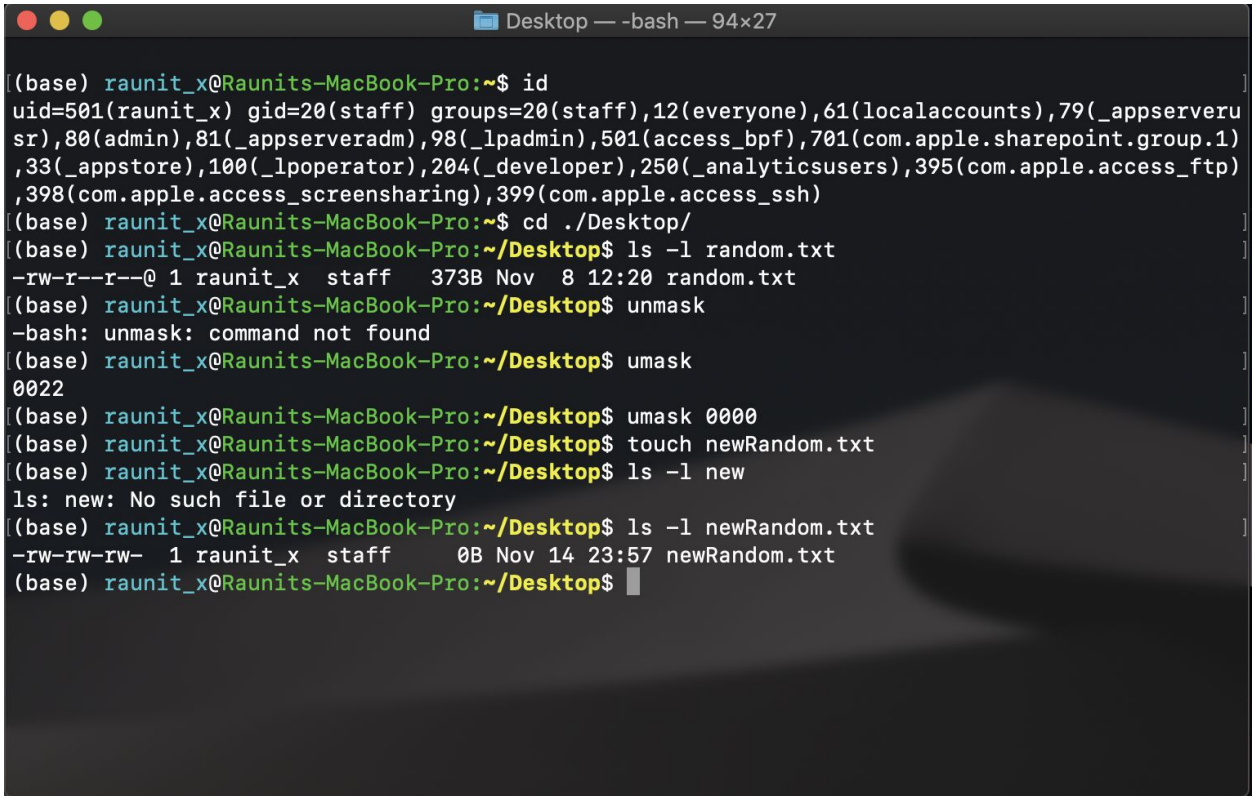
When user accounts are created, users are assigned a number called a user ID or uid which is then, for the sake of the humans, mapped to a user name.The user is assigned a primary group ID or gid and may belong to additional groups.

User accounts are defined in the /etc/passwd file and groups are defined in the /etc/group file. When user accounts and groups are created, these files are modified along with /etc/shadow which holds information about the user's password.For each user account, the /etc/passwd file defines the user (login) name, uid, gid,the account's real name, home directory, and login shell. If you examine the contents of/etc/passwd and /etc/group, you will notice that besides the regular useraccounts, there are accounts for the superuser (uid 0) and various other system users.

When viewed in long format, the first ten characters of the listing are the file attributes. The first of these characters is the file type.
The remaining nine characters of the file attributes, called the file mode, represent the read, write, and execute permissions for the file's owner, the file's group owner, and everybody else.

chmod – Change file mode
To change the mode (permissions) of a file or directory, the chmod command is used. Be aware that only the file’s owner or the superuser can change the mode of a file or directory. chmod supports two distinct ways of specifying mode changes: octal number representation, or symbolic representation. Symbolic or octal notations can be used to change permissions.

umask – Set Default Permissions
The umask command controls the default permissions given to a file when it is created.



Changing Identities

At various times, we may find it necessary to take on the identity of another user. Often we want to gain superuser privileges to carry out some administrative task, but it is also possible to "become" another regular user for such things as testing an account. There are three ways to take on an alternate identity:

1.Log out and log back in as the alternate user.
2.Use the **su** command.
3.Use the **sudo** command.

Superuser - system administrator

su – Run A Shell With Substitute User And Group IDs
The su command is used to start a shell as another user.

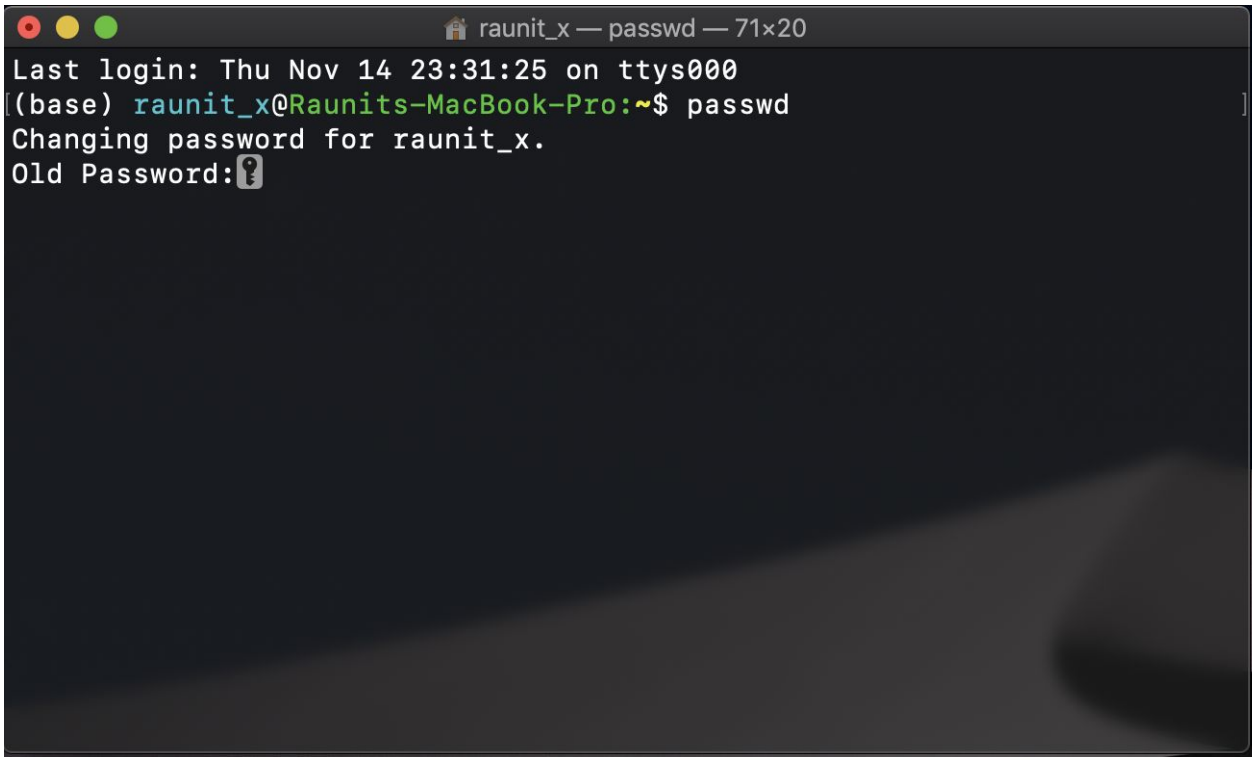
sudo – Execute A Command As Another User
The sudo command is like su in many ways, but has some important additional capabilities. The administrator can configure sudo to allow an ordinary user to execute commands as a different user (usually the superuser) in a very controlled way. In particular, a user may be restricted to one or more specific commands and no others. Another important difference is that the use of sudo does not require access to the superuser's password. To authenticate using sudo, the user uses his/her own password.

After entering the command, we are prompted for our password (not the superuser's) andonce the authentication is complete, the specified command is carried out.
One important difference between su and sudo is that sudo does not start a new shell, nor does it load another user's environment.
By default,Ubuntu disables logins to the root account (by failing to set a password for the account), and instead uses sudo to grant superuser privileges.

Apt-get, snap etc. are used to install/remove packages along with the sudo command.

chown – Change File Owner And Group
The chown command is used to change the owner and group owner of a file or directory. Superuser privileges are required to use this command.

To set or change a password, the **passwd** command is used.


raunit_x — passwd — 71×20
Last login: Thu Nov 14 23:31:25 on ttys000
(base) raunit_x@Raunits-MacBook-Pro:~$ passwd
Changing password for raunit_x.
Old Password:

Writing Shell Scripts

Chapter-24

What Are Shell Scripts?

In the simplest terms, a shell script is a file containing a series of commands. The shell reads this file and carries out the commands as though they have been entered directly on the command line.

The shell is somewhat unique, in that it is both a powerful command line interface to the system and a scripting language interpreter. As we will see, most of the things that can be done on the command line can be done in scripts, and most of the things that can be done in scripts can be done on the command line.

We have covered many shell features, but we have focused on those features most often used directly on the command line. The shell also provides a set of features usually (but not always) used when writing programs.

How To Write A Shell Script

To successfully create and run a shell script, we need to do three things:

1. Write a script. Shell scripts are ordinary text files. So we need a text editor to write them. The best text editors will provide syntax highlighting, allowing us to see a color-coded view of the elements of the script. Syntax highlighting will help us spot certain kinds of common errors. vim, gedit, kate, and many other editors are good candidates for writing scripts.

2. Make the script executable. The system is rather fussy about not letting any old text file be treated as a program, and for good reason! We need to set the script file's permissions to allow execution.

3. Put the script somewhere the shell can find it. The shell automatically searches certain directories for executable files when no explicit pathname is specified. For maximum convenience, we will place our scripts in these directories.

Script File Format

In keeping with programming tradition, we'll create a "hello world" program to demonstrate an extremely simple script. So let's fire up our text editors and enter the following script:

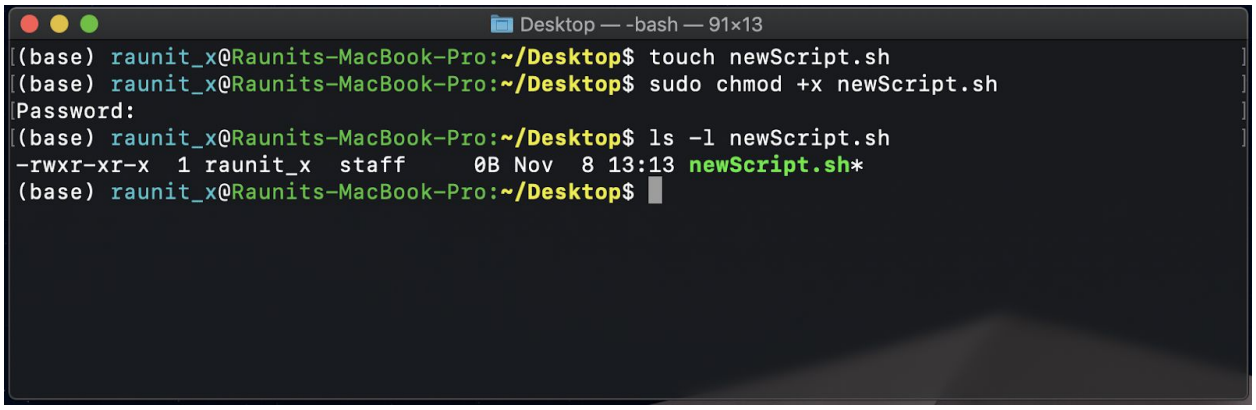
```
#!/bin/bash

# This is our first script.

echo 'Hello World!'
```

Executable Permissions

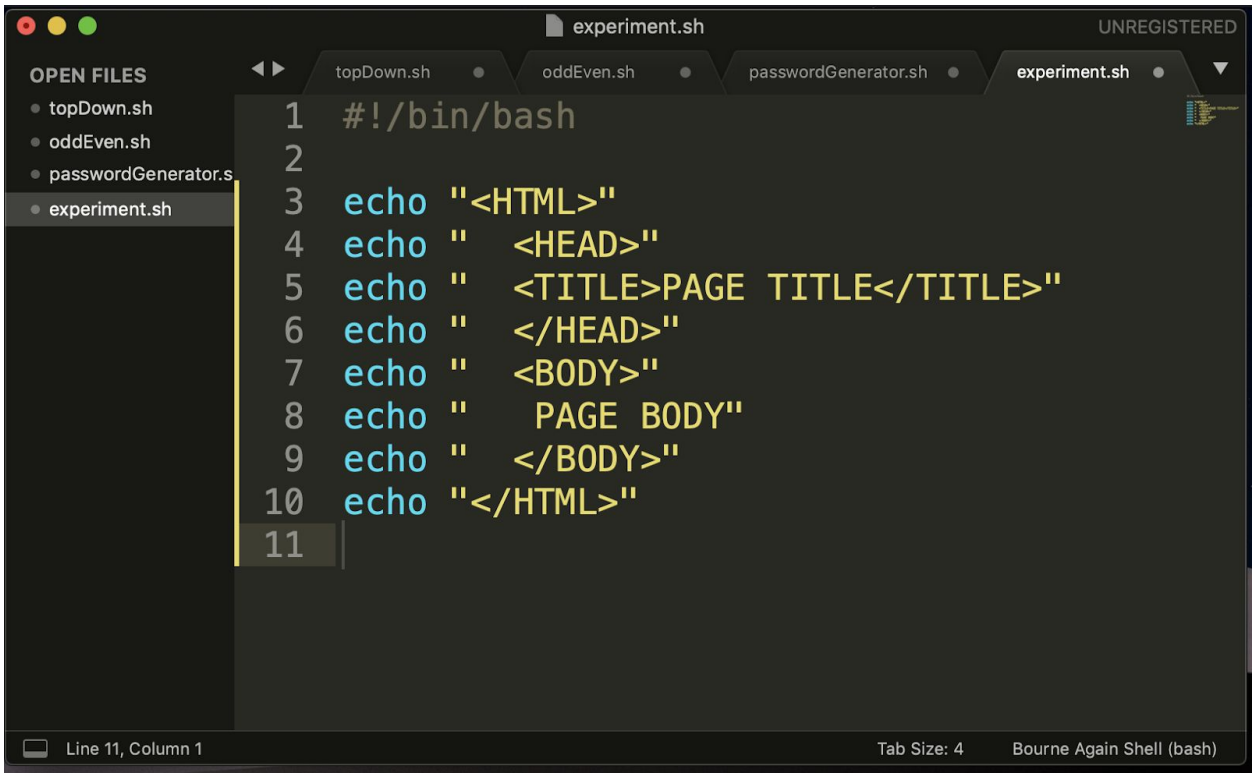
The next thing we have to do is make our script executable. This is easily done using chmod:



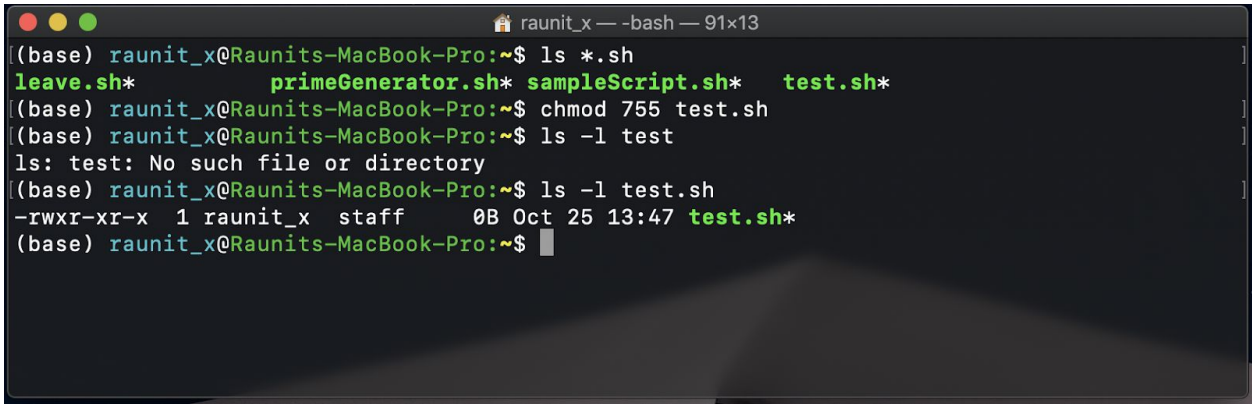
There are two common permission settings for scripts; 755 for scripts that everyone can execute, and 700 for scripts that only the owner can execute. Note that scripts must be readable in order to be executed.

Chapter 25 – Starting A Project

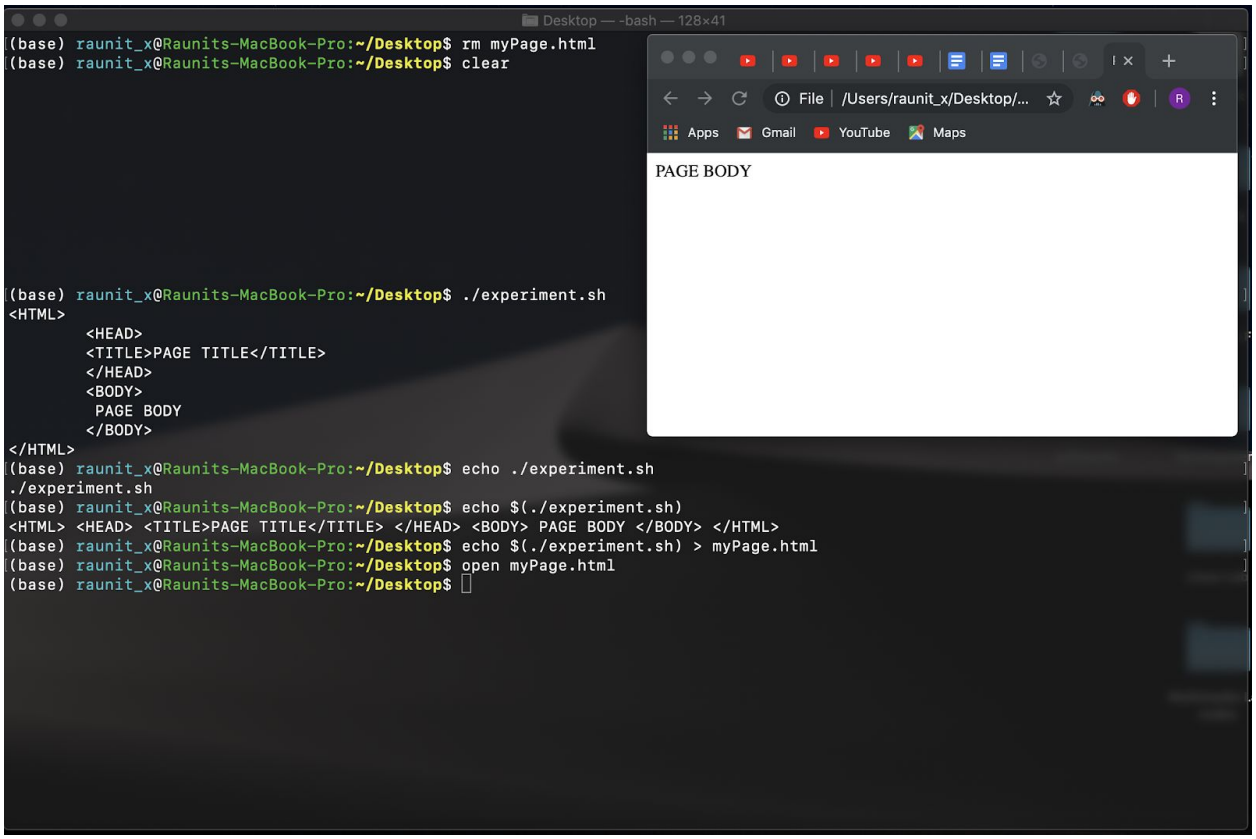
The program we will write is a report generator. It will present various statistics about our system and its status, and will produce this report in HTML format, so we can view it with a web browser such as Firefox or Chrome.



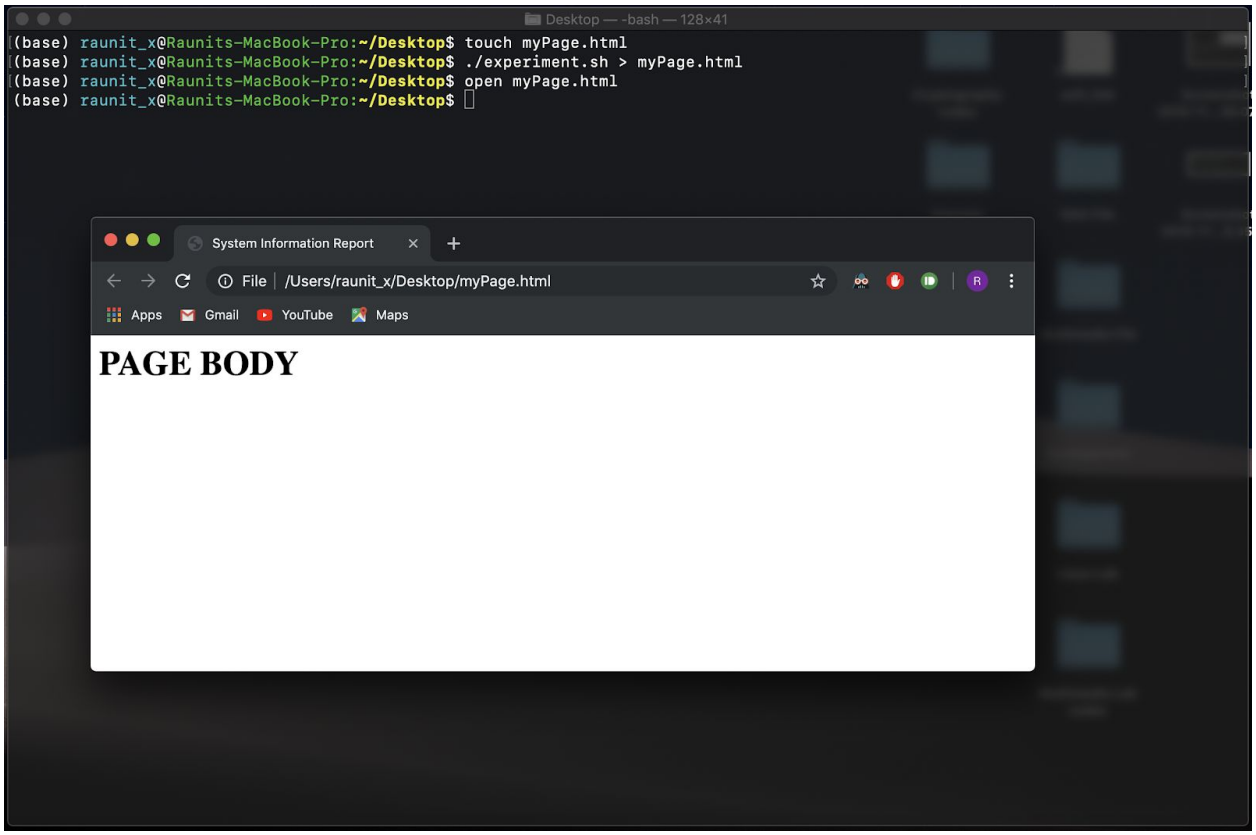
Our first attempt at this problem contains a shebang, a comment (always a good idea) and a sequence of echo commands, one for each line of output. After saving the file, we'll make it executable and attempt to run it:



When the program runs, we should see the text of the HTML document displayed on the screen, since the echo commands in the script send their output to standard output. We'll run the program again and redirect the output of the program to the file sys_info_page.html, so that we can view the result with a web browser:

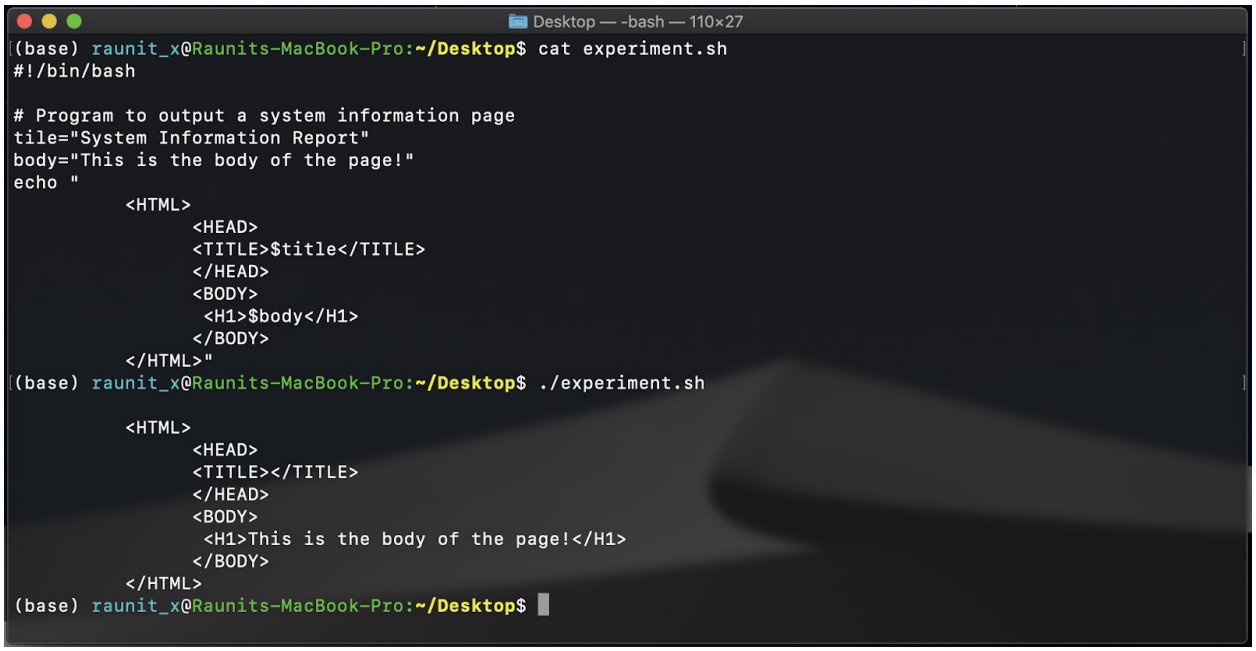


Second Stage: Adding A Little Data



We added a page title and a heading to the body of the report.

Variables And Constants

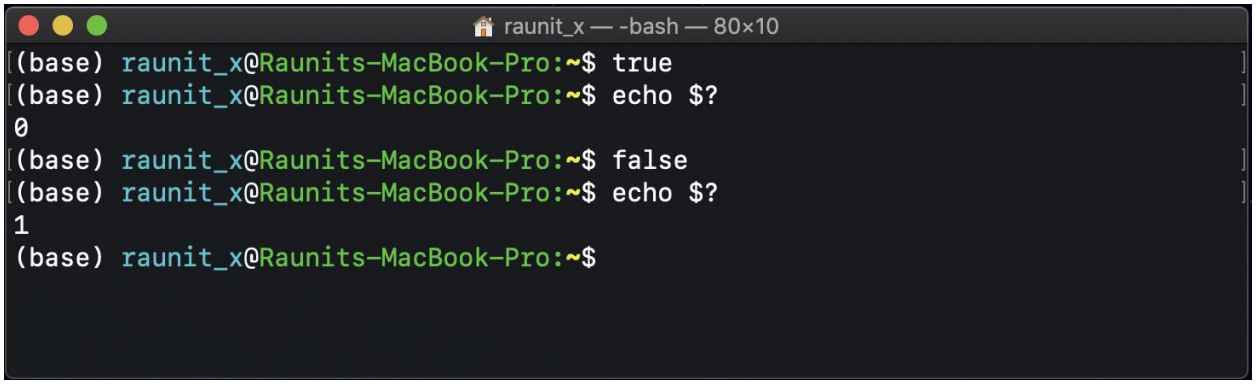


By creating a variable named title and assigning it the value “System Information Report”, we can take advantage of parameter expansion and place the string in multiple locations.

CHAPTER 28 - Flow Control: Branching With if

The **if** statement has the following syntax:
if commands; then commands
[elif commands;
Then commands...]
[else commands]
fi

Exit Status
Commands (including the scripts and shell functions we write) issue a value to the system when they terminate, called an exit status. This value, which is an integer in the range of0 to 255, indicates the success or failure of the command’s execution. By convention, a value of zero indicates success and any other value indicates failure.
The shell provides two extremely simple builtin commands that do nothing except terminate with either a zero or one exit status. The true command always executes successfully and the false command always executes unsuccessfully.
echo $? tells us the exit status value



Test

The test command performs a variety of checks and comparisons.

Syntax:

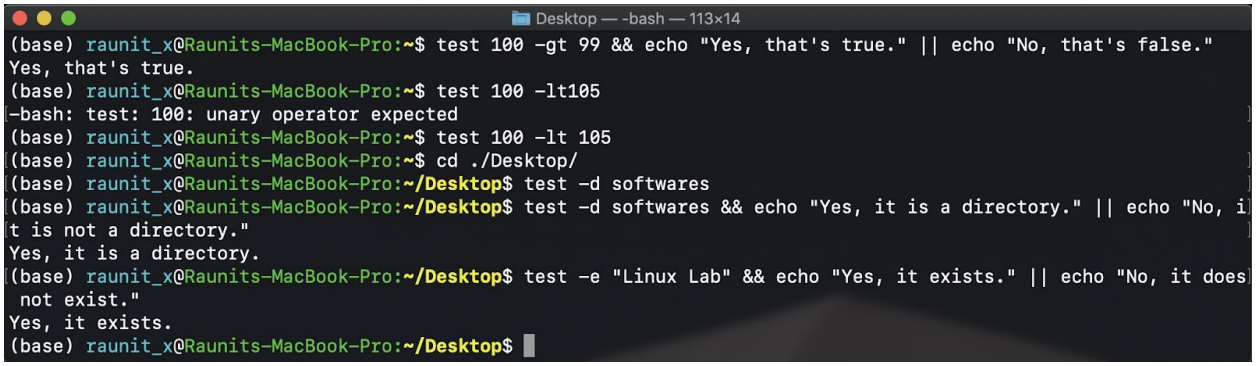
test expression

Or

[expression]

Some options:

-d file	file exists and is a directory.
-e file	file exists.
-f file	file exists and is a regular file
-w file	file exists and is writable
-x file	file exists and is executable



Compound test commands:

Recent versions of bash include a compound command that acts as an enhanced replacement for test. It uses the following syntax:

[[expression]]

where, like test, expression is an expression that evaluates to either true or false result. The [[ ]] command is very similar to test (it supports all of its expressions), but adds an important new string expression:

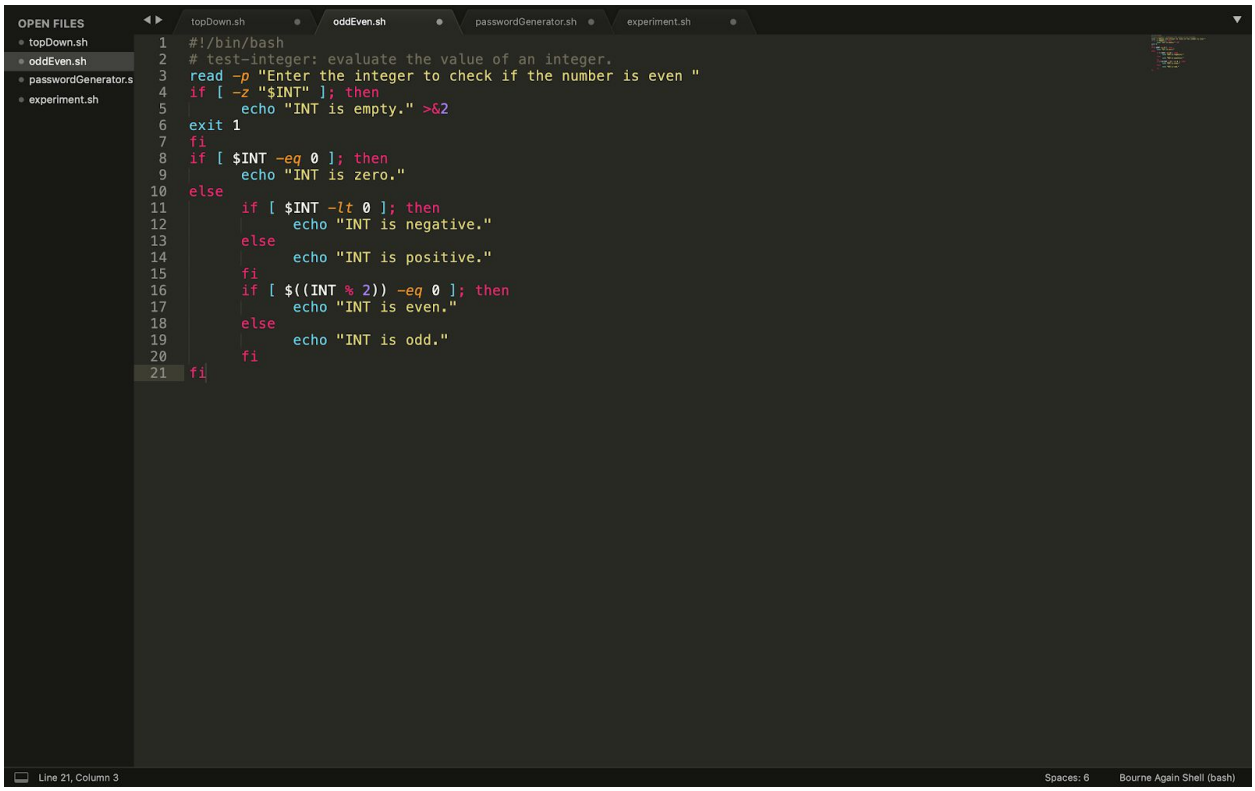
string1 =~ regex

which returns true if string1 is matched by the extended regular expression regex. This opens up a lot of possibilities for performing such tasks as data validation.

(( )) is used to perform arithmetic truth tests. An arithmetic truth test results in true if the result of the arithmetic evaluation is non-zero.

```
Desktop — -bash — 90×8
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ ./oddEven.sh
INT is negative.
INT is odd.
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$
```

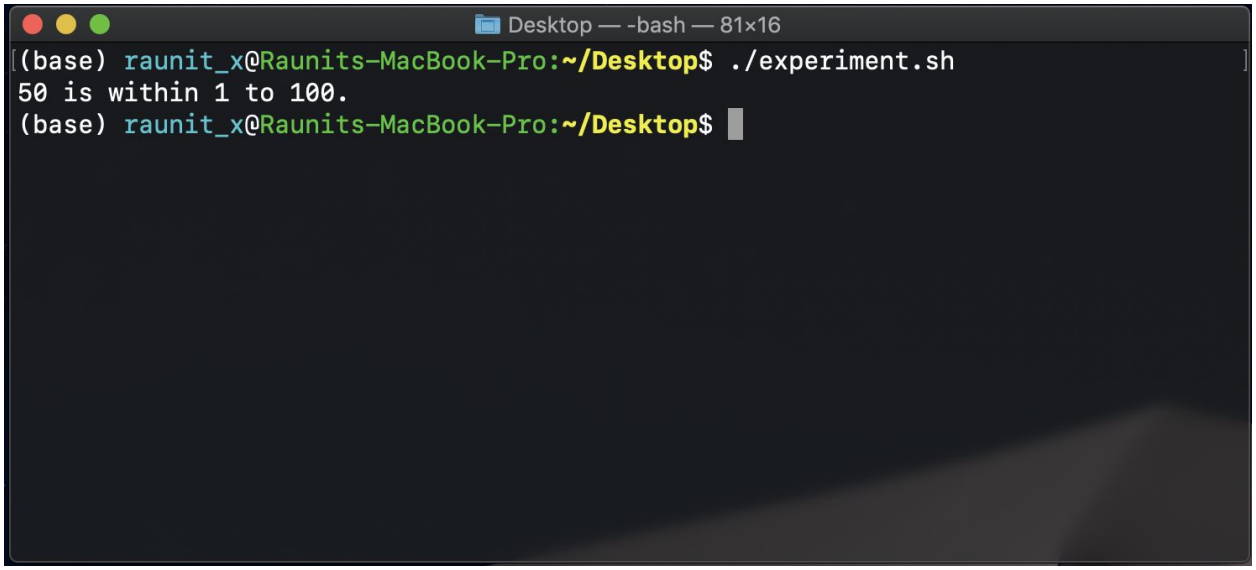
Corresponding Script to check sign and odd/even:

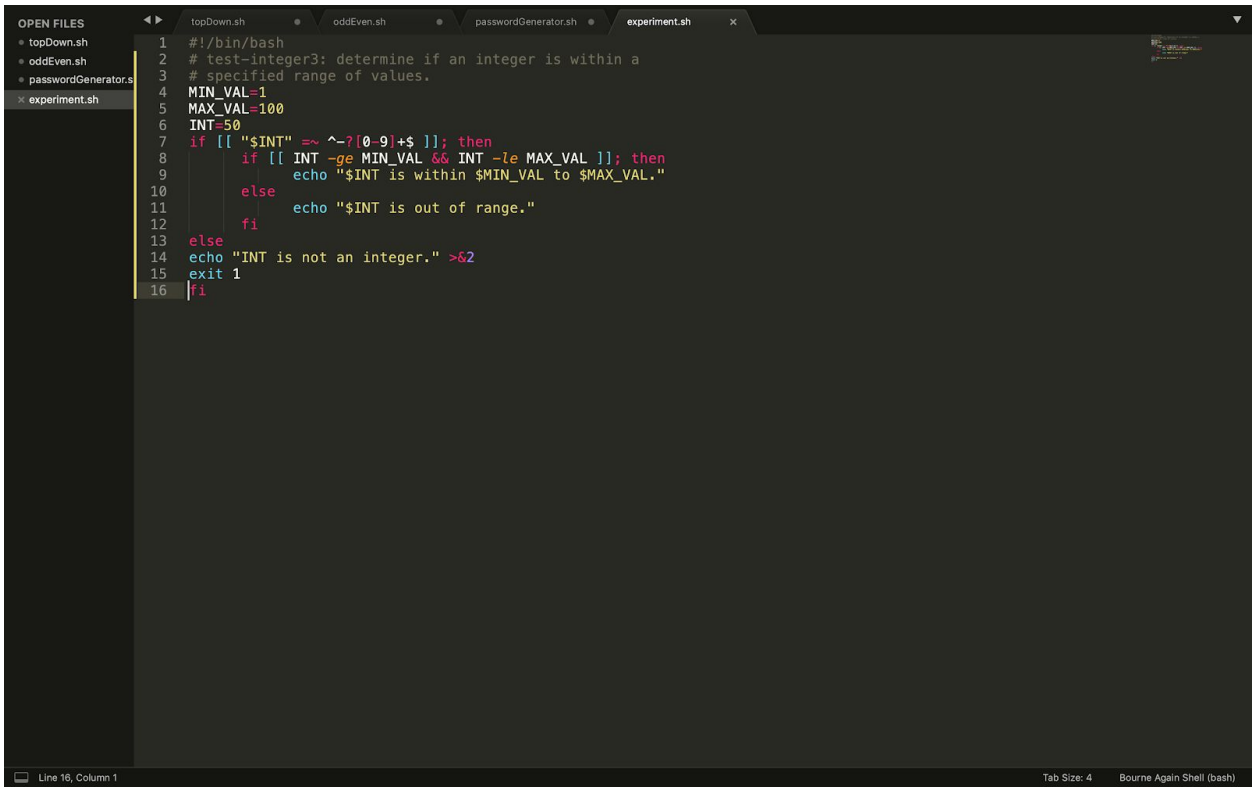


By applying the regular expression, we are able to limit the value of INT to only strings that begin with an optional minus sign, followed by one or more numerals. This expression also eliminates the possibility of empty values.

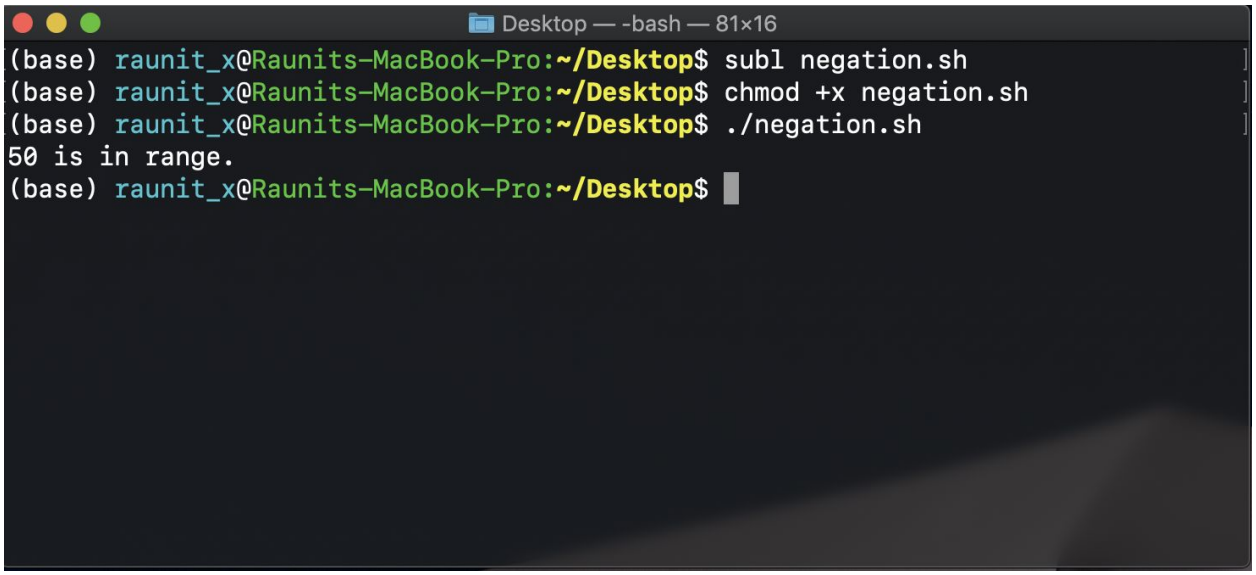
Logical Operators

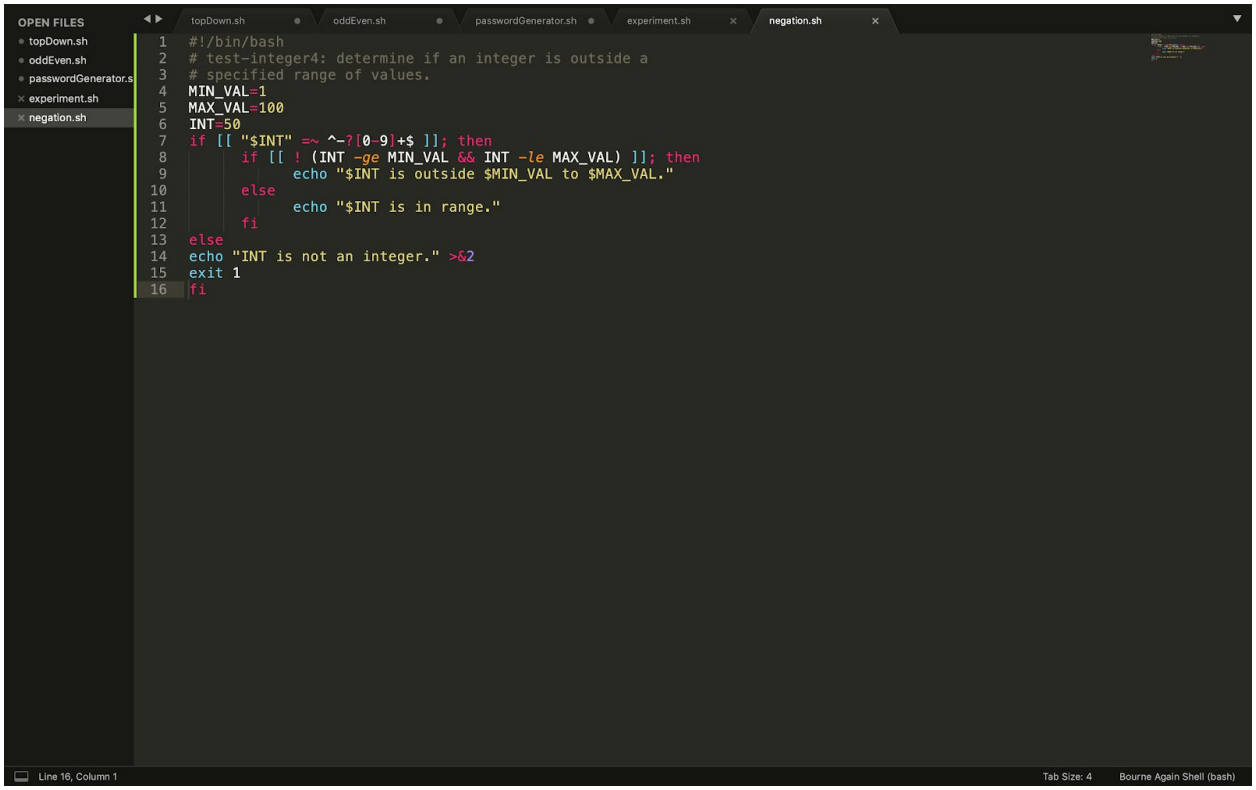
It's also possible to combine expressions to create more complex evaluations. Expressions are combined by using logical operators. We saw these in Chapter 17, when we learned about the find command. There are three logical operations for test and [[ ]]. They are AND, OR and NOT. test and [[ ]] use different operators to represent these operations.





Similar script with negation operator:





Control Operators

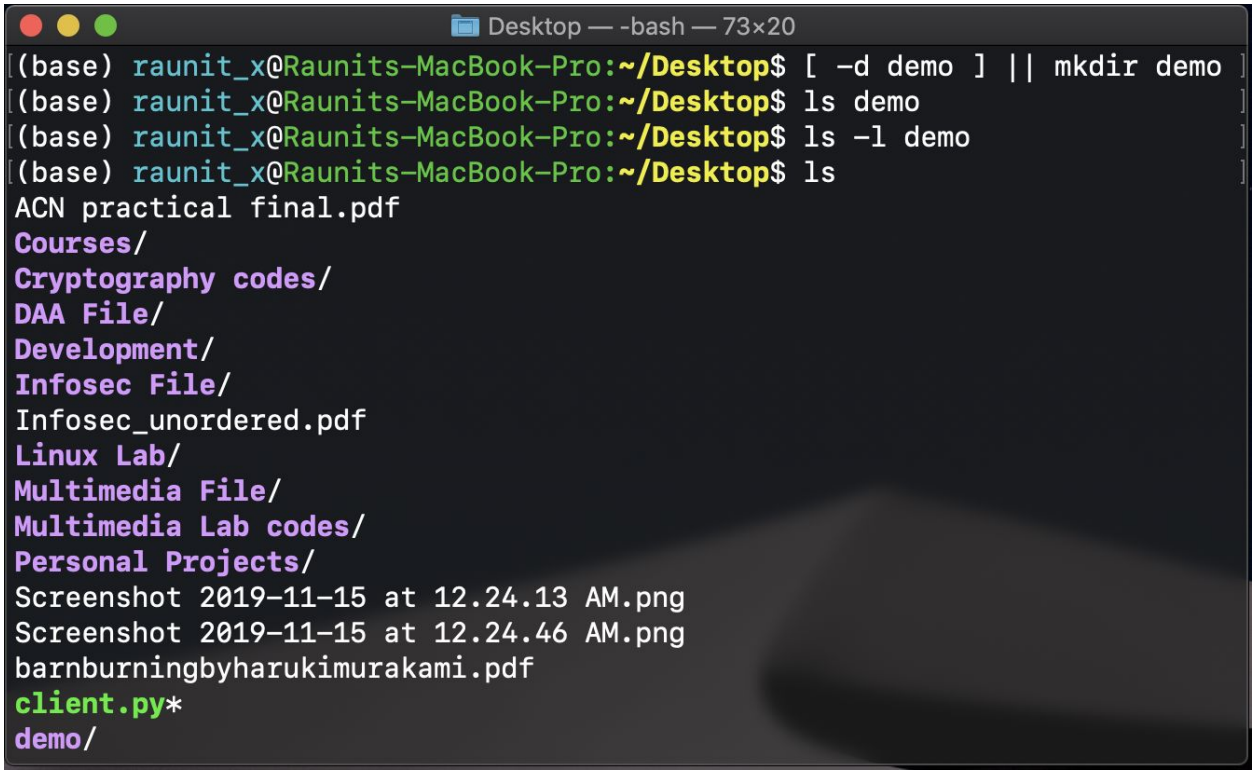
bash provides two control operators that can perform branching. The && (AND) and ||(OR) operators work like the logical operators in the [[ ]] compound command. This is the syntax:

command1 && command2

And

command1 || command2

It is important to understand the behavior of these. With the && operator, command1 is executed and command2 is executed if, and only if, command1 is successful. With the|| operator, command1 is executed and command2 is executed if, and only if, command1 is unsuccessful.

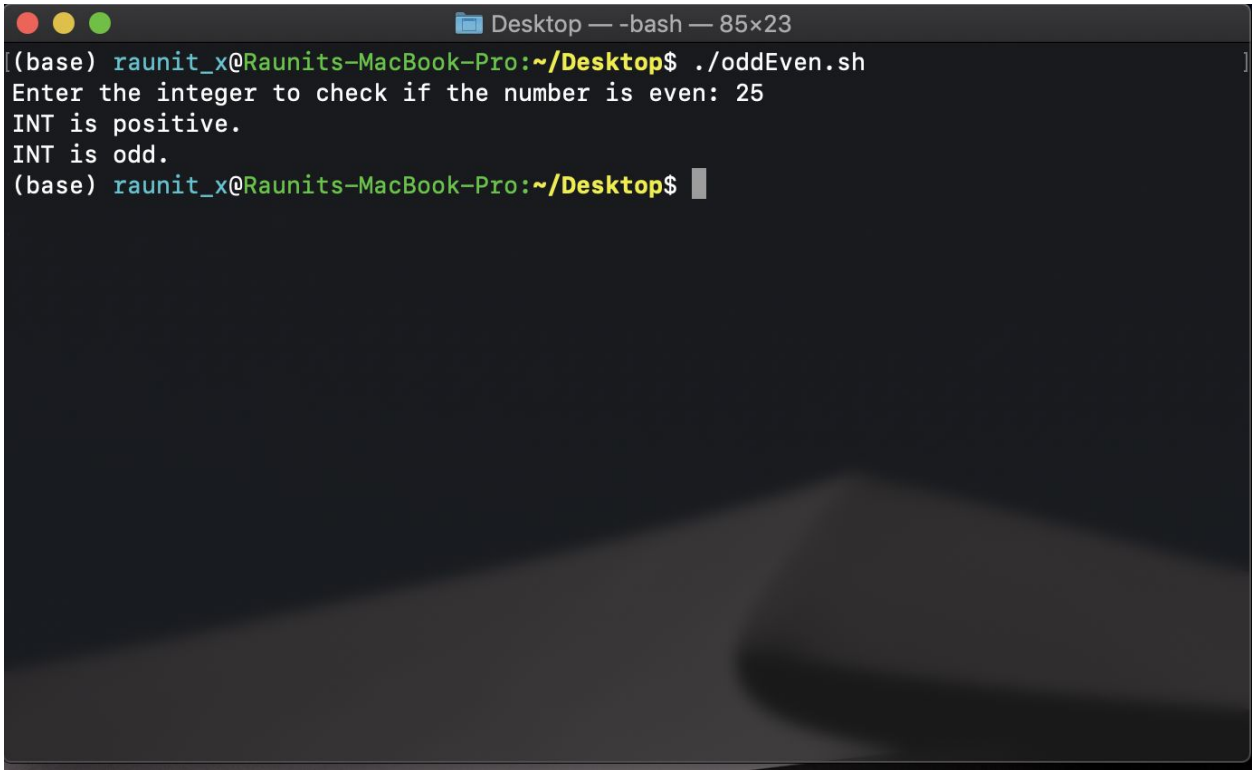

Desktop — -bash — 73×20
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ [-d demo] || mkdir demo
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ ls demo
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ ls -l demo
(base) raunit_x@Raunits-MacBook-Pro:~/Desktop$ ls
ACN practical final.pdf
Courses/
Cryptography codes/
DAA File/
Development/
Infosec File/
Infosec_unordered.pdf
Linux Lab/
Multimedia File/
Multimedia Lab codes/
Personal Projects/
Screenshot 2019-11-15 at 12.24.13 AM.png
Screenshot 2019-11-15 at 12.24.46 AM.png
barnburningbyharukimurakami.pdf
client.py*
demo/

Chapter 29-Reading Keyboard Input

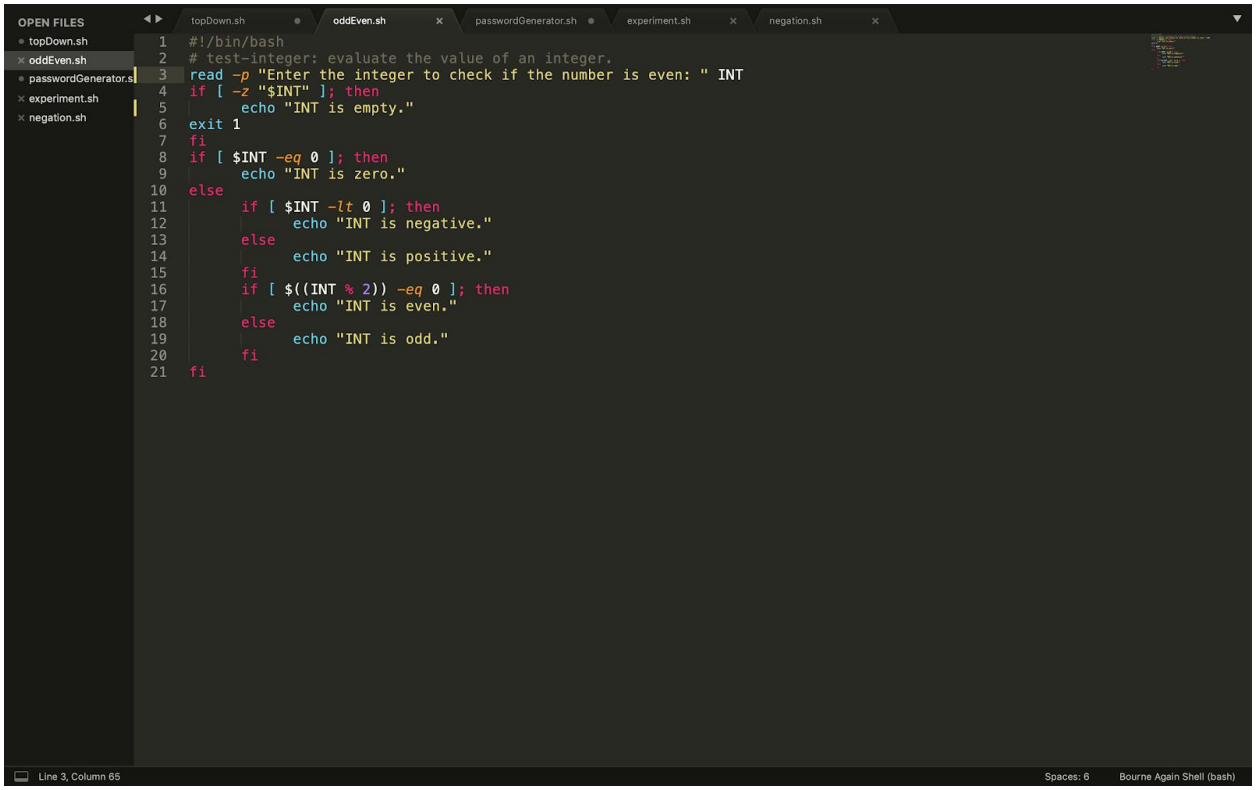
read – Read Values From Standard Input

The read builtin command is used to read a single line of standard input. This command can be used to read keyboard input or, when redirection is employed, a line of data from a file. The command has the following syntax:

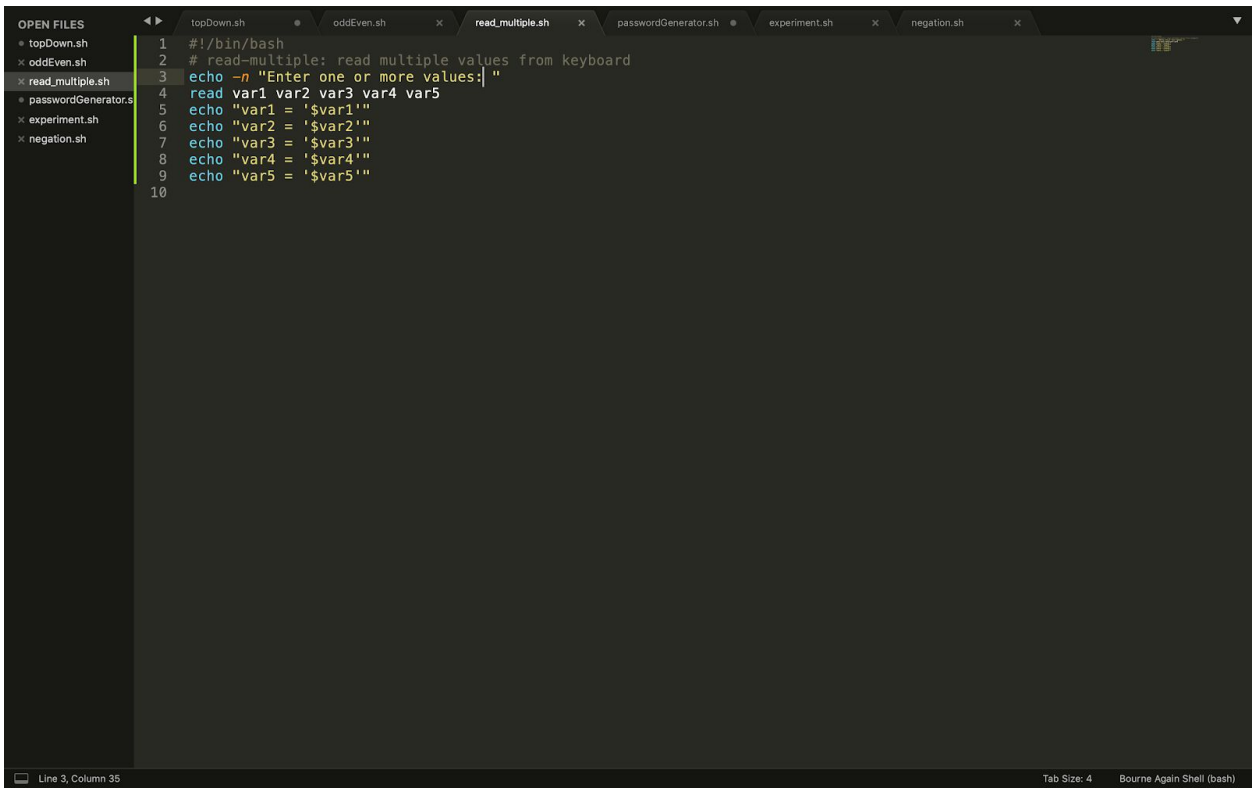
read [-options] [variable...]

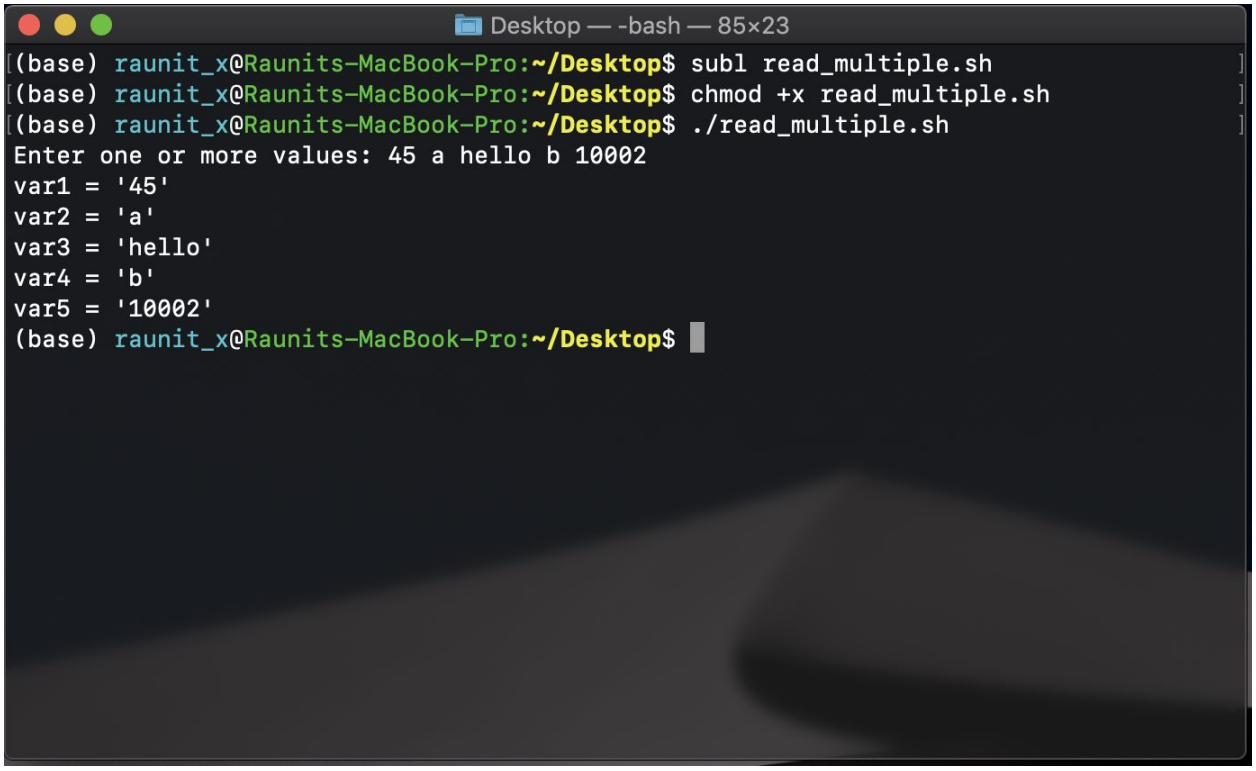


Script:

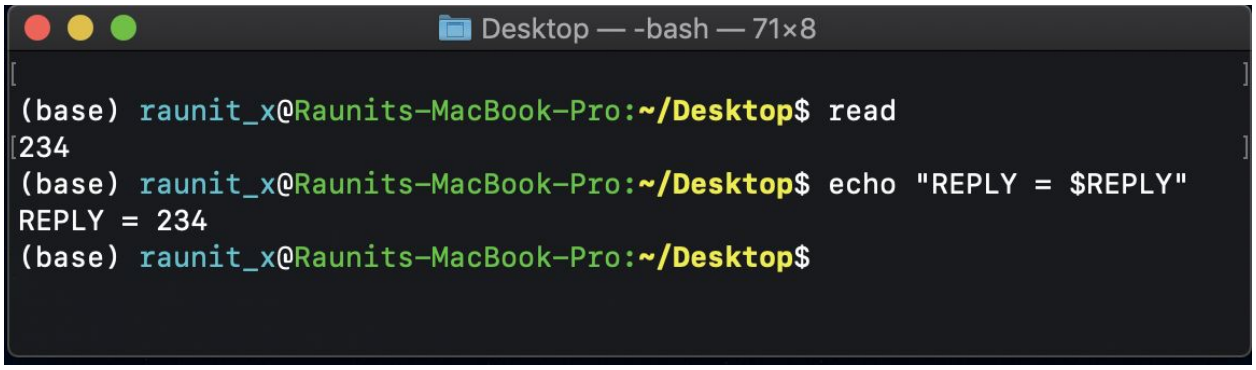


read can assign input to multiple variables, as shown in this script:





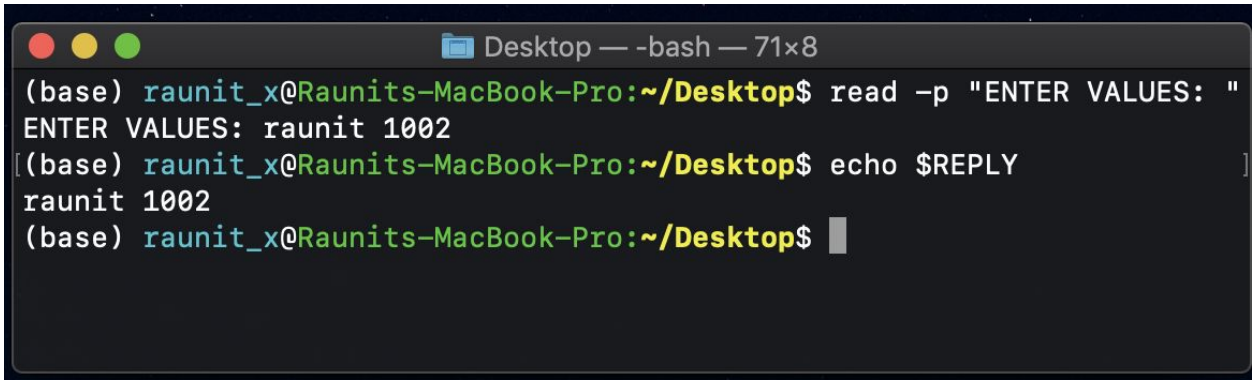
If read receives fewer than the expected number, the extra variables are empty, while an excessive amount of input results in the final variable containing all of the extra input.
If no variables are listed after the read command, a shell variable, REPLY, will be assigned all the input.



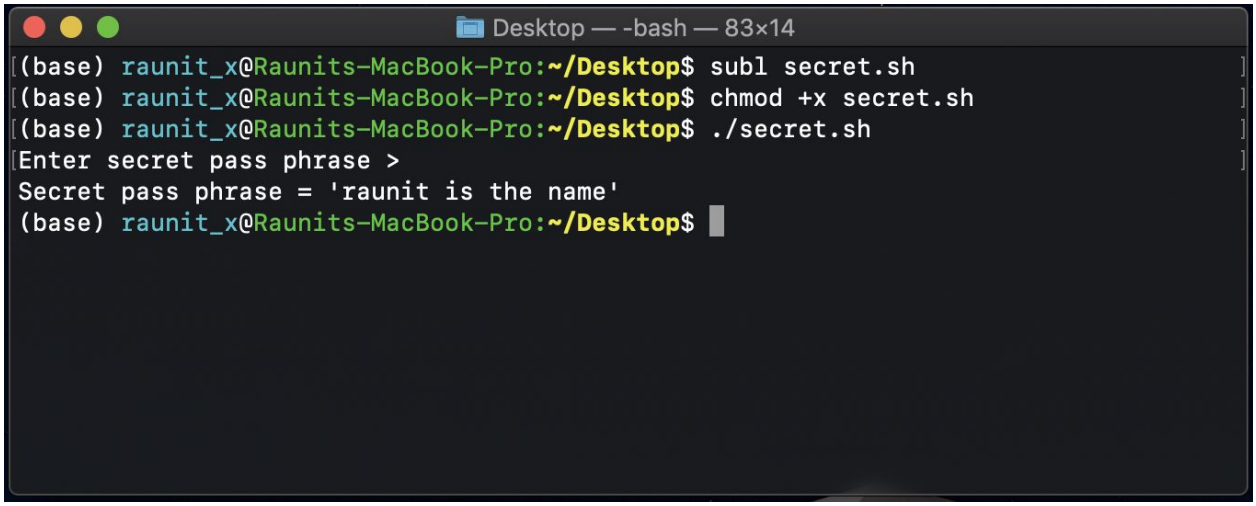
Options
read supports the following options:

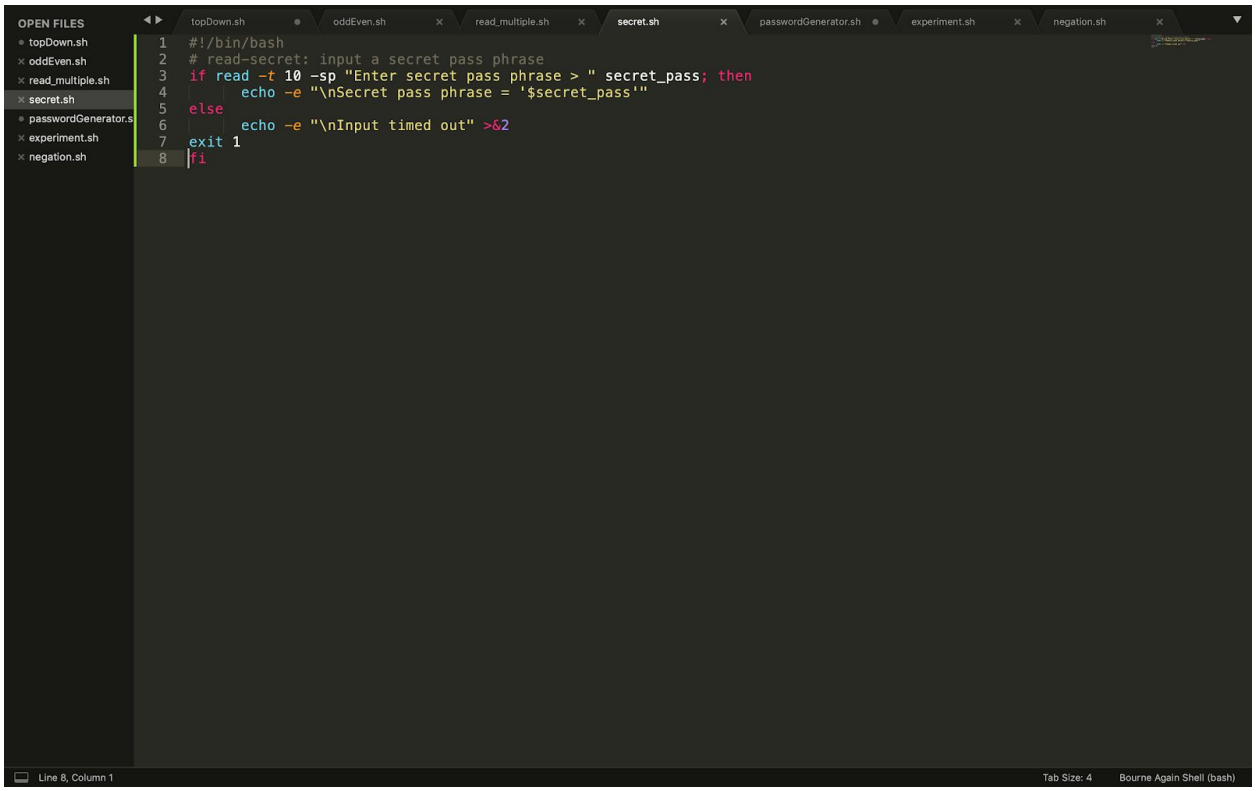
Option	Description
`-a array`	Assign the input to *array*, starting with index zero. We will cover arrays in Chapter 35.
`-d delimiter`	The first character in the string *delimiter* is used to indicate end of input, rather than a newline character.
`-e`	Use Readline to handle input. This permits input editing in the same manner as the command line.
`-i string`	Use *string* as a default reply if the user simply presses Enter. Requires the `-e` option.
`-n num`	Read *num* characters of input, rather than an entire line.
`-p prompt`	Display a prompt for input using the string *prompt*.
`-r`	Raw mode. Do not interpret backslash characters as escapes.
`-s`	Silent mode. Do not echo characters to the display as they are typed. This is useful when inputting passwords and other confidential information.
`-t seconds`	Timeout. Terminate input after *seconds*. `read` returns a non-zero exit status if an input times out.
`-u fd`	Use input from file descriptor *fd*, rather than standard input.

with the -p option, we can provide a prompt string:



With the -t and -s options we can write a script that reads “secret” input and times out if the input is not completed in a specified time:



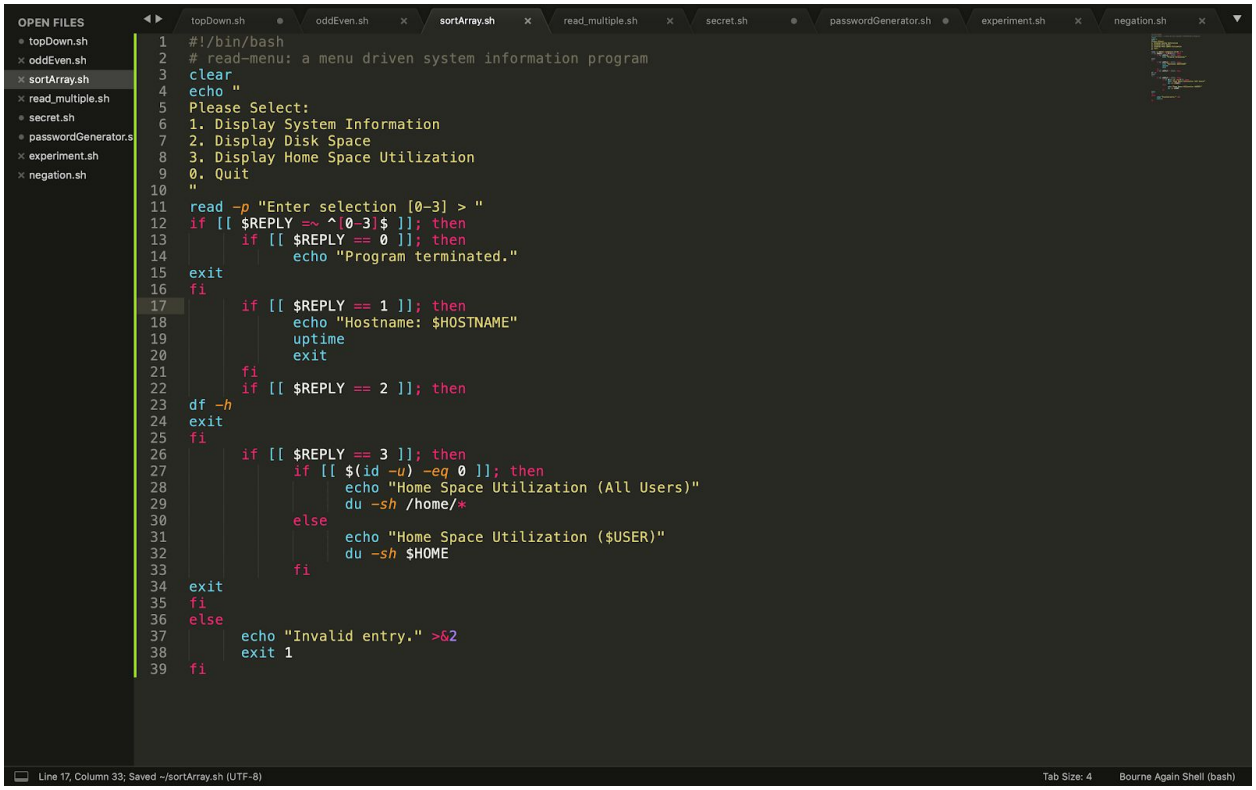


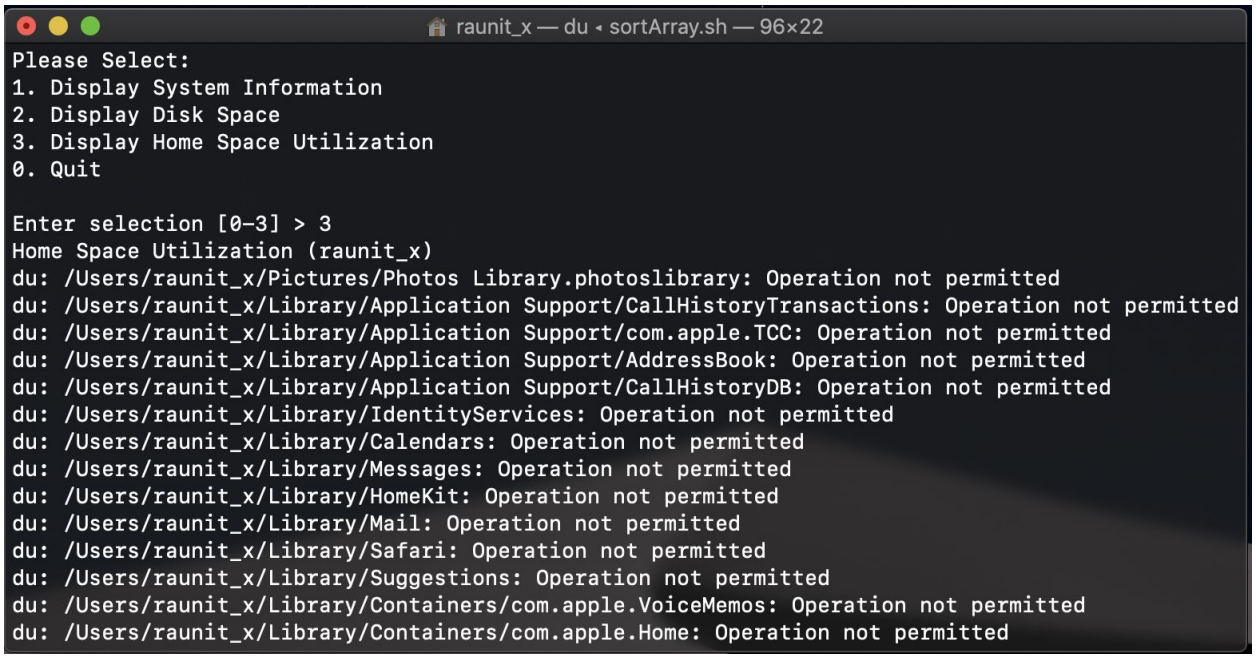
IFS

Normally, the shell performs word splitting on the input provided to read. As we have seen, this means that multiple words separated by one or more spaces become separate items on the input line, and are assigned to separate variables by read. This behavior is configured by a shell variable named IFS (for Internal Field Separator). The default value of IFS contains a space, a tab, and a newline character, each of which will separate items from one another. We can adjust the value of IFS to control the separation of fields input to read. For example, the /etc/passwd file contains lines of data that use the colon character as a field separator. By changing the value of IFS to a single colon, we can use read to input the contents of /etc/passwd and successfully separate fields into different variables.

Menus

A common type of interactivity is called menu-driven. In menu-driven programs, the user is presented with a list of choices and is asked to choose one. For example, we could imagine a program that presented the following:

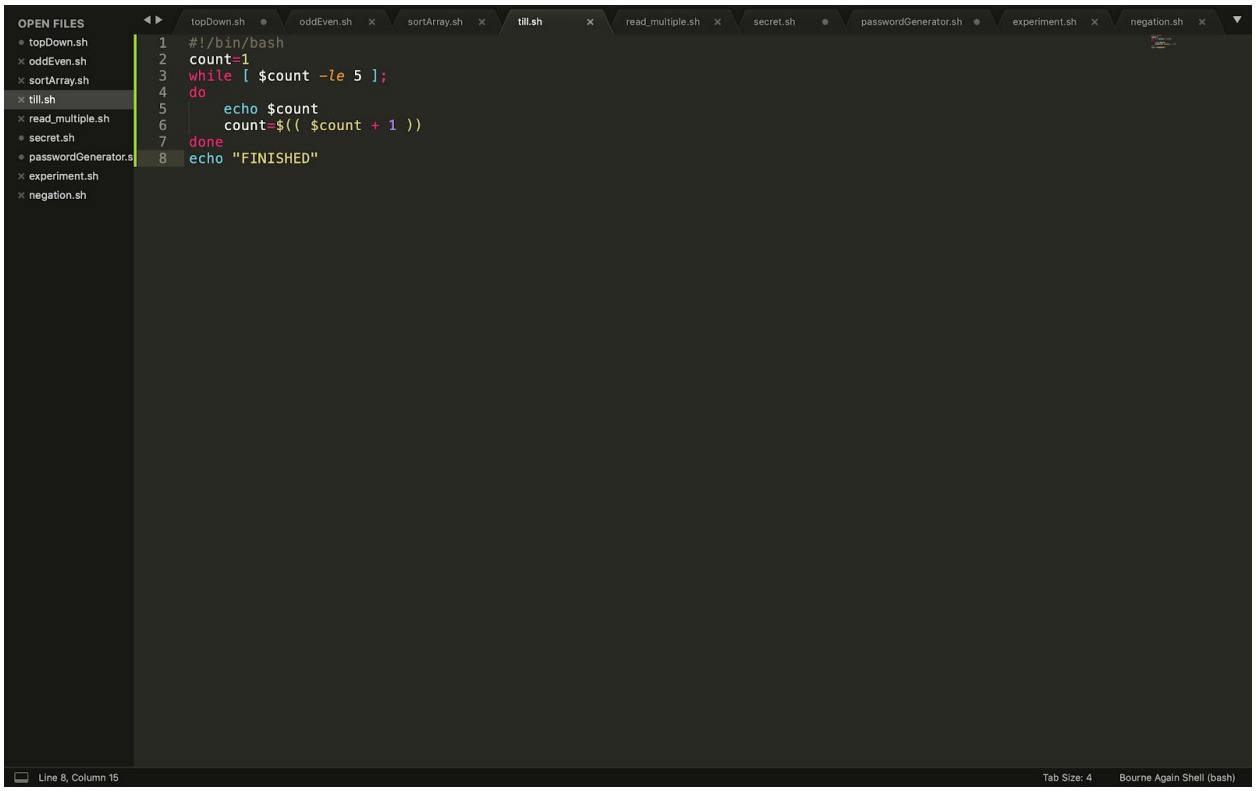

OPEN FILES
topDown.sh
oddEven.sh
sortArray.sh
read_multiple.sh
secret.sh
passwordGenerator.s
experiment.sh
negation.sh
topDown.sh
oddEven.sh
sortArray.sh
read_multiple.sh
secret.sh
passwordGenerator.sh
experiment.sh
negation.sh
#!/bin/bash
read-menu: a menu driven system information program
clear
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
if [[$REPLY =~ ^[0-3]$]]; then
if [[$REPLY == 0]]; then
echo "Program terminated."
exit
fi
if [[$REPLY == 1]]; then
echo "Hostname: $HOSTNAME"
uptime
exit
fi
if [[$REPLY == 2]]; then
df -h
exit
fi
if [[$REPLY == 3]]; then
if [[$(id -u) -eq 0]]; then
echo "Home Space Utilization (All Users)"
du -sh /home/*
else
echo "Home Space Utilization ($USER)"
du -sh $HOME
fi
exit
fi
else
echo "Invalid entry." >&2
exit 1
fi
Line 17, Column 33; Saved ~/sortArray.sh (UTF-8)
Tab Size: 4
Bourne Again Shell (bash)

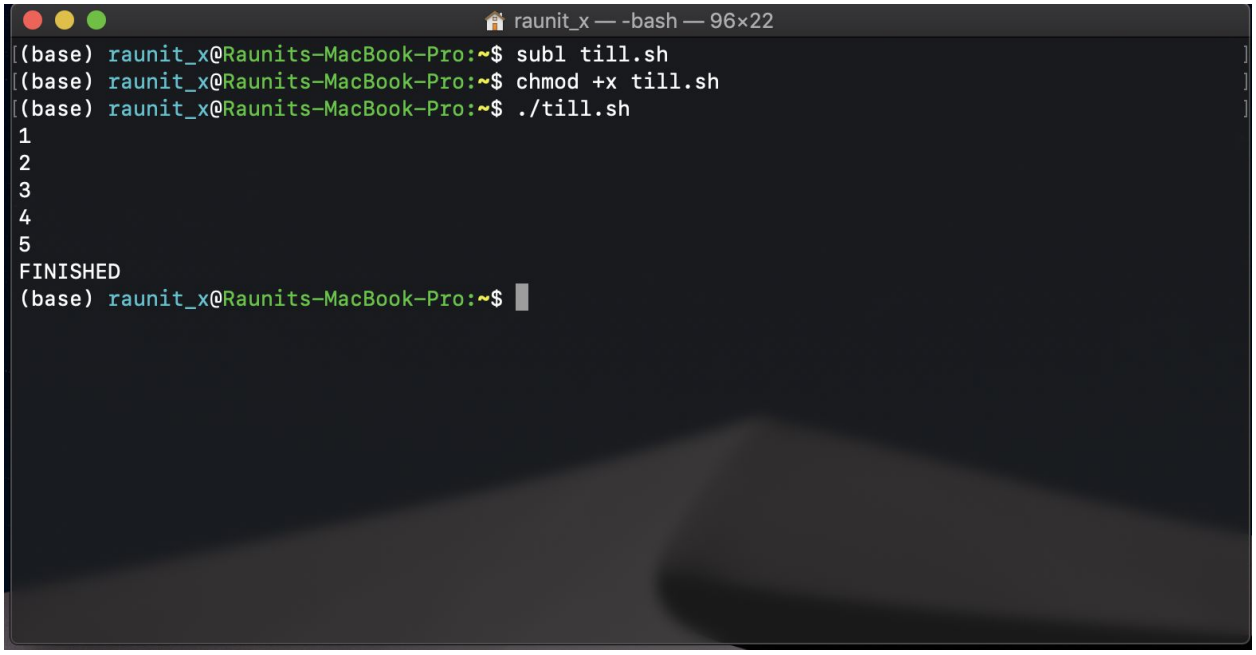

raunit_x — du ◂ sortArray.sh — 96×22
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
Enter selection [0-3] > 3
Home Space Utilization (raunit_x)
du: /Users/raunit_x/Pictures/Photos Library.photoslibrary: Operation not permitted
du: /Users/raunit_x/Library/Application Support/CallHistoryTransactions: Operation not permitted
du: /Users/raunit_x/Library/Application Support/com.apple.TCC: Operation not permitted
du: /Users/raunit_x/Library/Application Support/AddressBook: Operation not permitted
du: /Users/raunit_x/Library/Application Support/CallHistoryDB: Operation not permitted
du: /Users/raunit_x/Library/IdentityServices: Operation not permitted
du: /Users/raunit_x/Library/Calendars: Operation not permitted
du: /Users/raunit_x/Library/Messages: Operation not permitted
du: /Users/raunit_x/Library/HomeKit: Operation not permitted
du: /Users/raunit_x/Library/Mail: Operation not permitted
du: /Users/raunit_x/Library/Safari: Operation not permitted
du: /Users/raunit_x/Library/Suggestions: Operation not permitted
du: /Users/raunit_x/Library/Containers/com.apple.VoiceMemos: Operation not permitted
du: /Users/raunit_x/Library/Containers/com.apple.Home: Operation not permitted

Chapter 30-Flow Control: Looping With while / until

Looping
while

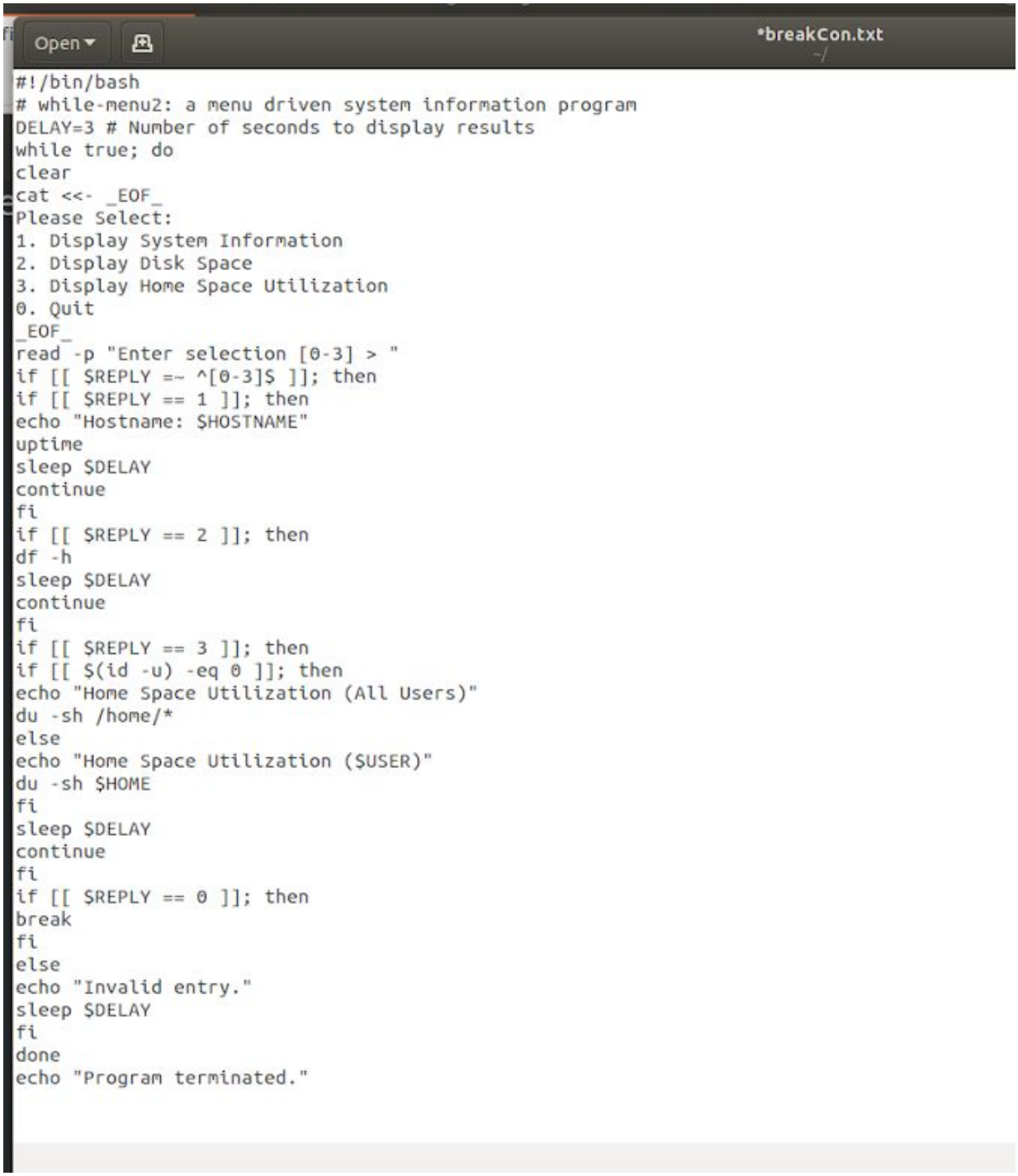
Let's say we wanted to display five numbers in sequential order from one to five. a bash script could be constructed as follows:





Breaking Out Of A Loop

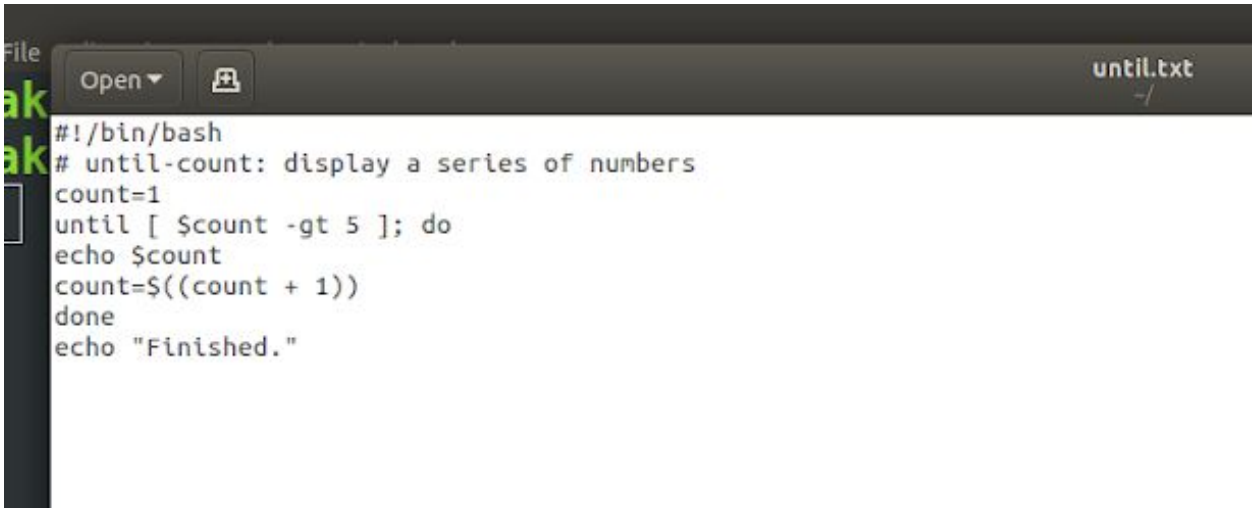
bash provides two builtin commands that can be used to control program flow inside loops. The break command immediately terminates a loop, and program control resumes with the next statement following the loop. The continue command causes the remainder of the loop to be skipped, and program control resumes with the next iteration of the loop. Here we see a version of the while-menu program incorporating both break and continue:



until

The until command is much like while, except instead of exiting a loop when a nonzero exit status is encountered, it does the opposite. An until loop continues until it receives a zero exit status. In our while-count script, we continued the loop as long as the value of the count

variable was less than or equal to 5. We could get the same result by coding the script with until:



Chapter 32-Flow Control: Branching With case

We constructed some simple menus and built the logic used to act on a user's selection. To do this, we used a series of `if` commands to identify which of the possible choices has been selected. This type of construct appears frequently in programs, so much so that many programming languages (including the shell) provide a flow control mechanism for multiple-choice decisions.

case *word* in
    [pattern [| pattern]...) commands ;;]...

```
#!/bin/bash
# case-menu: a menu driven system information program
echo "
Please Select:
1. Display System Information
2. Display Disk Space
3. Display Home Space Utilization
0. Quit
"
read -p "Enter selection [0-3] > "
case $REPLY in
     0) echo "Program terminated."
     exit
     ;;
     1) echo "Hostname: $HOSTNAME"
     uptime
     ;;
     2) df -h
     ;;
     3) if [[ $(id -u) -eq 0 ]]; then
          echo "Home Space Utilization (All Users)"
```

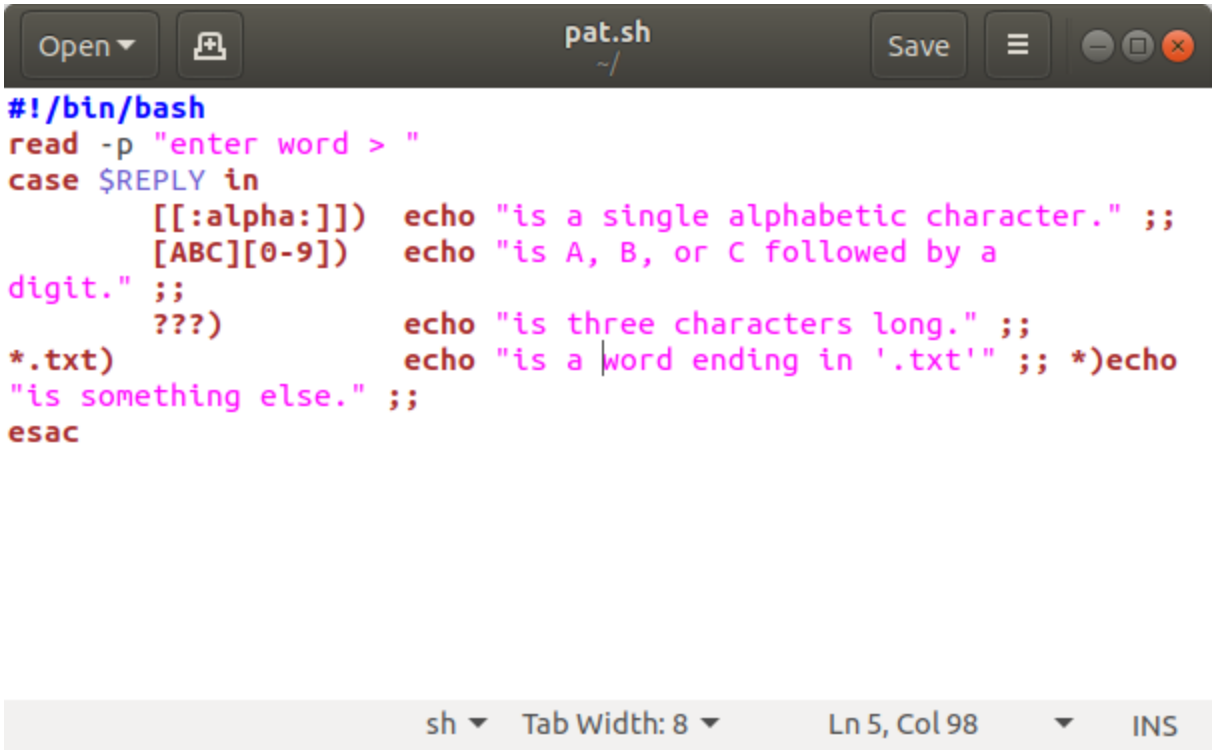
The `case` command looks at the value of word, in our example, the value of the `REPLY` variable, and then attempts to match it against one of the specified patterns.

When a match is found, the commands associated with the specified pattern are executed. After a match is found, no further matches are attempted.

Patterns

The patterns used by `case` are the same as those used by pathname expansion. Patterns are terminated with a ")" character. Here are some valid patterns:

Pattern	Description
`a)`	Matches if word equals "a".
`[[:alpha:]])`	Matches if word is a single alphabetic
character.`???)`	Matches if word is exactly three characters long.
`*.txt)`	Matches if word ends with the characters ".txt".
`*)`	Matches any value of word.


Open
pat.sh
~/
Save
#!/bin/bash
read -p "enter word > "
case $REPLY in
[[:alpha:]]) echo "is a single alphabetic character." ;;
[ABC][0-9]) echo "is A, B, or C followed by a
digit." ;;
???) echo "is three characters long." ;;
*.txt) echo "is a word ending in '.txt'" ;; *)echo
"is something else." ;;
esac
sh
Tab Width: 8
Ln 5, Col 98
INS

Chapter 34-Flow Control: Looping With for

The for loop differs from the while and until loops in that it provides a means of processing sequences during a loop. This turns out to be very useful when programming. Accordingly, the for loop is a very popular construct in bash scripting.A for loop is implemented, naturally enough, with the for command. In modern versions of bash, for is available in two forms.

for：Traditional Shell Form

The original `for` command's syntax is:

```
for variable [in words];

do
     commands
Done
```

Where variable is the name of a variable that will increment during the execution of the loop, words is an optional list of items that will be sequentially assigned to variable, and commands are the commands that are to be executed on each iteration of the loop.

In this example, `for` is given a list of four words: "A," "B," "C," and "D." With a list of our words, the loop is executed four times. Each time the loop is executed, a word is assigned to the variable `i`.

Inside the loop, we have an `echo` command that displays the value of i to show the assignment.
As with the while and until loops, the done keyword closes the loop.

for: C Language Form

Recent versions of bash have added a second form of for command syntax, one that resembles the form found in the C programming language. Many other languages support this form, as well:

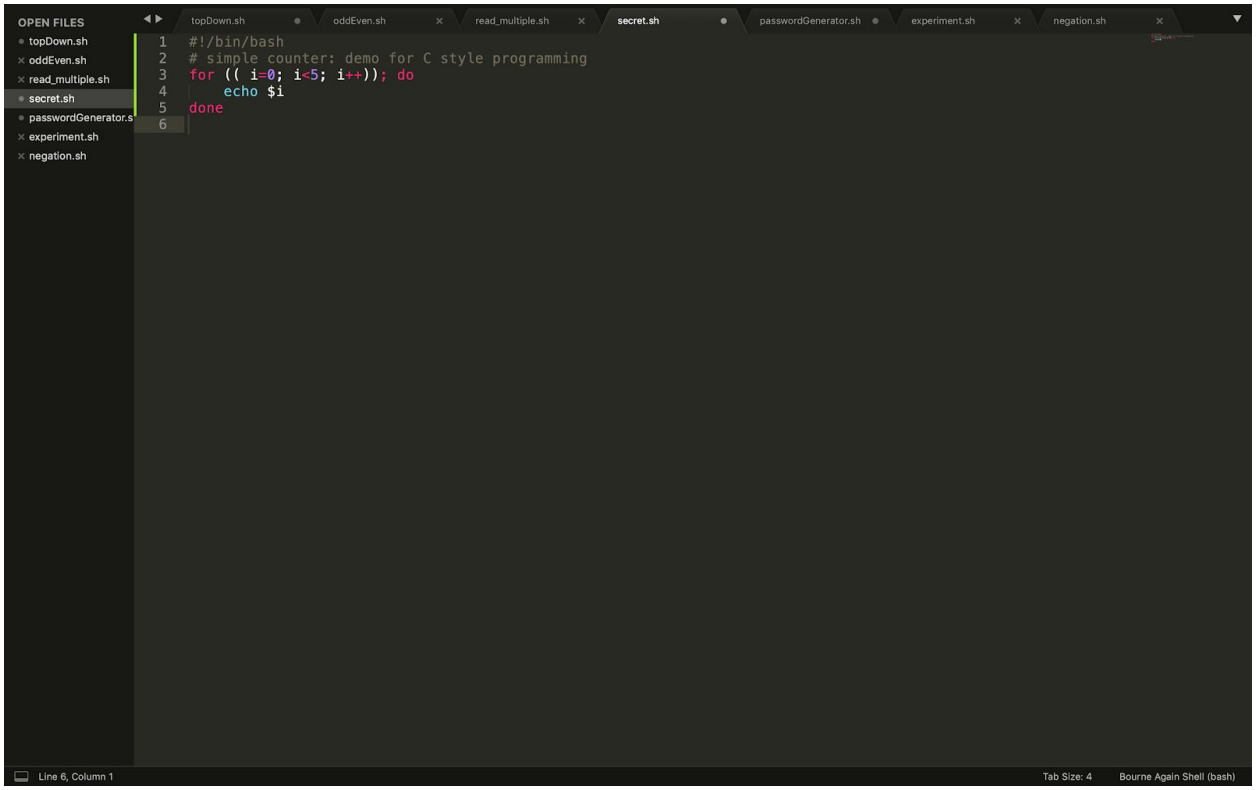
```
for (( expression1; expression2; expression3 )); do
     Commands
done
```

where expression1, expression2, and expression3 are arithmetic expressions and commands are the commands to be performed during each iteration of the loop.

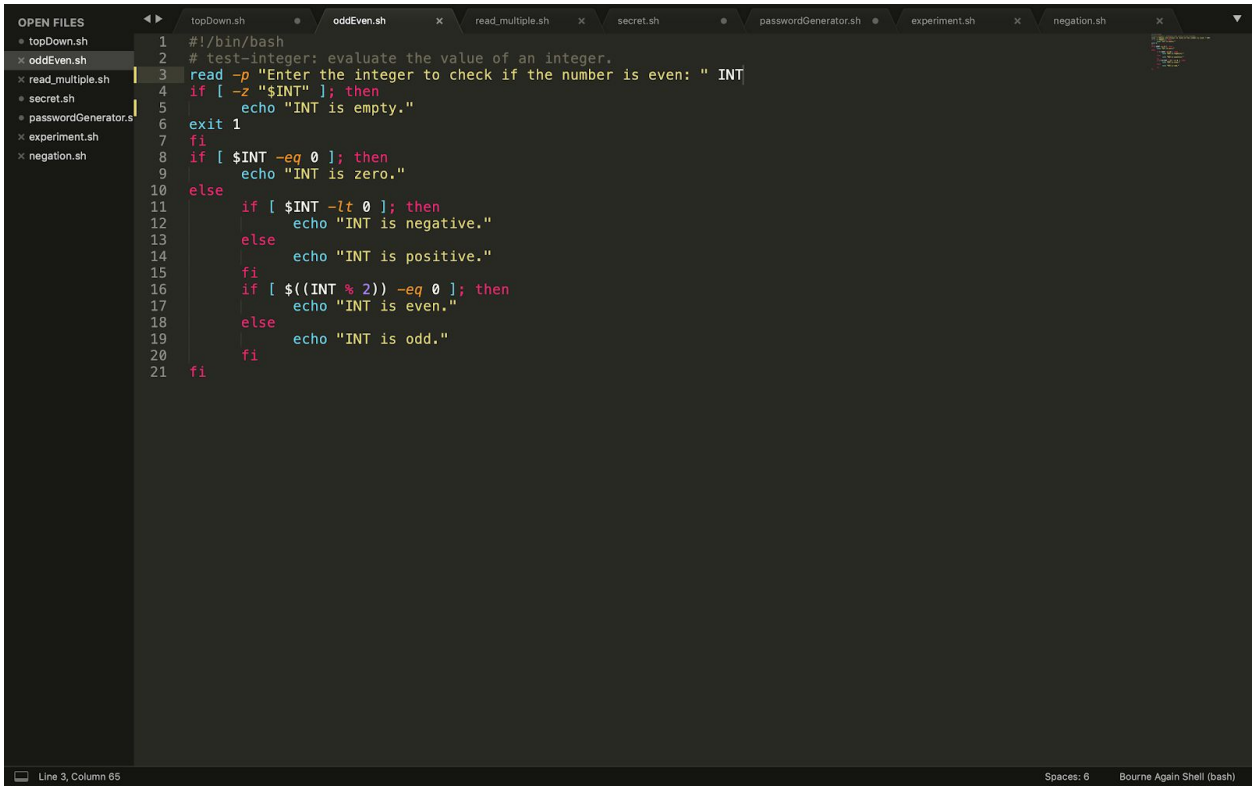
In terms of behavior, this form is equivalent to the following construct:

```
(( expression1 ))
while (( expression2 )); do
     Commands
(( expression3 ))
done
```

expression1 is used to initialize conditions for the loop, expression2 is used to determine when the loop is finished and expression3 is carried out at the end of each iteration of the loop.



In this example, expression1 initializes the variable i with the value of zero, expression2 allows ethe loop to continue as long as the value of i remains less than five, and expression3 increments the value of i by one each time the loop repeats.


OPEN FILES
topDown.sh
oddEven.sh
read_multiple.sh
secret.sh
passwordGenerator.s
experiment.sh
negation.sh
topDown.sh
oddEven.sh
read_multiple.sh
secret.sh
passwordGenerator.sh
experiment.sh
negation.sh
#!/bin/bash
test-integer: evaluate the value of an integer.
read -p "Enter the integer to check if the number is even: " INT
if [-z "$INT"]; then
echo "INT is empty."
exit 1
fi
if [$INT -eq 0]; then
echo "INT is zero."
else
if [$INT -lt 0]; then
echo "INT is negative."
else
echo "INT is positive."
fi
if [$((INT % 2)) -eq 0]; then
echo "INT is even."
else
echo "INT is odd."
fi
fi
Line 3, Column 65
Spaces: 6
Bourne Again Shell (bash)

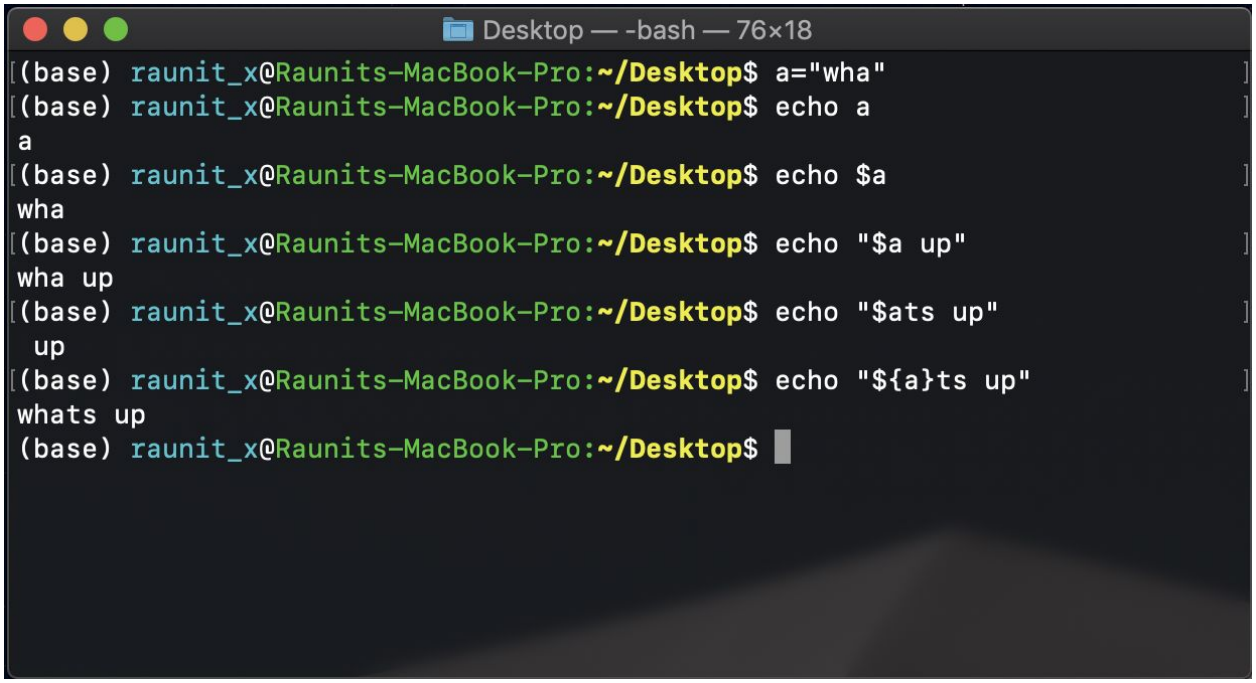
Chapter 35-Strings And Numbers

Basic Parameters

The simplest form of parameter expansion is reflected in the ordinary use of variables. For example: $a when expanded, becomes whatever the variable a contains. Simple parameters may also be surrounded by braces:${a}

This has no effect on the expansion, but is required if the variable is adjacent to other text, which may confuse the shell.

In this example, we attempt to create a filename by appending the string "_file" to the contents of the variable a.



Expansions To Manage Empty Variables

1. `${parameter:-word}` If parameter is unset (i.e., does not exist) or is empty, this expansion results in the valueof word. If parameter is not empty, the expansion results in the value of parameter.

2. `${parameter:=word}` If parameter is unset or empty, this expansion results in the value of word. In addition,the value of word is assigned to parameter. If parameter is not empty, the expansion results in the value of parameter.

3. `${parameter:?word}` If parameter is unset or empty, this expansion causes the script to exit with an error, and the contents of word are sent to standard error. If parameter is not empty, the expansion results in the value of parameter.

4. `${parameter:+word}` If parameter is unset or empty, the expansion results in nothing. If parameter is not empty, the value of word is substituted for parameter; however, the value of parameter is not changed.

Arithmetic Evaluation And Expansion

`$((expression))`

where expression is a valid arithmetic expression.This is related to the compound command (( )) used for arithmetic evaluation (truth tests)

Unary Operators

There are two unary operators, the + and -, which are used to indicate if a number is positive or negative, respectively. For example, -5.

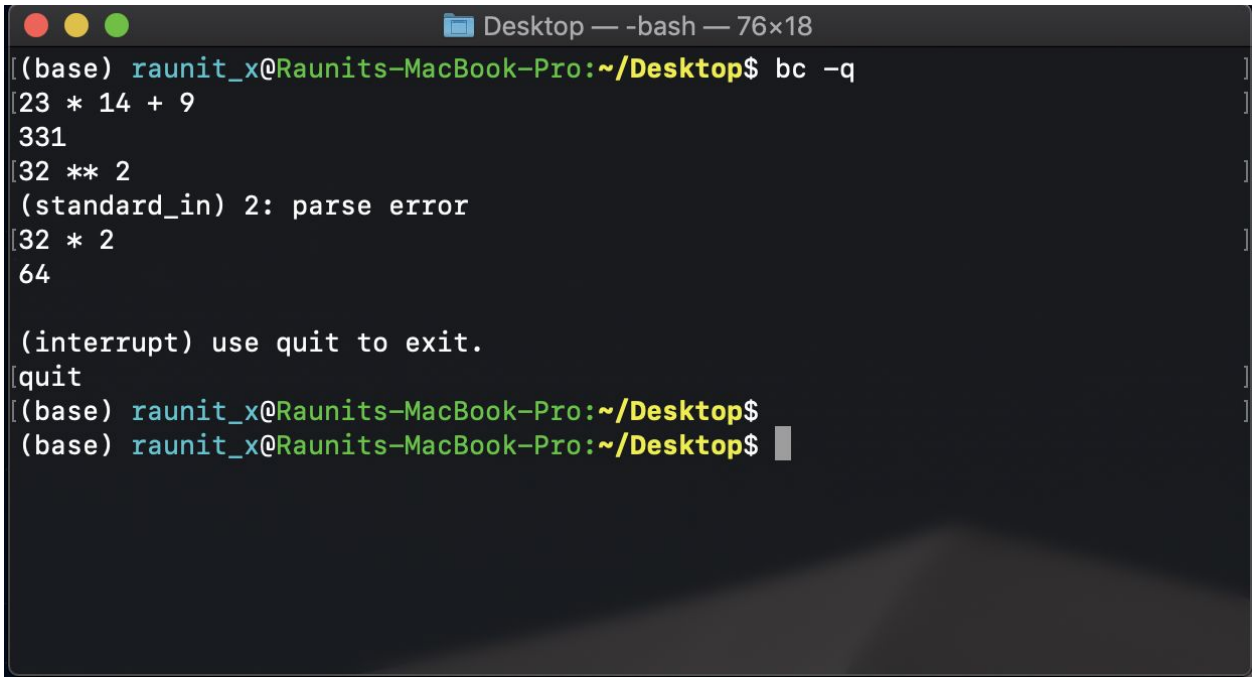
Bit Operations

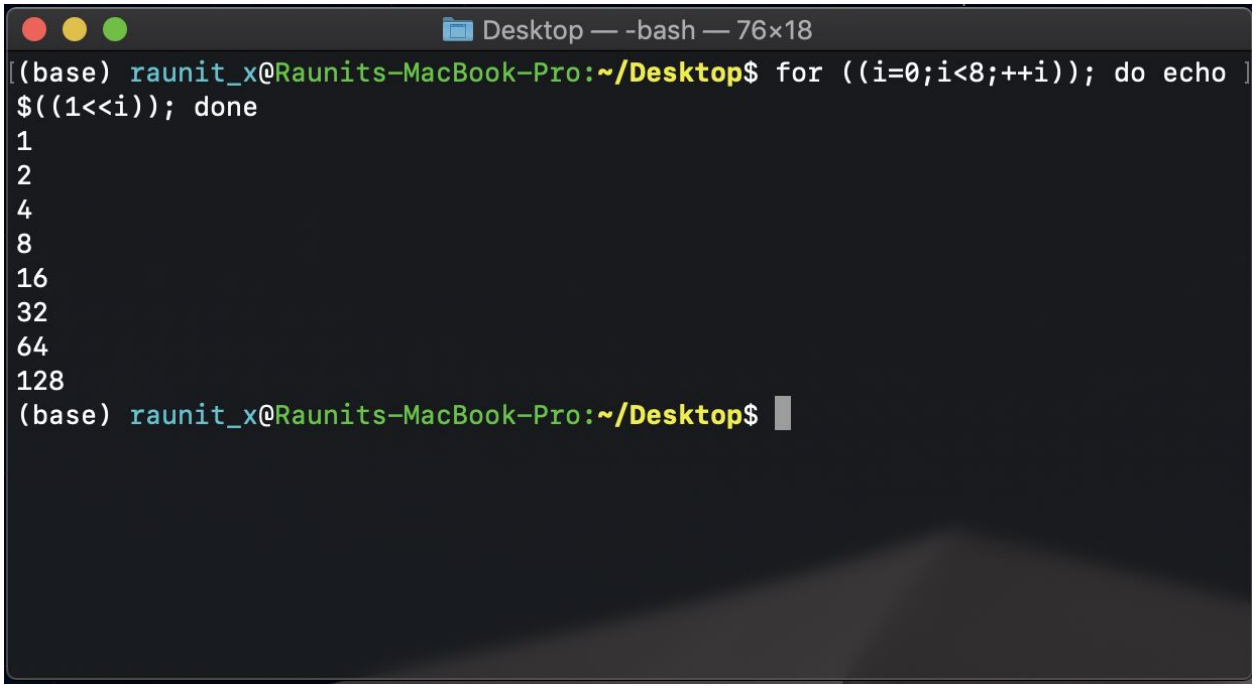
One class of operators manipulates numbers in an unusual way. These operators work at the bit level. They are used for certain kinds of low level tasks, often involving setting or reading bit-flags.

bc – An Arbitrary Precision Calculator Language

We have seen how the shell can handle all manner of integer arithmetic, but what if we need to perform higher math or even just use floating point numbers? The answer is, we can't. At least not directly with the shell. To do this, we need to use an external program.

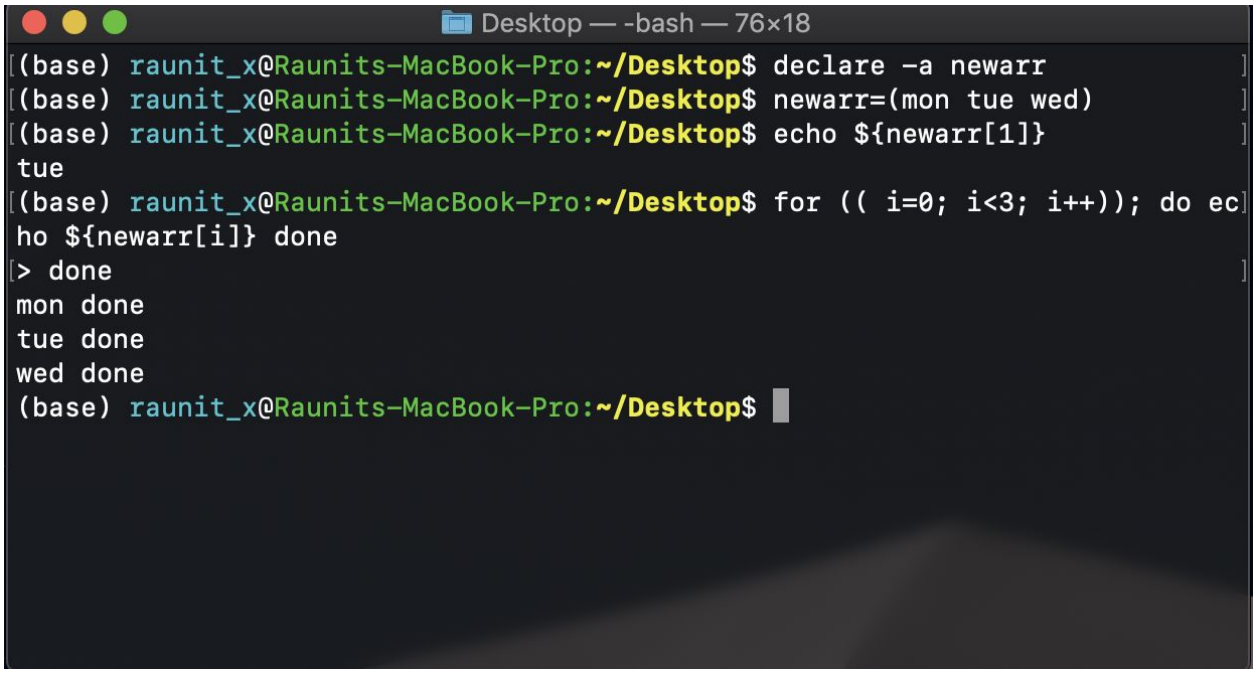
The bc program reads a file written in its own C-like language and executes it. A bc script may be a separate file or it may be read from standard input. The bc language supports quite a few features including variables, loops, and programmer-defined functions.

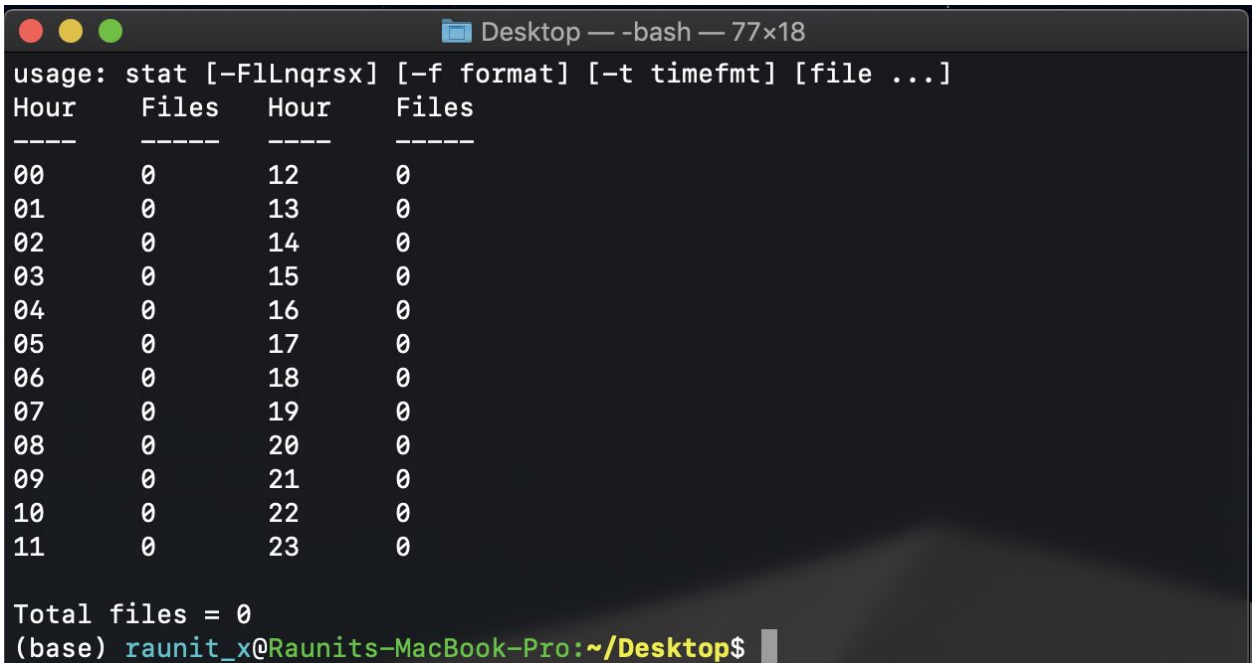




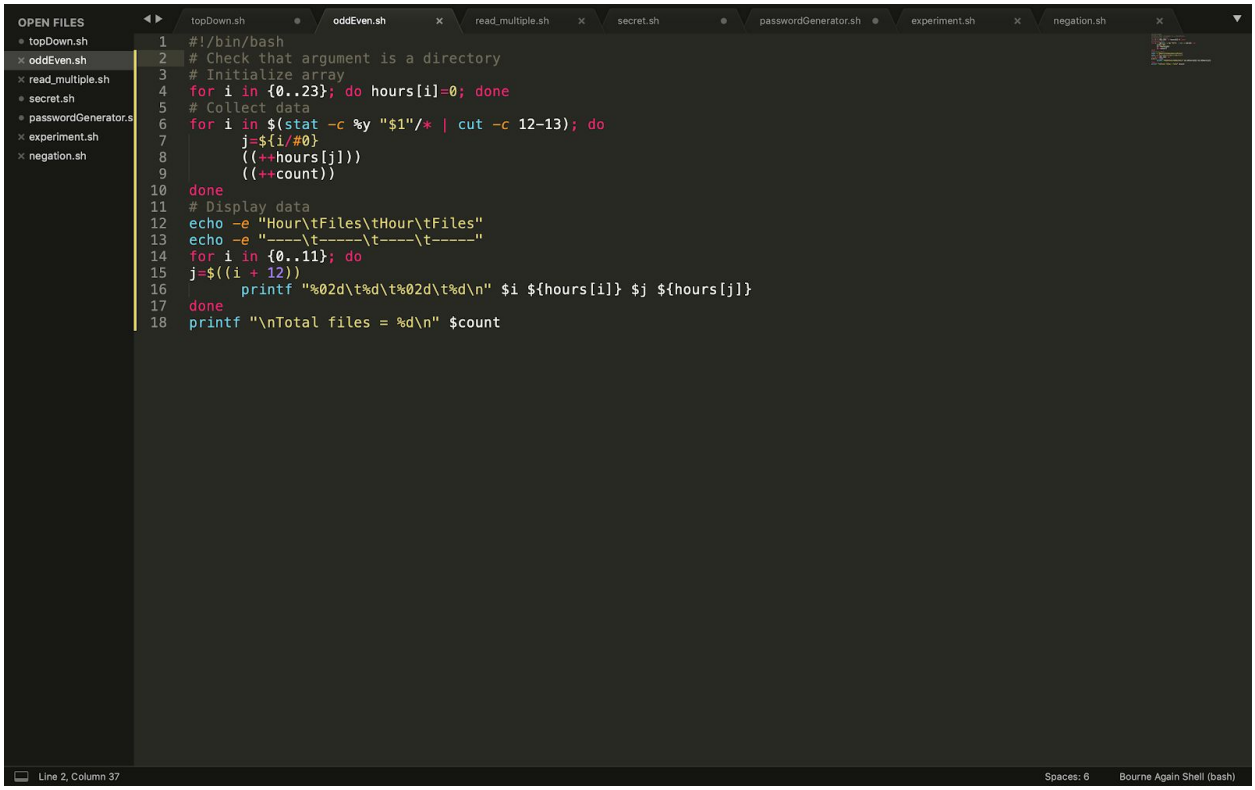
Chapter 36-Arrays

Arrays are variables that hold more than one value at a time. Arrays are organized like a table. A spreadsheet acts like a two-dimensional array. It has both rows and columns, and an individual cell in the spreadsheet can be located according to its row and column address. An array behaves the same way.
An array has cells, which are called elements, and each element contains data. An individual array element is accessed using an address called an index or subscript.





Script for the above output:

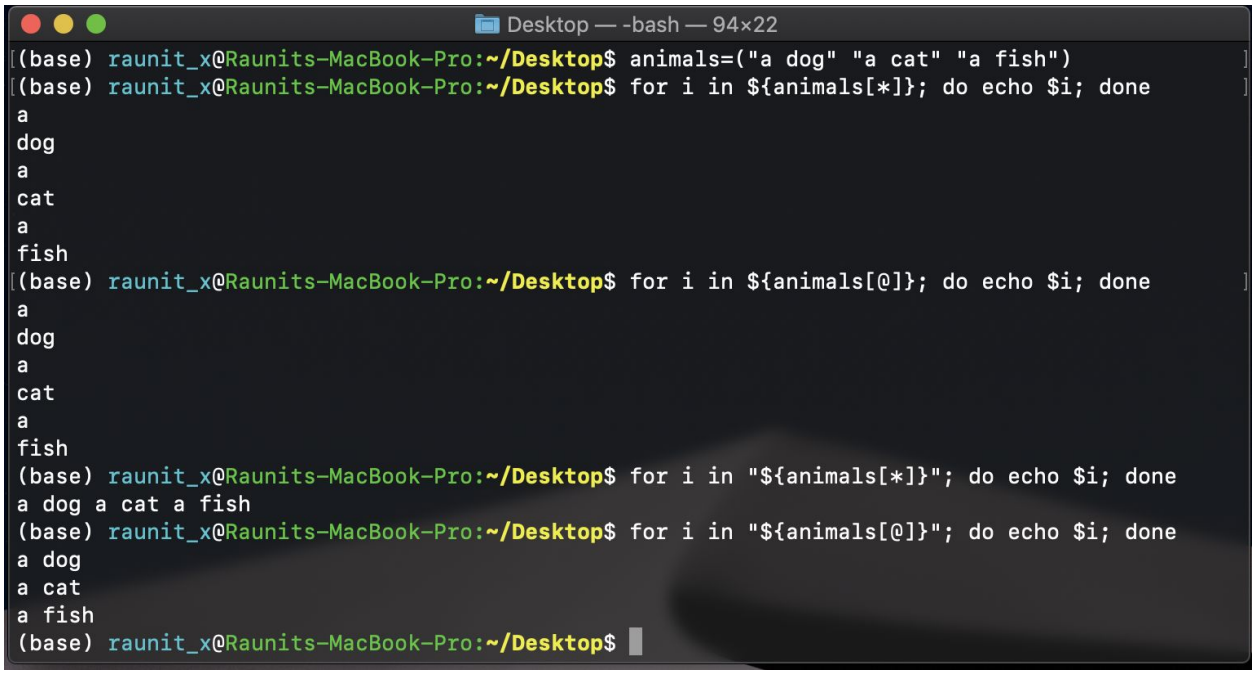


Array Operations

There are many common array operations. Such things as deleting arrays, determining their size, sorting, etc. have many applications in scripting.

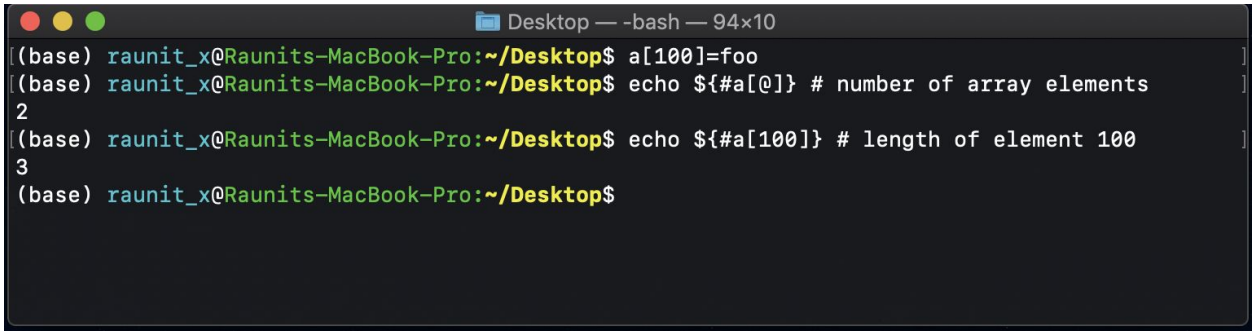
Outputting The Entire Contents Of An Array

The subscripts * and @ can be used to access every element in an array. As with positional parameters, the @ notation is the more useful of the two. Here is a demonstration:



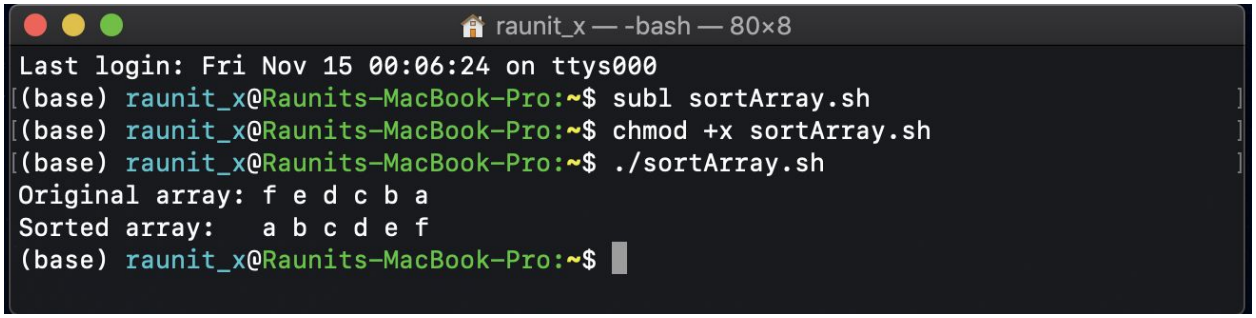
Determining The Number Of Array Elements

Using parameter expansion, we can determine the number of elements in an array in much the same way as finding the length of a string. Here is an example:



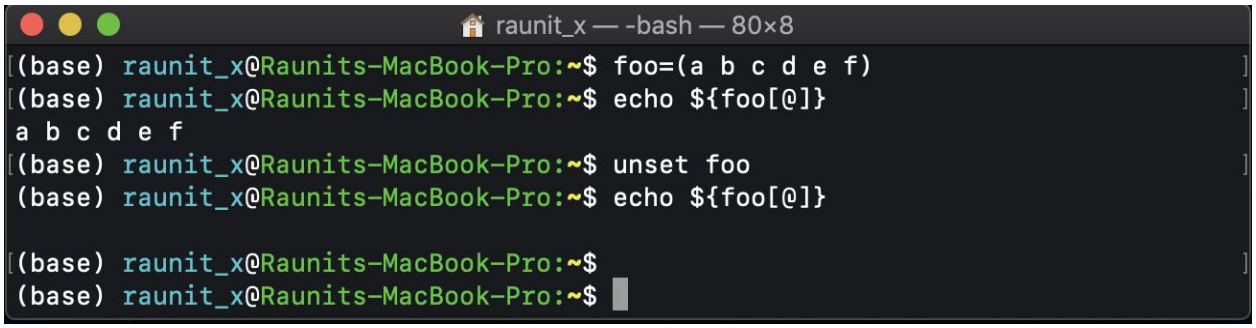
Sorting An Array

Just as with spreadsheets, it is often necessary to sort the values in a column of data. The shell has no direct way of doing this, but it's not hard to do with a little coding:



Deleting An Array

To delete an array, use the unset command:


raunit_x — -bash — 80×8
(base) raunit_x@Raunits-MacBook-Pro:~$ foo=(a b c d e f)
(base) raunit_x@Raunits-MacBook-Pro:~$ echo ${foo[@]}
a b c d e f
(base) raunit_x@Raunits-MacBook-Pro:~$ unset foo
(base) raunit_x@Raunits-MacBook-Pro:~$ echo ${foo[@]}
(base) raunit_x@Raunits-MacBook-Pro:~$
(base) raunit_x@Raunits-MacBook-Pro:~$